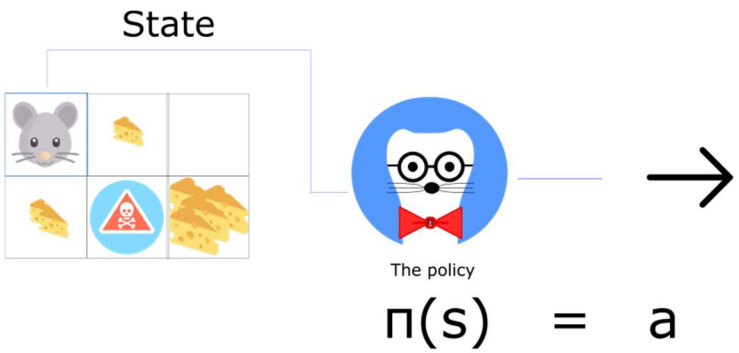
# Why using Policy-Based methods?

**Two types of policy**

A policy can be either deterministic or stochastic.

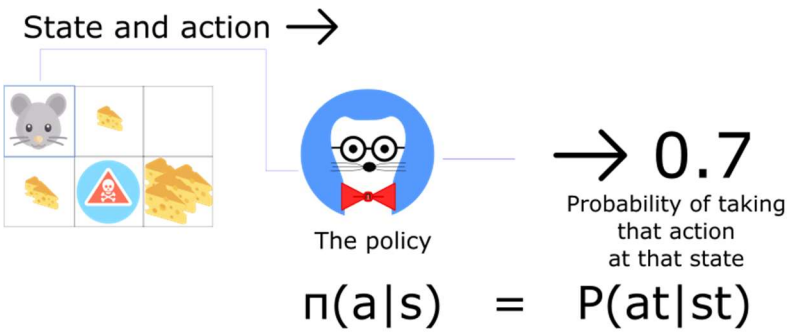
A deterministic policy is policy that maps state to actions. You give it a state and the function returns an action to take.




*Deterministic policy*

Deterministic policies are used in deterministic environments. These are environments where the actions taken determine the outcome. There is no uncertainty. For instance, when you play chess and you move your pawn from A2 to A3, you're sure that your pawn will move to A3.

On the other hand, a stochastic policy outputs a probability distribution over actions.




*Stochastic policy*

It means that instead of being sure of taking action *a* (for instance left), there is a probability we'll take a different one (in this case 30% that we take south).

The stochastic policy is used when the environment is uncertain. We call this process a Partially Observable Markov Decision Process (POMDP).

Most of the time we'll use this second type of policy.

**Advantages**

*But Deep Q Learning is really great! Why using policy-based reinforcement learning methods?*

There are three main advantages in using Policy Gradients.

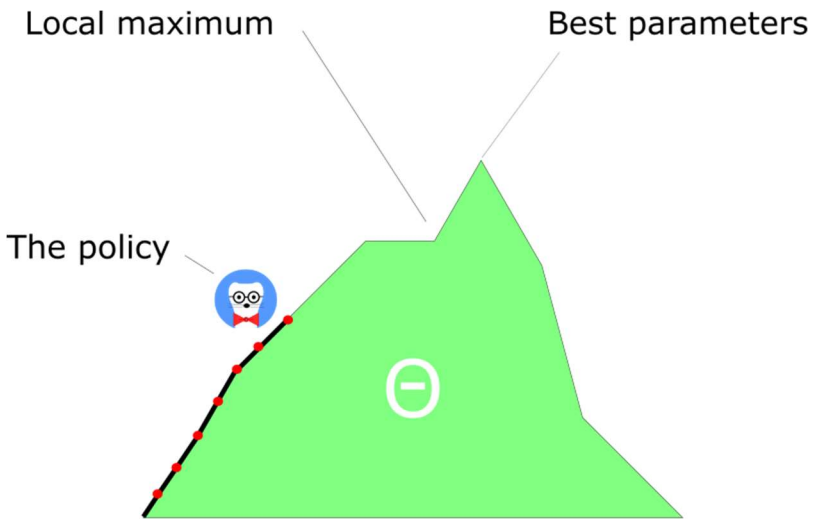
**Convergence**

For one, policy-based methods have better convergence properties.

The problem with value-based methods is that they can have a big oscillation while training. This is because the choice of action may change dramatically for an arbitrarily small change in the estimated action values.

On the other hand, with policy gradient, we just follow the gradient to find the best parameters. We see a smooth update of our policy at each step.

Because we follow the gradient to find the best parameters, we're guaranteed to converge on a local maximum (worst case) or global maximum (best case).




**Policy gradients are more effective in high dimensional action spaces**

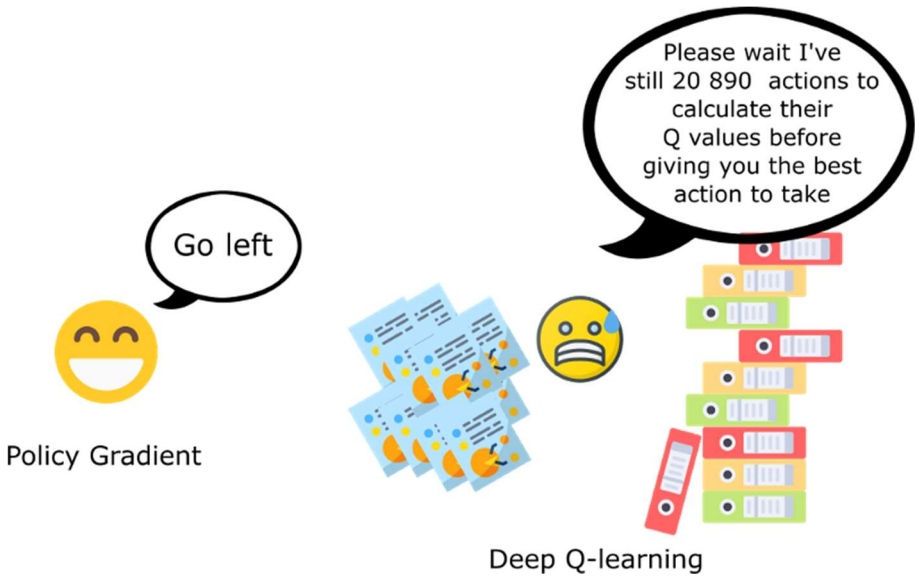
The second advantage is that policy gradients are more effective in high dimensional action spaces, or when using continuous actions.

The problem with Deep Q-learning is that their predictions assign a score (maximum expected future reward) for each possible action, at each time step, given the current state.

But what if we have an infinite possibility of actions?

For instance, with a self driving car, at each state you can have a (near) infinite choice of actions (turning the wheel at 15°, 17.2°, 19,4°, honk…). We'll need to output a Q-value for each possible action!

On the other hand, in policy-based methods, you just adjust the parameters directly: thanks to that you'll start to understand what the maximum will be, rather than computing (estimating) the maximum directly at every step.



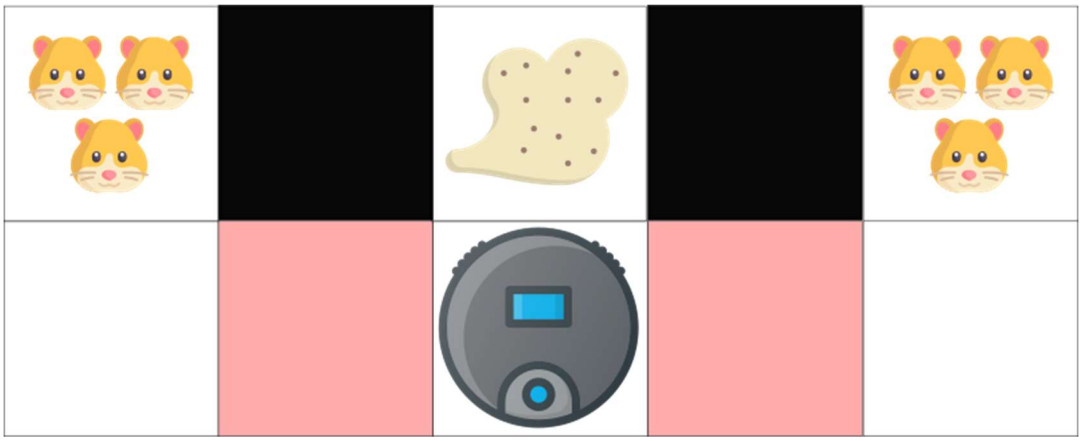

**Policy gradients can learn stochastic policies**

A third advantage is that policy gradient can learn a stochastic policy, while value functions can't. This has two consequences.

One of these is that we don't need to implement an exploration/exploitation trade off. A stochastic policy allows our agent to explore the state space without always taking the same action. This is because it outputs a probability distribution over actions. As a consequence, it handles the exploration/exploitation trade off without hard coding it.

We also get rid of the problem of perceptual aliasing. Perceptual aliasing is when we have two states that seem to be (or actually are) the same, but need different actions.

Let's take an example. We have a intelligent vacuum cleaner, and its goal is to suck the dust and avoid killing the hamsters.

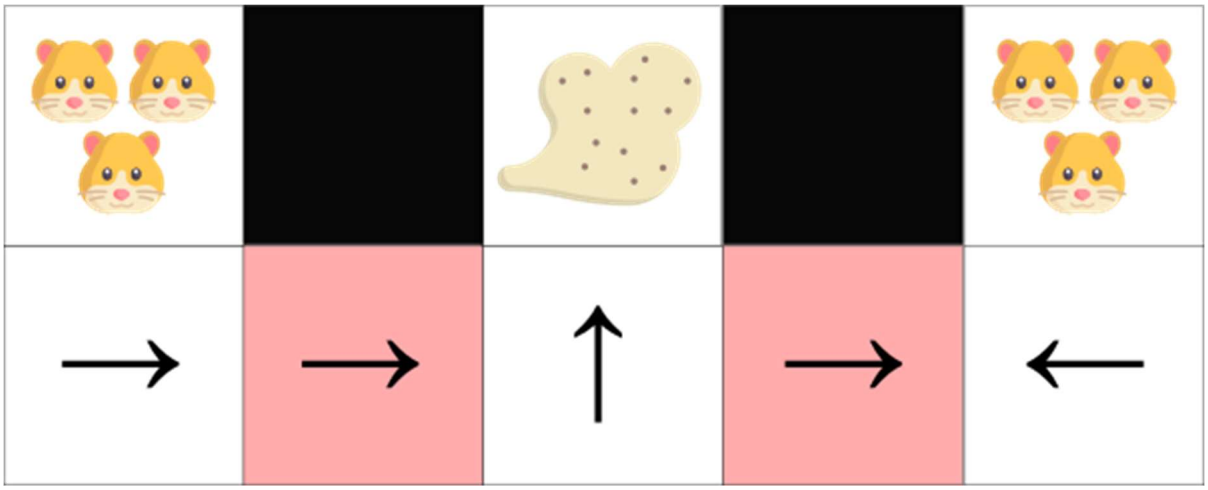


This example was inspired by the excellent course make by David Silver: [http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf](http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf)

Our vacuum cleaner can only perceive where the walls are.

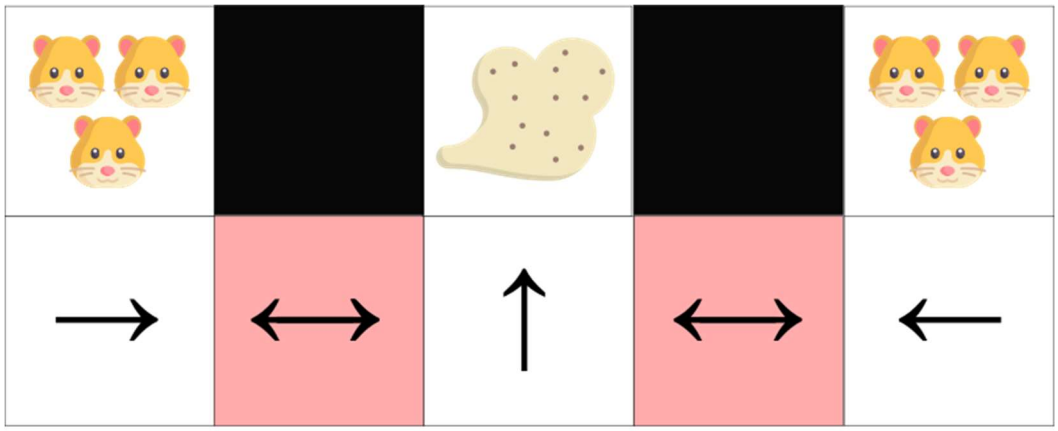
The problem: the two red cases are aliased states, because the agent perceives an upper and lower wall for each two.

Under a deterministic policy, the policy will be either moving right when in red state or moving left. Either case will cause our agent to get stuck and never suck the dust.



Under a value-based RL algorithm, we learn a quasi-deterministic policy ("epsilon greedy strategy"). As a consequence, our agent can spend a lot of time before finding the dust.

On the other hand, an optimal stochastic policy will randomly move left or right in grey states. As a consequence it will not be stuck and will reach the goal state with high probability.



$$\pi = (\text{wall UP and DOWN} \mid \text{Go LEFT}) = 0.5$$
$$\pi = (\text{wall UP and DOWN} \mid \text{Go RIGHT}) = 0.5$$

**Disadvantages**

Naturally, Policy gradients have one big disadvantage. A lot of the time, they converge on a local maximum rather than on the global optimum.

Instead of Deep Q-Learning, which always tries to reach the maximum, policy gradients converge slower, step by step. They can take longer to train.

However, we'll see there are solutions to this problem.

**Policy Search**

We have our policy π that has a parameter θ. This π outputs a probability distribution of actions.

$$\pi_\theta(a|s) = P[a|s]$$

Probability of taking action a given state s with parameters theta.

Awesome! But how do we know if our policy is good?

Remember that policy can be seen as an optimization problem. We must find the best parameters (θ) to maximize a score function, J(θ).

$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

There are two steps:

- Measure the quality of a π (policy) with a policy score function J(θ)
- Use policy gradient ascent to find the best parameter θ that improves our π.

The main idea here is that J(θ) will tell us how good our π is. Policy gradient ascent will help us to find the best policy parameters to maximize the sample of good actions.

**First Step: the Policy Score function J(θ)**

To measure how good our policy is, we use a function called the objective function (or Policy Score Function) that calculates the expected reward of policy.

Three methods work equally well for optimizing policies. The choice depends only on the environment and the objectives you have.

First, in an episodic environment, we can use the start value. Calculate the mean of the return from the first time step (G1). This is the cumulative discounted reward for the entire episode.

$$J_1(\theta) = E_\pi[G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + ...] = E_\pi(V(s_1))$$

Cumulative discounted rewards starting at start state

Equivalent

Value of state 1

The idea is simple. If I always start in some state s1, what's the total reward I'll get from that start state until the end?

We want to find the policy that maximizes G1, because it will be the optimal policy. This is due to the reward hypothesis explained in the first article.

For instance, in Breakout, I play a new game, but I lost the ball after 20 bricks destroyed (end of the game). New episodes always begin at the same state.

I calculate the score using J1(θ). Hitting 20 bricks is good, but I want to improve the score. To do that, I'll need to improve the probability distributions of my actions by tuning the parameters. This happens in step 2.

In a continuous environment, we can use the average value, because we can't rely on a specific start state.

Each state value is now weighted (because some happen more than others) by the probability of the occurrence of the respected state.

$$J_{avgv}(\theta) = E_\pi(V(s)) = \sum d(s)V(s)$$

$$where\ d(s) = \frac{N(s)}{\sum_{s'} N(s')}$$

Number of occurrences of the state

Total nb occurences of all states

Third, we can use the average reward per time step. The idea here is that we want to get the most reward per time step.

$$J_{avR}(\theta) = E_\pi(r) = \sum_s d(s) \sum_a \pi\theta(s,a)\ R_s^a$$

Probability that I'm in state s

Probability that I take this action a from that state under this policy

Immediate reward that I'll get

**Second step: Policy gradient ascent**

We have a Policy score function that tells us how good our policy is. Now, we want to find a parameter θ that maximizes this score function. Maximizing the score function means finding the optimal policy.

To maximize the score function J(θ), we need to do gradient ascent on policy parameters.

Gradient ascent is the inverse of gradient descent. Remember that gradient always points to the steepest change.

In gradient descent, we take the direction of the steepest decrease in the function. In gradient ascent we take the direction of the steepest increase of the function.

Why gradient ascent and not gradient descent? Because we use gradient descent when we have an error function that we want to minimize.

But, the score function is not an error function! It's a score function, and because we want to maximize the score, we need gradient ascent.

The idea is to find the gradient to the current policy π that updates the parameters in the direction of the greatest increase, and iterate.

$$
\begin{aligned}
Policy: \ & \pi_\theta \\
Objective\ function\ : & J(\theta) \\
Gradient: \ & \nabla_\theta J(\theta) \\
Update: \theta & \leftarrow \theta + \alpha \nabla_\theta J(\theta)
\end{aligned}
$$

Okay, now let's implement that mathematically. This part is a bit hard, but it's fundamental to understand how we arrive at our gradient formula.

We want to find the best parameters θ*, that maximize the score:

$$\theta^* = \underset{\theta}{argmax} \underbrace{E_{\pi\theta}[\sum_t R(s_t, a_t)]}_{J(\theta)}$$

Our score function can be defined as:

$$J(\theta) = \underbrace{E_\pi}_{\text{Expected given policy}}\underbrace{[R(\tau)]}_{\text{Expected future reward}}$$

s0, a0, r0,
s1, a1, r1...

Which is the total summation of expected reward given policy.

Now, because we want to do gradient ascent, we need to differentiate our score function J(θ).

Our score function J(θ) can be also defined as:

$$J_1(\theta) = V_{\pi\theta}(s_1) = E_{\pi\theta}[v_1] = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) R_s^a$$

State distribution — Action distribution

We wrote the function in this way to show the problem we face here.

We know that policy parameters change how actions are chosen, and as a consequence, what rewards we get and which states we will see and how often.

So, it can be challenging to find the changes of policy in a way that ensures improvement. This is because the performance depends on action selections and the distribution of states in which those selections are made.

Both of these are affected by policy parameters. The effect of policy parameters on the actions is simple to find, but how do we find the effect of policy on the state distribution? The function of the environment is unknown.

As a consequence, we face a problem: how do we estimate the ∇ (gradient) with respect to policy θ, when the gradient depends on the unknown effect of policy changes on the state distribution?

The solution will be to use the Policy Gradient Theorem. This provides an analytic expression for the gradient ∇ of J(θ) (performance) with respect to policy θ that does not involve the differentiation of the state distribution.

So let's calculate:

$$J(\theta) = E_\pi[R(\tau)]$$

s0, a0, r0, s1, a1, r1...

Expected given policy — Expected future reward

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

Put the gradient inside

Remember, we're in a situation of stochastic policy. This means that our policy outputs a probability distribution π(τ ; θ). It outputs the probability of taking these series of steps (s0, a0, r0…), given our current parameters θ.

But, differentiating a probability function is hard, unless we can transform it into a logarithm. This makes it much simpler to differentiate.

Here we'll use the likelihood ratio trick that replaces the resulting fraction into log probability.

$$\pi(\tau;\theta)\frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)} \qquad \nabla log\, x = \frac{\nabla x}{x}$$

Likelihood ratio trick

$$= \sum_\tau \pi(\tau;\theta)\nabla_\theta(log\pi(\tau;\theta))R(\tau)$$

Now let's convert the summation back to an expectation:

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta(log\pi(\tau|\theta))R(\tau)]$$

Policy function — Score function

As you can see, we only need to compute the derivative of the log policy function.

Now that we've done that, and it was a lot, we can conclude about policy gradients:

$$Policy\ gradient : E_\pi[\nabla_\theta(log\pi(s,a,\theta))R(\tau)]$$

Policy function — Score function

$$Update\ rule : \Delta\theta = \alpha * \nabla_\theta(log\pi(s,a,\theta))R(\tau)$$

Change in parameters — Learning rate

This Policy gradient is telling us how we should shift the policy distribution through changing parameters θ if we want to achieve an higher score.

R(tau) is like a scalar value score:

- If R(tau) is high, it means that on average we took actions that lead to high rewards. We want to push the probabilities of the actions seen (increase the probability of taking these actions).
- On the other hand, if R(tau) is low, we want to push down the probabilities of the actions seen.

This policy gradient causes the parameters to move most in the direction that favors actions that has the highest return.