Reinforcement Learning

Approximate Methods

Jakub Chojnacki

About the Author

Who Am I

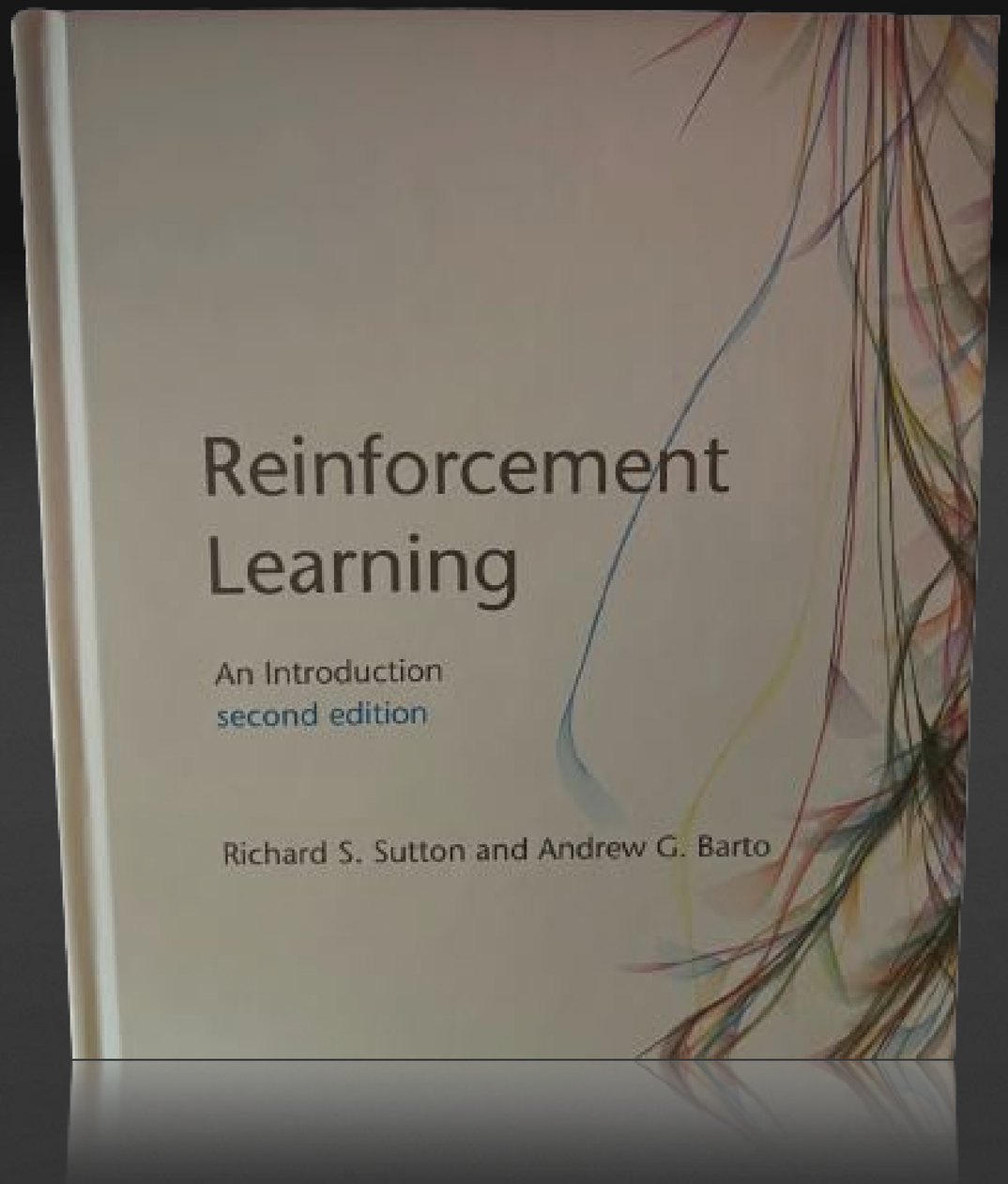
Admin
What & When

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 - { DQN, PPO, DDPG, TD3, SAC, *TQC, *QR-DQN }
    2. Tips and tricks
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. Deep Reinforcement Learning
2. Value Based Methods
3. Baselines Improvements
4. Policy Based Methods
5. Actor Critic Methods

Deep Reinforcement Learning

Solving real world problems

Boston Dynamics, Atlas

Deep Reinforcement Learning

Adding approximation layer

1. In **tabular** reinforcement learning either
   - MDP was given beforehand
   - Value function /or policy could be stored as a **lookup table** in memory
2. With real world problems the story is different **as we do not have access to the MDP**
   - We **can not** say for sure what will be the outcome of our actions
   - **If we had** access to crypto MDP we would know **EXACTLY** what would be the outcome of our investment in crypto
     → **but we don't have access : (**
3. The value function space is huge and trying to store it is **impractical**
   - Storing **all consecutive** frames in Atari

Dogecoin, crypto

Deep Reinforcement Learning

MDP Examples

1. Given the **baseline**
   - Atoms in the known universe

     $$\sim 10^{82}\ states$$

2. The research community has successfully **solved** problems
   - Game of Backgammon

     $$10^{20}\ states$$

   - Game of Go

     $$10^{170}\ states$$

   - Starcraft II

     $$10^{270}\ states$$

   - Helicopter manoeuvres

     $$continuous\ [\ \infty\ ]\ states$$

   - Crypto

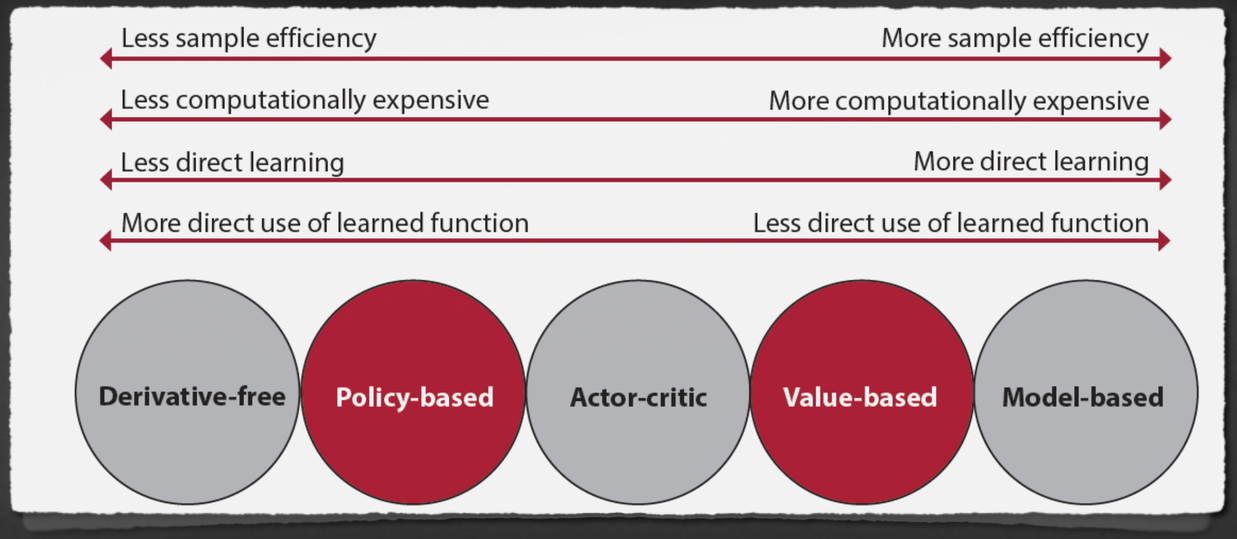
     $$continuous\ [\ \infty\ ]\ states$$

Starcraft II imperfect observations

Deep Reinforcement Learning

Types of algorithmic approaches

1. Derivative free
    - Genetic algorithms
    - Evolution strategies
2. Policy based
    - Direct policy optimisation
    - Value function optimisation **is optional**
3. Actor critic
    1. Direct policy optimisation
    2. Value function optimisation **via bootstrapping**
4. Value based
    1. Value function optimisation **with or without bootstrapping**
5. Model based
    1. Direct MDP modelling
    2. Value function /or policy **is optional**



Deep Reinforcement Learning Algorithms

Value Based Methods

Aggressive approximation method

Boston Dynamics, Atlas

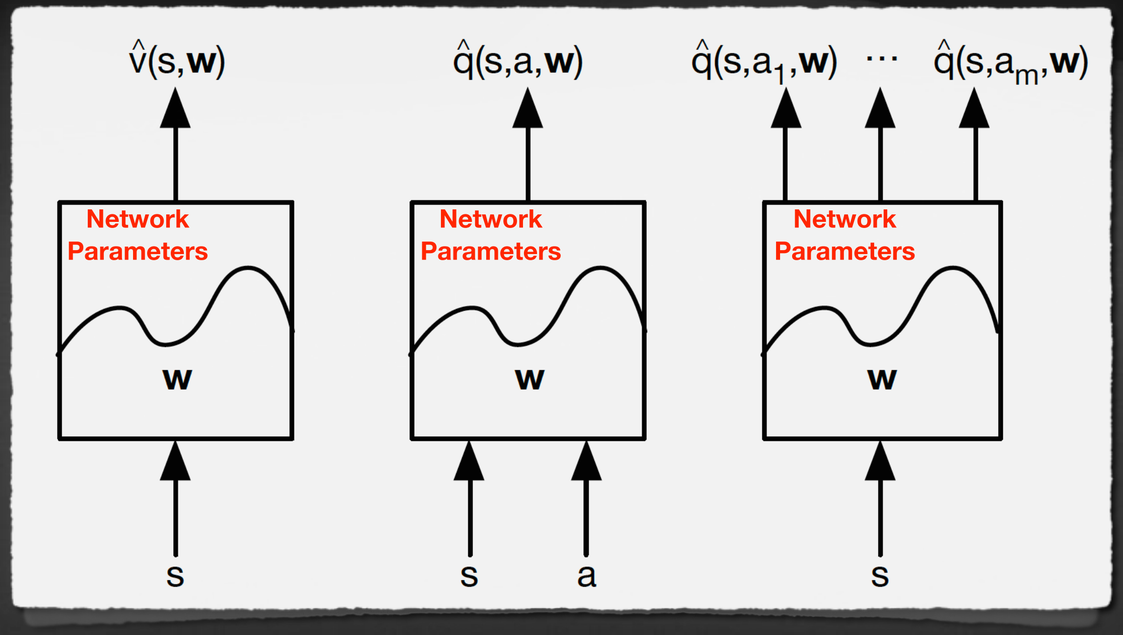
Incremental Value Function Approximation

Learning on the fly

1. Deep reinforcement learning is used to solve MDPs
   - That have **too many** states /or actions to store in memory
   - It is **too slow** to learn the value of each state individually
2. Therefore the solution is to estimate value function with function approximation **[e.g. Neural Network]**
   - $\hat{v}(s, \mathbf{w}) \approx \mathbf{v}_{\pi}(\mathbf{s})$

     Find approximation to **true state value function**
   - $\hat{q}(s, a, \mathbf{w}) \approx \mathbf{q}_{\pi}(\mathbf{s}, \mathbf{a})$

     Find approximation to **true action value function**
3. It means that we want to **generalise** from **seen states to unseen states**



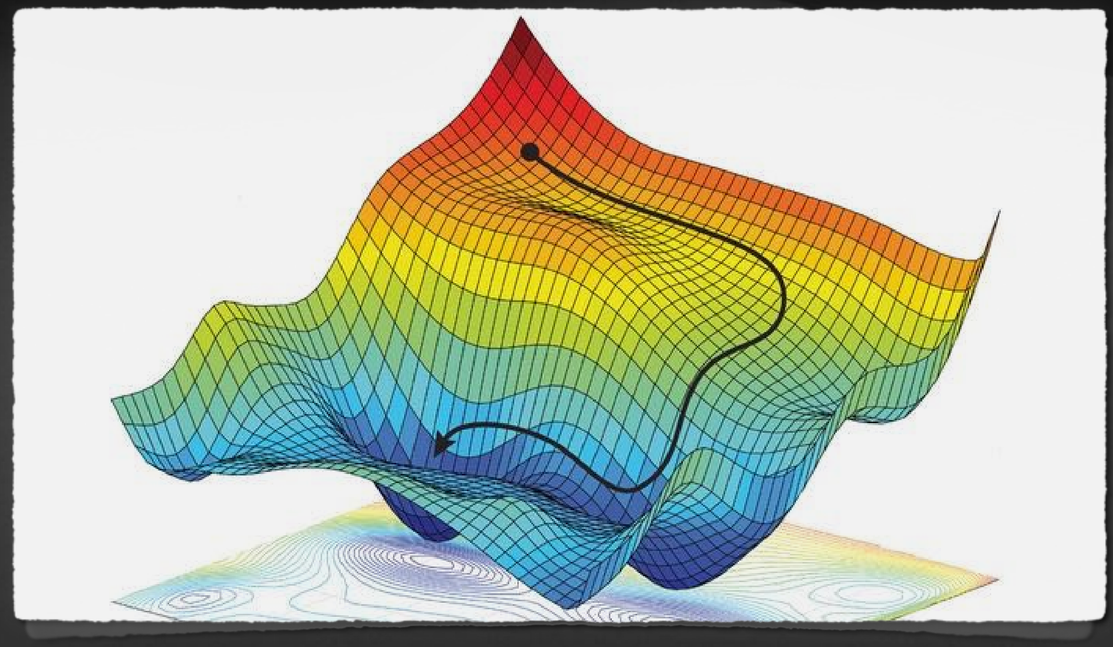
Objective Function
Numerical Gradient Descent

1. Let $J(w)$ be a **differentiable** function of parameter vector [w]
2. We can define the gradient of $J(w)$ to be

$$\nabla_w J(w) = \begin{bmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{bmatrix}$$

3. To find a **local minimum** of $J(w)$
4. Adjust the [w] parameters in direction of **-ve gradient**

$$\triangle w = -\frac{1}{2} \nabla_w J(w)$$



Finding local minimum in parameter space [w]

Objective Function

Mean-squared-error

1. Let $J(w)$ be a **differentiable** function of parameter vector [w]
2. And **true state value function** [$v_\pi(s)$] given by **supervisor**
3. Find parameter vector [w] **minimising MSE** between **approximate value function and true value function**

$$J(w) = \mathbb{E}_\pi[(v_\pi(s) - \hat{v}(s, w))^2]$$

4. Calculate gradient descent to find **local minimum**

$$\triangle w = \alpha \mathbb{E}_\pi[(v_\pi(s) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)]$$

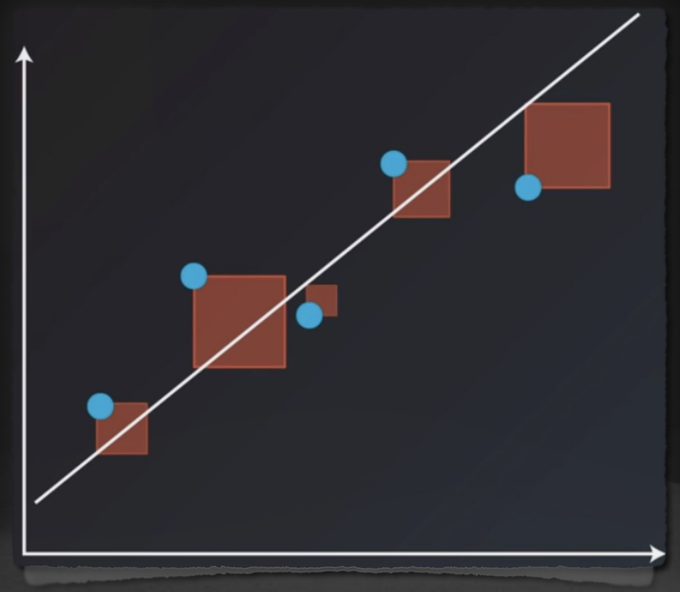
5. Stochastic gradient descent **samples** the gradient so we can **get rid of expectation**

$$\triangle w = \alpha[(v_\pi(s) - \hat{v}(s, w)] \nabla_w \hat{v}(s, w)$$

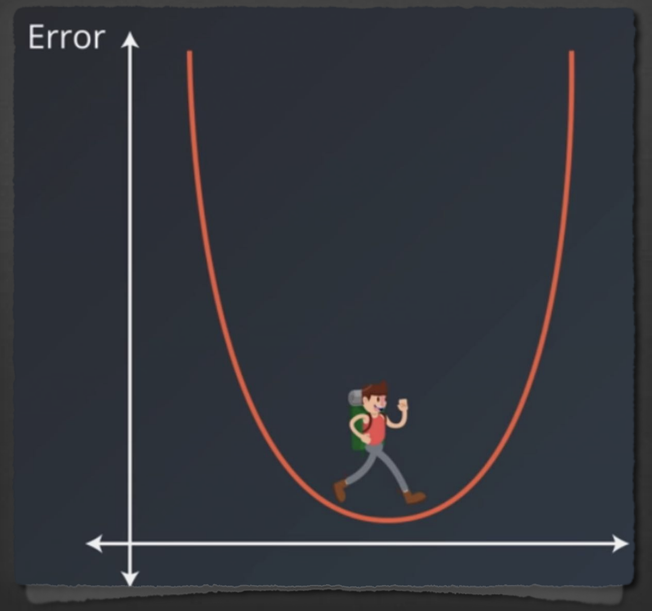
Vector of partial derivatives is calculated only $w.r.t$ current estimate

$\nabla_w \hat{v}(s, w)$

It tell us how to adjust the gradient such to minimise error generated



Error calculation



Error minimisation

But in reinforcement learning there is no supervisor

Therefore we have no access to true value function

Incremental Prediction Algorithms

A more realistic target

1. In practice we substitute a target for [$v_\pi(s)$]

$$\triangle w = \alpha[(v_\pi(s_t) - \hat{v}(s_t, w)]\nabla_w \hat{v}(s_t, w)$$

2. With **model free prediction** algorithm
   - Monte Carlo

$$\triangle w = \alpha[(G_t - \hat{v}(s_t, w)]\nabla_w \hat{v}(s_t, w)$$

   - TD [0]

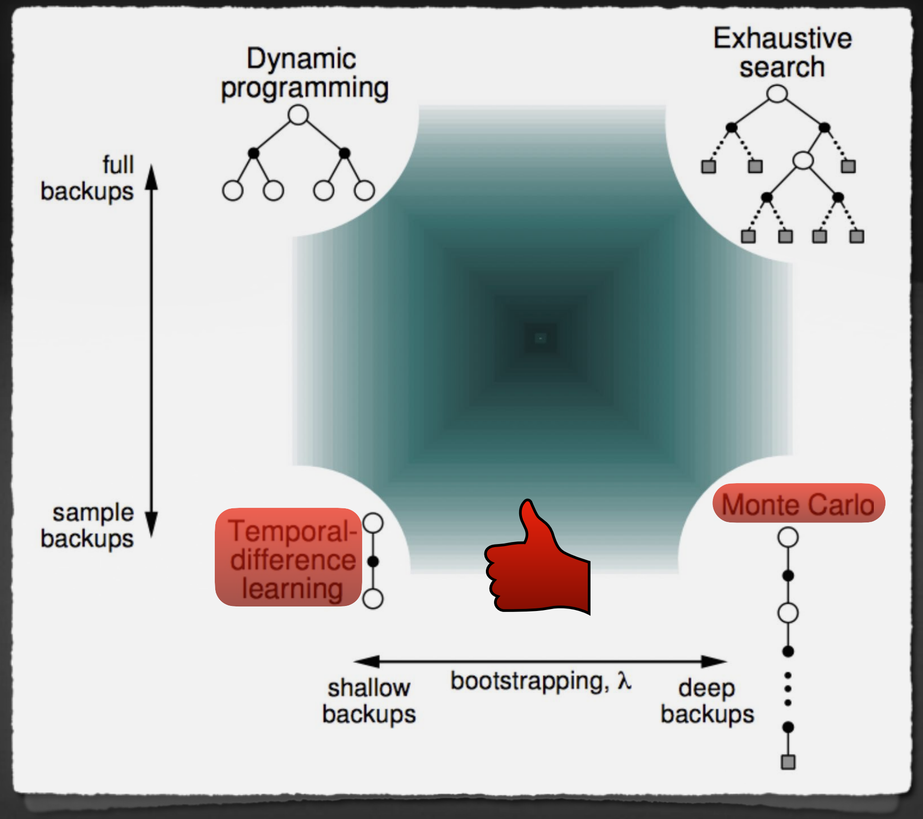
$$\triangle w = \alpha[(R_{t+1} + \gamma\hat{v}(s_{t+1}, w) - \hat{v}(s_t, w)]\nabla_w \hat{v}(s_t, w)$$

   - TD [λ] → forward view [MC]

$$\triangle w = \alpha[(G_t^\lambda - \hat{v}(s_t, w)]\nabla_w \hat{v}(s_t, w)$$

3. The general form of update equation **[Bellman Expectation Equation]**

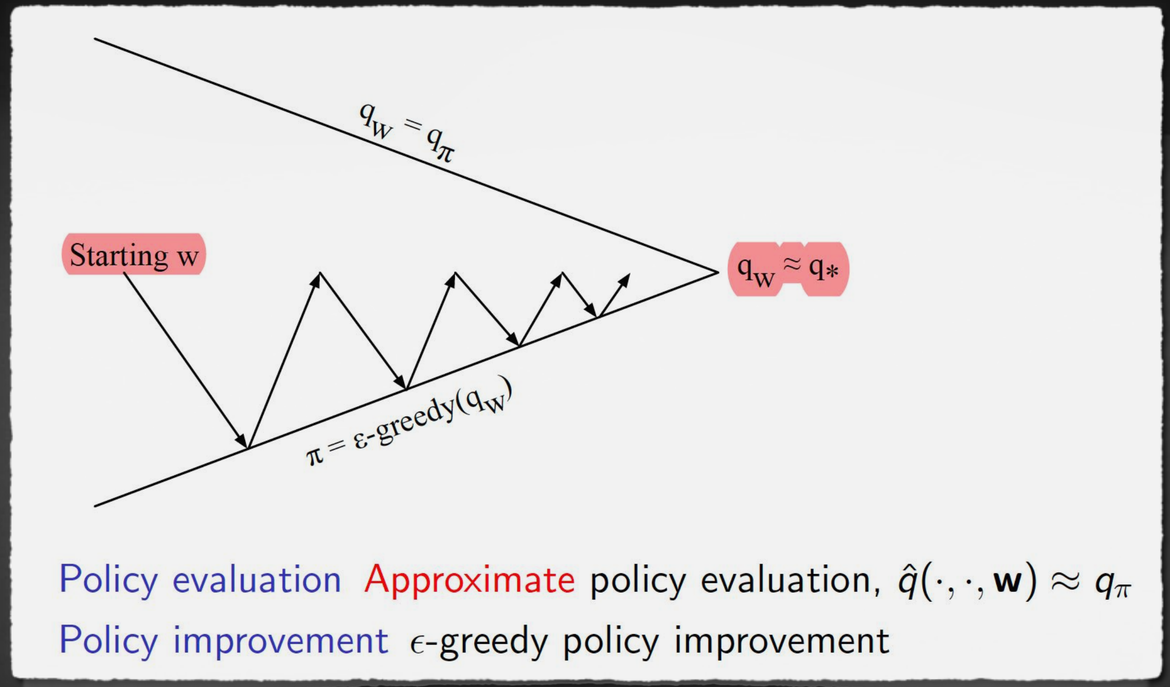
$$param\_update = learning\_rate \times error\_generated \times grad\_min\_direction$$



Action Value Function Approximation

Let's talk control

1. The story starts the same as per **tabular control methods**
2. We want to approximate the action value function
   - $\hat{q}(s, a, \mathrm{w}) \approx q_\pi(s, a)$
3. Minimise MSE between **approximate action-value function** and **true action-value function**
   - $J(w) = \mathbb{E}_\pi[(q_\pi(s, a) - \hat{q}(s, a, w))^2]$
4. Use SGD to find a **local minimum**
   - $\triangle w = \alpha[q_\pi(s, a) - \hat{q}(s, a, w)] \nabla_w \hat{q}(s, a, w)$



Control with action-value function approximation

Incremental Control Algorithm

Neural Fitted Q-Learning

1. Let's **naively** apply function approximation to **Q-Learning** algorithm
   - Policy **evaluation** [Bellman Optimality Equation]

$$J(w) = \mathbb{E}_{\pi}[(R_{t+1} + \gamma \max_{a_{t+1}} \hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)) \nabla_w \hat{q}(s_t, a_t, w)]$$

   - Policy **improvement**

$$\epsilon - greedy$$

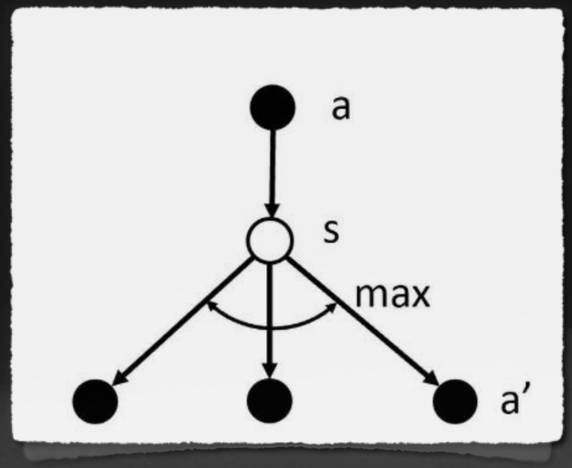
2. Lets design our network as **state-in-values-out** architecture
   - Network **input**

     Current observation

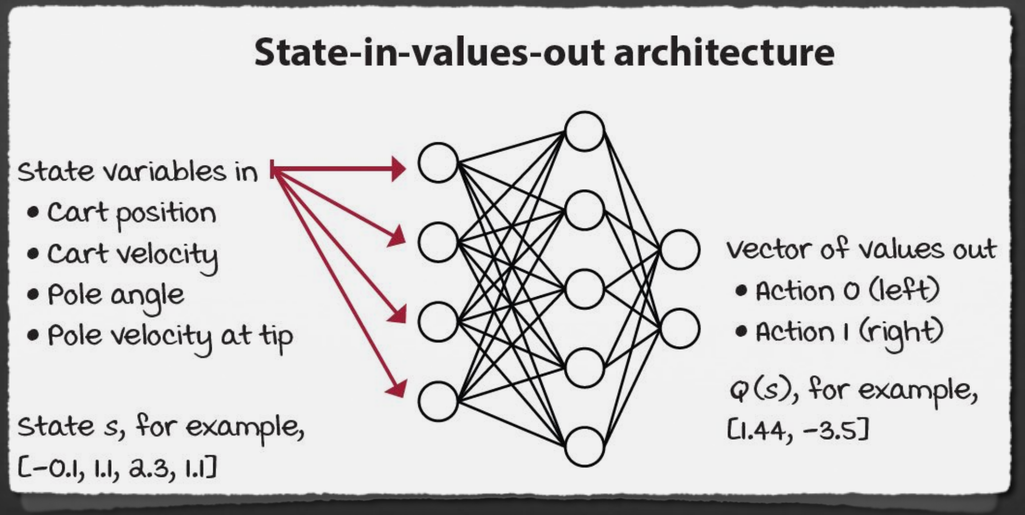
   - Network **output**

     Q-values for all possible actions

3. The learnt **policy** will be **deterministic** due to policy improvement algorithm used in **value based methods**



Q-Learning backup diagram

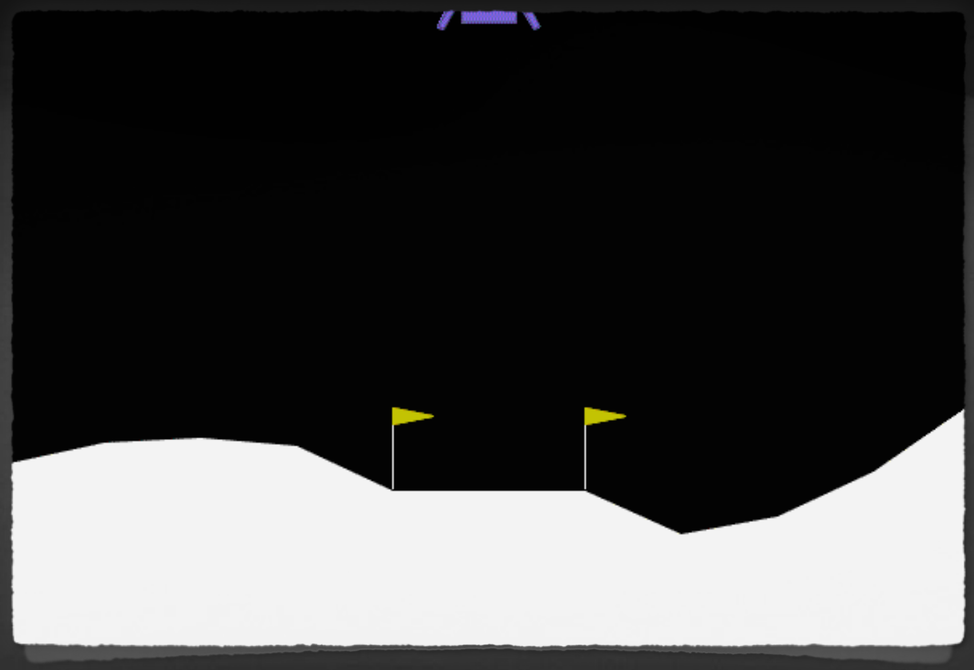


NN design for cart pole environment

Evaluation Environment

Lunar Lander environnement

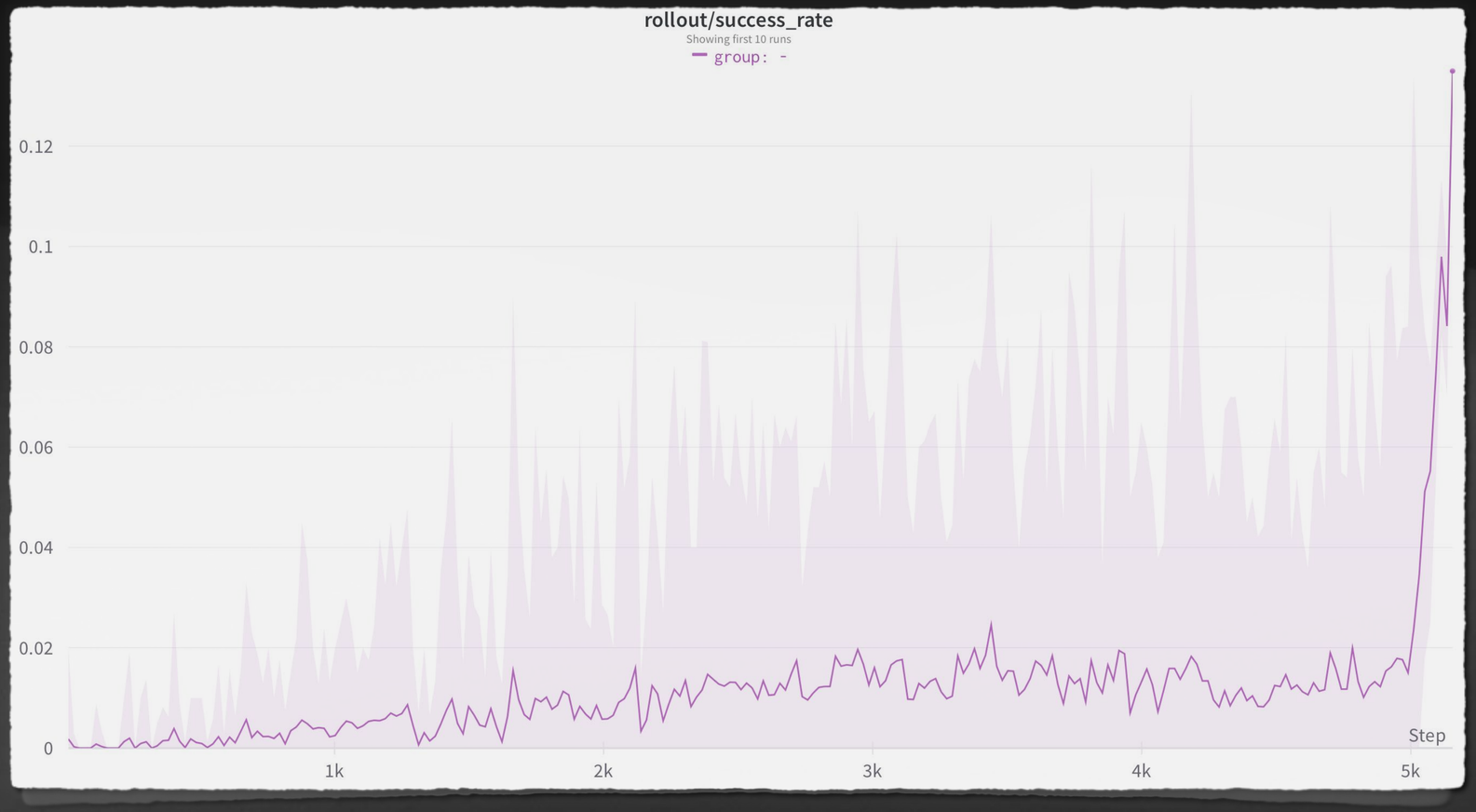
1. Consider the **discrete action space** environment from **OpenAI Gym**
2. Namely LunarLander-v2
3. Action space - Discrete(4) → **discrete**
    - Action [0] - do nothing
    - Action [1] - fire left orientation engine
    - Action [2] - fire main engine
    - Action [3] - fire right orientation engine
4. Observation space - Box(8) → **continuous**
    - Lander [x, y] coordinates
    - Linear velocities in coordinates [x, y]
    - Angle
    - Velocity
    - Two booleans representing whether each leg is in contact with the ground



LunarLander-v2 environment

NFQ Results …

~ 15% success rate



NFQ results on LunarLander-v2 env

Neural Fitted Q-Learning

Sucks

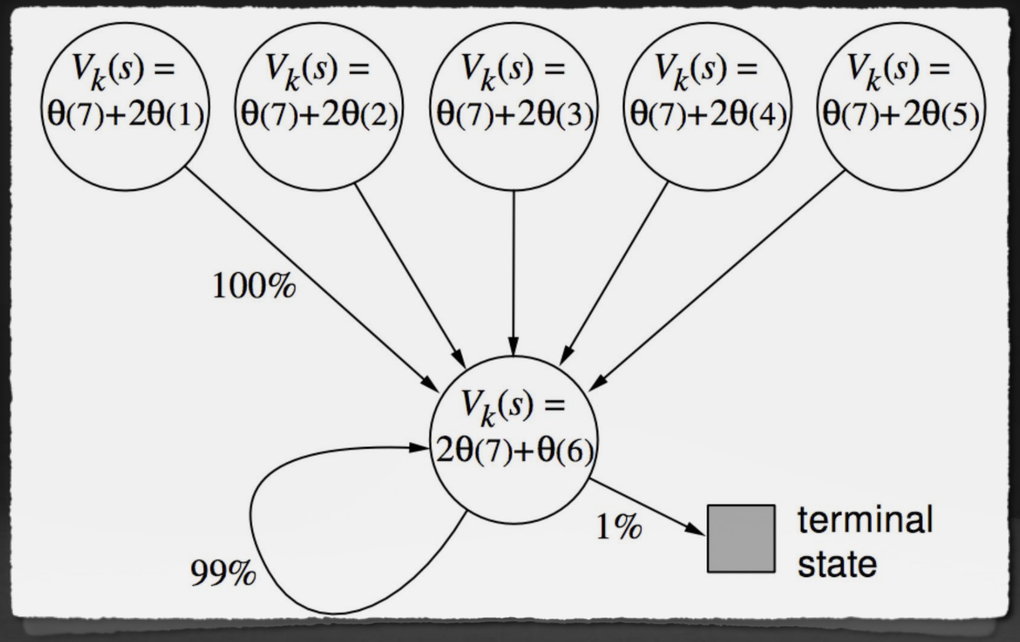
But why ?

The Deadly Triad

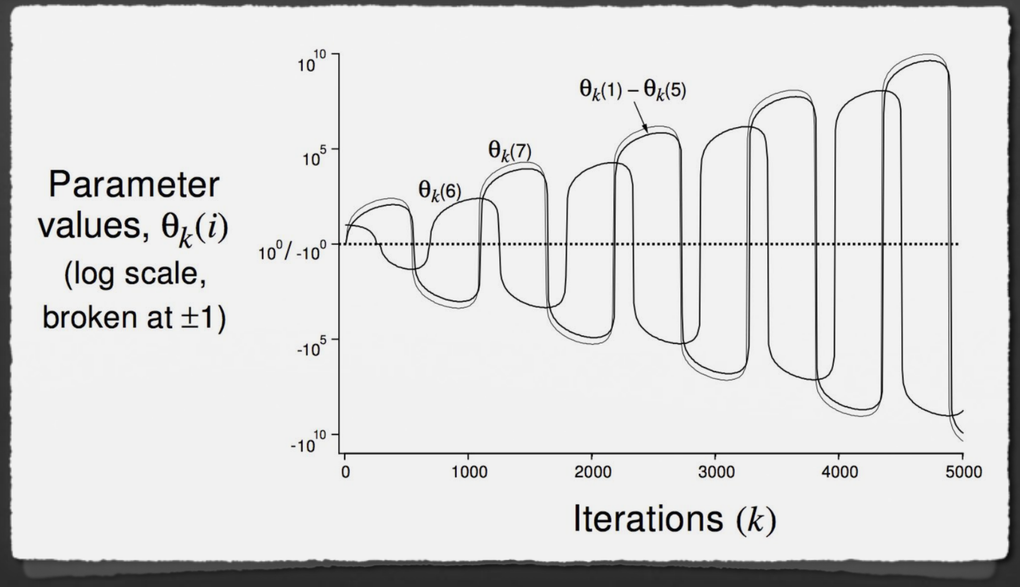
Miserable fate of incremental methods

1. Function approximation
   - Using **any layer of approximation**, such as neural networks or linear function approximation
2. Bootstrapping
   - Update targets that include **existing estimates** → temporal difference learning
3. Off-Policy
   - Training on distribution of transitions **other that that produced by the target policy**

There is **no guarantee** that our method will **converge to any optima** given we use **ALL** of these at the same time



Baird's counterexample

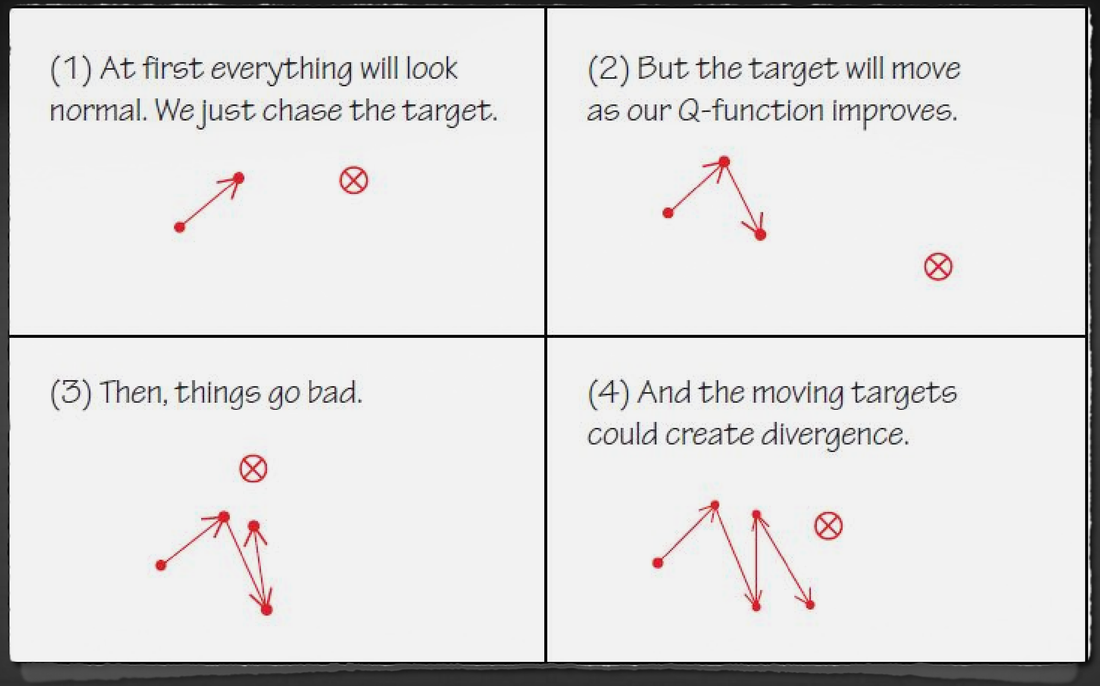


Parameters divergence in Baird's counterexample

Non Stationary Target

Chasing moving target

1. Given function approximation as the generalisation tool between state-action pairs
2. And TD [0] as policy evaluation mechanism
3. **Any change** to the network parameter space [w] will have an **impact on all similar states at once**
   - This happens due to **close correlation** of state-action pair which are being estimated and successor state-action pair
4. Additionally since the TD is using bootstrapping mechanism
   - Our target values depend on the values for the successor state
5. We are creating a non-stationary target for our learning updates
6. And thus since we are updating the parameters of our approximate action-value function
   - $\triangle w = \alpha[r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})]\nabla_w \hat{q}(s, a, w)$
   - Where $\nabla_w \hat{q}(s, a, w)$ is the vector of partial derivatives w . r . t current estimate of action value function
7. **The target's value changes and makes our most recent update out dated**



Moving target w . r . t network parameters change

Data Correlation with Time

Breaking the IID assumption

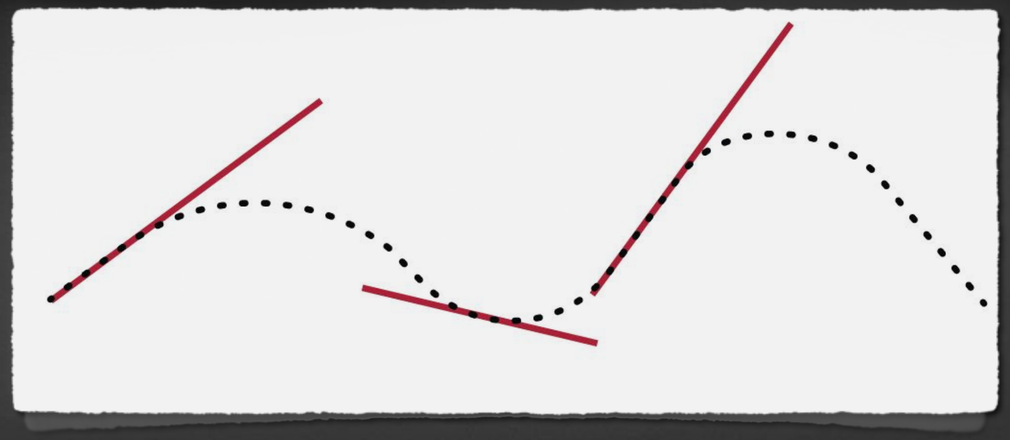
1. Most optimisers are designed to work with **independent and identically distributed data**

2. This assumption is obvious in supervised learning

   - To train a dog breed classifier take 1 billion images with ergodic class distribution and shuffle them

3. Incremental DRL methods on the other hand are dealing with the data that is both dependent and non-identically distributed

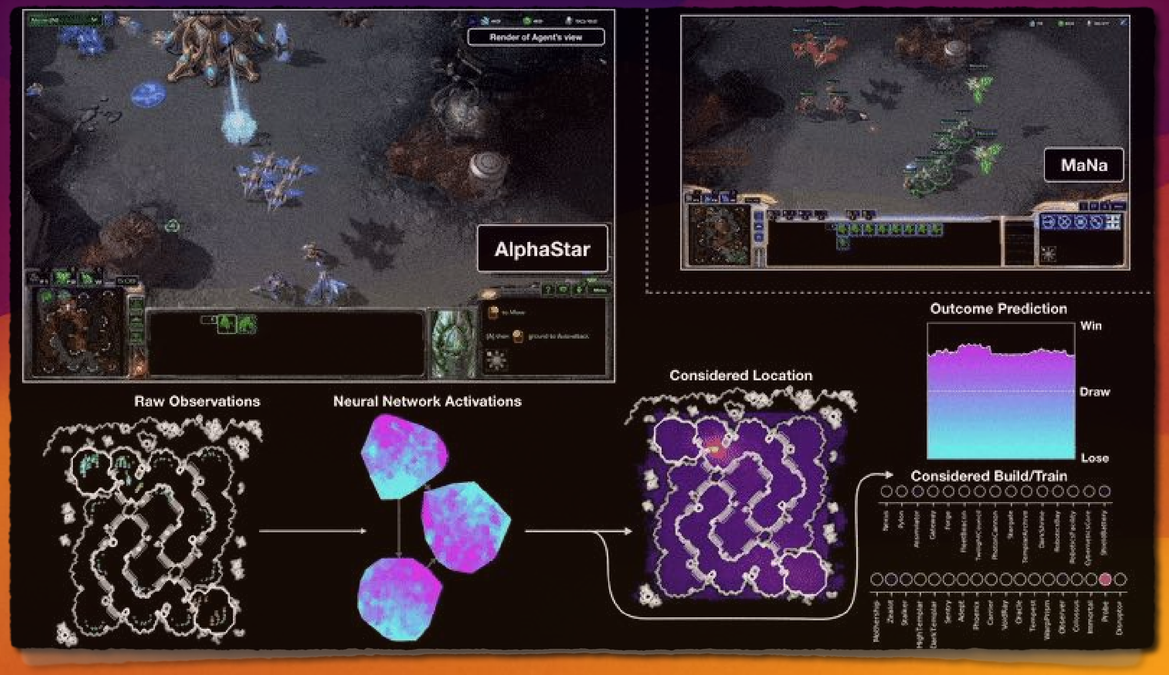
   - Video stream is an example of time correlated data



Similarity of the adjacent data points in single trajectory

Baseline improvements

Closing the gap between reinforcement learning and supervised learning



DeepMind, AlphaStar

Batch Value Function Approximation

Storing memories

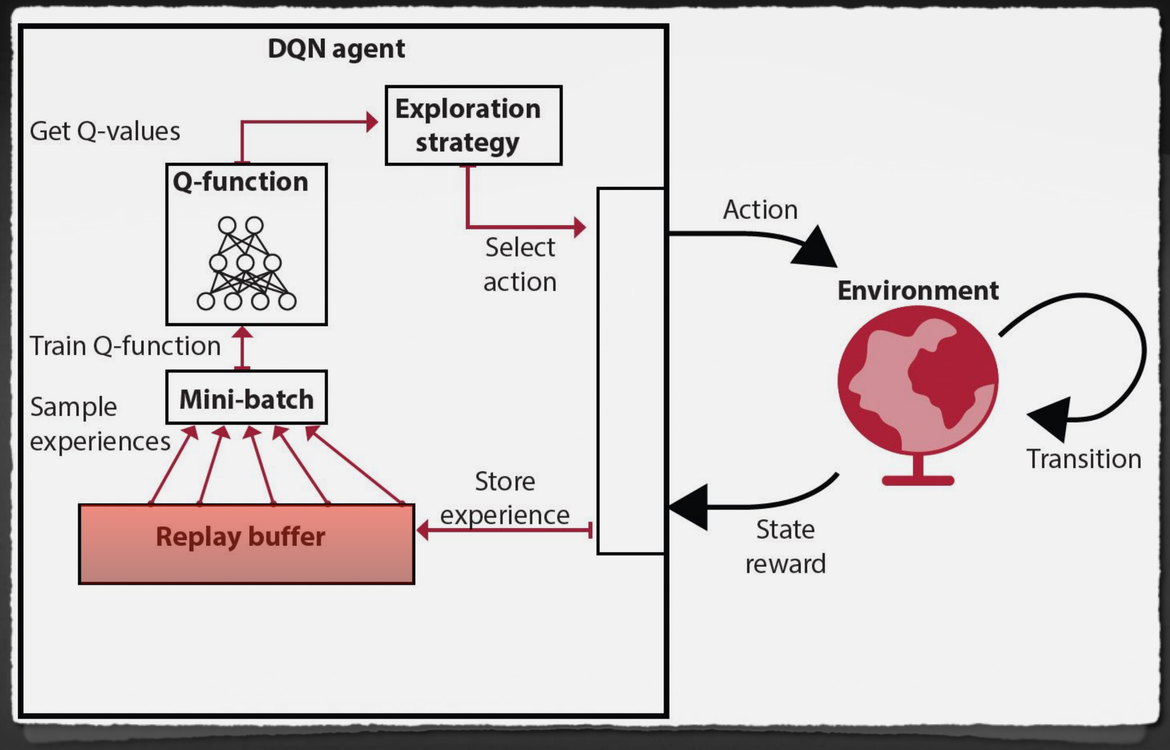
1. Consider a data structure called **replay buffer** **[experience replay** or **replay memory]**

2. Used to store **trajectories** [τ] in memory for **several steps**

   - $\tau_0 = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots)$
   - $\tau_1 = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots)$
   - $D = [\tau_0, \tau_1, \tau_2, \ldots]$

3. Therefore allowing us to **sample a mini-batch** from broad set of past experiences

4. This approach solves the issue with **data correlation with time**



Experience replay strategy

Target Networks
Freezing network weights

1. **Target network** is a **separate network** that we can **fix for multiple steps** and reserve for calculating **more stationary targets**

2. In practice the target network is a **copy** of our main network [online network] parameters

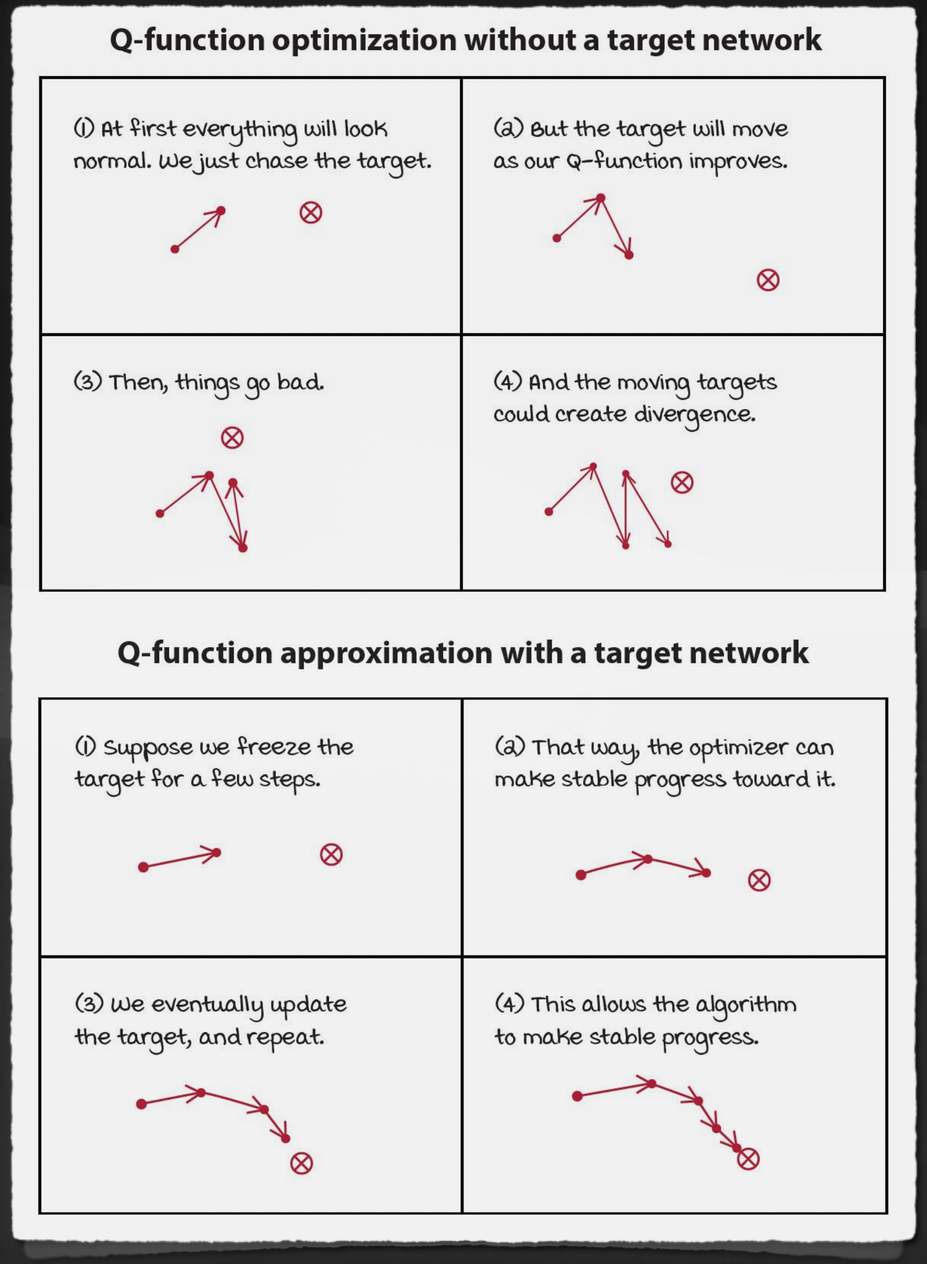
3. Which mean's that we have two instances of **neural network weights**

4. The **online** network parameters [w] are optimised on **every step**

5. The **target** network parameters [$\overline{w}$] are **copied** or **soft-copied** from **online** network every so often

   - Depending on the problem it varies from 10 to 10,000 steps

6. This approach solves the issue with **non stationary target**



Freezing the target for our network's update

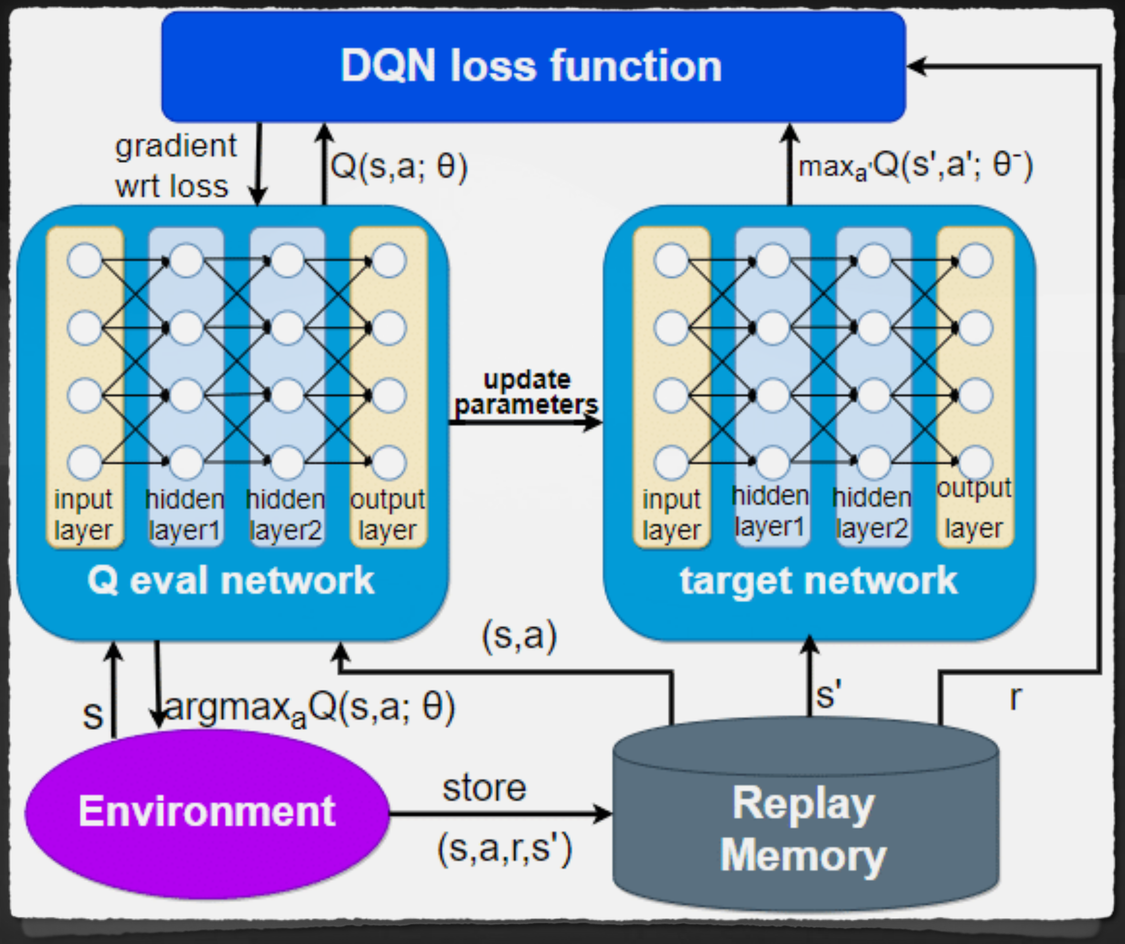
DQN
Deep Q-Learning Network

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**

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



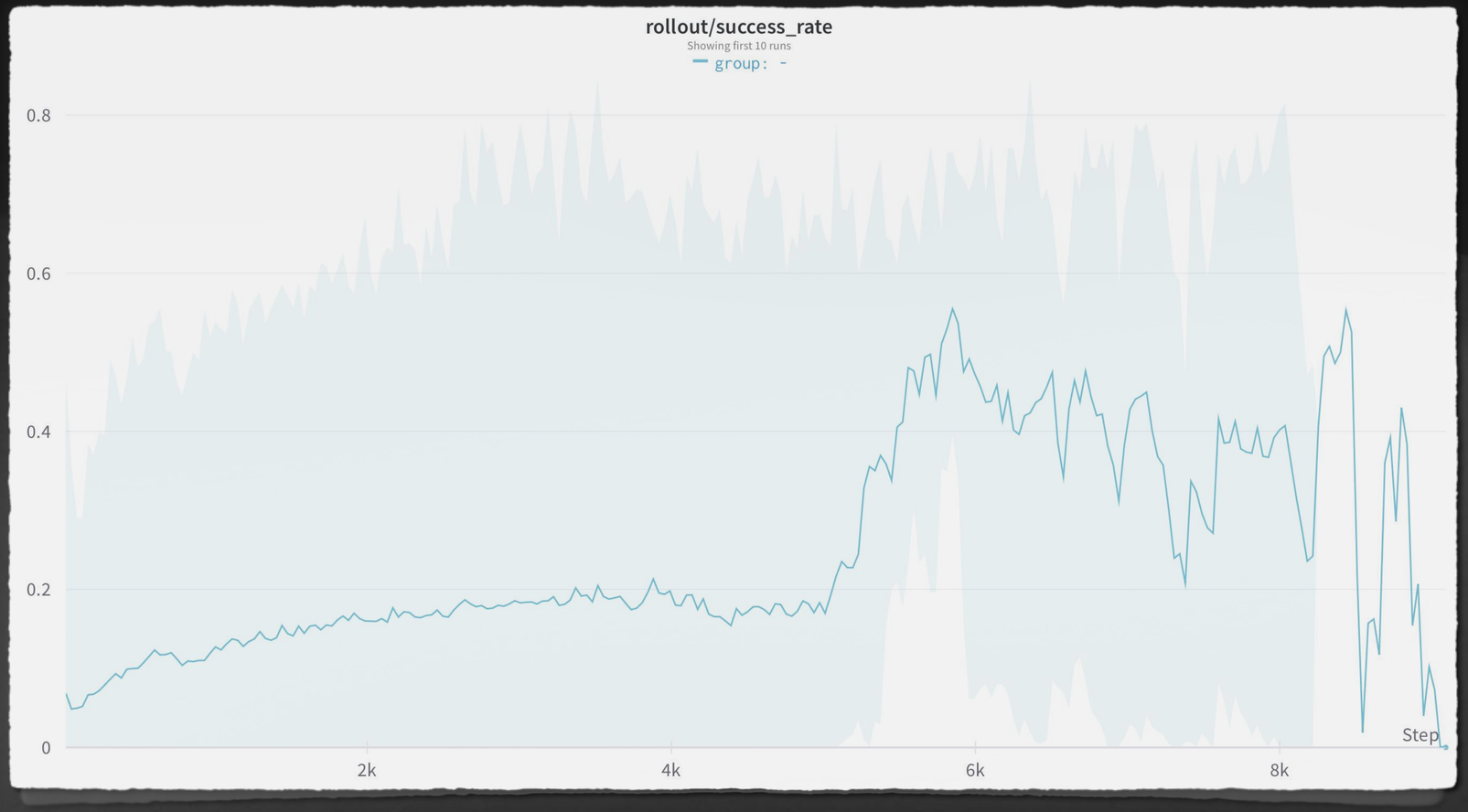
DQN architecture

DQN creates many small supervised learning problems

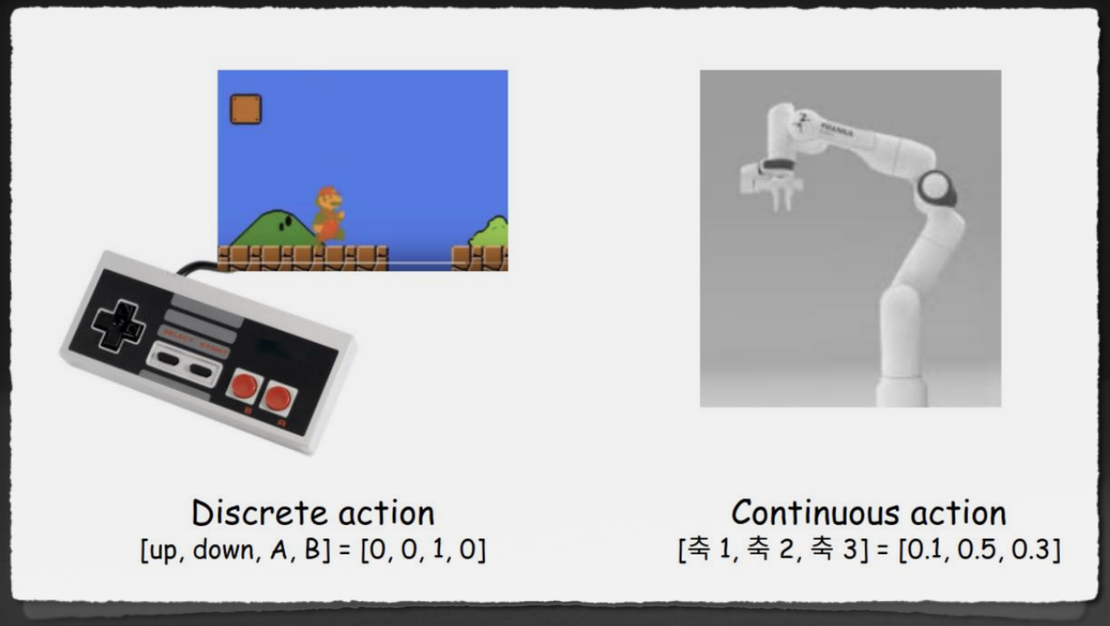
And solves them one by one

DQN Results …

~ 80% success rate



DQN results on LunarLander-v2 env



But there is one more issue with value based methods

And is connected to action space

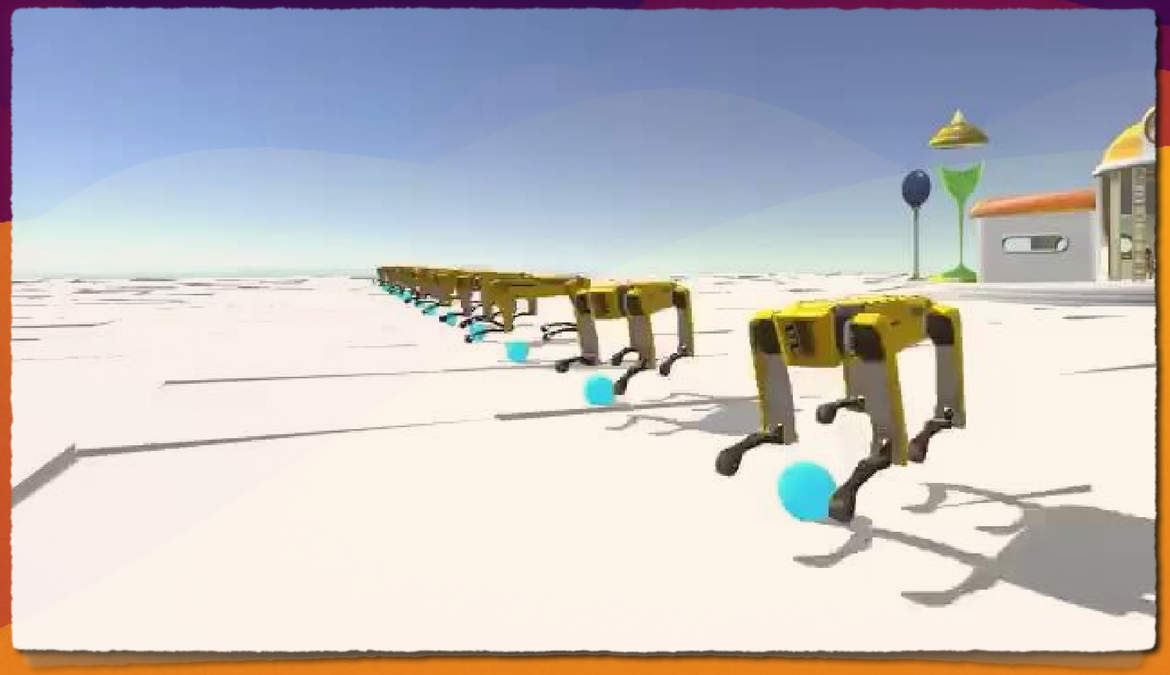
$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

Bellman Optimality Equation comes with a price [max operator]

Value based methods are not efficient in continuous action spaces

Policy Based Methods

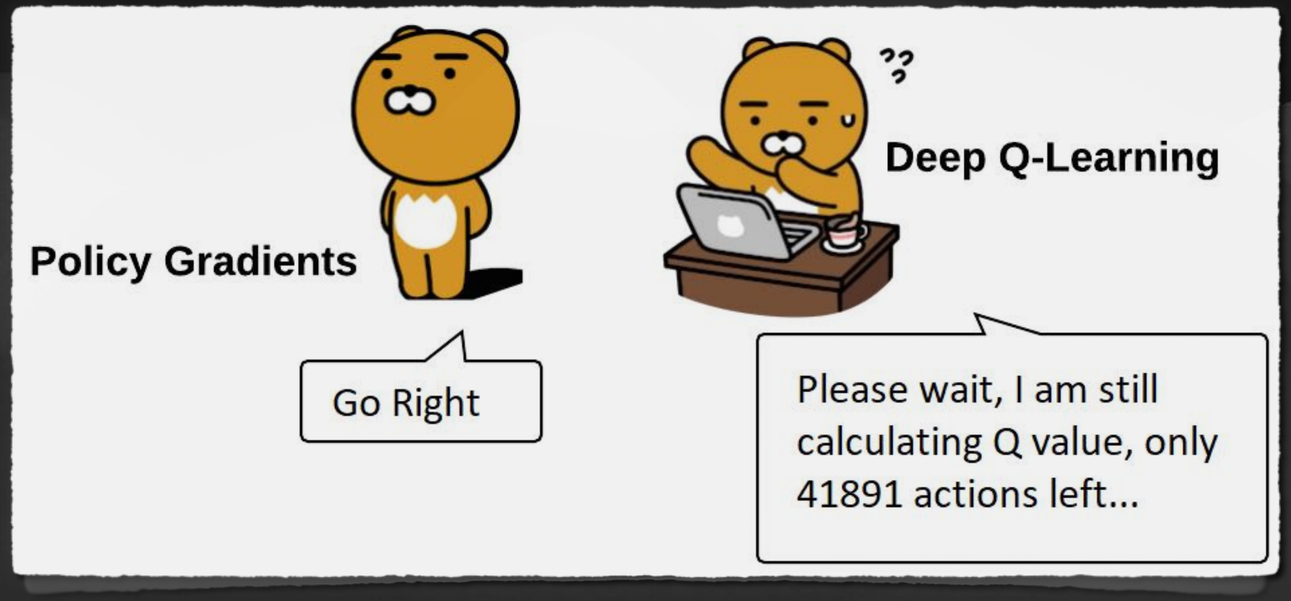
Optimising policy directly



Boston Dynamics, Quadrupedal Robot Sim Training

Optimising policies directly

It seems like a right objective



Policy Based Methods

Direct policy parametrisation

1. Policy based reinforcement learning is an **optimisation** problem
   - Given **policy** $\pi_\theta(s, a)$ with parameters θ

     $$\pi_\theta(s, a) = \mathbb{P}[a \mid s, \theta]$$

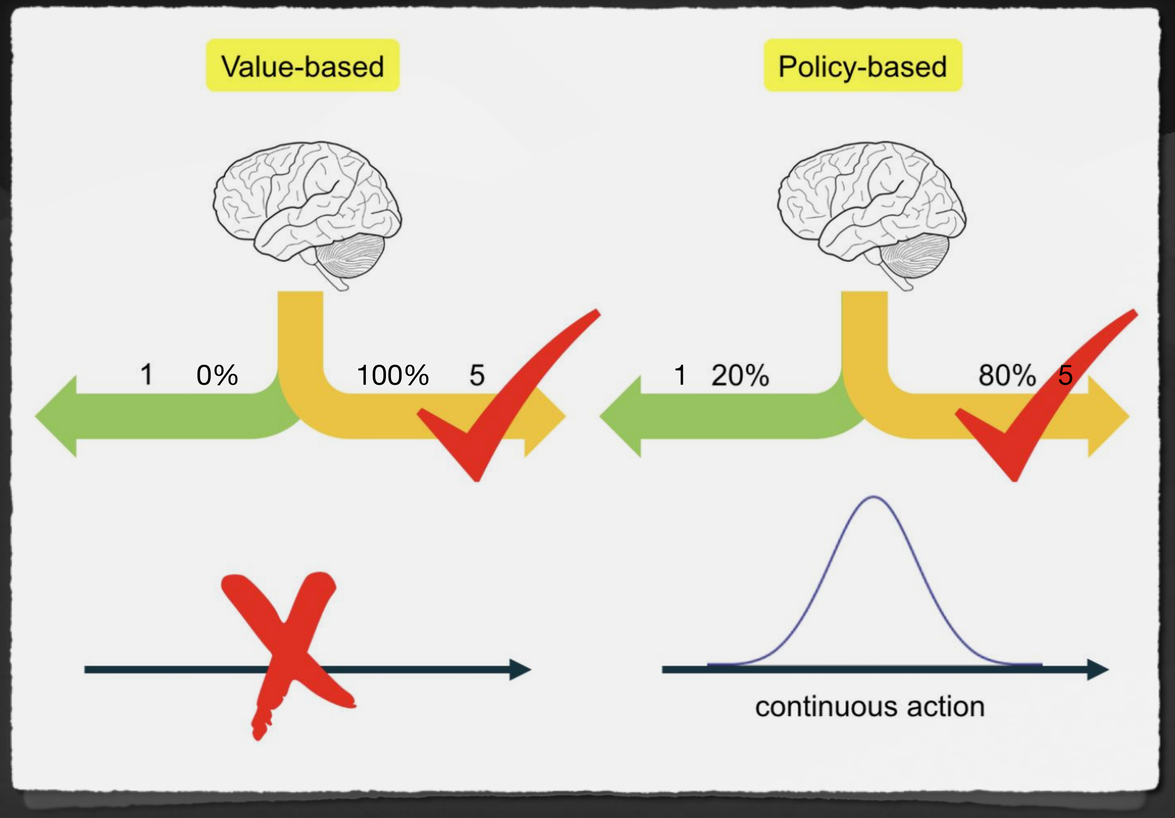
   - Find **best** θ
2. To measure the quality of policy [π_θ] we have to choose the **objective function**
   - **Episodic environments** can use the **start value**

     $$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

   - **Continuing environments** can use the **average value**

     $$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

   Where d^{π_θ} is **stationary distribution** of Markov chain for π_θ



Value Based vs Policy Based

Objective Function

Numerical Gradient Ascent

1. Let $J(\theta)$ be any **policy objective** function
2. We can define the gradient of $J(\theta)$ to be

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

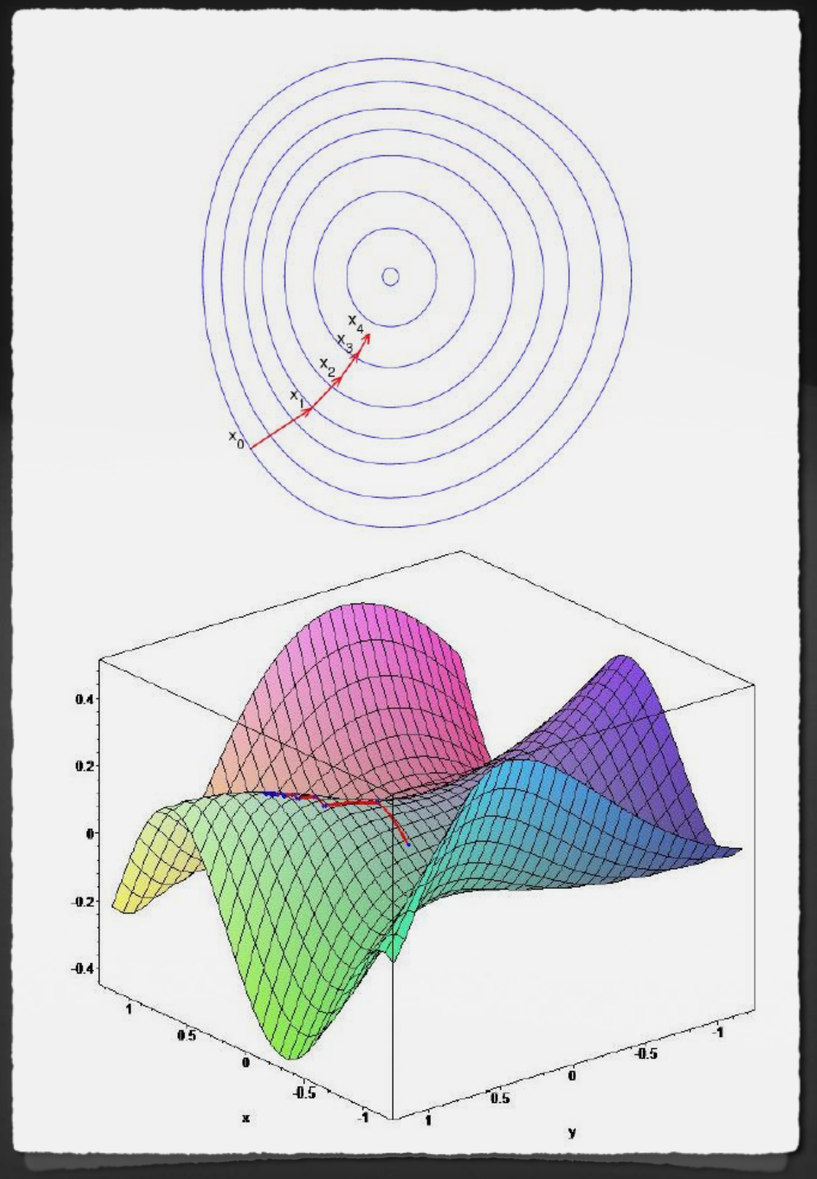
3. To find a **local maximum** of $J(\theta)$
4. Adjust the [θ] parameters in direction of +**ve gradient**

$$\triangle \theta = \alpha \nabla_\theta J(\theta)$$

5. In order to **evaluate** policy gradient of $\pi_\theta(s, a)$
6. Compute gradients by **finite differences**
   - Estimate k^{th} partial derivatives of objective function $w.r.t\ \theta$
   - By perturbing θ by small amount ϵ in k^{th} dimension

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

Where u_k is unit vector with 1 in k^{th} component 0 elsewhere



Finding local maximum in parameter space [θ]

Numerical computation of policy gradient is simple

But noisy and inefficient in policy based methods


calm down
calm down
NUMERICAL GRADIENT SUCKS

Policy Gradient Theorem

Reduce performance dependency

1. Let's define the performance measure
   - **value of the start state** of the episode

     $$J(\theta) = v_{\pi_\theta}(s_0)$$

     Where the v_{π_θ} is the **true value function** for π_θ, the policy determined by parameter θ
2. The problem is that **performance** depends on **both**
   - the **action selection**
   - and the **distribution of states** in which those selections are made
3. And that **both** of these are **affected** by the policy parameter [θ]
4. The **distribution of states** is the function of the **environment** and is typically **unknown** for the agent
5. Fortunately there is an excellent theoretical proof called **Policy Gradient Theorem**
6. Which provides an **ANALYTICAL** expression for the gradient of performance $w.r.t$ policy parameter [θ] which is what we need to approximate for **gradient ascent**
7. **That does not involve the derivative of the state distribution**
   - **the constant of proportionality** $\sum_s \mu(s)$

For episodic case is the average length of an episode

For continuing case it is 1

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.18)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a)\big(r + v_\pi(s')\big) \Big] \\
& && \text{(Exercise 3.19 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''|s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 199)} \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Eq. 9.3)} \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Q.E.D.)}
\end{aligned}$$

Score Function

Computing gradient analytically

1. Let's now compute the policy gradient **analytically**
2. Assume policy π_θ is **differentiable** whenever it is **non-zero**
3. And we know the gradient $\nabla_\theta \pi_\theta(s, a)$
   - **Likelihood ratios** exploit the following identity

$$\nabla_\theta \pi_\theta(s, a) = \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} = \pi_\theta(s, a) \nabla_\theta log \pi_\theta(s, a)$$

   - The **score function**

$$\nabla_\theta log \pi_\theta(s, a)$$

   - Tells us how to **maximise** the likelihood of a certain action [a]

A Better Objective Function

Friendly Policy Gradient Theorem

1. Consider **score function**

$$\nabla_\theta \pi_\theta(a,s) = \pi_\theta(a,s)\frac{\nabla_\theta \pi_\theta(a,s)}{\pi_\theta(a,s)} = \pi_\theta(a,s)\,\nabla_\theta log \pi_\theta(a,s)$$

2. And **Policy Gradient Theorem**

$$\nabla_\theta J(\theta) = \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(a,s) q_\pi(s,a)$$

3. Since the Policy Gradient Theorem applies to all objective function

4. We can define a **general policy gradient** objective

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a,s) Q^{\pi_\theta}(s,a)]$$

→ How to adjust our policy to get **more** or **less** of some action [a]

Direct policy differentiation

$$\theta^\star = \arg\max_\theta \underbrace{E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]}_{J(\theta)}$$

a convenient identity

$$\pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) = \pi_\theta(\tau)\frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau)$$

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\underbrace{r(\tau)}_{\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)}] = \int \pi_\theta(\tau) r(\tau) d\tau$$

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau = \int \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

REINFORCE

Monte-Carlo Policy Gradient Control

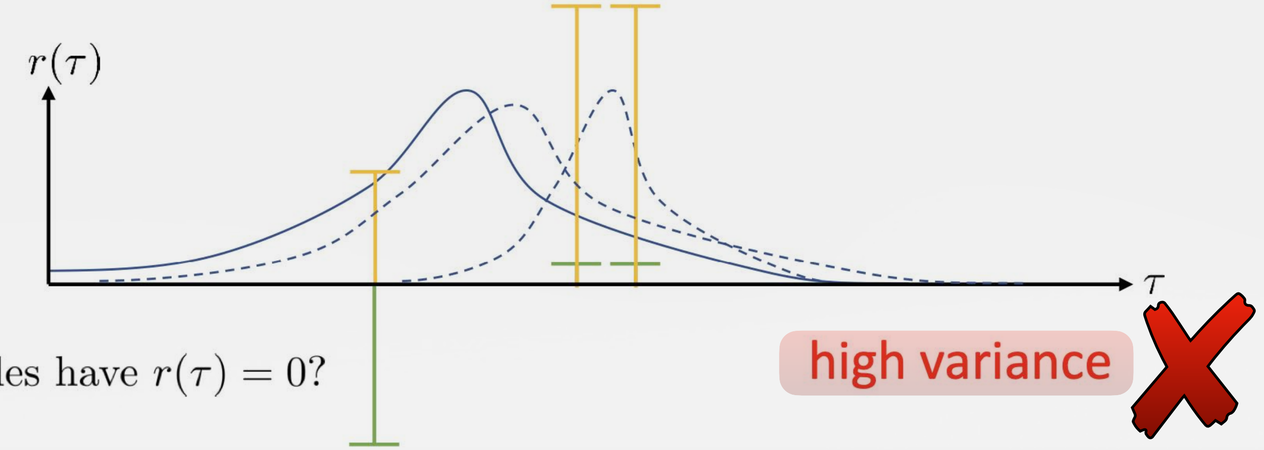
- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE
    Initialise θ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
        for $t = 1$ to $T - 1$ **do**
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return θ
end function

What is wrong with the policy gradient?

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau) r(\tau)$$



high variance

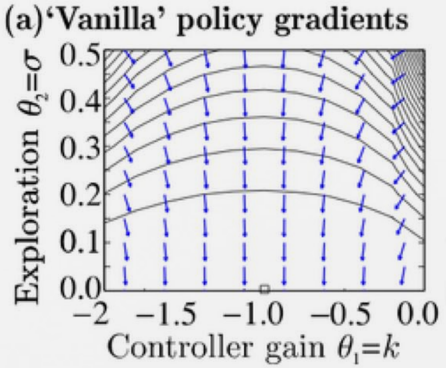
even worse: what if the two "good" samples have $r(\tau) = 0$?

$$\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = -\frac{1}{2\sigma^2}(k\mathbf{s}_t - \mathbf{a}_t)^2 + \text{const} \qquad \theta = (k, \sigma)$$

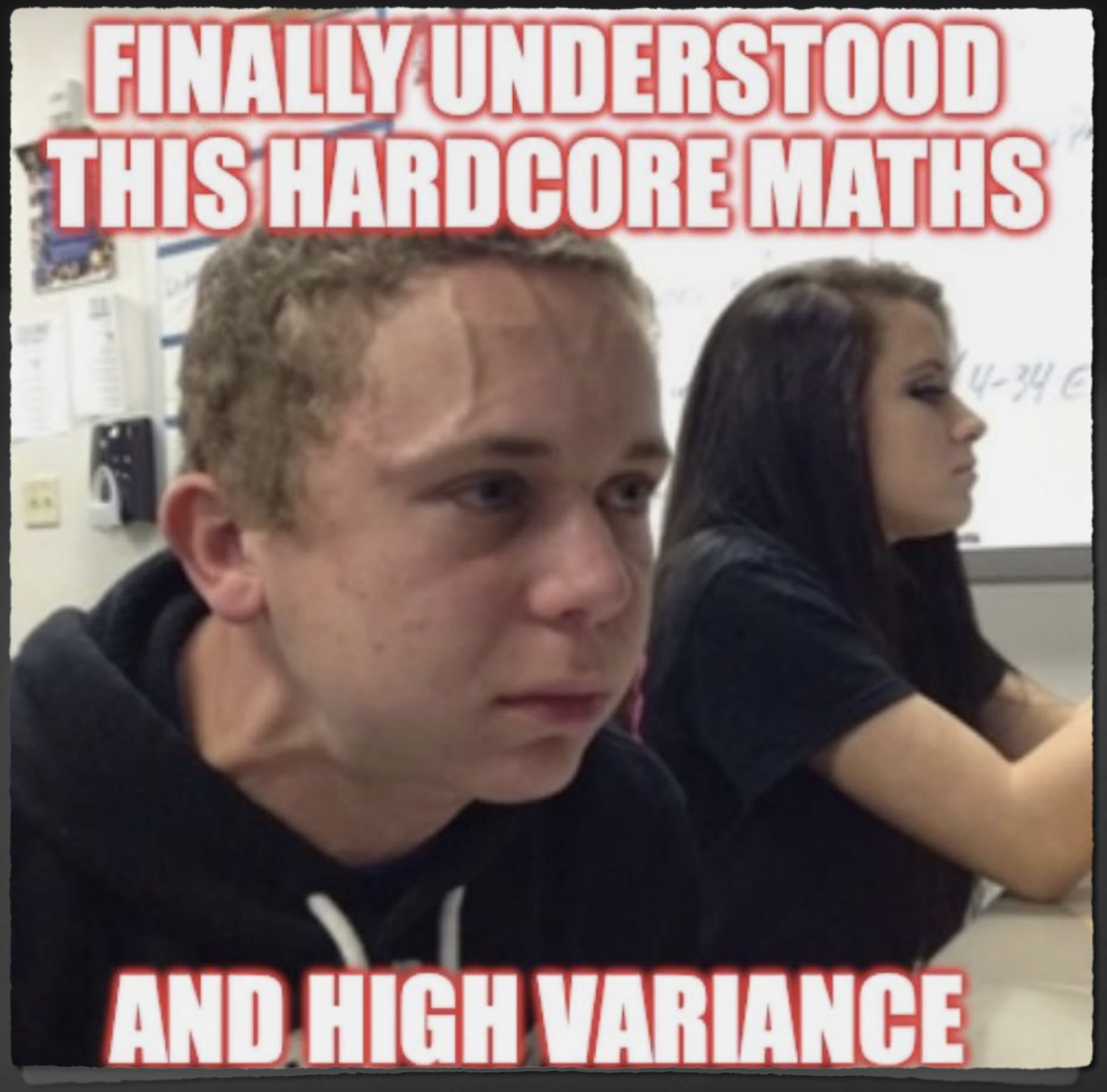
$$r(\mathbf{s}_t, \mathbf{a}_t) = -\mathbf{s}_t^2 - \mathbf{a}_t^2$$

slow convergence

hard to choose learning rate



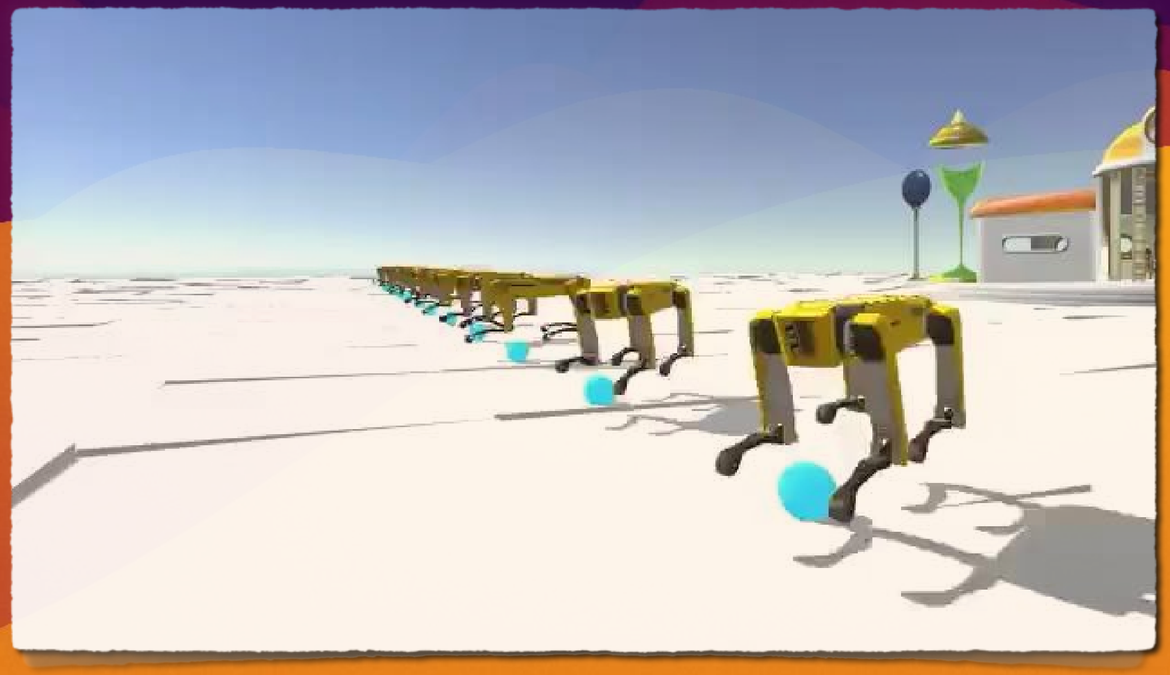
(image from Peters & Schaal 2008)


FINALLY UNDERSTOOD
THIS HARDCORE MATHS
AND HIGH VARIANCE

Is deep reinforcement learning even working?

Actor Critic

Reducing variance using a critic



Boston Dynamics, Quadrupedal Robot Sim Training

Actor Critic

Adding Q-value to policy gradient

1. Due to Monte Carlo target high variance
2. Let's consider a **critic** to **estimate** the action-value function
   - **Policy evaluation** problem

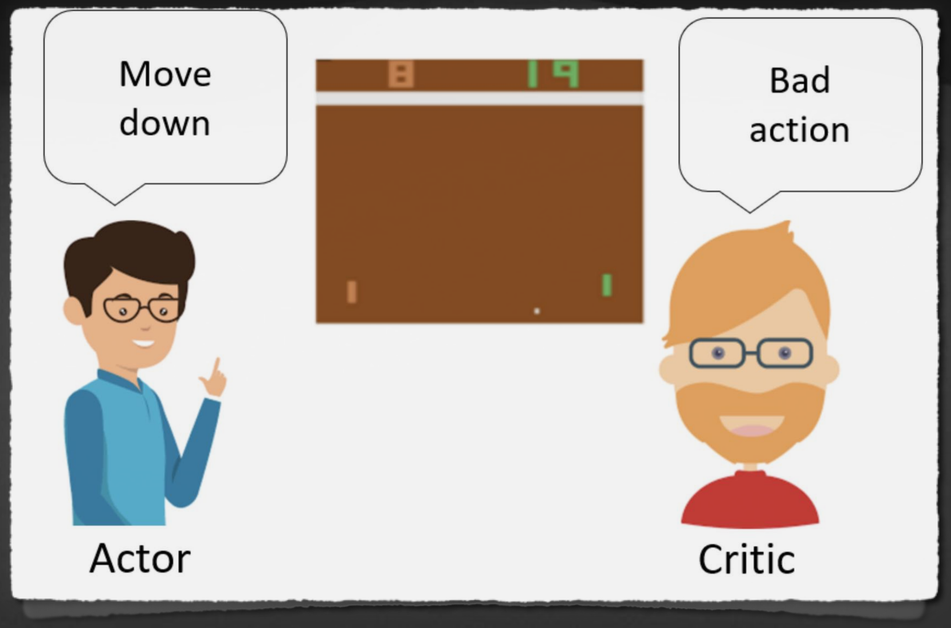
$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

   - **How good is policy** [π_θ] for **current** parameters [θ]
3. The **Policy Gradient Theorem** is defined as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a,s) Q^{\pi_\theta}(s,a)]$$

4. Adding **critic** to Policy Gradient Theorem results in

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a,s) Q_w(s,a)]$$



Actor and Critic duo

Actor Critic

Approximate policy gradient

1. Actor critic algorithms maintain **two** sets of parameters [w] and [θ]
   - Critic updates **action-value function** parameters [w]

$$\triangle w = \alpha[r + \gamma Q_w(s', a') - Q_w(s, a)] \nabla_w Q_w(s, a)$$

   - Actor updates policy parameters [θ], **in direction suggested by critic**

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

Stochastic gradient descent **samples** the gradient, therefore **expected** update is equal to full gradient update

According to Richard S. Sutton and Andrew G. Barto

Actor Critic methods must bootstrap

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

Adding Baseline

Reducing variance even further

1. Let's unroll the expectation in policy gradient theorem and subtract baseline function [B(s)]

$$\mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a,s)B(s)] = \sum_s \mu(s)B(s) \sum_a \nabla_\theta \pi_\theta(s,a)B(s)$$

$$\sum_s \mu(s)B(s) \sum_a \nabla_\theta \pi_\theta(s,a)B(s) = \sum_s \mu(s)B(s) \nabla_\theta \sum_a \pi_\theta(s,a)$$

- Probability distribution [$\mu(s)$] sums up to 1, which is a constant gradient
- This whole term is zero mean, zero expectation

$$\sum_s \mu(s)B(s) \nabla_\theta \sum_a \pi_\theta(s,a) = 0$$

- Therefore we can use the **baseline**

Adding Baseline
Critic adjustments

1. A good choice will be a **state value function**

$$B(s) = V^{\pi_\theta}(s)$$

2. So we can rewrite the policy gradient using the advantage function

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

   - **How much better** than usual is to take action [a]

3. **Critic's** role is now to approximate the **action advantage function**

$$A_w(s,a) \approx A^{\pi_\theta}(s,a)$$

Adding Baseline
Last touches

1. Policy Gradient Theorem with advantage function is the following

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a, s) A_w(s, a)]$$

2. The **intuition** behind this equation is
   - **Push policy parameters towards situations where we do better than usual**
3. Gradient direction
   - $A_w(s, a) > 0$

     The gradient will tell us how to move in a direction to achieve that gain
   - $A_w(s, a) < 0$

     The gradient will move us in the opposite direction

Actor Critic Methods

Are the building blocks of SOTA DRL algorithms

Coming soon …

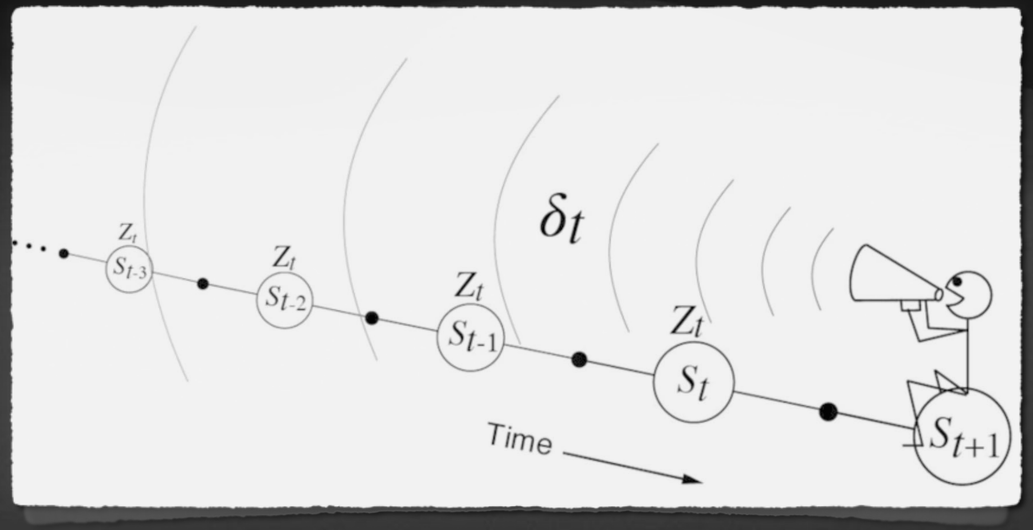
Thank You

Appendix

Incremental Prediction Algorithms

Backward view TD with approximation

1. Consider eligibility trace vector [E_t]
   - $E_t = \gamma\lambda E_{t-1} + \nabla_w \hat{q}(s_t, a_t, w)$
2. And TD error w . r . t approximate action-value function
   - $\delta_t = R_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)$
3. The update for network parameters [w] is the following
   - $\triangle w = \alpha\delta_t E_t$



Backward view TD