BIO-INSPIRED COMPUTING: APPLICATIONS AND INTERFACES

DEEP REINFORCEMENT LEARNING (DRL)

Slides created by Jo Plested

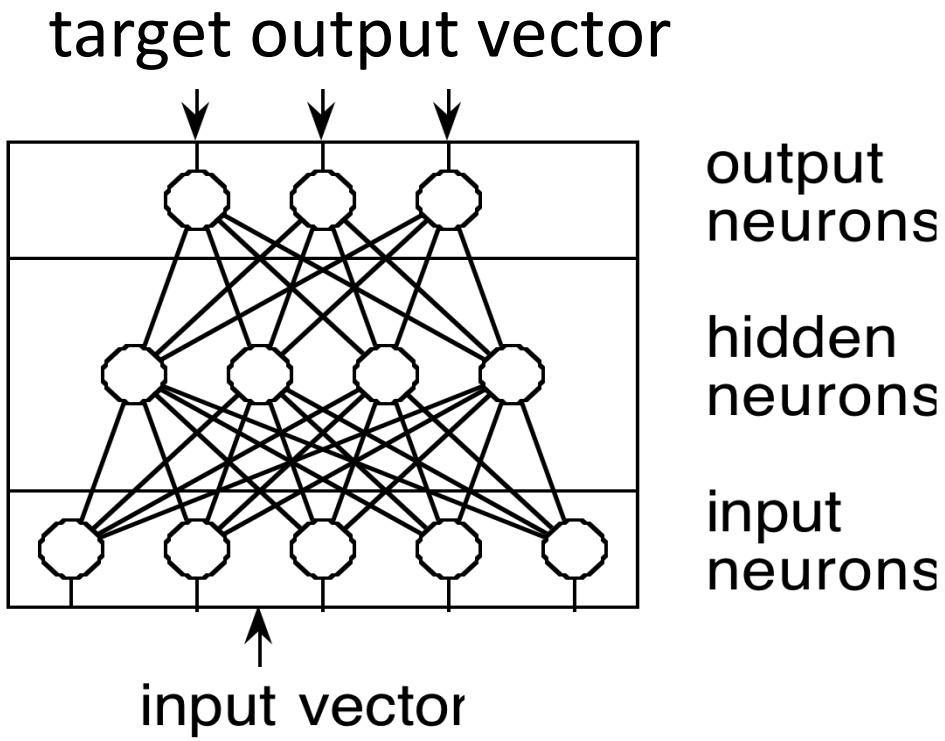
LECTURE OUTLINE

- Reinforcement Learning
  - What is it?
  - Q Learning
  - Policy Gradients

Recap - Varieties of learning algorithm

Supervised learning

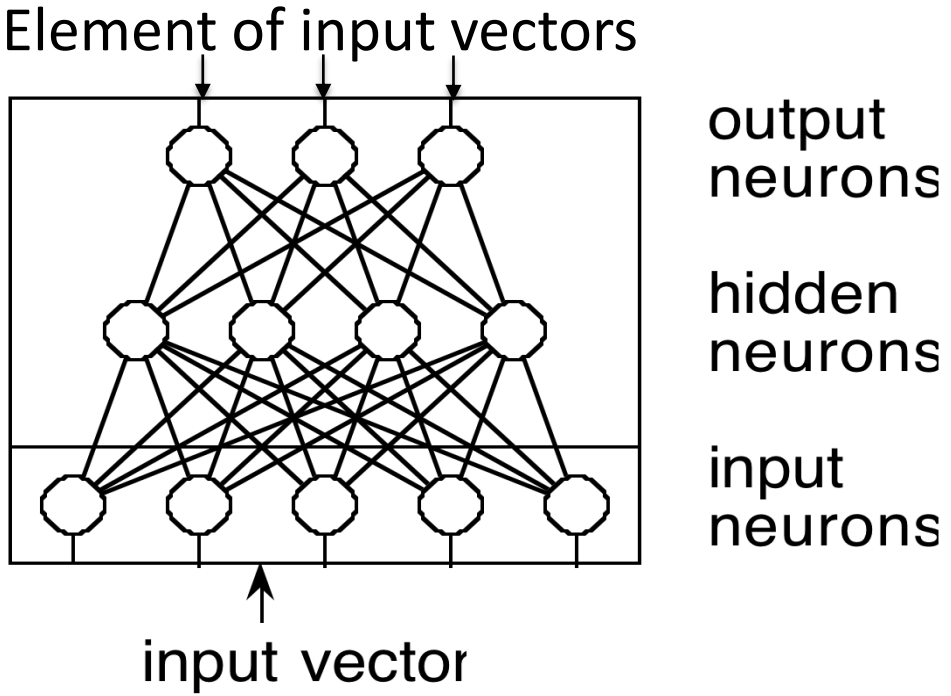
- Requires a 'teacher' who defines the target output vector for each input vector.



Recap - Varieties of learning algorithm

Unsupervised learning

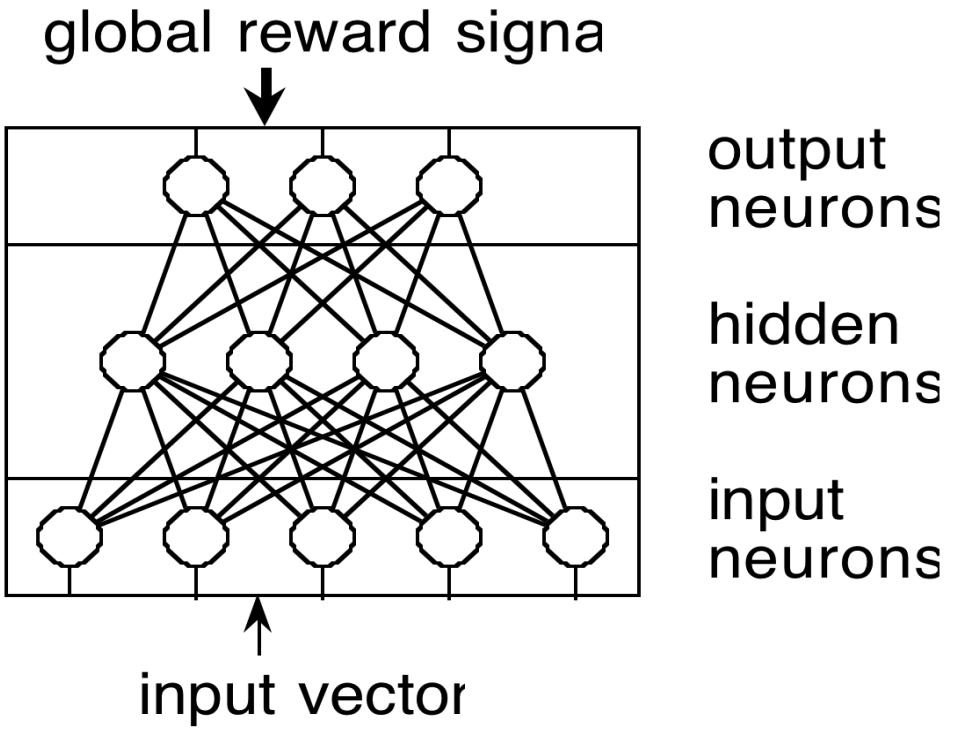
- The processing units model the higher-order statistical structure of the set of input vectors.
- The target output is some element of the set of input vectors, eg the next frame in a video.



Recap - Varieties of learning algorithm

Reinforcement learning

- Optimises sparse rewards
- Is very slow because many input presentations are required so it can assign credit correctly.



REINFORCEMENT LEARNING

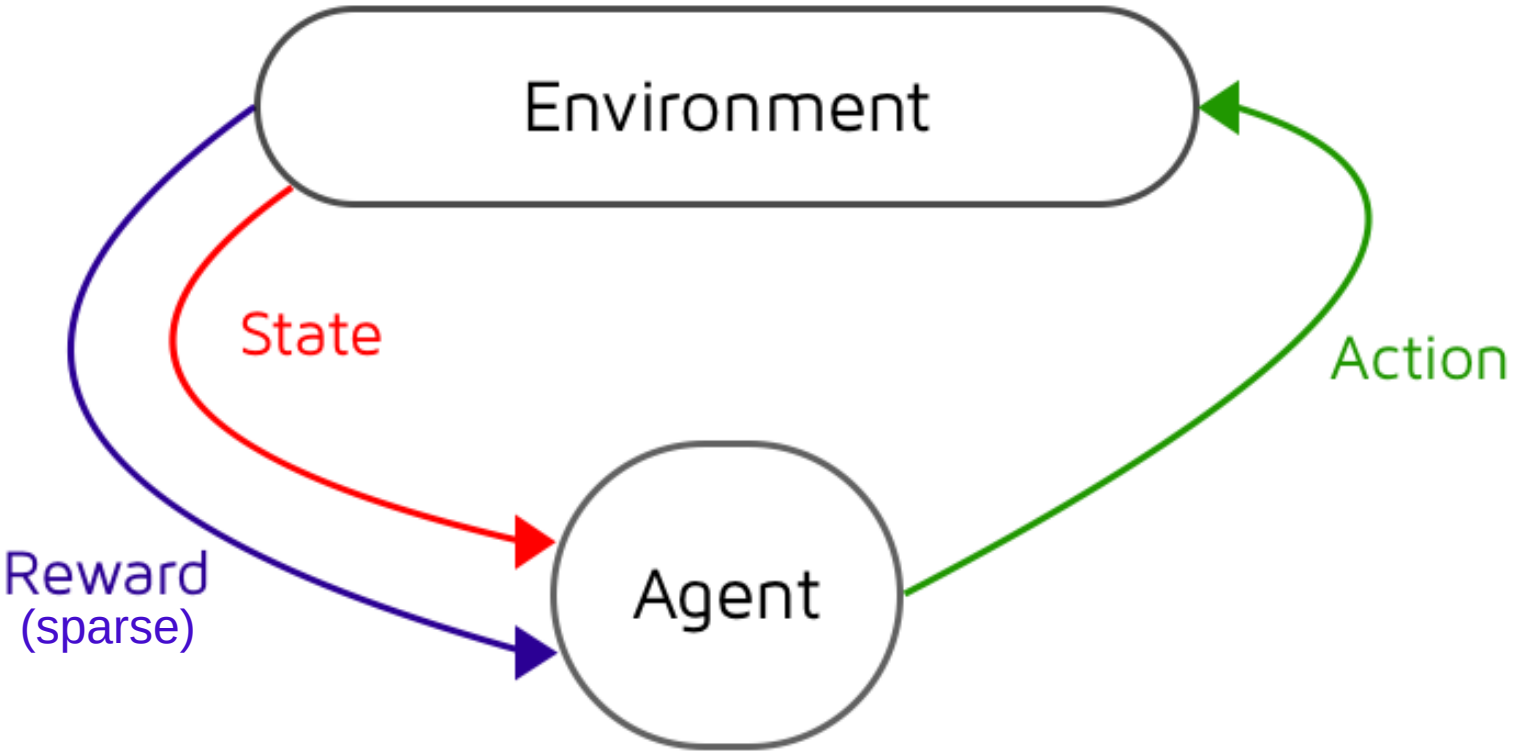
Reinforcement Learning introduction/refresher

REINFORCEMENT LEARNING

- Reinforcement learning learns to map state spaces to actions by maximizing the expected value of the given action

- Positive/negative rewards are given for pre-defined events and a discount factor γ, $0<\gamma<1$ is used to scale future rewards so immediate rewards are preferred/avoided more over those in the future

REINFORCEMENT LEARNING

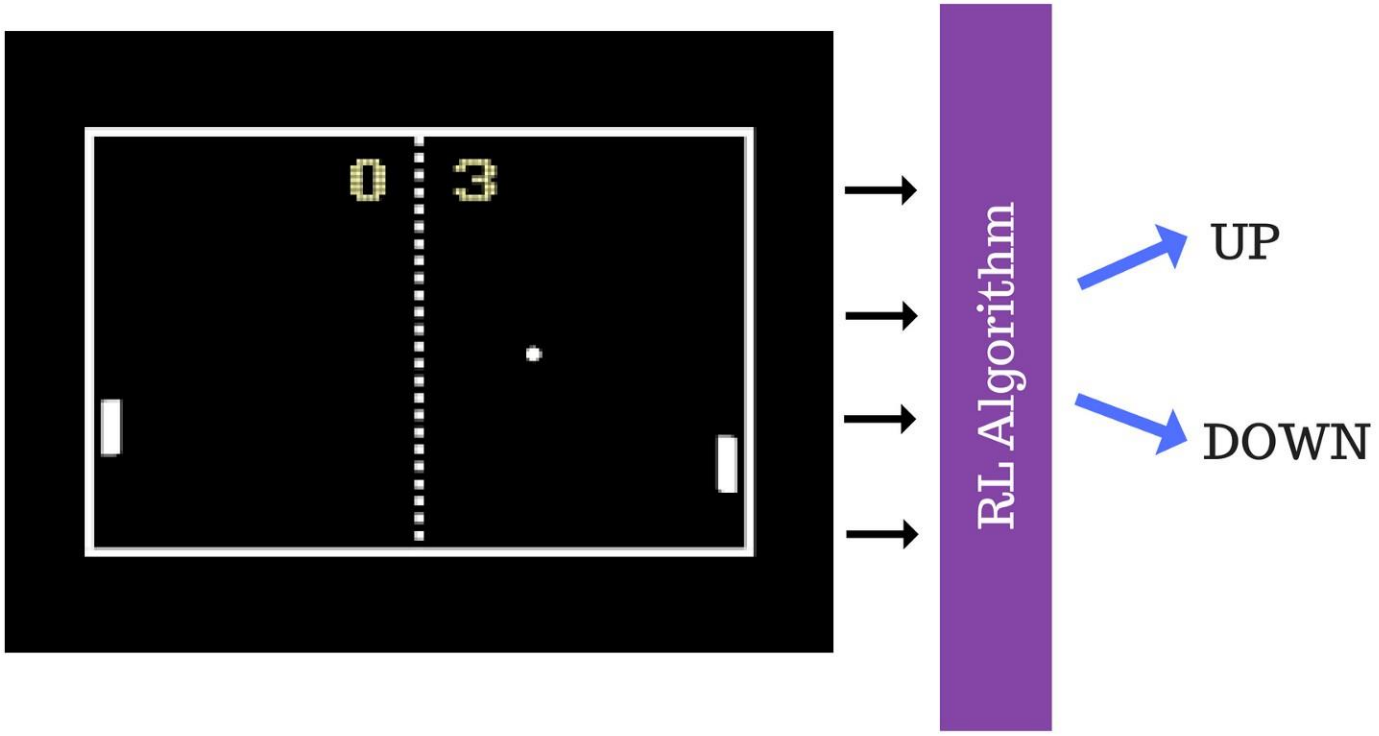


REINFORCEMENT LEARNING

- Reinforcement learning can be thought of as like supervised learning with sparse and/or uncertain labels.
  - For example: an agent may perform an action now that contributes to them getting a reward in another 50 time steps, but they might perform the same action in the same state another time and not get a reward due to the uncertainty of the environment
- Because of this sparsity and/or uncertainty and often the lack of any prior encoded knowledge of its environment, reinforcement learning often needs an extremely large number of episodes (training run throughs) to learn effectively

REINFORCEMENT LEARNING

For example: an agent may perform an action now that contributes to them getting a reward in another 50 time steps, but they might perform the same action in the same state another time and not get a reward due to the uncertainty of the environment



REINFORCEMENT LEARNING

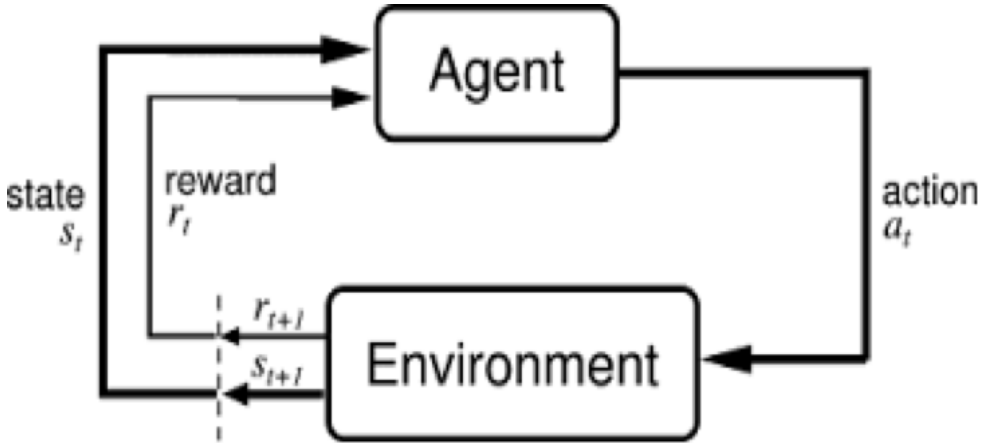
So reinforcement learning tends to excel at tasks where it can practice the same scenarios repeatedly or have lots of examples of scenarios to learn from

- Reinforcement learning has been shown to achieve above human expert level at a large number of game environments ranging from Atari games to board game like Chess and Go, to simulated driving environments

REINFORCEMENT LEARNING

- Reinforcement learning has also been used successfully to train robots to learn from their environment. Algorithms often train in a simulated physical environment before being placed in real physical robots.
  - For example Python has a training environment called gym which can create simulated physical environments to train algorithms
- Reinforcement learning algorithms also excel at optimisation

REINFORCEMENT LEARNING



Agent: An agent takes actions

Action (A): A is the set of all possible moves the agent can make

Environment: The world through which the agent moves. The environment takes the agent's current state and action as input, and returns as output the agent's reward and next state

State (S): A state is a concrete and immediate situation in which the agent finds itself

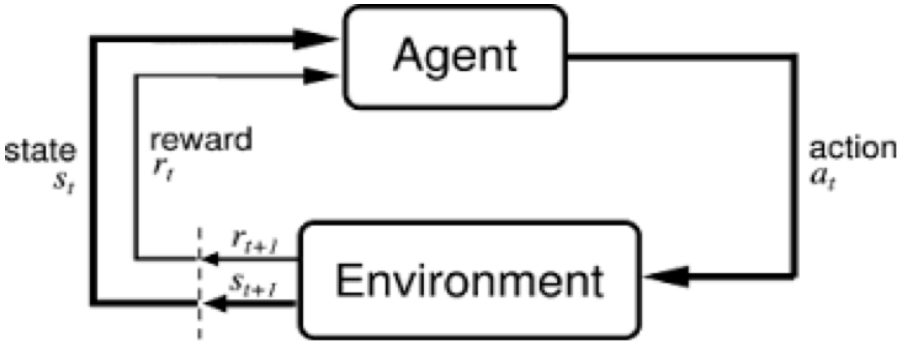
Reward (r): A reward is the feedback by which we measure the success or failure of an agent's actions and the training signal, they are usually sparse

Discount factor (γ): The discount factor is multiplied with future rewards in order to dampen their effect on the agent's choice of action. It makes future rewards worth less than immediate rewards

REINFORCEMENT LEARNING

- **Policy (π)**: The policy is the strategy the agent employs to determine the next action based on the current state. It maps states to actions
- **Value (V)**: The expected long-term return with discount, as opposed to the short-term reward r. $\mathbf{V_{\pi}(s)}$ is the expected long-term return of the current state under policy **π**
- **Q-value or action-value (Q)**: similar to Value, except that it takes an extra parameter, the current action a. $\mathbf{Q_{\pi}(s, a)}$ is the expected long-term return of the current state s, taking action a under policy π.

REINFORCEMENT LEARNING



So **environments** are functions that transform an action taken in the current state into the next state and a reward; **agents** are functions that transform the new state and reward into the next action. When we design a reinforcement learning model we assume that we can know the agent's function, but we cannot know the function of the environment. It is a black box where we only see the inputs and outputs. Reinforcement learning represents an agent's attempt to approximate the environment's function, so it can send actions into the black-box environment that maximize the rewards it spits out.

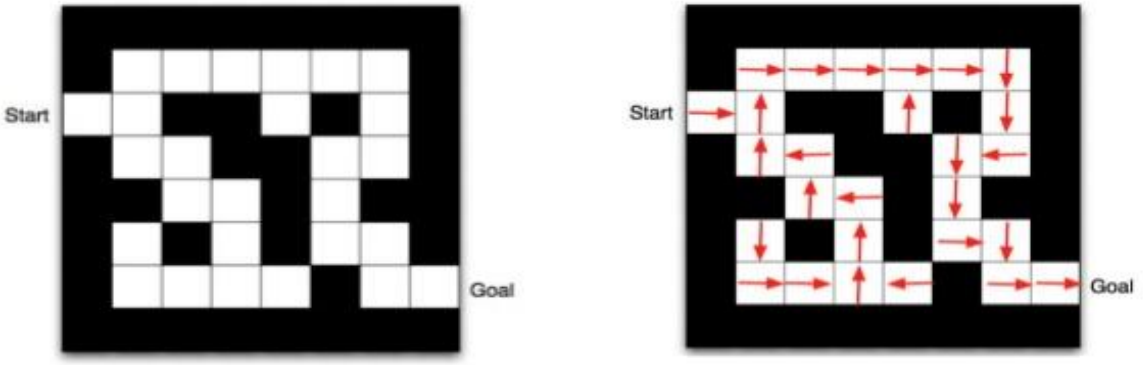
REINFORCEMENT LEARNING

- The return of following policy π from state s is:
  - $$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$$
  - Where r_t is the reward obtained after t steps starting from state s and following policy π thereafter

REINFORCEMENT LEARNING

An RL algorithm designed to solve a maze can either be trained in an environment that gives a reward of 1 when the agent finishes the maze or -1 for each time step. Either way after many training episodes the agent will learn to minimise the number of steps it takes to complete the maze.

Maze example and policy



[David Silver, Advanced Topics: RL]

REINFORCEMENT LEARNING

- The goal of reinforcement learning is to pick the best action (the action that results in the highest return) for any given state, which means the actions have to be ranked, and assigned values relative to one another
- Since those actions are state-dependent, what we are really gauging is the value of state-action pairs; i.e. an action taken from a certain state, something you did somewhere

REINFORCEMENT LEARNING

For example:

If the action is yelling “Fire!”, then performing the action in a crowded theatre should mean something different from performing the action next to a squad of people with rifles. We can’t predict an action’s outcome without knowing the context

REINFORCEMENT LEARNING

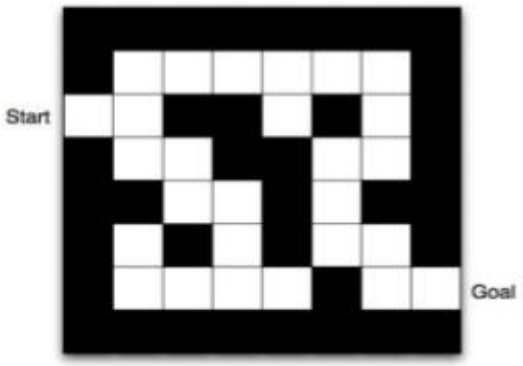
- To achieve the goal of picking the best action for a given state the model needs to learn what actions are most likely to result in the highest long term total reward
- Historically this was done by explicitly learning and storing for every single state a:
  - dynamics or reward model
  - value
  - state-action value
  - policy

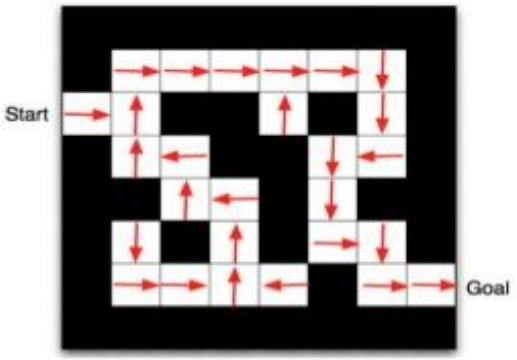
REINFORCEMENT LEARNING

Historically this was done by explicitly learning and storing for every single state a:

- dynamics or reward model
- value
- state-action value
- policy

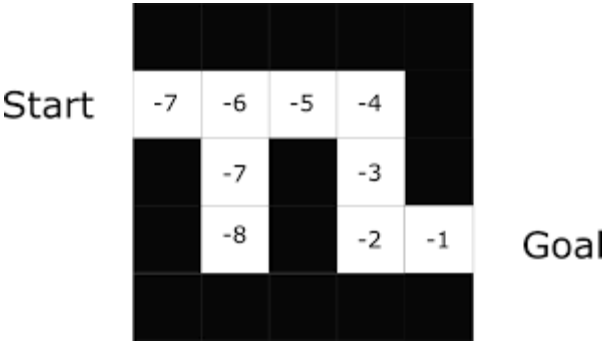
Maze example and policy





[David Silver, Advanced Topics: RL]

$V\pi^*(s)$



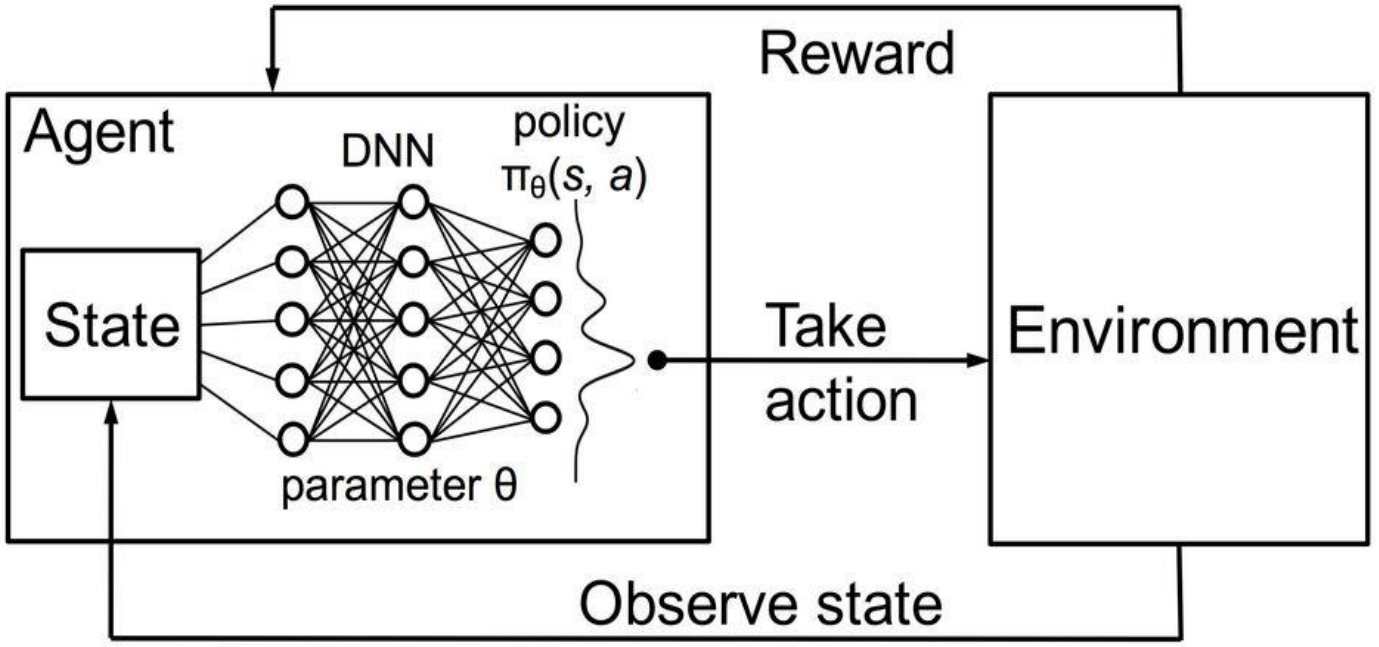
Value of each state under the optimal policy with a reward of -1 for each step and a discount factor of 1 (no discount).

REINFORCEMENT LEARNING

- This is not an efficient way of doing things when the state space gets large e.g. for image inputs
- A more compact representation that generalizes across states or states and action is needed
- Better to represent a (state-action/state) value function with a parameterized function instead of a table

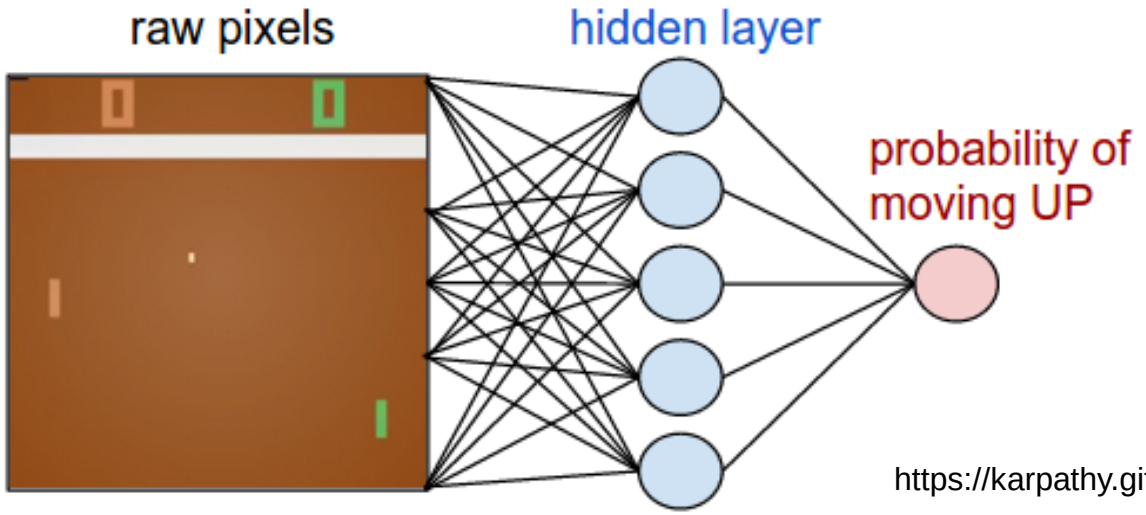
DEEP REINFORCEMENT LEARNING

- Complex RL problems can often be modelled well with a neural network. The type of neural network depends on the input and the problem to be solved



DEEP REINFORCEMENT LEARNING

- The game of pong is an excellent example of an RL task with a large state space
- The input is an image frame and the output is a move of the paddle, up or down



https://karpathy.github.io/assets/rl/policy.png

This is a very simple example. In reality because it's an image input a CNN would likely be used

DEEP REINFORCEMENT LEARNING

- It's essentially a two class image classification problem and the setup of the Neural Network itself doesn't change. It would likely be a CNN
- The difference comes in the loss function. Because it's not supervised learning we don't get a target value, i.e. we don't get any direct information about whether the chosen move was right or wrong
- We get an occasional reward, positive or negative. In this case a 1 if we win or a -1 if we lose

DEEP REINFORCEMENT LEARNING

Our aim is to train a policy that tries to maximize the total discounted, cumulative reward

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$$

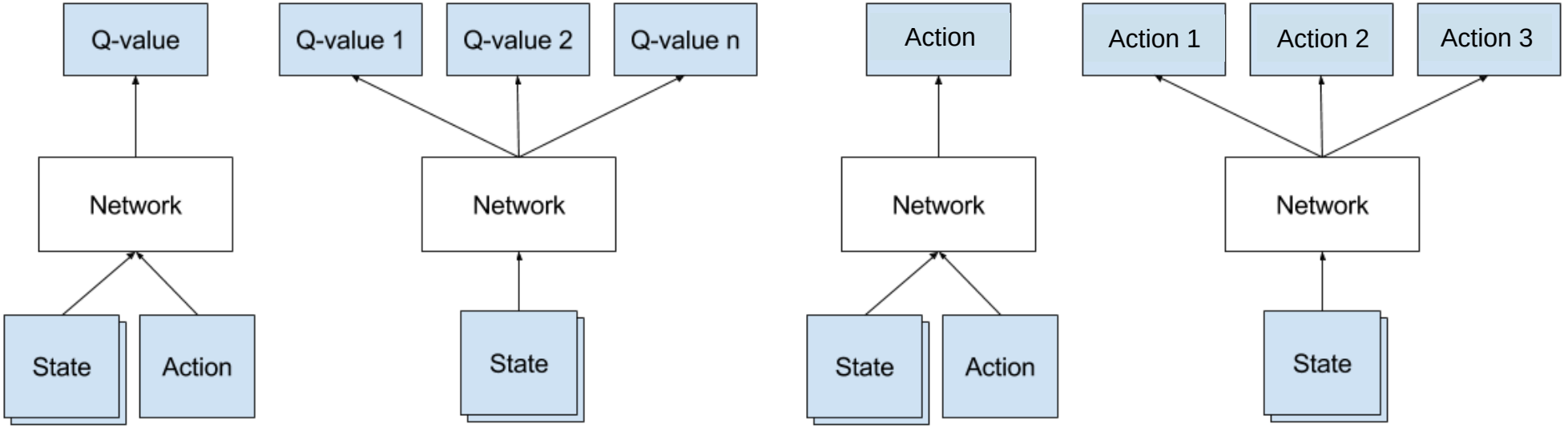
where R_{t_0} is known as the return. Note the difference between R_{t_0} the actual current value of future rewards and $V_{\Pi}(s)$ the expected value of future rewards

DEEP REINFORCEMENT LEARNING

- The two most popular general categories of reinforcement learning algorithms at the moment are Q Learning and Policy Gradients. There are also some hybrid methods.
- Q learning aims to approximate the Q function (the long term reward from taking action a in state s under policy π), while Policy Gradients is a method to directly optimize the policy in the action space

DEEP REINFORCEMENT LEARNING

- Q learning aims to approximate the Q function (the long term reward from taking action a in state s under policy π), while Policy Gradients is a method to directly optimize the policy in the action space



Q Learning

Policy Gradients

DEEP Q LEARNING

The main idea behind Q-learning is that if we knew the exact Q function Q^*:State×Action$\rightarrow\mathbb{R}$, so we knew exactly what our return would be for a given action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi^*(s) = \text{argmax}_a \, Q^*(s, a)$$

We don't have access to Q^*. But, since neural networks are universal function approximators, we can train one to resemble Q^*

DEEP Q LEARNING BELLMAN EQUATION

Returns at successive time steps are related to each other in a specific way:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \\ &= r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \cdots) \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

This is called the Bellman equation. Every value function for some policy obeys this equation, so can be unrolled recursively as above.

DEEP Q LEARNING BELLMAN EQUATION

Because it obeys the Bellman equation the Q value function can also be unrolled recursively:

$$Q^{\pi}(\text{s,a}) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s, a]$$
$$= \mathbb{E}\,[r_t + \gamma Q^{\pi}(\text{s}',\text{a}')|s, a]$$

where s′ is the state we ended up in after taking action a from state s, a′ is the action that was chosen in state s′ under policy π and r_t is the immediate reward received after taking action a from state s (this might be 0 or negative)

The optimal Q value function Q^*(s,a) can also be unrolled recursively

$$Q^*(\text{s,a}) = \mathbb{E}\,[r + \gamma\, max_{a'}\, Q^*(\text{s}',\text{a}')|s, a]$$

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf

DEEP Q LEARNING BELLMAN EQUATION

Value iteration algorithms solve the Bellman equation

If we consider the deterministic case:

- Under an optimal policy:

$$Q^*(s,a) = r + \gamma\, Q^*(s', \pi^*(s'))$$

- When the current policy is not optimal the difference between the two sides of the equality is known as a temporal difference (TD) error, δ :

$$\delta = Q(s,a) - (r + \gamma\, max_{a'} Q(s', a'))$$

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf

DEEP Q LEARNING BELLMAN EQUATION

$$\delta = Q(s, a) - (r + \gamma \, max_{a'} Q(s', a'))$$

Term one in this equation is our current estimate of the value of taking action a in state s and term two is the reward we received after taking action a from s plus the discounted estimate of the value of taking the action that maximises the expected future rewards from state s'

DEEP Q LEARNING BELLMAN EQUATION

$$\delta = Q(s,a) - (r + \gamma \, max_{a'} Q(s',a'))$$

The second term can differ from the first term in two ways:

1. The expected immediate reward is different to the actual immediate reward we receive for taking action a from state s
2. We end up in a different state s' than expected.

In the maze example the first difference would be if we unexpectedly found the end and the second example would be if we unexpectedly found a shortcut that got us closer to the end than expected

DEEP Q LEARNING BELLMAN EQUATION

$$\delta = Q(s,a) - (r + \gamma \, max_{a'} Q(s',a'))$$

For a comparison with supervised learning δ can be thought of as the difference between the predicted value (term 1) and actual value (term 2) for one transition so

$$(r + \gamma \, max_{a'} Q(s',a'))$$

is sometimes referred to as the target value.

DEEP Q LEARNING

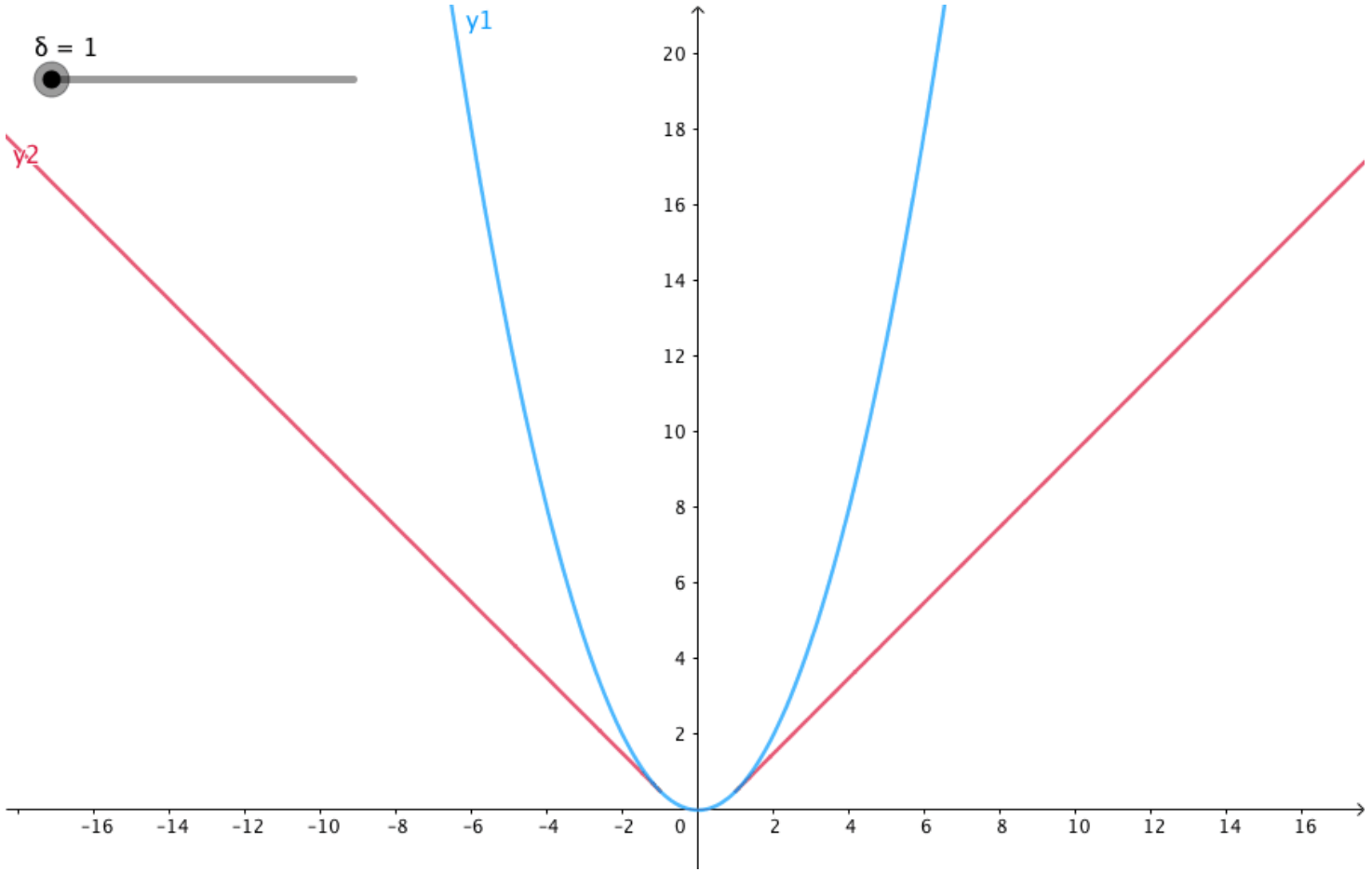
$$\delta = Q(s, a) - (r + \gamma \, max_{a'} Q(s', a'))$$

Now that we have an error value we can minimise it using a loss function. We could use a MSE loss, but in practice the Huber loss is often used in Deep Q Learning (DQN)

The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy

http://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

DEEP Q LEARNING



y1 is the squared error and y2 is the Huber Loss

DEEP Q LEARNING

We calculate the Huber loss over a batch of transitions, B:

$$L = \frac{1}{|B|} \sum_{(s,a,s',a') \epsilon B} L(\delta)$$

where $$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 \; for \; |\delta| \leq 1 \\ |\delta| - \frac{1}{2} \; otherwise \end{cases}$$

DEEP Q LEARNING

The Huber loss is called SmoothL1Loss in Pytorch

Now that we have the loss function we can backpropagate the gradients and update the weights in the usual way

In Pytorch
loss.backward()
and
optimizer.step()

DEEP Q LEARNING TIPS AND TRICKS

Exploration vs exploitation:

- The trade off between taking the current estimated optimal action and exploring new actions which could lead to better rewards is called exploration vs exploitation
- In the commonly used epsilon greedy policy, each iteration, with a probability of epsilon, the agent takes a random action as opposed to the action with the current highest estimated future rewards. This forces the agent to explore more of the state space early in training. As training progresses epsilon is slowly reduced to some predefined minimum in order to favour exploitation over exploration

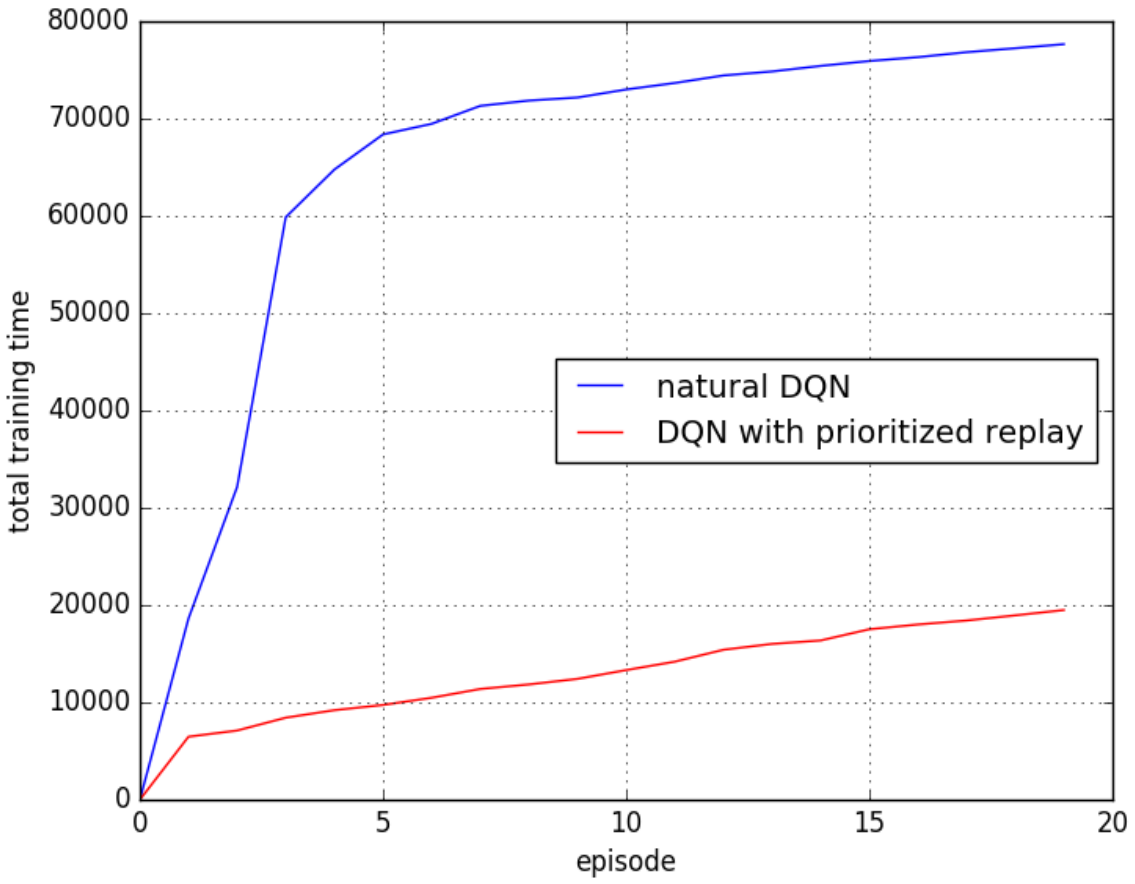
DEEP Q LEARNING TIPS AND TRICKS

Replay Memory:

- Rather than learning from transitions in the order that they appear when taking actions, replay memory stores the transitions that the agent observes then samples from them randomly
- That way the transitions that build up a training batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure

DEEP Q LEARNING TIPS AND TRICKS

Replay Memory:



Prioritized Replay:
Extends DQN's experience replay function by learning to replay memories where the real reward significantly diverges from the expected reward, letting the agent adjust itself in response to developing incorrect assumptions. Makes training time and stability significantly faster

DEEP Q LEARNING TIPS AND TRICKS

- A separate target neural network is generally used to compute $max_{a'} Q(s', a') = V(s_{t+1})$ for added stability
- This target neural network has its weights frozen most of the time but the policy network's weights are copied across to it after a set number of transitions or episodes
- An episode is a full set of transitions from a starting state to a terminal state. For example from the start of a maze to the end of a maze for the maze learning problem

DEEP Q LEARNING TIPS AND TRICKS

Double Q Learning: The max operator in the error function in standard DQN, uses the same network model both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. This is clearly shown if we expand out the target value:

$$(r + \gamma\, max_{a'} Q(s', a')) = (r + \gamma\, Q(s', argmax_{a'}\, Q(s', a')))$$

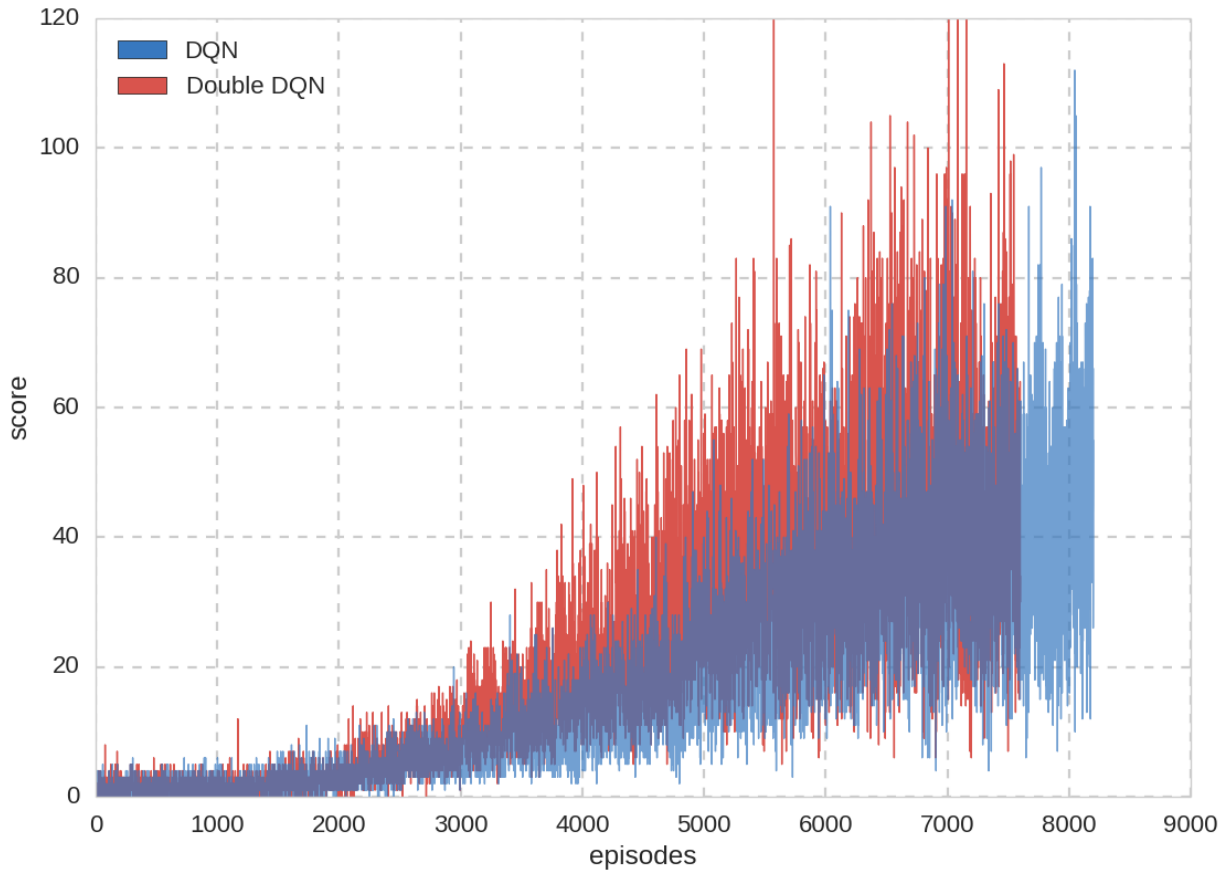
In Double Q Learning a second Q network is learned to fairly evaluate the current greedy policy:

$$(r + \gamma\, Q_1(s', argmax_{a'}\, Q_2(s', a')))$$

The role of Q_1 and Q_2 is swapped each iteration

http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847

DEEP Q LEARNING TIPS AND TRICKS

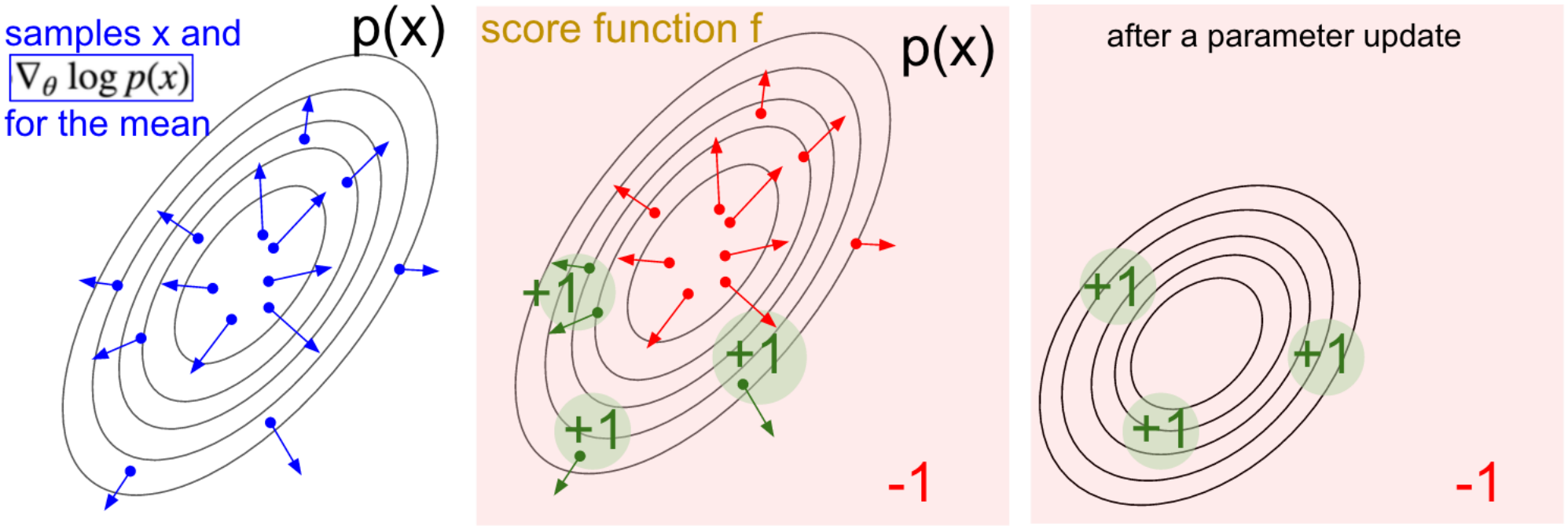


POLICY GRADIENTS

While the goal of Q Learning is to approximate the Q function and use it to infer the optimal policy π* (i.e. $argmax_a Q(s,a)$), Policy Gradients (PG) seeks to directly optimize in the policy space

https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS



https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS

- In Policy Gradients, we usually use a neural network (or other function approximators) to directly model the action probabilities
- Each time the agent interacts with the environment (hence generating a data point <s,a,r,s′>), we tweak the weights of the neural network so that “good” actions will be sampled more likely in the future
- We repeat this process until the policy network converges to the optimal policy π*

https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html

POLICY GRADIENTS

So as per Q Learning, the overall goal is to maximise the expected return

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$$

This is a special case of the generalised formula:

$$\mathbb{E}_{x \sim p(x|\theta)} [f(x)]$$

The expectation of some scalar valued score function f(x) (the return function) under some probability distribution p(x;θ) parameterized by θ

We are interested in calculating how we should shift the distribution (through its parameters θ) to increase the scores of its samples, as judged by f (i.e. how to change network's parameters so that action samples get higher rewards)

POLICY GRADIENTS

Deriving Policy gradients (not examinable):

$$\nabla_\theta \mathbb{E}_{x\sim p(x|\theta)}[f(x)]$$

$$= \nabla_\theta \sum_x p(x|\theta) f(x)$$ definition of expectation

$$= \sum_x f(x) \nabla_\theta p(x|\theta)$$ swap sum and gradient

$$= \sum_x p(x|\theta) f(x) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}$$ both multiply and divide by p(x)

$$= \sum_x p(x|\theta) f(x) \nabla_\theta \log p(x|\theta)$$ use $\nabla \log(z) = \frac{1}{z} \nabla z$ (chain rule of differentiation)

$$= \mathbb{E}_{x\sim p(x|\theta)}[f(x) \nabla_\theta \log p(x|\theta)]$$ definition of expectation

https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS

$$\mathbb{E}_{x\sim p(x|\theta)}[f(x)\nabla_\theta \log p(x|\theta)]$$

In English, the term $\nabla_\theta \log p(x|\theta)$ gives us the direction we would need to move the parameters θ to increase the probability of action x and $f(x)$ is the score of x (the return)

So if the score (return) is positive this will move the parameters in the direction to make x more likely and if it's negative it will move the parameters in the opposite direction

POLICY GRADIENTS PROCEDURE

So if we go back to the pong game example:

- The neural network takes in an image and outputs log probabilities of the moves up and down (probabilities once they've been through a softmax function)
- In supervised learning we know the correct answer straight away and can calculate how far off these answers were so our objective is to maximise $\sum_i \log p(y_i|x_i)$, the sum of the log probabilities of outputting the correct label given each input

https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS PROCEDURE

In Reinforcement learning we don't have the labels, so instead we assume the actions we took were correct at the time, then scale them by eventual outcomes

Our objective becomes to maximise $\sum_i R_i \log p(y_i|x_i)$ which in reinforcement learning terms is:

$$\sum_i R_i \log \pi(a|s,\theta)$$

and the loss we want to minimise is

$$\sum_i -R_i \log \pi(a|s,\theta)$$

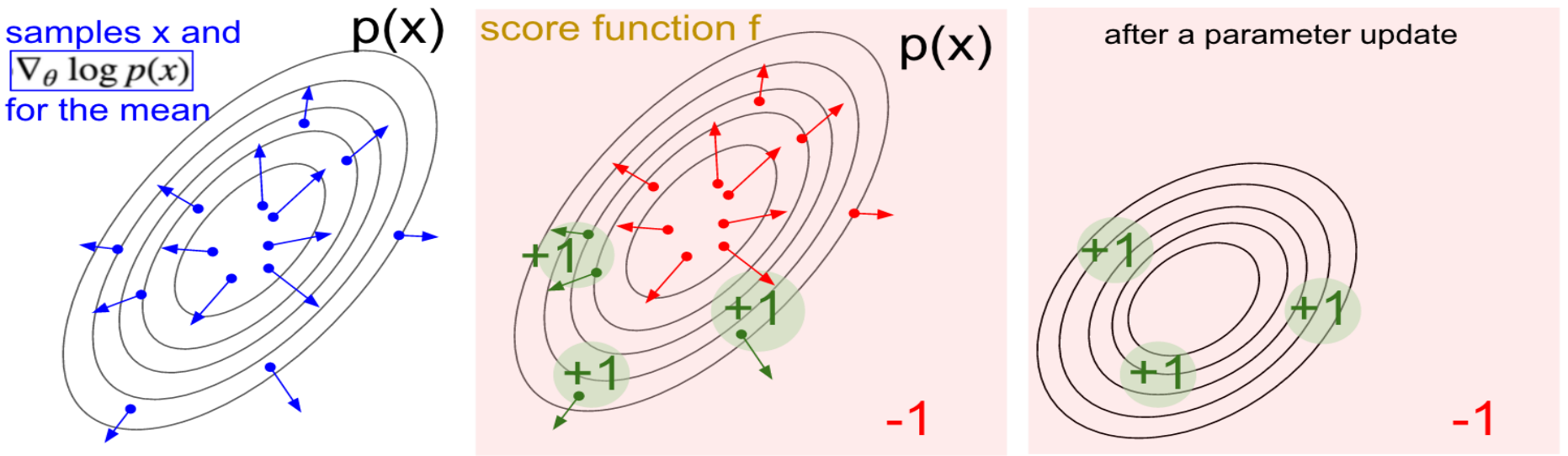
From there we can just call backwards() and optimizer.step() as usual

POLICY GRADIENTS PROCEDURE

Example:

- If we take the simplest case for Pong where we get 1 if we win and -1 if we lose and we have no discount factor, every move that led to a win gets multiplied by 1 and every move that led to a loss gets multiplied by a -1
- This may seem counterintuitive as you could eventually lose after a good move or win after a bad move, but after many, many repetitions the good moves will become more likely and the bad moves less likely

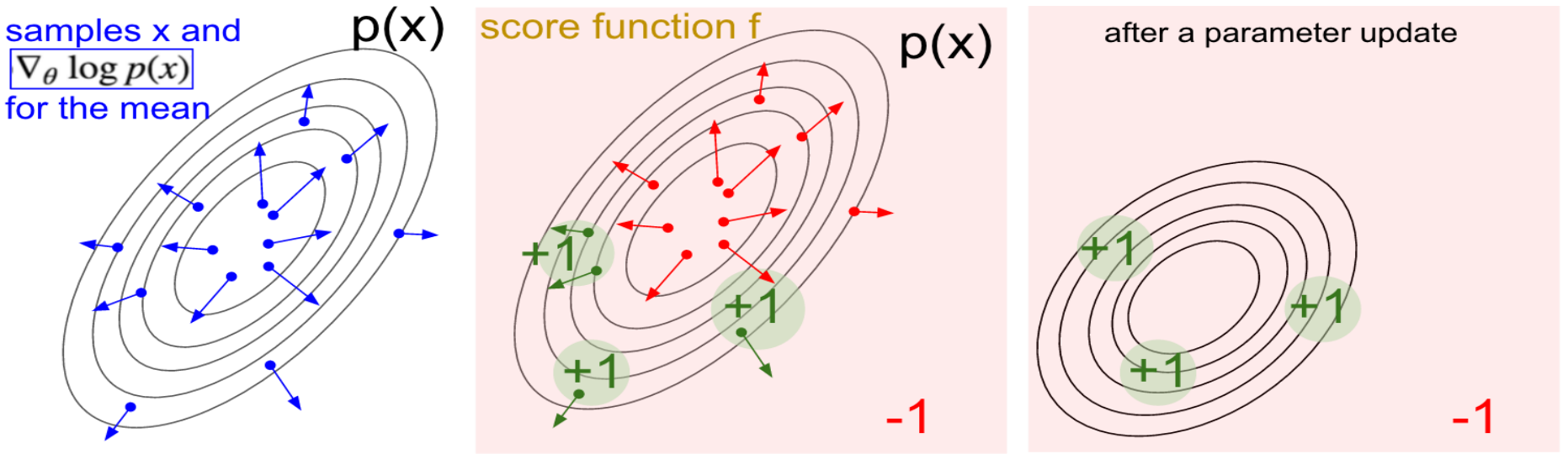
POLICY GRADIENTS PROCEDURE



A visualization of the score function gradient estimator. Left: A gaussian distribution and a few samples from it (blue dots). On each blue dot we also plot the gradient of the log probability with respect to the gaussian's mean parameter. The arrow indicates the direction in which the mean of the distribution should be nudged to increase the probability of that sample. Middle: Overlay of some score function giving -1 everywhere except +1 in some small regions (note this can be an arbitrary and not necessarily differentiable scalar-valued function). The arrows are now colour coded because due to the multiplication in the update we are going to average up all the green arrows, and the negative of the red arrows. Right: after parameter update, the green arrows and the reversed red arrows nudge us to the left and towards the bottom. Samples from this distribution will now have a higher expected score, as desired.

https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS PROCEDURE



Note: $\nabla_\theta \log p(x|\theta) = \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}$ is intuitive as the gradient of the probability of taking the action scaled by the probability of taking the action and tells us what direction to move the parameters in to make that action more likely. This value is then scaled by the return on that action

https://karpathy.github.io/2016/05/31/rl/

POLICY GRADIENTS VS Q LEARNING

The main difference between Q Learning (DQN) and Policy Gradients is DQN estimates the value of taking a particular action in a particular state, whereas Policy Gradients is outputting a distribution over actions which gives the probability that a particular action in a particular state is the best action.

In Policy Gradients the policy can approach deterministic, whereas in DQN if action selection is ε-greedy there is always some probability that a random action will be taken. Even if a softmax over action values is taken the action values will converge on their true values which differ by a finite amount resulting in probabilities between 0 and 1

POLICY GRADIENTS VS Q LEARNING

Advantages of Policy-Based RL

- Better convergence properties. In DQN an arbitrarily small change in action values can result in a large change in policies. Convergence can depend on the specific problem though. Problems vary in the complexity of their policies and action-value functions. If the action-value function is significantly simpler DQN will be a better choice
- Effective in high-dimensional or continuous action spaces (without computing max)

POLICY GRADIENTS VS Q LEARNING

Advantages of Policy-Based RL …

Can learn stochastic policies - in problems with significant function approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker

POLICY GRADIENTS VS Q LEARNING

Disadvantage of Policy Gradients:

High variance in estimating the gradient of E[Rt]:

- Essentially, each time we perform a gradient update, we are using an estimation of gradient generated by a series of data points <s,a,r,s′> accumulated through a single episode of game play. This is known as Monte Carlo method
- The estimation can be very noisy, and a bad gradient estimate could adversely impact the stability of the learning algorithm

POLICY GRADIENTS VS Q LEARNING

When Q Learning does work, it often shows a better sample efficiency and more stable performance

There have been a number of recent solutions to this trade off between policy based methods and Q Learning

Policy Gradients with baseline:

A simple way to reduce the variance in Policy Gradients is to subtract a baseline from the reward in the Policy Gradients formula

$$\nabla_\theta \mathbb{E}[R_t] = \mathbb{E}[\,(R_t - B)\nabla_\theta \log \pi(a|s,\theta)]$$

POLICY GRADIENTS WITH BASELINE

$$\nabla_\theta \mathbb{E}[R_t] = \mathbb{E}[\,(R_t - B)\nabla_\theta \log \pi(a|s,\theta)]$$

- Assume R_t is always positive, with the absence of the baseline B, P(a) will always get pushed up even if a is a bad action. This causes the gradient estimator to have high variance since even bad actions get positive updates
- Subtracting a proper baseline ensures that the scaling factor (R_t −B) is only positive if the action is good and negative if the action is bad. Empirically, this trick reduces the variance of the gradient estimate quite drastically

POLICY GRADIENTS WITH BASELINE

$$\nabla_\theta \mathbb{E}[R_t] = \mathbb{E}[\,(R_t - B)\nabla_\theta \log \pi(a|s,\theta)]$$

- The baseline can be any function, even a random variable.
- The baseline should vary with state. In some states all actions have high values and we need a high baseline to only have positive updates for the better actions; in other states all actions will have low values and a low baseline is appropriate

POLICY GRADIENTS WITH BASELINE

$$\nabla_\theta \mathbb{E}[R_t] = \mathbb{E}[\,(R_t - B)\nabla_\theta \log \pi(a|s,\theta)]$$

- So the loss function with a baseline is now:

$$\sum_i (R_i - b(s))(-\log \pi(a|s,\theta))$$

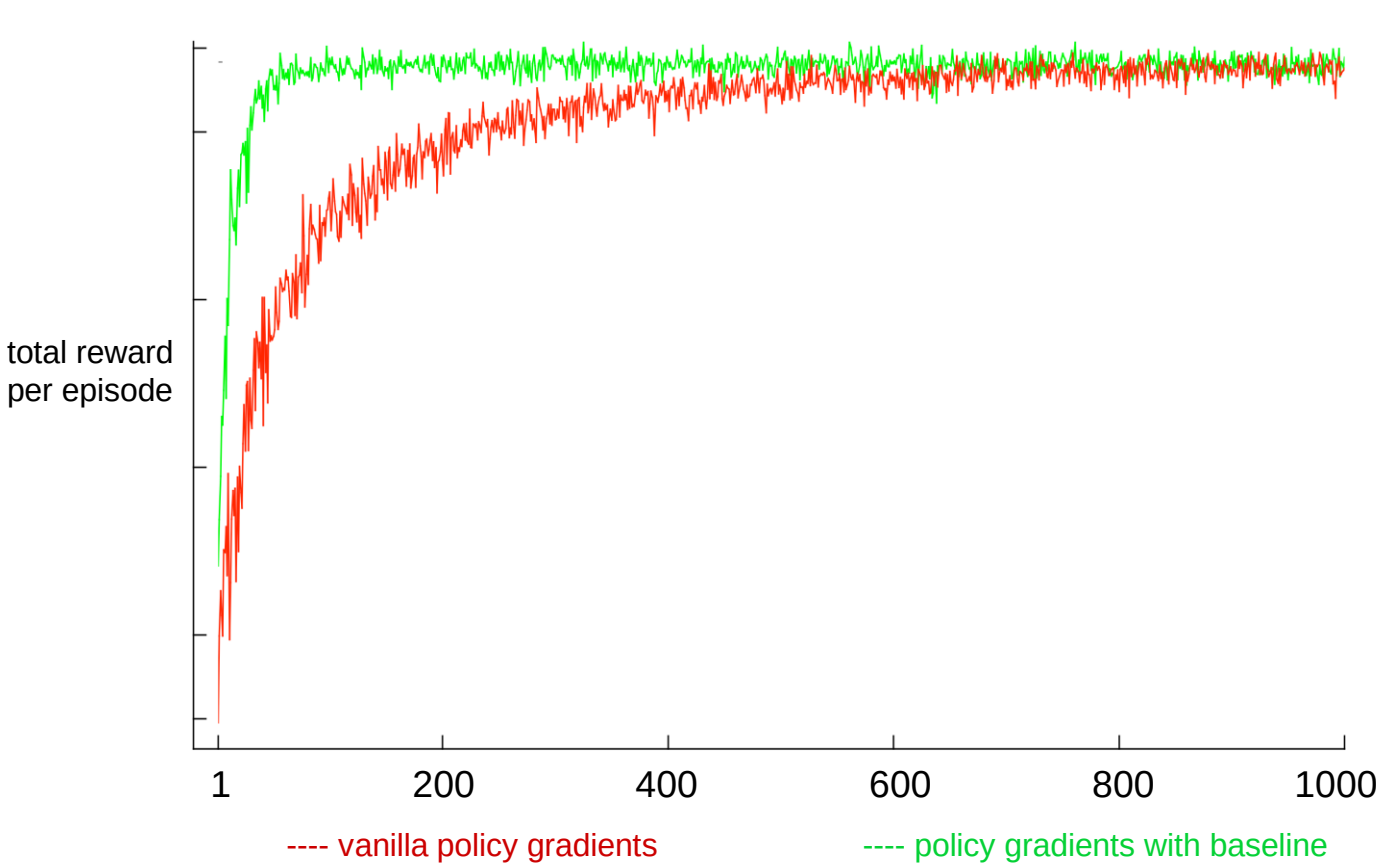
- This is just a generalisation of the standard policy gradients algorithm as b(s) could be 0

POLICY GRADIENTS WITH BASELINE

One natural and common choice for the baseline is an estimate of the state value, $\hat{v}(s, \phi)$. The value function is the expected value of total future rewards in state s. So $(R_t - B)$ is telling us how good the action is compared to the average. As a result, only better than average actions can get positive updates. Hence the term $(R_t - B)$ is often called the Advantage function

POLICY GRADIENTS WITH BASELINE

Policy Gradients with baseline



Adding a baseline to Policy Gradients can make it learn much faster, as illustrated here. The step size used here for plain Policy Gradients is that at which it performs best. Each line is an average over 100 independent runs

http://incompleteideas.net/book/the-book-2nd.html

ACTOR-CRITIC METHODS

- Another problem with Policy Gradients methods is that they are difficult to implement for continuing problems. Problems that don't have episodes. To overcome this and to reduce variance and accelerate learning Actor-Critic methods are used.
- Actor Critic methods reintroduce a temporal difference error over one transition to replace $(R_i - b(s))$in the Policy Gradients with Baseline algorithm:

$$\nabla_\theta \, \mathbb{E}[R_t] = \mathbb{E}[\, (r_t + \gamma v(s', \phi) - v(s, \phi)) \log \pi(a|s, \theta)]$$

Where s' is the next state after taking action a in state s and v is a separate value estimation function with parameters ϕ to estimate the expected value of a state.

ACTOR-CRITIC METHODS

$$\nabla_\theta \, \mathbb{E}[R_t] = \mathbb{E}[\,(r_t + \gamma v(s', \phi) - v(s, \phi)) \nabla_\theta \log \pi(a|s, \theta)]$$

Like with Q-Learning the $v(s, \phi)$ is the predicted value and $(r_t + \gamma v(s', \phi))$ is the target value over one transition (the actual reward and the discounted expected value after the transition). v is a separate function (neural network) whose loss is just the temporal difference.

The loss for the policy function is now:

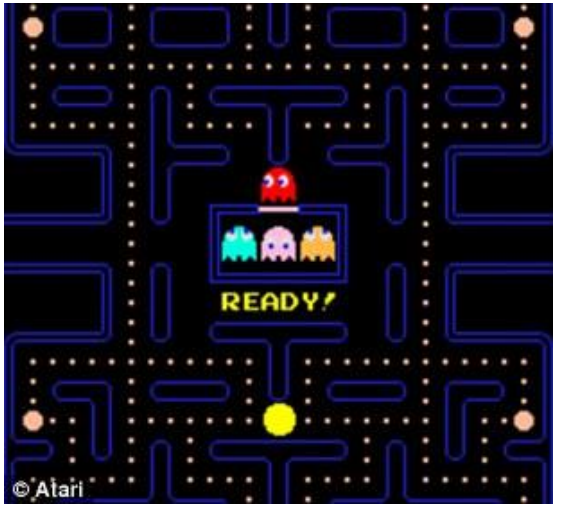
$$\sum_i (r_t + \gamma v(s', \phi) - v(s, \phi))(-\log \pi(a|s, \theta))$$

Replacing the return of an entire episode with an estimate of the loss over one time step means updates can be done online after each transition and variance is reduced.

RECENT SUCCESSES IN REINFORCEMENT LEARNING

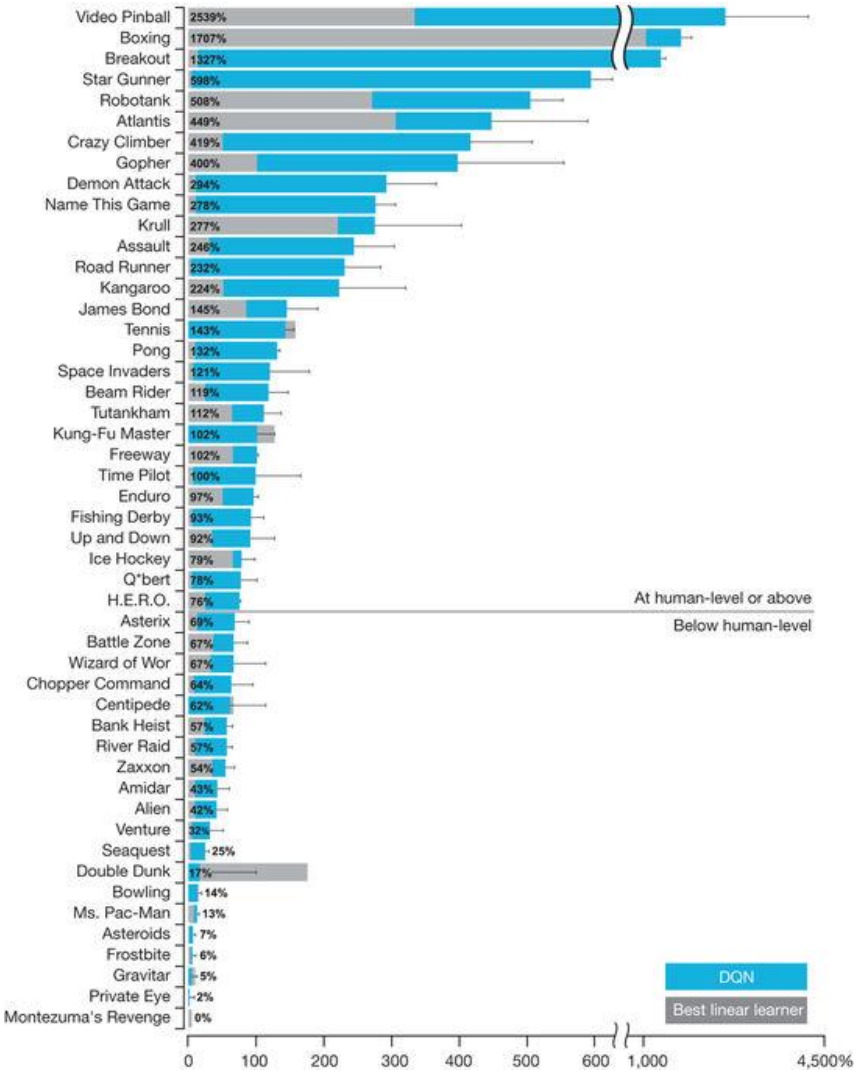
ATARI using deep Q-learning:
https://arxiv.org/pdf/1312.5602.pdf

With Policy Gradients:
www.jmlr.org/proceedings/papers/v37/schulman15.pdf
www.jmlr.org/proceedings/papers/v48/mniha16.pdf



RECENT SUCCESSES IN REINFORCEMENT LEARNING

ATARI using deep Q-learning: www.davidqiu.com:8888/research/nature14236.pdf



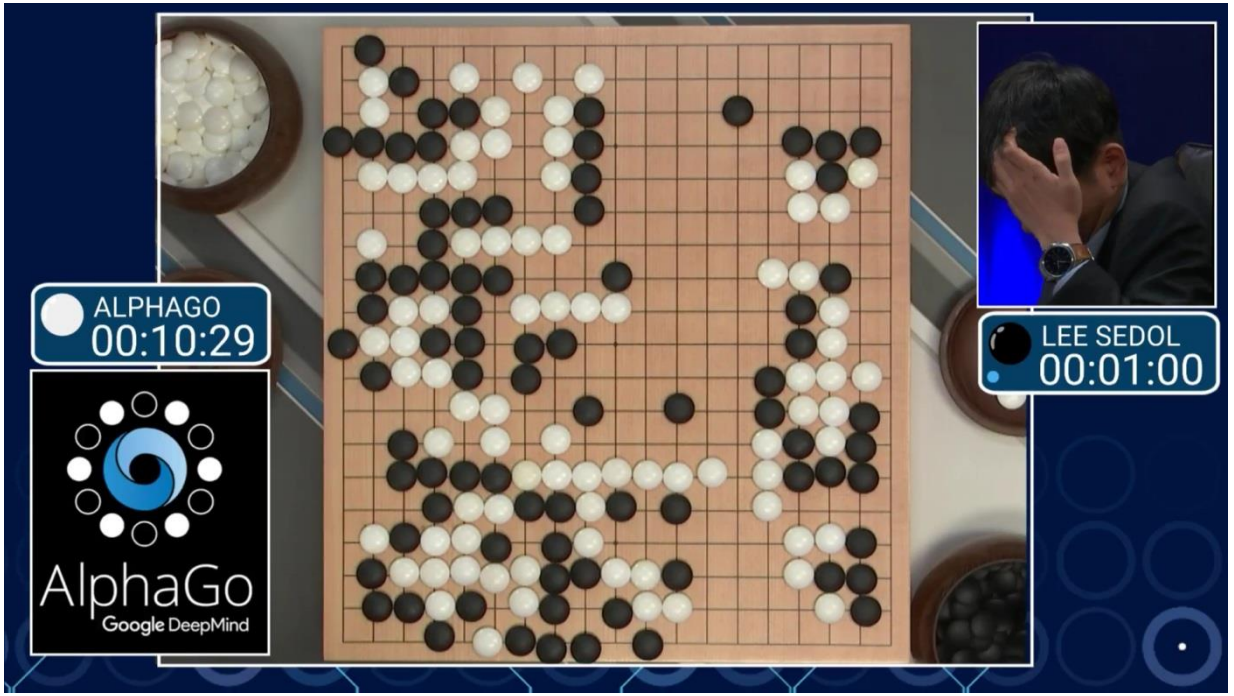
RECENT SUCCESSES IN REINFORCEMENT LEARNING

Robotic locomotion using policy gradients:
https://arxiv.org/pdf/1506.02438
https://www.youtube.com/watch?v=SHLuf2ZBQSw

https://arxiv.org/abs/1707.02286
https://www.youtube.com/watch?v=g59nSURxYgk

RECENT SUCCESSES IN REINFORCEMENT LEARNING

AlphaGo: supervised learning + policy gradients + value functions + Monte-Carlo tree search: https://pdfs.semanticscholar.org/1740/eb993cc8ca81f1e46ddaadce1f917e8000b5.pdf



RECENT SUCCESSES IN REINFORCEMENT LEARNING

Using Policy Gradient reinforcement learning to optimise the design of a neural network.
Neural architecture search with reinforcement learning: https://arxiv.org/pdf/1611.01578