Reinforcement Learning

Modern Deep Reinforcement Learning

Jakub Chojnacki

About the Author

Who Am I

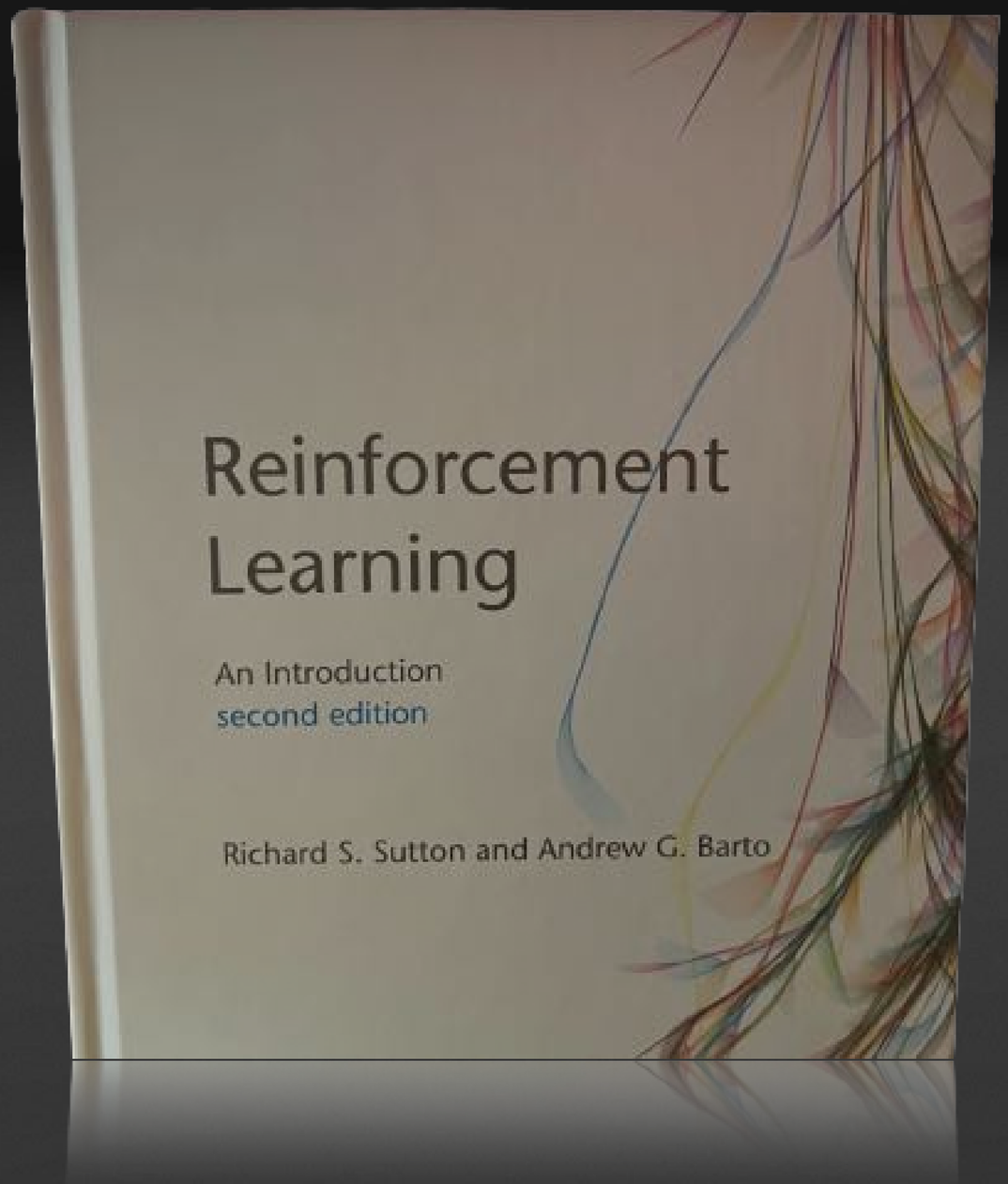
Admin
What & When

1. **Introduction** - **[Office: 2.11.2022] & [Online: 3.11.2022] @ 12.00 pm - 1.00 pm**
    1. Deep dive into RL systems
    2. Intuition building
2. **Tabular methods** - **[Office: 9.11.2022] & [Online: 10.11.2022] @ 12.00 pm - 1.00 pm**
    1. Dynamic programming
    2. Monte Carlo methods
    3. Temporal difference methods
3. Approximate methods - **[Office: 16.11.2022] & [Online: 17.11.2022] @ 12.00 pm - 1.00 pm**
    1. Intro to Deep Reinforcement Learning
    2. Value function approximation
    3. Policy gradient methods
4. **Modern Deep Reinforcement Learning** - **[Office: 23.11.2022] & [Online: 24.11.2022] @ 12.00 pm - 1.00 pm**
    1. Grokking modern algorithms - { DDQN, PER, QRDQN, Rainbow DQN, DDPG, TD3, PPO, SAC }
    2. Intrinsically Motivated Reinforcement Learning
5. **Coding session** - **[Office: 30.11.2022] & [Online: 1.12.2022] @ 12.00 pm - 1.00 pm**
    1. Learn how to code RL environment using OpenAI API
    2. Choose algorithm, solve the problem and evaluate your agent

Materials

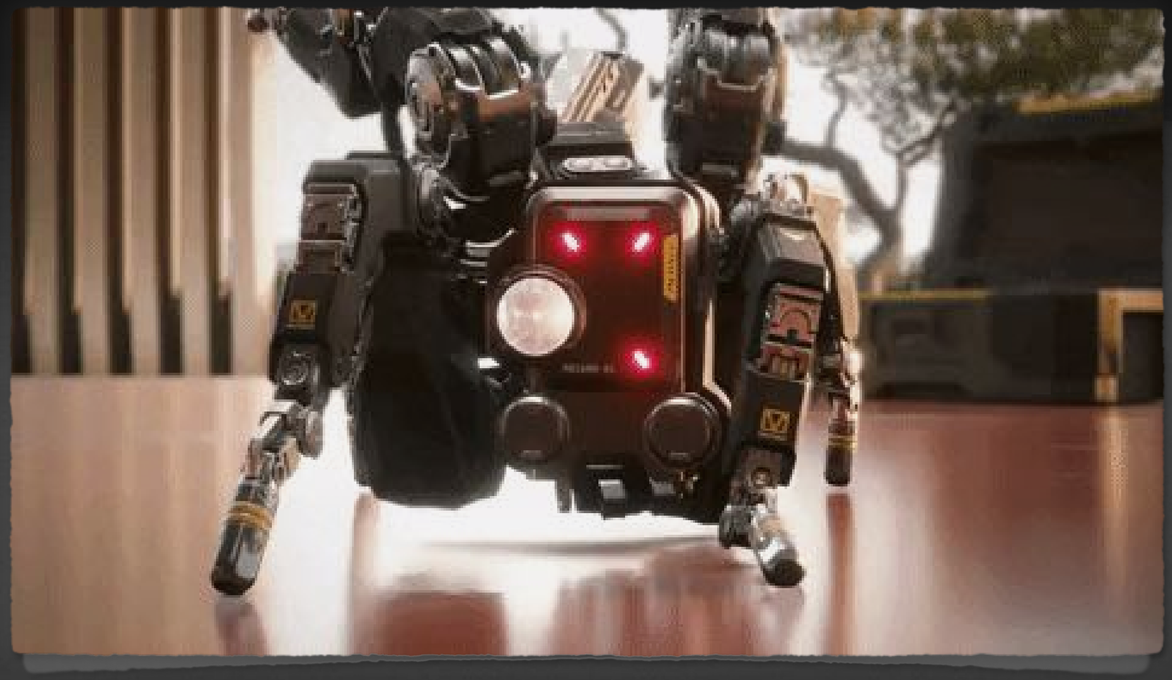
What was used

1. Videos: [David Silver] “Introduction to Reinforcement Learning 2015”
2. Book: [Sutton & Barto] “Reinforcement Learning An Introduction 2nd edition”
3. Book: [Miguel Morales] “Grokking Deep Reinforcement Learning”



Agenda
Topics covered

1. Improvements to Value Networks
2. Advanced Experience Replay Strategies
3. Distributional Reinforcement Learning
4. Improvements to Actor-Critic Methods
5. Advanced Actor-Critic Methods



Improvements to Value Networks

Double and Duelling Networks

Boston Dynamics, Atlas

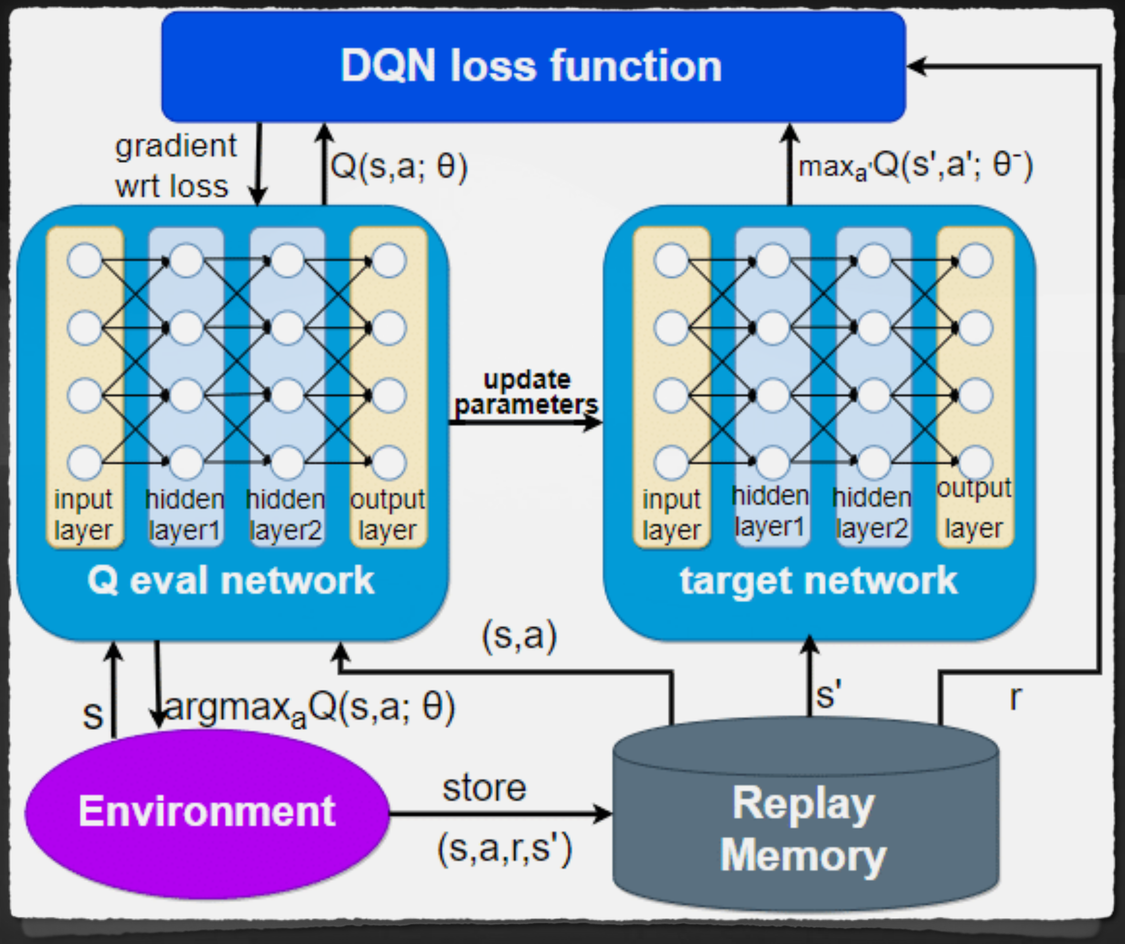
DQN
Deep Q-Learning Network

1. DQN [off-policy] uses **experience replay** and **target networks**
   - Take action a_t according to $\epsilon - greedy$ policy
   - Store trajectory [τ] (s, a, r, s') in **replay memory** [D]
   - Sample random **mini-batch** of trajectories from replay memory
   - Compute Q-learning targets $w\,.\,r\,.\,t$ **target network** parameters [$\overline{w}$]
   - Optimise **MSE** between q-network and q-learning targets

$$\mathscr{L}(w_i) = \mathbb{E}_{s,a,r,s' \sim D_i}[\left(r + \gamma \max_{a'} \hat{q}(s', a', \overline{w}_i) - \hat{q}(s, a, w_i)\right)^2]$$

   - Using some variant of SGD this **algorithm converges to least squares solution**

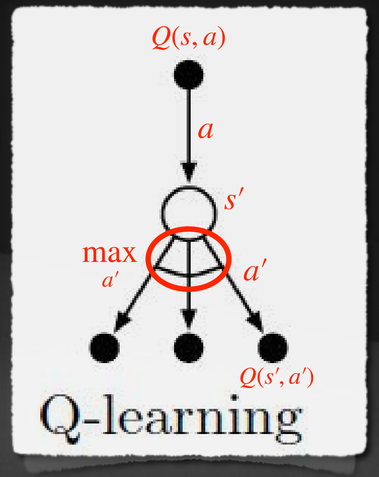
$$w^{\pi} = \arg\min_{w} LS(w)$$



DQN architecture

Positive Overestimation Bias

Maximisation bias



1. Q-Learning was mathematically proven to **overestimate the optimal action values** under mild assumptions by **Thrun and Shwartz**

2. The core idea is that there is some source of **random approximation error**
   - That can be either **positive** or **negative**

3. A **maximum** over estimated values is used implicitly as an estimate of the **maximum** value
   - Which can lead to a **significant positive bias**

1. Bellman **Optimality** Equation

$$Q(s,a) \leftarrow \mathbb{E}\left[R_s^a + \gamma \max_{a'} Q(s',a') \,|\, s,a\right]$$

2. Bellman **Optimality** Equation for **Q-Learning**

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R_s^a + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

3. Bellman **Optimality** Equation for **loss** calculation w . r . t network parameters [w_i] in **DQN**

$$\mathscr{L}(w_i) = \mathbb{E}_{s,a,r,s' \sim D_i}[\left(r + \gamma \max_{a'} \hat{q}(s',a',\overline{w_i}) - \hat{q}(s,a,w_i)\right)^2]$$

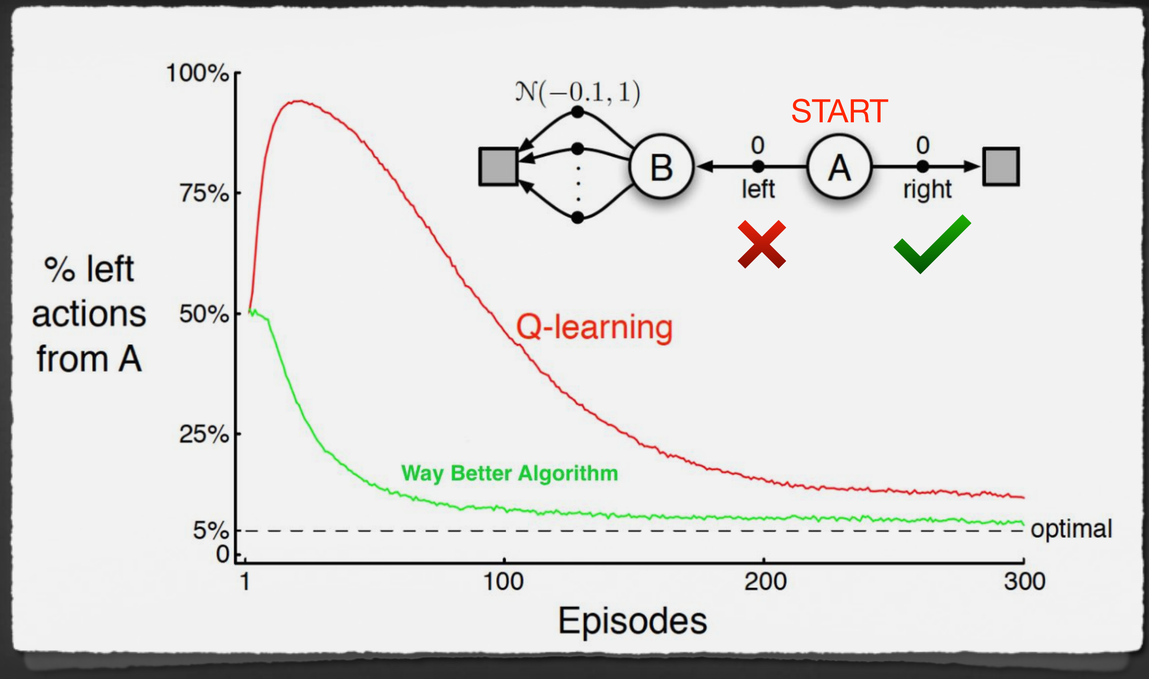
Positive Overestimation Bias

Maximisation bias example

1. Consider a **TD based** control algorithm, namely **Q-Learning**
2. And a small MDP that has two **non-terminal** states [A] and [B] and the episode **starts** in state [A]
3. With two possible actions [left] and [right] from [A]
   - Right actions transition immediately to terminal state with a **reward and return of 0**
   - Left action transitions to state [B] also with a reward of 0
   - **However from state** [B] there are many possible actions all of which cause immediate transition to terminal state
   - With a reward drawn from a **normal distribution** with **mean** [-0.1] and **variance** [1.0]

$$\mathcal{N}(-0.1,\ 1.0)$$

4. Thus the **expected return for any trajectory** starting with [left] is [-0.1]
5. therefore taking action [left] is **always a mistake**



Positive Overestimation Bias, Sutton & Barto

Super positive people [super optimists] example

Let’s consider a very optimistic person, let’s call her DQN.

She’s experienced many things in life from toughest defeat to 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 quite rewarding the value of [$Q(s, a)$] should be 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, DQN doesn't play the slot machines every time she goes to the casino. 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 machine area is under maintenance and not accessible (the environment transitions her somewhere else).

Additionally, most of the time when DQN plays the slot machines, **she doesn’t get the jackpot** (the environment is stochastic).

Mitigating the Overestimation Bias

Double Networks

Double DQN

[1] Adding Double Learning to DQN

1. Consider an update to our DQN parameters [w_i]

$$\nabla_{w_i} \mathscr{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D_i}[\left(r + \gamma \max_{a'} \hat{q}(s', a', \overline{w_i}) - \hat{q}(s, a, w_i)\right) \nabla_{w_i} \hat{q}(s, a, w_i]$$

Which is equivalent to

$$\triangle w_i = \mathbb{E}_{s,a,r,s' \sim D_i}[\left(r + \gamma \max_{a'} \hat{q}(s', a', \overline{w_i}) - \hat{q}(s, a, w_i)\right) \nabla_{w_i} \hat{q}(s, a, w_i]$$

2. Given the **max** and **argmax** correlation

$$\max_{\sigma} F(\mu, \sigma) = F\left(\mu, \arg\max_{\sigma} F(\mu, \sigma)\right)$$

3. Let's **unwrap** the **max** operation in our update

$$\max_{a'} \hat{q}(s', a', \overline{w_i}) = \hat{q}\left(s', \arg\max_{a'} \hat{q}(s', a', \overline{w_i}), \overline{w_i}\right)$$

Double DQN

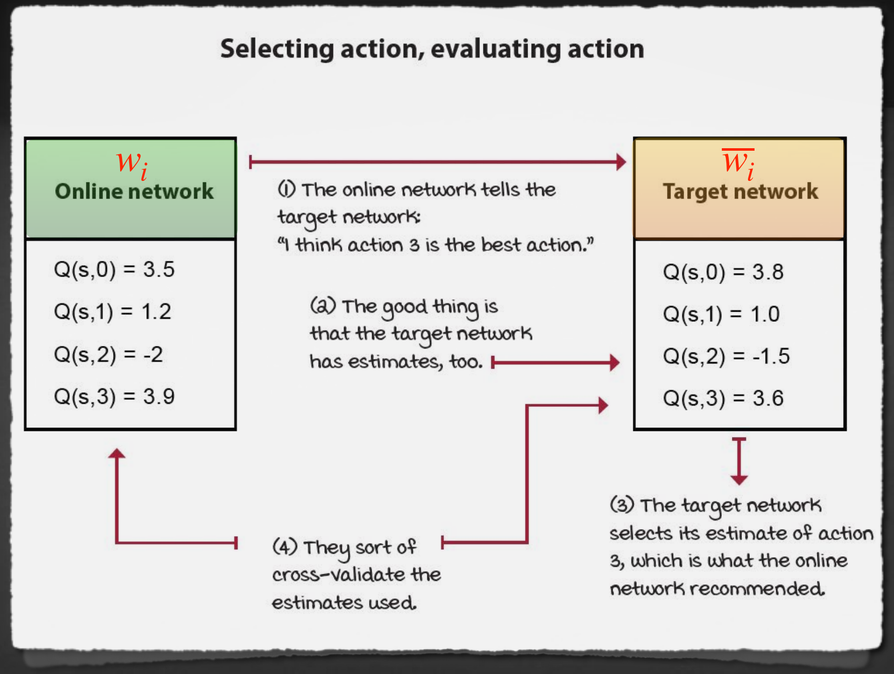
[2] Adding Double Learning to DQN

1. Let's rewrite the DQN update equation to use **argmax**

$$\triangle w_i = \mathbb{E}_{s,a,r,s' \sim D_i}\left[\left(r + \gamma \hat{q}\left(s', \arg\max_{a'} \hat{q}(s', a', \overline{w_i}), \overline{w_i}\right) - \hat{q}(s, a, w_i)\right) \nabla_{w_i} \hat{q}(s, a, w_i)\right]$$

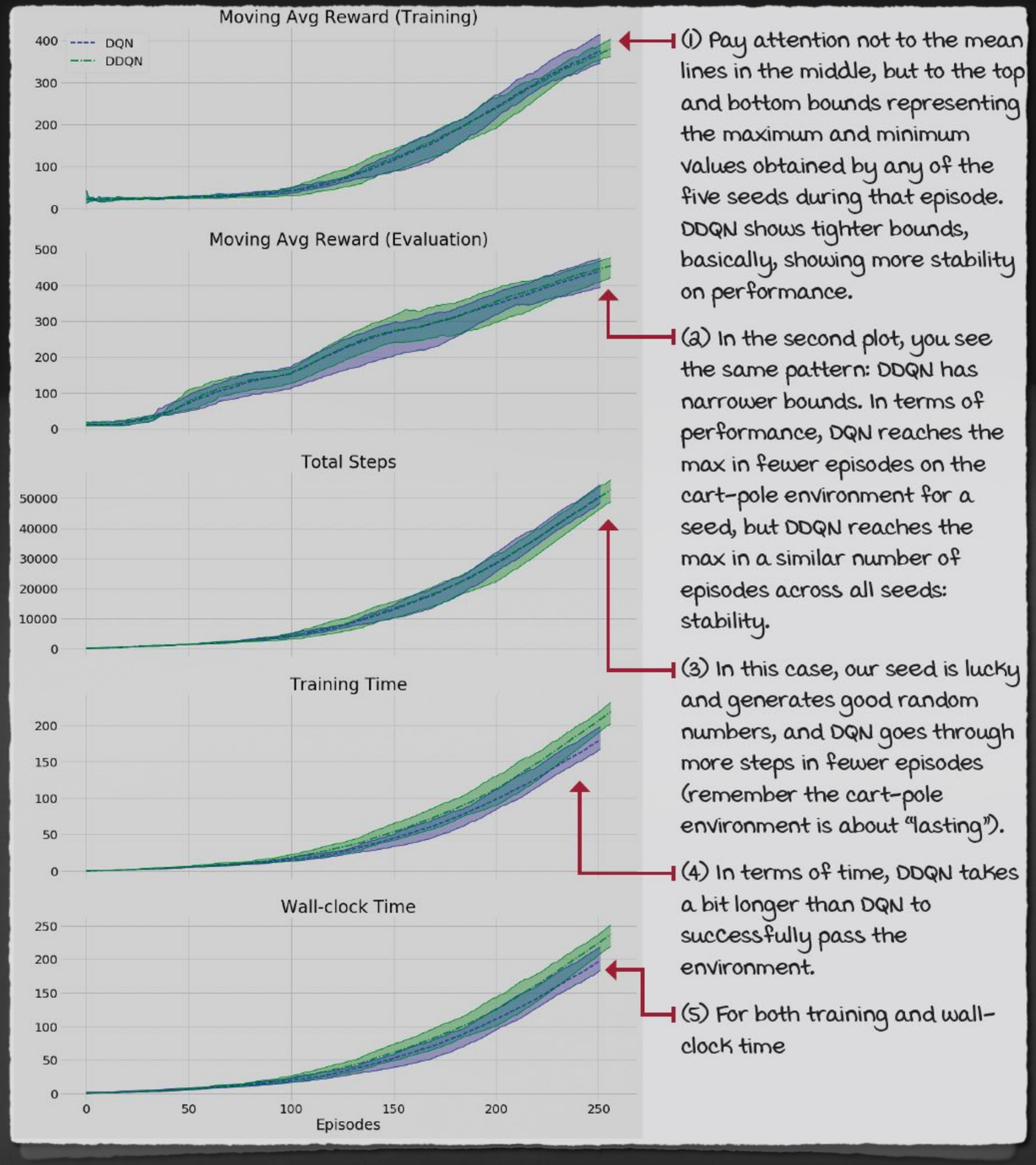
2. Notice that now we are **asking two question** the t**arget network**
   - **"Which action is the highest-valued action in state [s']"**
   - **"What is the value of this action"**
3. Asking **the same Q-function** those two questions is the **source of maximisation bias**
4. To solve this problem let the **online network** answer the **first** question
5. And the **target network** to answer the **second** question
6. Therefore the **final update** can be re-written as

$$\triangle w_i = \mathbb{E}_{s,a,r,s' \sim D_i}\left[\left(r + \gamma \hat{q}\left(s', \arg\max_{a'} \hat{q}(s', a', w_i), \overline{w_i}\right) - \hat{q}(s, a, w_i)\right) \nabla_{w_i} \hat{q}(s, a, w_i)\right]$$



Solving Maximisation Bias

- DQN vs DDQN


Moving Avg Reward (Training)
DQN
DDQN
400
300
200
100
0
Moving Avg Reward (Evaluation)
500
400
300
200
100
0
Total Steps
50000
40000
30000
20000
10000
0
Training Time
200
150
100
50
0
Wall-clock Time
250
200
150
100
50
0
0
50
100
150
200
250
Episodes
(1) Pay attention not to the mean lines in the middle, but to the top and bottom bounds representing the maximum and minimum values obtained by any of the five seeds during that episode. DDQN shows tighter bounds, basically, showing more stability on performance.
(2) In the second plot, you see the same pattern: DDQN has narrower bounds. In terms of performance, DQN reaches the max in fewer episodes on the cart-pole environment for a seed, but DDQN reaches the max in a similar number of episodes across all seeds: stability.
(3) In this case, our seed is lucky and generates good random numbers, and DQN goes through more steps in fewer episodes (remember the cart-pole environment is about "lasting").
(4) In terms of time, DDQN takes a bit longer than DQN to successfully pass the environment.
(5) For both training and wall-clock time

Replay Buffer, Target Networks, Double Learning …

We’ve been making reinforcement learning more and more like supervised learning

Many say that this is not a good direction
And instead should we focus more on reinforcement learning fundamentals

A Reinforcement Learning Aware Network

Dueling Networks

Dueling DDQN

[1] Adding Dueling Network to DDQN

1. The duelling network is an improvement that **applies only to the network architecture** and not to the algorithm
   - No additional networks or changes in objective function
2. Currently DDQN is learning Q-value for each action **separately with a bit of generalisation** happening because **networks** are internally connected
   - Therefore information is shared between nodes of a network
   - But when learning about Q-value for action [a_1] and state [s]

$$q(s, a_1)$$

   - We're ignoring the fact that we could use the same information to learn something about different actions under same state [s]

$$\begin{bmatrix} q(s, a_2) \\ q(s, a_3) \\ \vdots \end{bmatrix}$$

3. To mitigate this issue we can take advantage of the **relationship** between value functions

$$q_\pi(s, a) = v_\pi(s) + a_\pi(s, a)$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

(1) Recall the action-value function of a policy is its expectation of returns given you take action a in state s and continue following that policy.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

(2) The state-value function of state s for a policy is the expectation of returns from that state, assuming you continue following that policy.

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s)$$

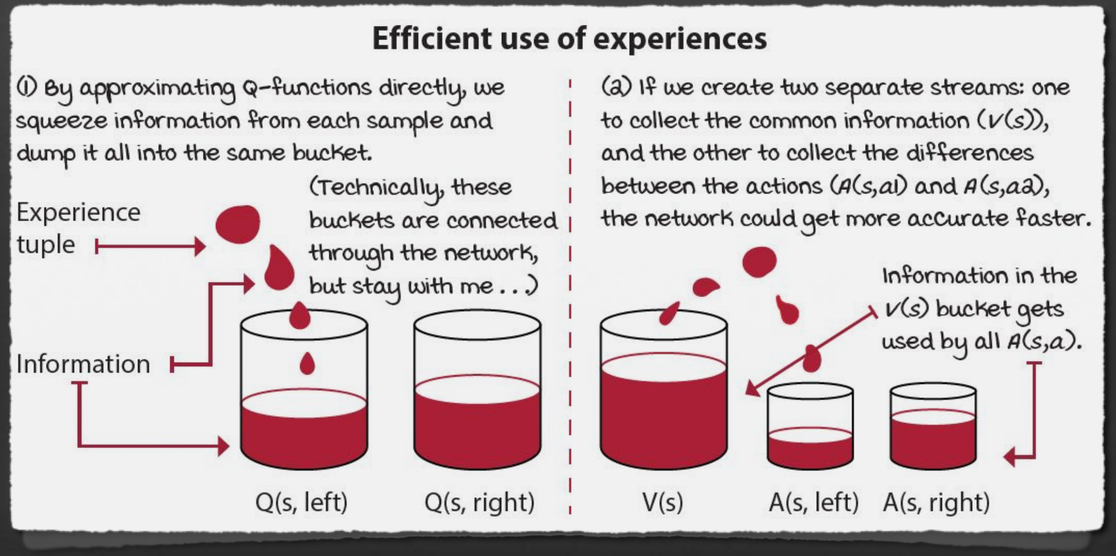
(3) The action-advantage function tells us the difference between taking action a in state s and choosing the policy's default action.

$$\mathbb{E}_{a \sim \pi(s)}\left[a_\pi(s, a)\right] = 0$$

(4) Infinitely sampling the policy for the state-action pair yields 0. Why? Because there's no advantage in taking the default action.

$$q_\pi(s, a) = v_\pi(s) + a_\pi(s, a)$$

(5) Finally, we wrap up with this rewrite of the advantage equation above. We'll use it shortly.



The bottom line is that the values of actions depend on the values of states

Taking the worst action in a god state could be better than taking the best action in bad state

Dueling DDQN

[2] Adding Dueling Network to DDQN

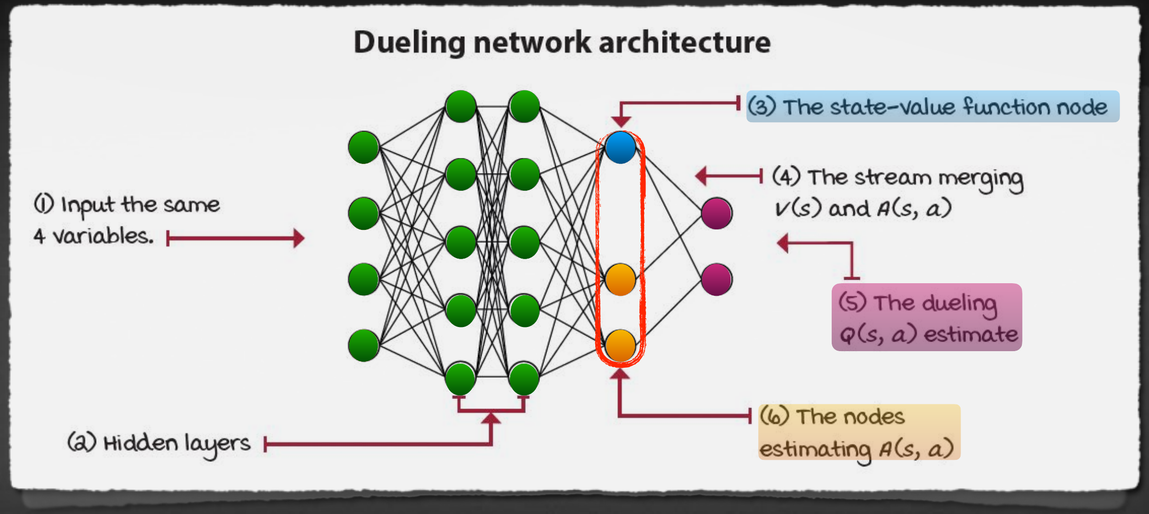
1. The dueling network consists of creating **<u>two separate estimators</u>**
   - State value function [$V(s)$]
   - Action advantage function [$A(s,a)$]
2. In order to **<u>reconstruct</u>** the **<u>action value function</u>** we have to **<u>aggregate</u>** these two estimators
   - Consider a Q-function parametrised by [θ, α, β]

     $\theta - network\ shared\ parameters$

     $\alpha - weights\ of\ action\ advantage\ function$

     $\beta - weights\ of\ state\ value\ function$
   - Therefore the reconstructed Q-function can be written as follows

     $$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + A(s,a,\theta,\alpha)$$
3. That way we end up with the **<u>Dueling</u>** Double DQN



Dueling DDQN

[3] Adding Dueling Network to DDQN

(1) The Q-function is parameterized by theta, alpha, and beta. Theta represents the weights of the shared layers, alpha the weights of the action-advantage function stream, and beta the weights of the state-value function stream.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

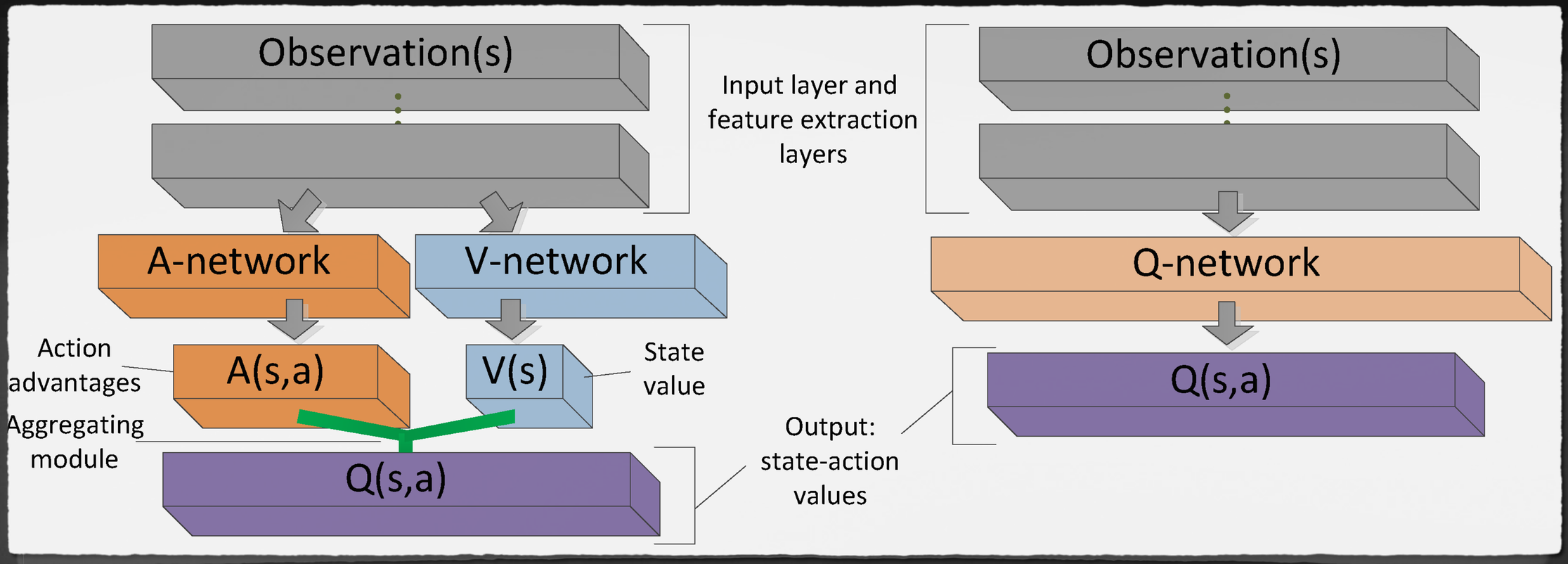
$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right)$$

(2) But because we cannot uniquely recover the Q from V and A, we use the above equation in practice. This removes one degree of freedom from the Q-function. The action-advantage and state-value functions lose their true meaning by doing this. But in practice, they're off-centered by a constant and are now more stable when optimizing.

Practical Reconstruction of Q-function

Dueling DDQN

[3] Adding Dueling Network to DDQN



Duelling DDQN vs DDQN Architectures

Advanced Experience Replay Strategies

Random sampling from replay memory is inefficient

Boston Dynamics, Atlas

Prioritised Experience Replay

PER

1. The goal is to allocate resources for experience tuples that **have the most significant potential for learning**

2. Let's start with calculating absolute TD error $|\delta_i|$ as a measure of **"surprise"**

   - **How wrong was our estimate at given time step**

3. Then we have to **balance** the positive, negative and neutral experience

   - [1] Sort samples by their absolute TD error in descending order and create a rank for each

     $$rank(i)$$

   - [2] Rank based prioritisation, to ensure **high TD errors** won't be sampled more often than those with low magnitudes

     $$p(i) = \frac{1}{rank(i)}$$

   - [3] Priorities to probabilities, where [α] controls the blend between uniform and prioritised experience sampling

     $$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

(i) I'm calling it "dueling DDQN" target to be specific that we're using a target network, and a dueling architecture. However, this could be more simply called TD target.

$$|\delta_i| = |\underbrace{r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta_i, \alpha_i, \beta_i); \theta^-, \alpha^-, \beta^-)}_{\text{Dueling DDQN target}} - Q(s, a; \theta_i, \alpha_i, \beta_i)|$$

Dueling DDQN error

Absolute Dueling DDQN error

Prioritisation Bias

The return of importance sampling

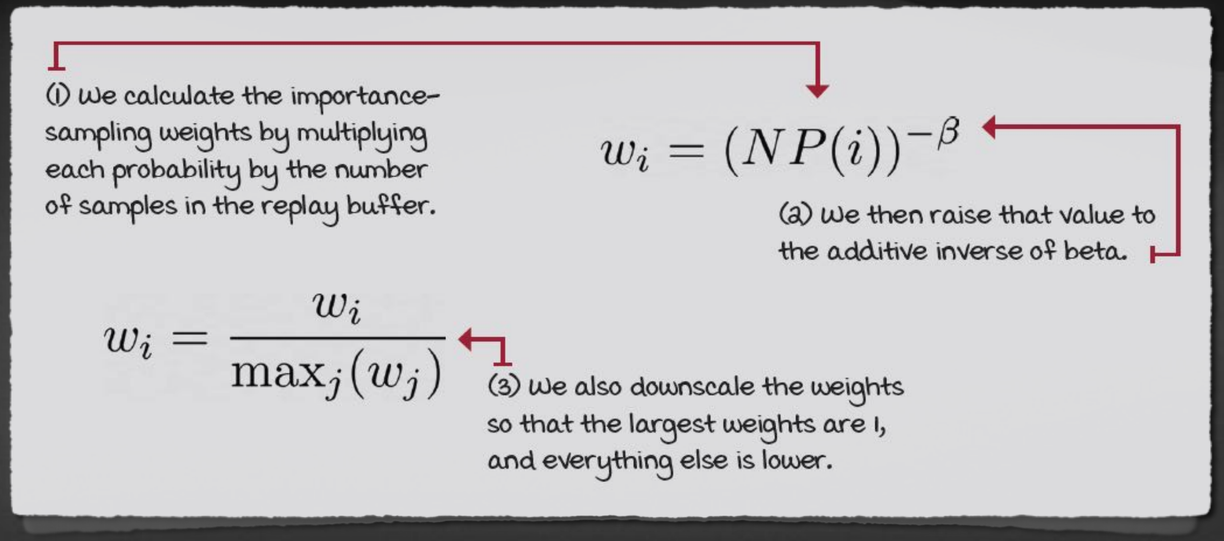
1. Using one distribution for **estimating** another one introduces bias in the estimates.

2. Because we're sampling based on these **probabilities**, **priorities**, and **TD errors**, we need to account for that.

   - The **distribution** of the updates must be from **the same distribution as its expectation**
   - When we update the action-value function of state [*s*] and an action [*a*], we must be cognisant that we **always update with targets**
   - Targets are **samples of expectations**
   - That means the reward and state at the next step could be **stochastic**
   - there could be **many possible different** rewards and states when taking action [*a*] in a state [*s*]

3. Otherwise if we update a single sample **more often than it appears in that expectation** we'd create a bias toward this value

4. The way to mitigate prioritisation bias is to a technique called **weighted importance sampling**



Dueling DDQN + PER

Network update

(1) I don't want to keep bloating this equation, so I'm only using theta to represent all parameters, the shared, for the action-advantage function, alpha, and for the state-value function, beta.

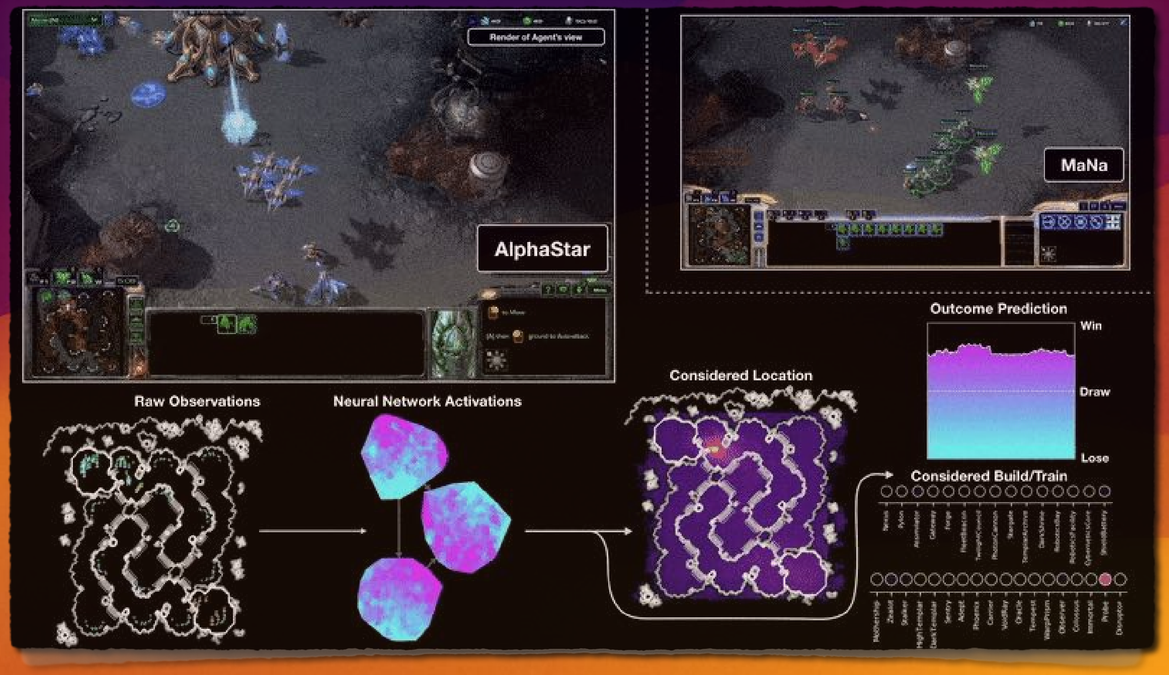
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(w,s,a,r,s')\sim\mathcal{P}(\mathcal{D})}\left[w\left(r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta_i); \theta^-) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

(2) Notice how I changed the $\mathcal{U}$ for a $\mathcal{P}$, because we're doing a prioritized sampling, and not uniformly at random.

(3) Finally, notice how we're using the normalized importance-sampling weights to modify the magnitude of the TD error.

Distributional Reinforcement Learning

Return is just a single number



DeepMind, AlphaStar

Categorical DQN

C51

1. C51 is a Q-learning algorithm based on DQN. Like DQN, it can be used on any environment with a **discrete action space**

2. The main difference between C51 and DQN is that rather than simply **predicting the Q-value** for each state-action pair

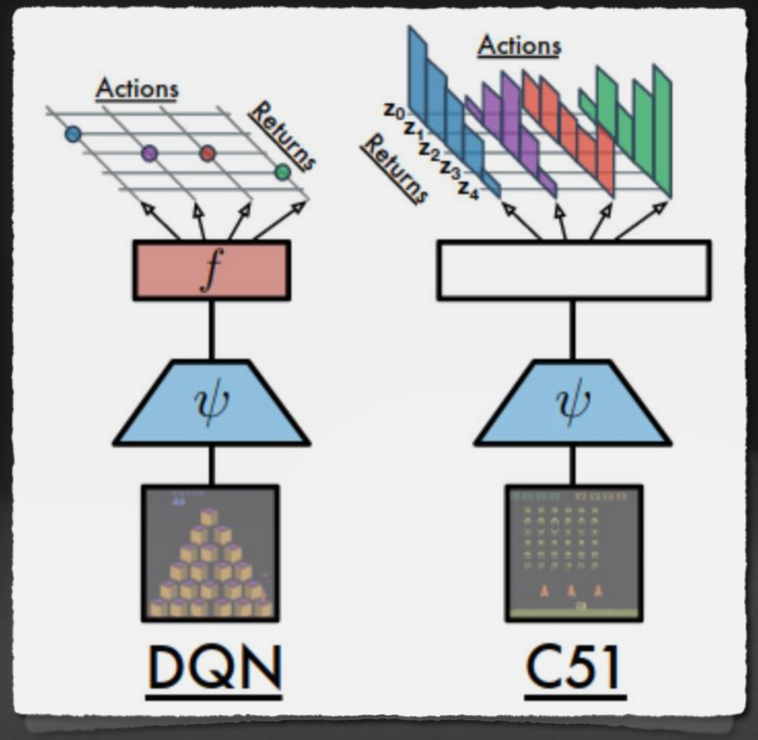
$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a]$$

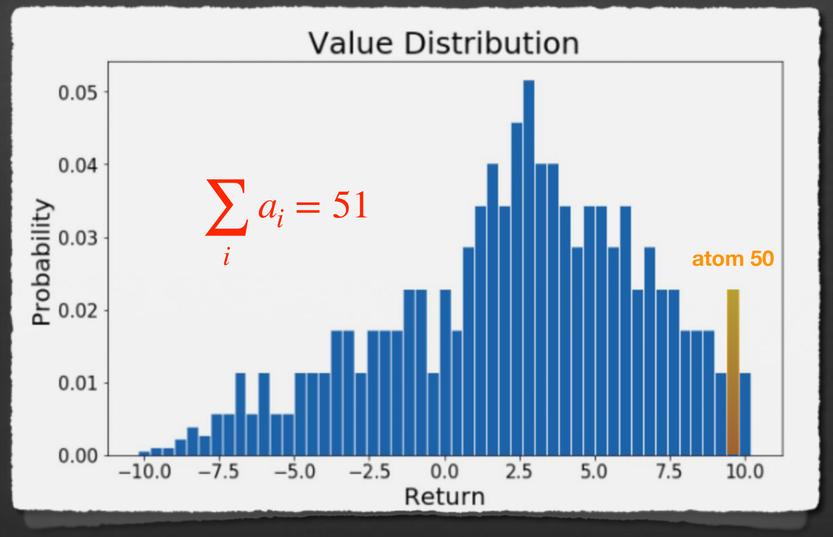
3. C51 predicts a **histogram model** for the probability distribution of the Q-value

   - By applying the **distributional Bellman operator**

   - Which is the contraction in a **maximal form of the Wasserstein metric** between probability distributions

4. C51 has **51 fixed locations** [atoms] for the **value** distribution and we **learn the probabilities** of the fixed location



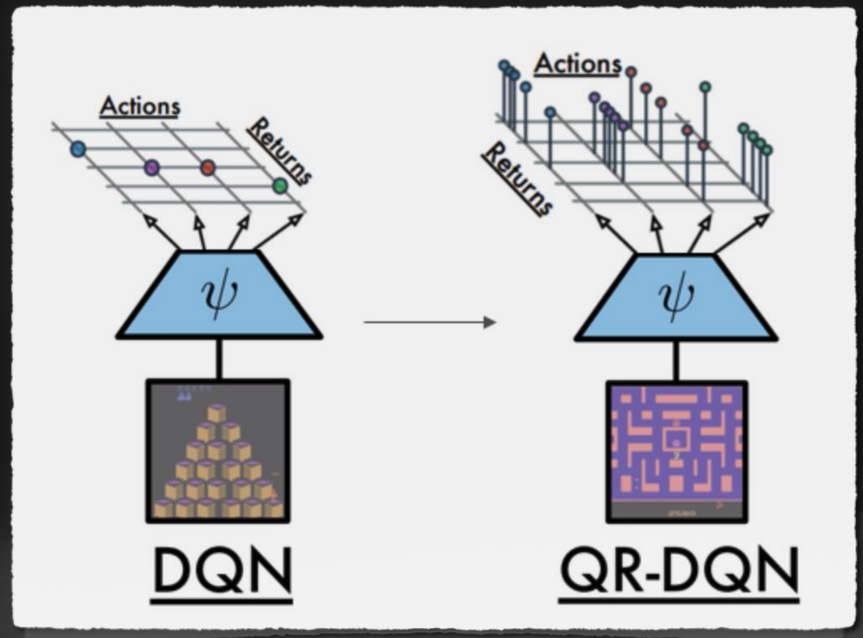


Value distribution in C51

Quantile Regression DQN

QR-DQN

1. QR-DQN differs from C51 that it learns the **quantile values directly**
2. By using **quantile approximation** mechanism
    1. QR-DQN computes the **return quantiles** on fixed, uniform quantile fractions using quantile regression
    2. And minimises the **quantile Huber loss** between the Bellman updated distribution and current return distribution
3. C51 has 51 **fixed locations** for the **value** distribution and we learn the probabilities **of the fixed location**.
4. QR-DQN aims to solve the restriction of C51 by considering a **fixed probability support** instead of a fixed value support.
5. Specifically, QR-DQN prescribes a discrete support taken on by the **cumulative probability function** [c.d.f.]
6. And estimates **quantiles of the target distributio**n via some parametric model



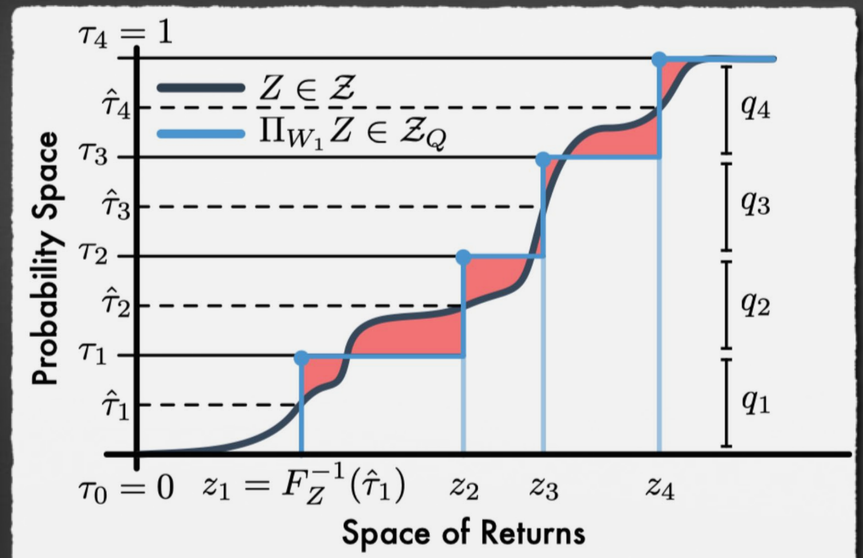


Figure 2: 1-Wasserstein minimizing projection onto $N = 4$ uniformly weighted Diracs. Shaded regions sum to form the 1-Wasserstein error.

Rainbow DQN

Kociołek Panoramiksa

1. It uses Double Q-Learning to tackle overestimation bias.
2. It uses Prioritised Experience Replay to prioritise important transitions.
3. It uses Dueling Networks
4. It uses Multi-Step Learning
5. It uses Distributional Reinforcement Learning instead of expected return
6. It uses Noisy Linear Layer for exploration

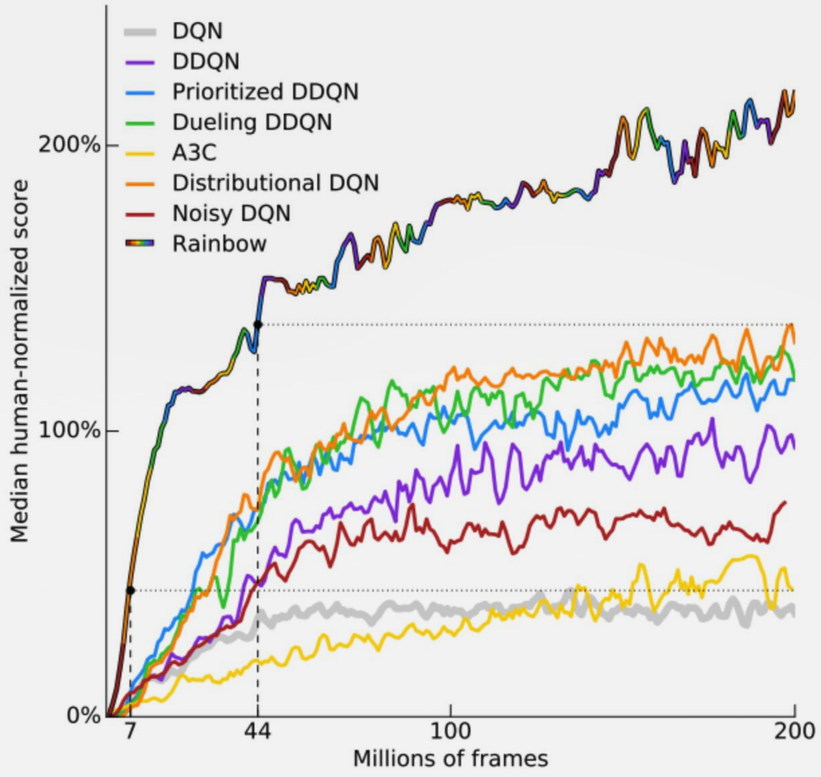
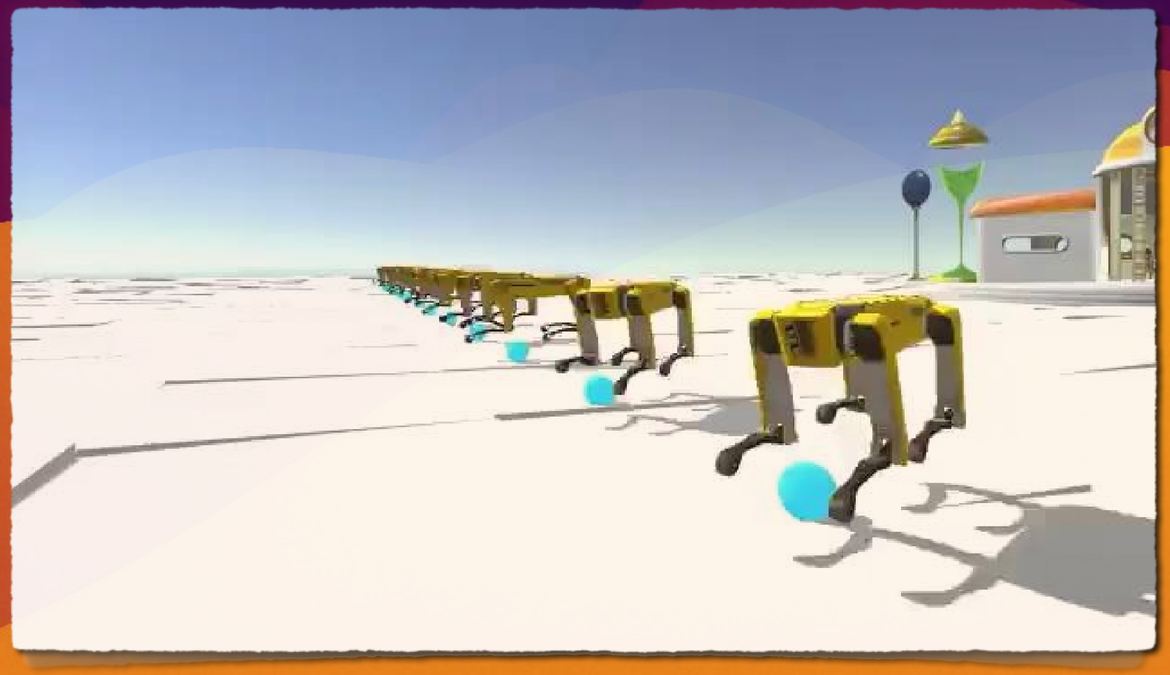


Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

Improvements to Actor Critic

The more actors the more data for critic to learn



Boston Dynamics, Quadrupedal Robot Sim Training

Q-Value Actor Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s,a) = \phi(s,a)^\top w$

  Critic Updates w by linear TD(0)

  Actor Updates θ by policy gradient

function QAC

Initialise s, θ

Sample $a \sim \pi_\theta$

for each step **do**

Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$

Sample action $a' \sim \pi_\theta(s', a')$

$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$

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

$w \leftarrow w + \beta \delta \phi(s, a)$

$a \leftarrow a', s \leftarrow s'$

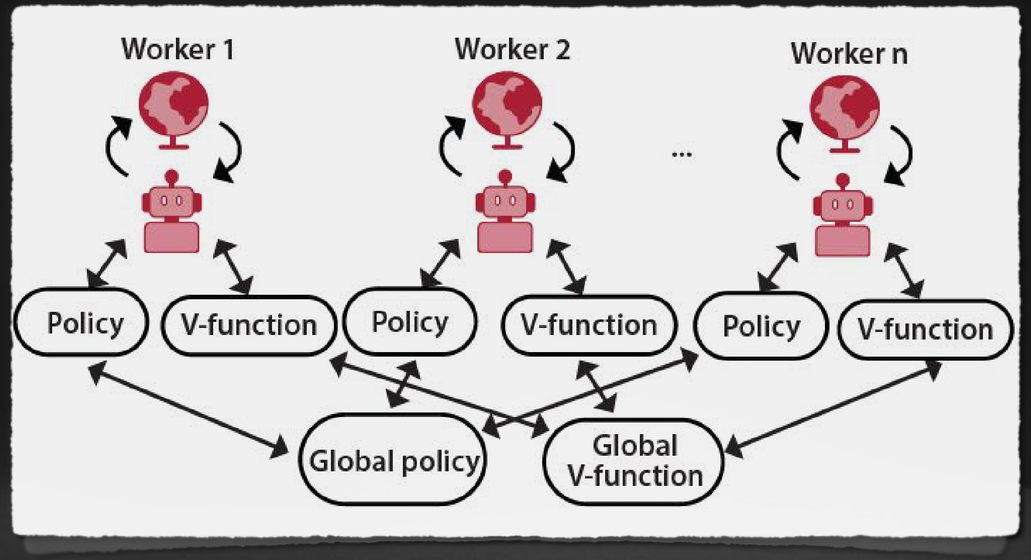
end for

end function

Asynchronous Advantage Actor-Critic

A3C

1. The idea is to update policy in **parallel** by use of multiple **agent-environment** entities
    - In **on-policy** manner
    - Without **replay buffer**
    - Each worker uses **n-step TD**
2. Having **multiple workers** generating experience on **multiple instances** of the environment in parallel
    - **decorrelates** the data used for training and reduces the variance of the algorithm
3. Each worker creates it's **own instance of the environment** and the policy and value function along the network parameters [θ, w]
    1. After an experience batch is collected, each worker updates the **global models** [policy and value function]
    2. Asynchronously **without the coordination** with other workers [hogwild!]
    3. Then each worker reloads their copy of the models and keeps at it
4. The most important resource is **CPU** due to multiple environment instances



Asynchronous Model Updates

(1) Before we were using full returns for our advantage estimates.

$$A(S_t, A_t; \phi) = G_t - V(S_t; \phi)$$

(2) Now, we're using n-step returns with bootstrapping.

$$A(S_t, A_t; \phi) = R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+n} + \gamma^{n+1} V(S_{t+n+1}; \phi) - V(S_t; \phi)$$

(3) We now use this n-step advantage estimate for updating the action probabilities.

$$L_\pi(\theta) = -\frac{1}{N} \sum_{n=0}^{N} \left[A(S_t, A_t; \phi) \log \pi(A_t|S_t; \theta) + \beta H(\pi(S_t; \theta)) \right]$$

(4) We also use the n-step return to improve the value function estimate. Notice the bootstrapping here. This is what makes the algorithm an actor-critic method.

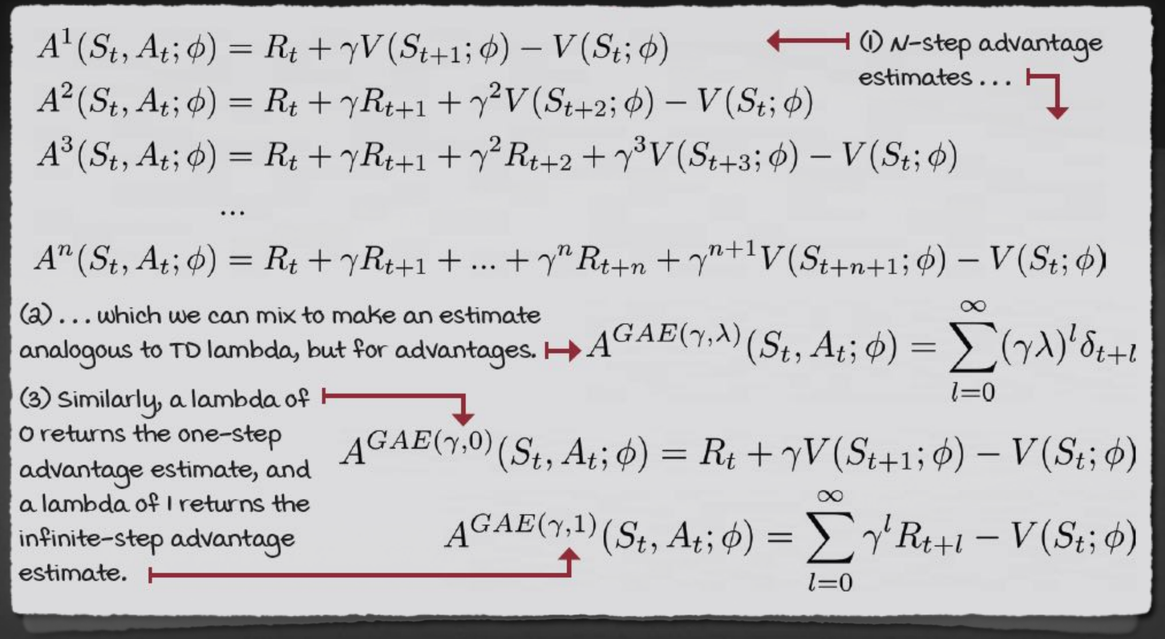
$$L_v(\phi) = \frac{1}{N} \sum_{n=0}^{N} \left[\left(R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+n} + \gamma^{n+1} V(S_{t+n+1}; \phi) - V(S_t; \phi) \right)^2 \right]$$

A3C Loss Computation and Critic Update

Generalised Advantage Estimation

GAE

1. Recall that raw policy gradients [REINFORCE] while unbiased, **have *high variance*.**
   - GAE is a technique to dramatically **reduce variance**
   - but this unfortunately comes at the cost of **introducing bias**
   - so one needs to be **careful** before applying tricks like this in practice
2. GAE uses an exponentially weighted combination of *n*-step **action-advantage function targets**
3. The same way the λ-target is an exponentially weighted combination of *n*-step **state-value function** targets
4. Advantage Estimator is a robust **estimate** of the action-advantage function [a_π]
   1. This way we **reduce the approximation** overhead
   2. By estimating the action-advantage function using only **state-value function**

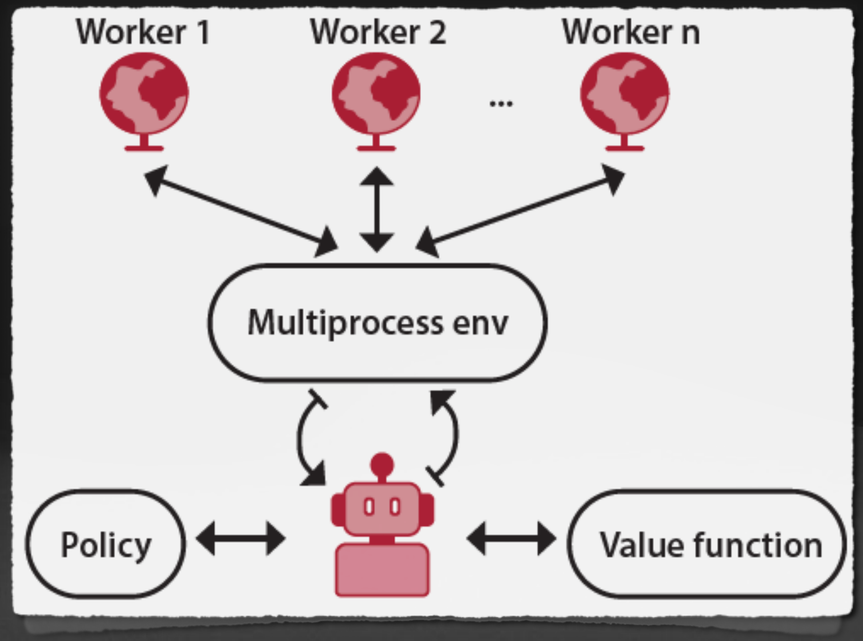


Advantage Estimator

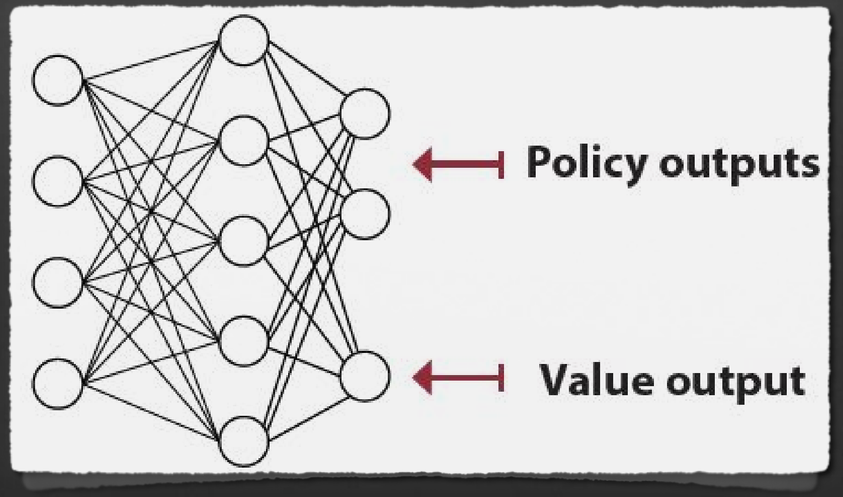
Advantage Actor-Critic

A2C

1. A2C uses **advantage estimator** [A_π] as a **critic**
2. Advantage Actor-Critic is a **synchronous** version of A3C
   - It uses **single neural network** [shared weights] for both policy and value function
   - In **on-policy** manner
   - Without **replay buffer**
3. Instead of having multiple workers with their own instance of environment
   - We have a **single agent** driving the interaction with the environment
   - But in this case the environment is a **multiprocess class** that gathers samples from multiple environments at once
4. The most important resource is **GPU** due to need to process multiple batches of data by a **single neural network** sat every timestep



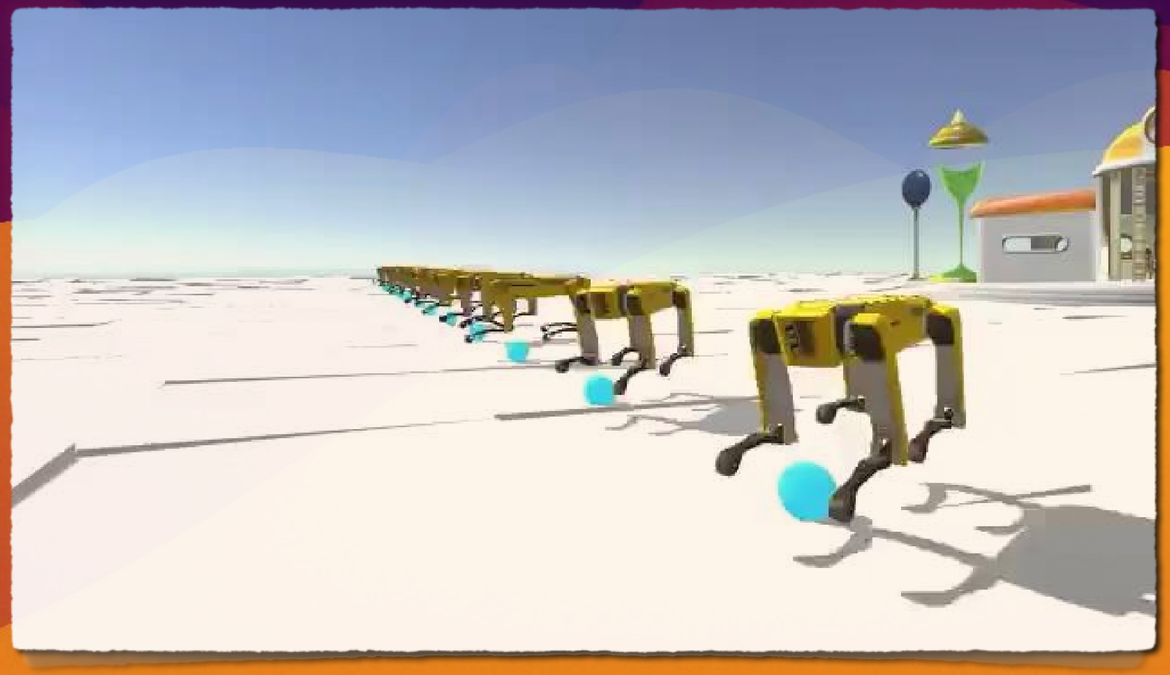
Synchronous Model Updates



Network Weight Sharing

Advanced Actor-Critic Methods

Towards Artificial General Intelligence



Boston Dynamics, Quadrupedal Robot Sim Training

Deep Deterministic Policy Gradients

DDPG

1. DDPG is a DQN for **continuous action spaces**
    - It is **off-policy**
    - It uses **replay buffer**
    - It uses **target networks**
2. **However** DDPG also trains a policy that **approximates** the optimal action
3. The main difference between **DQN** and DDPG is that while DQN uses the target Q-function for getting the greedy action using an **argmax**
4. **DDPG** uses a **target deterministic policy function** [μ]
   that is trained to **approximate** that greedy action.
    - Instead of using the argmax of the Q-function of the next state to get the greedy action as we do in DQN
    - In DDPG, we directly approximate the best action in the next state using a **policy function** [μ]
    - Then, in both, we use that action with the Q-function to get the max value
5. In order to train a **policy function** [μ]
    1. The network **must be differentiable** $w.r.t$ **action**
    2. Therefore the **action** must be **continuous**
    3. And the **objective function** is computed $w.r.t$ online network instead of target network

(1) Recall this function. This is the DQN loss function for the Q-function. It's straightforward.

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

(2) We sample a mini-batch from the buffer D, uniformly at random.

(3) Then, calculate the TD target using the reward and the discounted maximum value of the next state, according to the target network.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \theta^-); \theta^-) - Q(s, a; \theta_i)\right)^2\right]$$

(4) Also, recall this rewrite of the same exact equation. We change the max for the argmax.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma Q(s', \mu(s'; \phi^-); \theta^-) - Q(s, a; \theta_i)\right)^2\right]$$

(5) In DDPG, we also sample the mini-batch as in DQN.

(6) But, instead of the argmax according to Q, we learn a policy, mu.

(7) Mu learns the deterministic greedy action in the state in question. Also, notice phi is also a target network (-).

DDPG Objective Function

(1) Learning the policy is straightforward as well; we maximize the expected value of the Q-function using the state, and the policy's selected action for that state.

$$J_i(\phi_i) = \mathbb{E}_{s\sim\mathcal{U}(\mathcal{D})}\left[Q(s, \mu(s; \phi); \theta)\right]$$

(2) For this we use the sampled states from the replay buffer.

(3) Query the policy for the best action in those states.

(4) And then query the Q-function for the Q-value.

Approximation of Deterministic Policy

Policy network is trained to approximate the greedy action selection

Are we done with exploration?

To ensure continuous exploration once approximating deterministic policies

We inject Gaussian noise into the actions selected by the policy

Twin Delayed Deep Deterministic Policy Gradients

TD3

1. Twin Delayed DDPG is a **improvement** over DDPG
2. That introduces **three main changes** to the DDPG
   - It adds a double learning with a **unique** **TWIN** network architecture
   - It adds noise **not only** to to the action passed to the environment, but also to **target actions**

     Making the policy more **robust** to approximation error
   - It **delays the updates** to the policy network, target network and the TWIN target network

     So that the TWIN **online** network updates **more** frequently, making online Q-function updates at a higher rate than the rest

     Delaying these networks is beneficial because often, the online Q-function changes **shape abruptly early** on in the training process
3. TWIN network learned two Q-functions instead of one
   - Both Q-functions are treated as **separate** streams
   - The only thing that is **shared** is the **optimiser**
   - The **smaller** of Q-functions is taken to form the targets for **Bellman error loss functions**

(1) The twin network loss is the sum of MSEs of each of the steams.

$$J_i(\theta_i^a) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(\mathcal{TWIN}^{target} - Q(s,a;\theta_i^a)\right)^2\right]$$

$$J_i(\theta_i^b) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(\mathcal{TWIN}^{target} - Q(s,a;\theta_i^b)\right)^2\right]$$

(2) We calculate the target using the minimum between the two streams. This isn't a complete TD3 target. We'll add to it in a couple of pages.

$$\mathcal{TWIN}^{target} = r + \gamma \min_n Q(s', \mu(s';\phi^-);\theta^{n,-})$$

(3) But, notice how we use the target networks for both the policy and value networks.

TWIN Target [1]

(1) Let's consider a clamp function, which basically "clamps" or "clips" a value x between a low l, and a high h.

$$\text{clamp}(x,l,h) = \max(\min(x,h),l)$$

$$a'^{,smooth} = \text{clamp}(\mu(s';\phi^-) + \text{clamp}(\epsilon, \epsilon\_l), \epsilon\_h), a\_l, a\_h))$$

(2) In TD3, we smooth the action by adding clipped Gaussian noise, ε. We first sample ε, and clamp it to be between a preset min and max for ε. We add that clipped Gaussian noise to the action, and then clamp the action to be between the min and max allowable according to the environment. Finally, we use that smoothed action.

$$\mathcal{TD3}^{target} = r + \gamma \min_n Q(s', a'^{,smooth};\theta^{n,-})$$

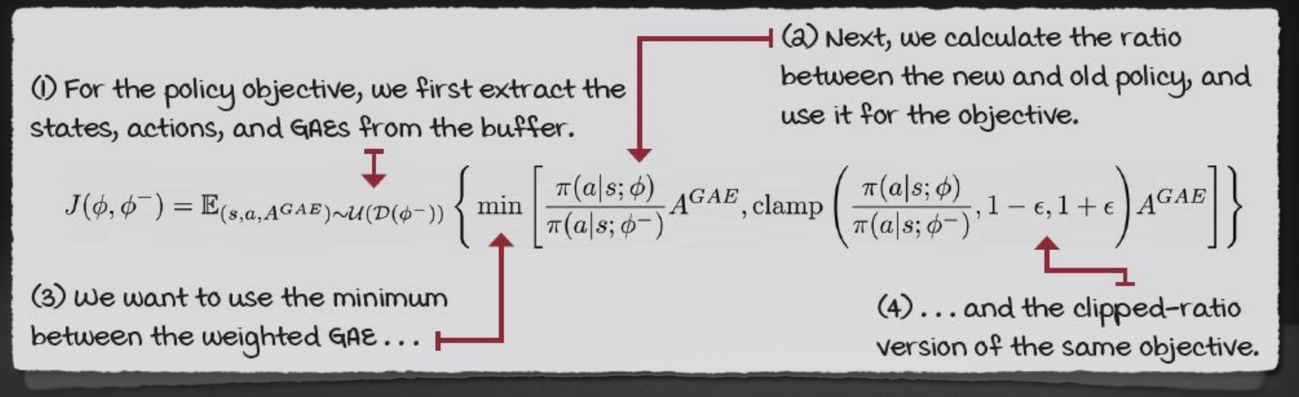
TWIN Target [2]

Proximal Policy Optimisation

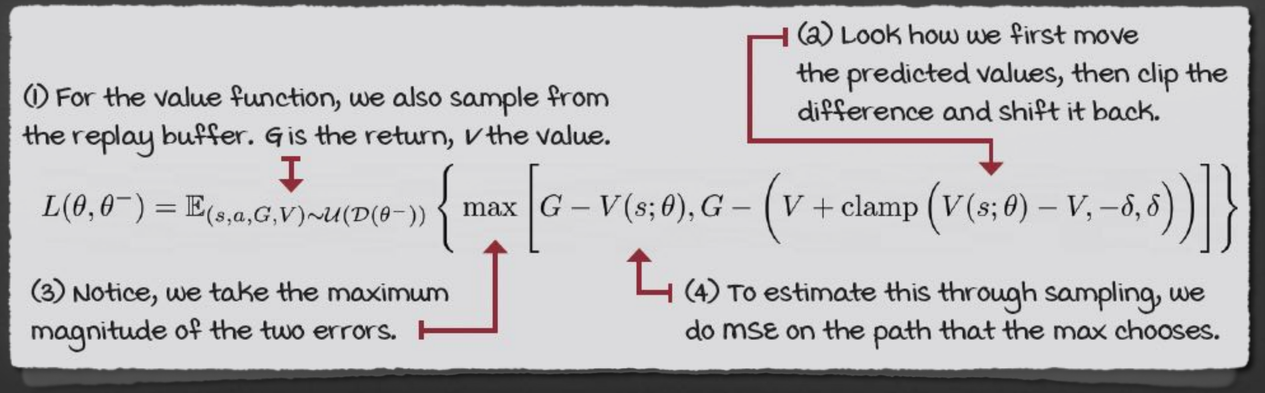
PPO

1. PPO introduces the mechanism to **restrict optimisation steps**
2. PPO is a algorithm with the same underlying structure as A2C
    - With the **critical** innovation of **surrogate objective function**
    - That allows an **on-policy algorithm** to perform **multiple gradient steps** on the **same** mini batch of experience

    Which makes PPO **more sample efficient** than other on-policy methods
3. Moreover PPO introduces the **clipped objective function**
    - Which **prevents** the policy from getting **too different** after an optimisation step
4. Clipping the **policy** updates
    - The main issue with the regular policy gradient is that **even a small change** in parameter space can lead to a **big difference in performance**
    - Intuitively, you can think of this clipped objective as a **coach preventing overreacting to outcomes**
5. Clipping the **value** updates
    - Intuitively, let the changes in parameter space change the Q-values only this much, **but not more**
    - As you can tell, this clipping technique **keeps the variance** of the things we care about smooth



Clipped Policy Objective

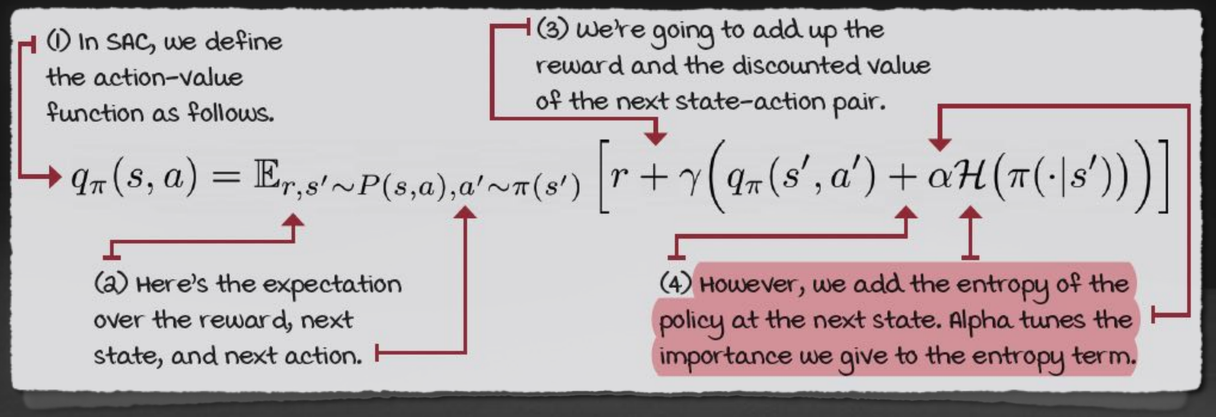


Clipped Value Objective

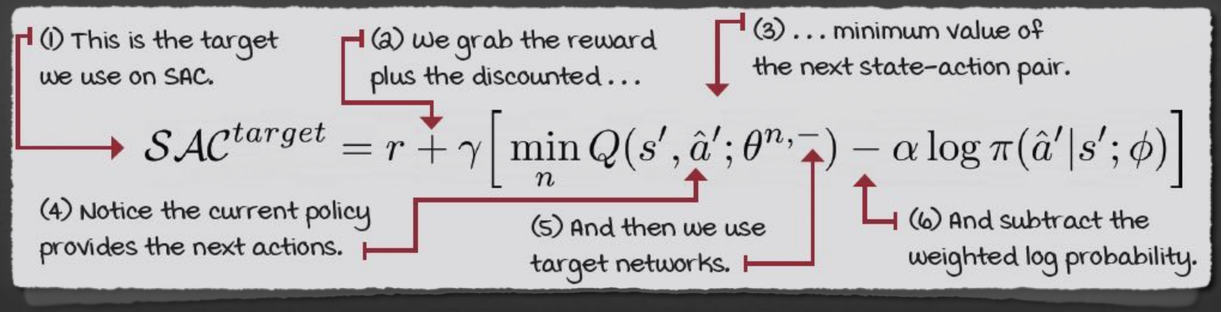
Soft Actor-Critic

[1] SAC

1. SAC **maximises both** the **expected return** and **entropy**
2. SAC is a similar algorithm to DDPG and TD3
3. But it trains **stochastic policy** instead of deterministic policy
4. The most crucial characteristic of SAC
   - Is that the **entropy** of the stochastic policy **becomes part of the value function** that the agent attempts to maximise
   - In practise SAC learns Q-value in a way similar to TD3
   - But the Q-functions are **optimised independently**, that is the optimiser is **not shared**
   - And the **entropy term** is added to the **target values**



Q-Value with Entropy



Q-Value Target

Hold on!

What is the purpose of entropy maximisation?

Entropy Maximisation

Intelligent Exploration

1. Entropy is a quantity which, roughly speaking, says how **random a random variable is**.
2. If a coin is weighted so that it almost always comes up heads, it has **low** entropy.
3. if it's evenly weighted and has a half chance of either outcome, it has **high** entropy.

In entropy-regularised reinforcement learning, the agent gets a **bonus reward** at each time step **proportional to the entropy** of the policy at that timestep

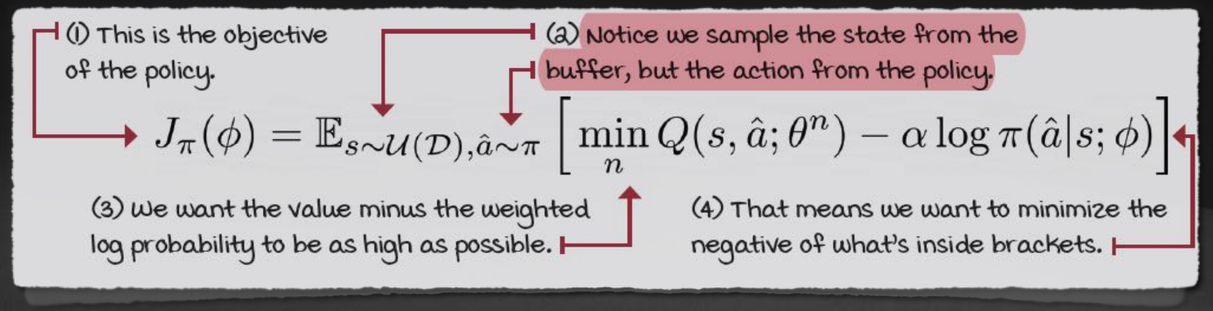
Soft Actor-Critic

[2] SAC

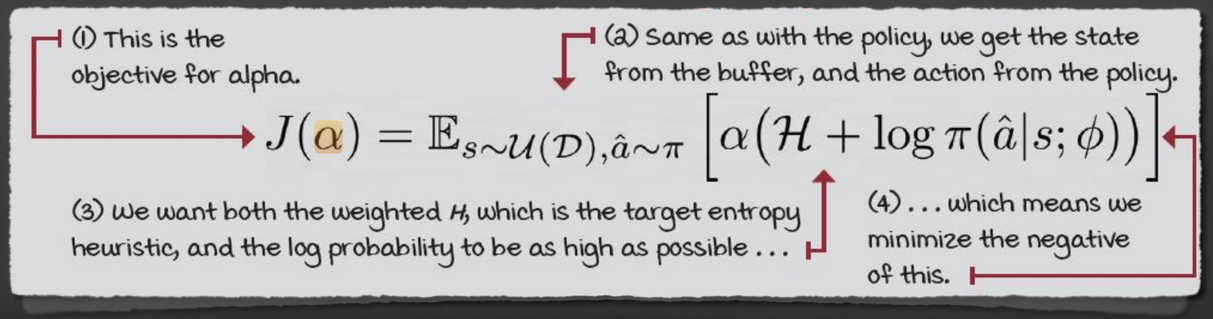
1. To learn stochastic policy instead of deterministic one
   - SAC uses squashed Gaussian policy that, in the forward pass, outputs **mean** and **standard deviation**
   - Then these can be used to **sample** actions from the distribution
   - SAC uses the **reparameterization** trick.

     This “trick” consists of moving the **stochasticity** out of the network and **into an input**

     This way, **the network is deterministic**, and we can train it without problems.
2. To tune the entropy coefficient
   - SAC employs **gradient-based optimisation** of entropy network parameters [α]
   - Toward a **heuristic expected entropy**
   - The recommended target entropy is based on the **shape of the action space**
   - More specifically, the **negative of the vector product** of the action shape.



Policy Objective



Tuning the Entropy Coefficient

Thank You

Appendix