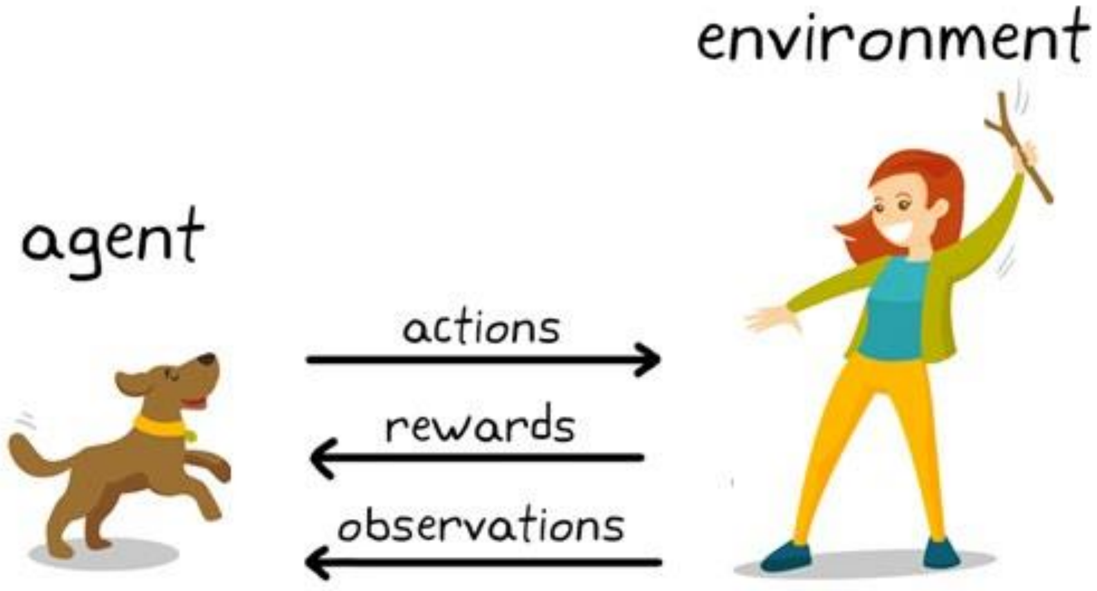


Deep Reinforcement learning

Iheb Bouariche | 17.03.2023

This research provides a simplified and accessible explanation of RL and Deep RL, covering the most crucial topics in the field. While the paper is still a work in progress, the current version aims to offer readers a clear understanding of these complex concepts.

Stat of art:

Artificial Intelligence (AI) and especially machine leaning field, focused on the development of intelligent machines that can perform tasks that typically require human intelligence, such as reasoning, perception, and decision making. AI aims to create algorithms and systems that can learn from data and experiences, improve over time, and adapt to changing environments. These intelligent machines can be programmed to perform a variety of tasks, ranging from natural language processing and image recognition to game playing, optimal control and autonomous driving. The ultimate goal of AI is to create machines that can operate autonomously and intelligently, exhibiting human-like behavior in a wide range of domains.

Machine learning, along with its subfield deep learning, has seen significant advances and success over the last decade. The three main categories of machine learning are supervised learning, unsupervised learning, and reinforcement learning.

In supervised learning, the algorithm is trained on labeled data, where the desired output is already known. The algorithm learns to map input data to the correct output by adjusting the model's parameters. Unsupervised learning, on the other hand, involves training algorithms on unlabeled data, where the desired output is not known. The algorithm must identify patterns and relationships in the data without human supervision.

Reinforcement learning has made significant progress in recent years, enabling agents to learn how to perform complex tasks in a variety of domains by interacting with the environment. One of the most important developments in reinforcement learning has been the introduction of deep reinforcement learning, which combines deep learning and reinforcement learning to enable agents to learn from high-dimensional sensory inputs and solving more complex tasks. Some notable examples of successful applications of deep reinforcement learning include learning to play Atari games, beating world champions in games such as Go and Chess, and achieving breakthroughs in robotics and autonomous systems.

One of the advanced applications of DRL in robotics is in the field of robotic manipulation and control, where agents are trained to optimize control signals and trajectories and perform complex tasks, such as grasping objects with robotic hands, assembling parts, and sorting objects. DRL has also been used to train robotic agents to walk, run, and jump, enabling the development of agile and dynamic robots that can navigate rough terrains and perform acrobatic movements.

What is Reinforcement Learning?

Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing at certain state an action and seeing the results of this decision. For each good action, the agent gets positive reward, and for each bad action, the agent gets negative reward or penalty. The primary goal of an agent in reinforcement learning is to adjust its behavior and improve its decision-making process for getting the maximum positive rewards by learning an optimal policy. Over time, agent explores and exploit its experiences to take actions that maximize its reward and minimize its penalty.

Markov decision process:

Markov process: states with probabilities. (it's like automata)

Markov reward process: states with probabilities, rewards and discount factor.

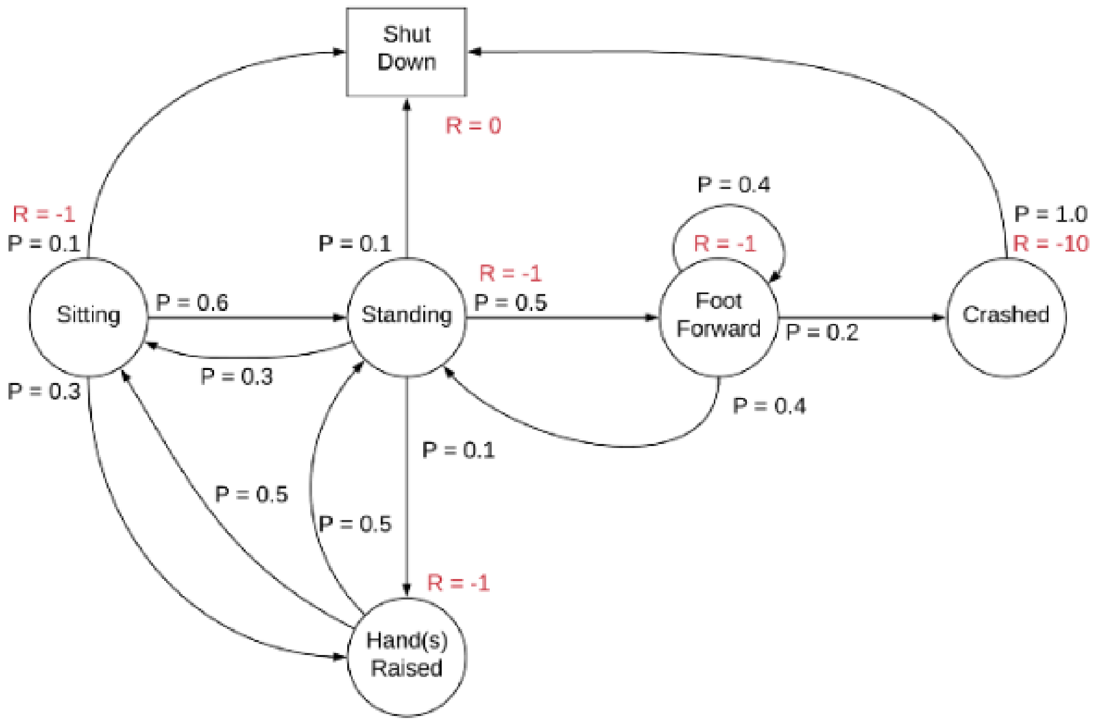


Figure 01: Markov reward process

Markov decision process: <u>decision</u> means that it depends on the agent to take actions, so, we have states with probabilities, rewards, discount factor and <u>stochastic policies</u>.

Markov Decision Process is a mathematical framework for defining an RL system, first the environment that the agent interacts with defined by states 'S', then the actions taken by the agent defined by 'A', and the received rewards defined by 'R'. In MDP, transitions are defined by probabilities for moving from one state to another state after taking a specific action. MDP can be defined as a tuple:

$$(S, A, P, R, \gamma)$$

Where: P: is the transition probability, $\mathrm{P} = \mathrm{p}(\mathrm{s}(\mathrm{t}+1) \mid \mathrm{s}(\mathrm{t}), \mathrm{a}(\mathrm{t}))$.

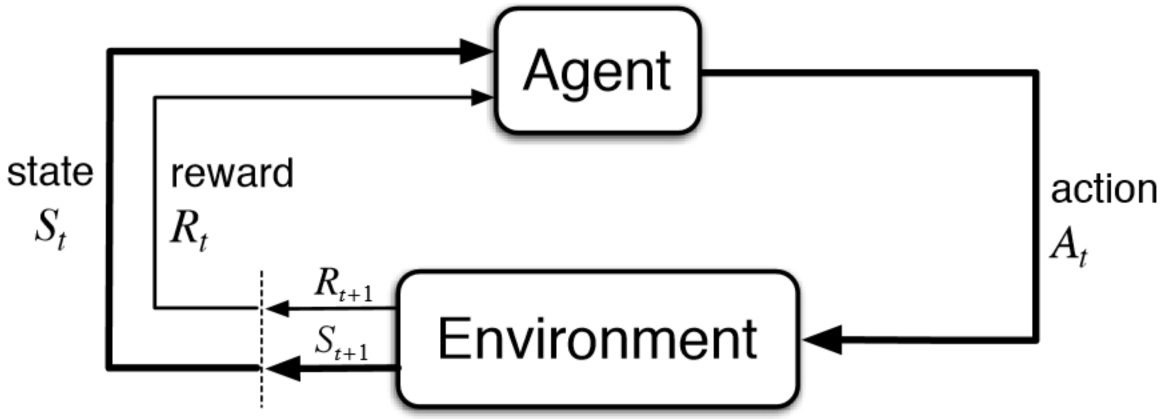
R: is the reward function, $\mathrm{R} = \mathrm{E}[\mathrm{r}(\mathrm{t}+1)|\mathrm{s}(\mathrm{t}), \mathrm{a}(\mathrm{t})]$.

γ: is the discount factor between 0 and 1.

S: is the finite set of states of the environment.

A: is the set of actions taken by the agent.

Agent interacts with the environment starting from initial state and by taking an action in each step, each step returns a reward 'r' and the agent will be in the next state 's(t+1)' until achieving the terminal state in the case of finite sequence and this usually called a training episode. We use then, reinforcement learning methods to learn from each episode and optimize the agent's policy.



What is policy?

A policy is mapping between states and actions that an agent uses to decide what action to take in a particular state, there are two types of policies: Deterministic Policy which maps each state to a single action. And Stochastic Policy which maps each state to a probability distribution over possible action. The primary objective of reinforcement learning is to find the optimal policy by interacting with the environment.

Deterministic policy is defined by: $\pi(s) = a$, Stochastic policy is defined by: $\pi(s|a) = a$.

Model-free and model-based:

In Model-Free Learning the agent learns directly from experiences by interacting with the environment. The agent does not have a model of the environment and it learns optimal policies through trial and error, our research here is based only on model-free RL methods. In the other side, the model-based learning, it requires a model which the agent can be used to learn a policy.

Cumulative reward: as the agent interact with the environment the cumulative rewards G{t} on each episode and at each step{t} can be defined by:

$$G = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Methods that learn values to solve MDP problems called <u>Value-Based</u> methods, in the other hand, there are <u>policy-based</u> methods and they optimize directly the policy.

Some of the RL challenges is defining the reward function (objective) of the agent.

State-value: is the expected reward from a particular state. And it defines how a state is better than other states in the environment.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], for\ all\ s \in S,$$

Action-value: is the expected reward conditioned by taking a particular action in a particular state. And it defines how an action in a particular state is better than the other actions in the same state.

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Optimal policy: state and action values are used to finding the optimal policy, then the MDP problem can be solved by these values, the problem here is how to find the optimal state or action values of an environment. RL value-based methods are designed to solve this problem.

$$\pi_*(\mathrm{s}) = \arg\max_\pi \mathcal{V}_\pi(\mathrm{s}) = \arg\max_\pi \mathrm{Q}_\pi(\mathrm{s}, \mathrm{a})$$

Bellman equation: is a recursive relationship that expresses the value of a state in a Markov Decision Process (MDP) in terms of the immediate reward obtained in that state and the expected value of the successor states. In other words, it defines the relationship between the value of a state and the values of its successor states. The bellman equation defines the expected values as follows:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s_{t+1})|S_t = s]$$

And the action-values as follows:

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})|S_t = s, A = a]$$

Temporal difference learning TD(0): is a value-based method used to learn state-values and mostly it converges to the optimal values. it's an On-policy iterative method that estimate the action values by using the bellman equation. it uses the immediate reward and the difference between the current state-value and the next state-value. TD(0) updates state-values with experiencing in the environment as follows:

- At each step {t}:

$\mathcal{V}_t \rightarrow \mathcal{V}_t + \alpha * (r_{t+1} + \gamma * \mathcal{V}_{t+1} - \mathcal{V}_t)$, Where: α is the learning rate.

SARSA: is a value-based and ON-Policy learning method used to learn action-values, it uses a similar update rule as TD, the only different is on action-values update. SARSA learning is defined as follows:

- At each step {t}:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Q-learning: one of the most important value-based methods used in RL. It's an OFF-Policy method and its update is similar to SARSA, but the main difference is that Q-learning uses the maximum estimated action-value of the next state, regardless of the action that the agent actually takes next.

- At each step {t}:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

To ensure good training, one approach is to use an exploration-exploitation strategy such as the ε-greedy method. In the real world, it is often more preferable to use action values rather than state values because we cannot know the value of the next state before taking an action. However, we can estimate the value of each action in a given state, which allows us to make more informed decisions.

TD methods can be used to solve problems in low-dimensional and simple environments, as well as in high-dimensional and complex environments. However, as the dimensionality of the state space increases, the amount of memory required to store the value function can become prohibitively large. To address this issue, function approximation methods such as neural networks can be used.

Deep learning:

Deep learning is a subfield of machine learning that is based on the use of artificial neural networks with multiple layers. These networks are designed to learn from large amounts of data and to generalize that learning to new situations.

The use of function approximation methods, particularly neural networks is to address the issue of memory when dealing with large dimensional environments in the context of TD methods. By using a neural network as a function approximator, the agent can learn to generalize its value estimates across similar states, even when the number of states is too large to store explicitly in memory and in complex and high-dimensional environments. Additionally, the use of deep neural networks can enable the agent to learn more sophisticated representations of the environment, leading to better perception and decision-making capabilities.

There are many different types of neural network architectures that have been developed for various applications. Most common architectures are:

1- Feedforward Neural Networks: This is the simplest type of neural network architecture, which consists of one or more layers of interconnected nodes, with no loops or cycles in the network structure.
2- Convolutional Neural Networks (CNNs): CNNs are designed specifically for image and video processing tasks, and use a specialized type of layer called a convolutional layer with filters to extract features from the input.

3- Recurrent Neural Networks (RNNs): RNNs are designed for processing sequential data, such as time sequence signals, speech and text. They use feedback connections between the nodes to store information about the previous inputs in the sequence.

Deep Reinforcement Learning: is a subfield of modern artificial intelligent that combines deep neural networks and reinforcement learning methods to enable machines to learn and make decisions in complex environments.

Deep Q-learning:

[Playing Atari with Deep Reinforcement Learning" by Volodymyr Mnih et al., published in 2013], this research paper was the explosion of Deep reinforcement learning when Deep Q-learning solves ATARI games and surpass the human expert. The basic idea of this algorithm is estimating the action values in an environment using the Q-learning and the bellman equation to learn deep neural network the action values, this network called Q-Network. It can be trained by minimizing a sequence of loss functions $Li(\theta i)$ (Q-learning loss) that changes at each step i,

$$L_i\ (\theta_i)\ =\ (r + \gamma \max_{a\prime} Q(s'\ , a'\ ;\ \theta_{i-1}) - Q(s, a, \theta_i))^2$$

This algorithm uses a technique called experience replay memory to train the neural network. During the training steps it selects minibatch random samples from the experience memory and apply the Q-learning update using the gradient descent. This method provides data efficiency and it breaks the correlation and reduces variance of the updates. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

Actually, DQN uses two neural networks. The target network is a copy of the Q-network that is frozen for a certain number of steps and is only updated periodically with the weights from the Q-network. A target network is used to calculate the TD error target value and stabilize the learning process and prevent overfitting of the Q-network and helps to reduce the variance and improve convergence.

Deep Q-learning algorithm is defined as follows:

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Value-based methods are used only on discrete-action space. For control problems, we will address methods for continuous action space.

Policy Gradient: (check this link to understand https://towardsdatascience.com/policy-gradients-in-reinforcement-learning-explained-ecec7df94245))

in contrast to value-based methods like TD learning and Q-learning, policy-based methods optimize the policy $\pi(a|s,\theta)$ directly with function approximation. And update the parameters θ by gradient ascent (Negative of gradient descent) on $E_{\tau\sim\pi_\theta}(R(\tau))$ the expected reward for "τ" trajectories generated from a stochastic policy π. We have REINFORCE algorithm (Williams, 1992) that address this problem, it's a policy gradient method updating θ in the direction of $\nabla_\theta log\pi(a_t|s_t,\theta)R_t$. Usually, a baseline $b_t(s_t)$ is subtracted from the return to reduce the variance of gradient estimate, and stabilize the training.

The objective function defined as follows:

$$J(\theta) = E_{\tau\sim\pi_\theta}(R(\tau)) = \sum_{\tau} P(\tau,\theta)R(\tau)$$

The gradient direction:

$$\nabla_\theta log\pi(a_t|s_t,\theta)R_t$$

The gradient direction with baseline is defined as follows:

$$\nabla_\theta log\pi(a_t|s_t, \theta)(R_t - b_t(s_t)$$

In the reinforce algorithm, the expected reward at each step R_t can be calculated at the end of an episode. $b_t(s_t)$ can be represent the average reward in a trajectory at the end of an episode. So, updating is performed at the end of an episode.

Another technique uses the advantage function $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$, can also be defined in the place of $(R_t - b_t(s_t))$. to measure how good an action is compared to other actions. $Q(a_t, s_t)$ replaces the expected reward, and the challenge is to determine V and Q. This can be achieved by using the Bellman equation and estimating only the value function using the actor-critic method. The advantage method provides an update of the policy approximation <u>at each step</u> (<u>Online training</u>) compared to the reinforce algorithm which can update only at the end of the episode (<u>Offline training</u>) the end of an episode.

Actor-critic method work with what called Policy iteration and it alternates between policy evaluation and policy improvement, to generate a sequence of improving policies. In policy evaluation, the value function of the current policy is estimated from the outcomes of sampled trajectories. In policy improvement, the current value function is used to generate a better policy.

Deep Deterministic Policy Gradient (DDPG):

DDPG is an actor-critic algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. while Deep Q-network solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. Deep Q-network cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function.

DDPG address the problem of continuous action space, and it uses as deep Q-learning a replay buffer memory technique and separate target networks for calculating the target value of the TD error. The difference from the Q-network is that we have the actor action and the state as an input to the Q-network and, its output is an action value.

DDPG learn a deterministic actor policy $\mu_\theta(s)$ which gives the action that maximize the critic $Q_\phi(s, a)$. By sampling a mini-batch from the buffer experiences, the actor network's ($\mu_\theta(s)$) weights are updated using the gradients of the Q-function (with a direction that maximize the actions values). In the other hand, Q-network {the critic network $Q_\phi(s, a)$} will be updated by the bellman equation and by using the actor and critic target networks. <u>Note that DDPG is not using the critic Q-value as a baseline, it updates directly the actor network from the gradient of the output of the Q-network, in order maximize the Q-value.</u>

To encourage exploration and exploitation, a noise will be added to the action of the actor, and usually, this noise will be decayed over episodes.

DDPG algorithm is defined as follows:

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$

    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

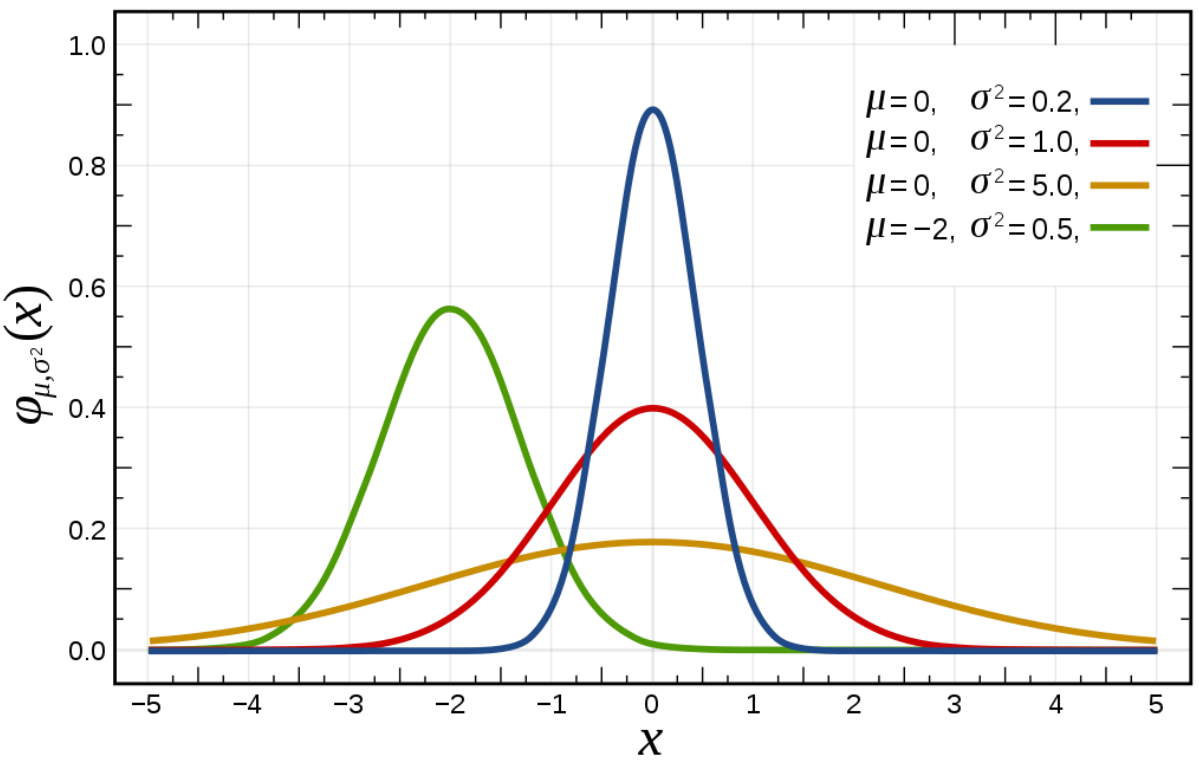
Training stochastic policies:

- DDPG is an off-policy actor-critic algorithm and it's used for continuous control tasks. It combines the benefits of deep neural networks with the power of Q-learning, allowing for better learning and optimization of complex policies. DDPG uses a deterministic policy to determine the optimal action to take at any given state, and it uses an experience replay buffer to learn from past experiences.
- PPO, on the other hand, is an on-policy algorithm that is used for both discrete and continuous control tasks. PPO optimizes the policy by maximizing the expected reward while simultaneously enforcing a constraint on the change in policy distribution. This helps to prevent the policy from changing too drastically and destabilizing the learning process. PPO achieves this by using a clipped surrogate objective function to update the policy in small steps.

Standard (Gaussian) normal distribution:

From Wikipedia, a normal distribution or Gaussian distribution is a type of continuous probability distribution for a real-valued random variable. The general form of its probability density function is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



The parameter μ is the mean or expectation of the distribution, while the parameter σ is its standard deviation. The variance of the distribution is σ^2. A random sampled number with respect to the probability distribution (Gaussian distribution) is said to be normally distributed, and is called a normal deviate.

Note that, the integral of a normal distribution over its entire domain is equal to 1.

Normal distribution can be learned by a neural network with mean and standard deviation as the outputs of the neural network. This idea will be used for continuous action space of policy gradient methods (TRPO & PPO).

Manifold probability distribution:

A probability manifold is a mathematical concept that describes a space where each point in the space corresponds to a probability distribution. More formally, a probability manifold is a manifold equipped with a family of probability distributions, one for each point on the manifold.

Fisher information:

Fisher information is a concept in mathematical statistics that quantifies how much information a random variable X carries about an unknown parameter θ. More specifically, it measures how sensitive the probability distribution of X is to changes in θ.

Entropy:

Entropy is a measure of the uncertainty or randomness of a probability distribution. For example, if we have coin with two faces, and we have a probability of 0.5 for each face, then we will have the highest value of the entropy. On the other hand, if we have one face with a probability of 1 and the other with a probability of 0, then we will have the lowest value of entropy. Entropy function can be calculated by this equation:

$$H(x) = -\sum_{x\in\chi} p(x)\log\big(p(x)\big)\,,$$

Entropy defined by the negative value of the sum of the probability of each case in a sample space multiplied by the log of each sample's probability.

KL-Divergence:

Is a measure of how one probability distribution P is different from a second, reference probability distribution Q. the KL-divergence can be calculated as follows:

$$D_{KL}(P||Q) = \sum_{x\in\chi} p(x)\log\left(\frac{P(x)}{Q(x)}\right),$$

Natural policy gradient:

The natural policy gradient is a method for optimizing the parameters of a policy. The key idea behind this method is to perform gradient ascent on the objective function that maximizes the expected reward, subject to a constraint on the KL-divergence between the updated policy and the old policy. This is achieved by transforming the parameter space using the Fisher information matrix, which captures the local curvature of the policy's objective function. The advantage of this method compared to the traditional policy gradient is that it is less sensitive to the choice of step size and can converge faster, particularly in high-dimensional or complex environments. It can also handle non-linear function approximators, such as neural networks.

We assume that π is the current policy and π_{old} is the old policy. F is the Fisher matrix and $||\Delta\theta||$ is the gradient update constraint (trust region distance). $\hat{A}_n$ is the advantage function, which contains the expected reward minus a baseline.

The objective of natural policy gradient is:

$$\underset{\pi_\theta}{\text{maximze}}\, \mathcal{L}(\pi_\theta),$$

$$subject\ to\ ||\Delta\theta|| < \delta,\ where\ ||\Delta\theta|| = (\Delta\theta^T F \Delta\theta)^{1/2}\ ,$$

$$where{:}\ \Delta\theta = \theta - \theta_{old}$$

Since the policy space is not flat, the trust region distance is measured with a metric tensor F (Fisher matrix) instead of the Euclidean distance (a square distance).

$$\mathcal{L}(\pi_\theta) = \hat{\mathbb{E}}_t \frac{\pi_{old}(a_n|s_n)}{\pi(a_n|s_n)}\ \hat{A}_n$$

$\mathcal{L}(\pi_\theta)$ is the objective function, which contains the ratio between the old policy and the current policy. By optimizing this ratio, the natural policy gradient algorithm is able to constrain changes to be small, resulting in stable and efficient learning.

We use KL divergence to measure the difference between two probability distributions and constrain objective updates by limiting the KL divergence.

$$\underset{\pi_\theta}{\text{maximze}}\, \mathcal{L}(\pi_\theta),$$
$$subject\ to\ D_{KL}(\pi_{old}\|\pi) < \delta$$

The KL divergence is defined by the following expression:

$$D_{KL}(\pi_{old}\|\pi) = \sum_{x\in\chi} \pi_{old} log\left(\frac{\pi_{old}(x)}{\pi(x)}\right)$$

We will see later, that there is a relation between the KL divergence and the Fisher matrix.

After defining the objective. Next, we will use some mathematical approximations (Taylor and Lagrange approximations), to transform this problem into numerically solvable equations.

$$\mathcal{L}_{\theta_k}(\theta) \approx \cancelto{0}{\mathcal{L}_{\theta_k}(\theta_k)} + g^T(\theta-\theta_k) + \ \ldots \textit{(the second order term for } \mathcal{L} \textit{ is ignored)}$$
$$\bar{D}_{KL}(\theta\|\theta_k) \approx \cancelto{0}{\bar{D}_{KL}(\theta_k\|\theta_k)} + \cancelto{0}{\nabla_\theta \bar{D}_{KL}(\theta\|\theta_k)|_{\theta_k}}(\theta-\theta_k) + \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k)$$

$$\mathcal{L}_{\theta_k}(\theta) \approx g^T(\theta-\theta_k) \qquad g \doteq \nabla_\theta \mathcal{L}_{\theta_k}(\theta)\,|_{\theta_k}$$
$$\bar{D}_{KL}(\theta\|\theta_k) \approx \frac{1}{2}(\theta-\theta_k)^T F\,(\theta-\theta_k) \qquad F \doteq \nabla_\theta^2 \bar{D}_{KL}(\theta\|\theta_k)\,|_{\theta_k}$$

As we derive the KL-divergence equation, we find that the Fisher information matrix is defined as the second derivative of the KL-divergence.

$$F = \nabla_\theta^2 \bar{D}_{KL}(\theta\|\theta_k)\,|_{\theta_k}$$
$$D_{KL}(P\|Q) = \sum_{x=1}^{N} P(x)\log\frac{P(x)}{Q(x)}$$

The following iterative equation can be used to find the value of θ that maximizes the objective function while satisfying the KL constraint.

$$\theta_{k+1} = \arg\max_\theta\ g^T(\theta-\theta_k)$$
$$\text{s.t.}\ \frac{1}{2}(\theta-\theta_k)^T F\,(\theta-\theta_k) \le \delta$$

By developing the previous equation into an iterative and numerical form, we arrive at the following natural gradient equation:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T F^{-1} g}} F^{-1} g$$

where $g = \nabla_\theta L$

The problem here is how to calculate the Fisher matrix?. Natural policy gradient provides a theoretical method for calculating the Fisher matrix (not defined here), but it may not be practical for high-dimensional parameter spaces such as neural networks due to the significant computational resources it requires. To overcome this problem, a new method called Trust Region Policy Approximation has been developed.

We did not cover the mathematical development of the natural policy gradient in depth due to its complexity.

Algorithm 1 Natural Policy Gradient

```
Input: initial policy parameters $\theta_0$
for $k = 0, 1, 2, \ldots$ do
    Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
    Form sample estimates for
    •    policy gradient $\hat{g}_k$ (using advantage estimates)
    •    and KL-divergence Hessian / Fisher Information Matrix $\hat{H}_k$
    Compute Natural Policy Gradient update:
        $$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$$
end for
```

Trust region policy optimization (TRPO):

TRPO is a policy optimization algorithm in reinforcement learning that addresses the problem of choosing step sizes for policy updates in a stable and effective way. It is a type of natural policy gradient method that has been shown to perform well in large-scale, high-dimensional environments.

TRPO uses the natural gradient of the objective function with respect to the policy parameters, which takes into account the geometry of the parameter space, to compute the step direction. This ensures that the step direction is the one that maximally improves the objective function while staying within the trust region.

Compared to natural policy gradient, TRPO address the problem due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO adds a modification to this update rule, a backtracking line search.

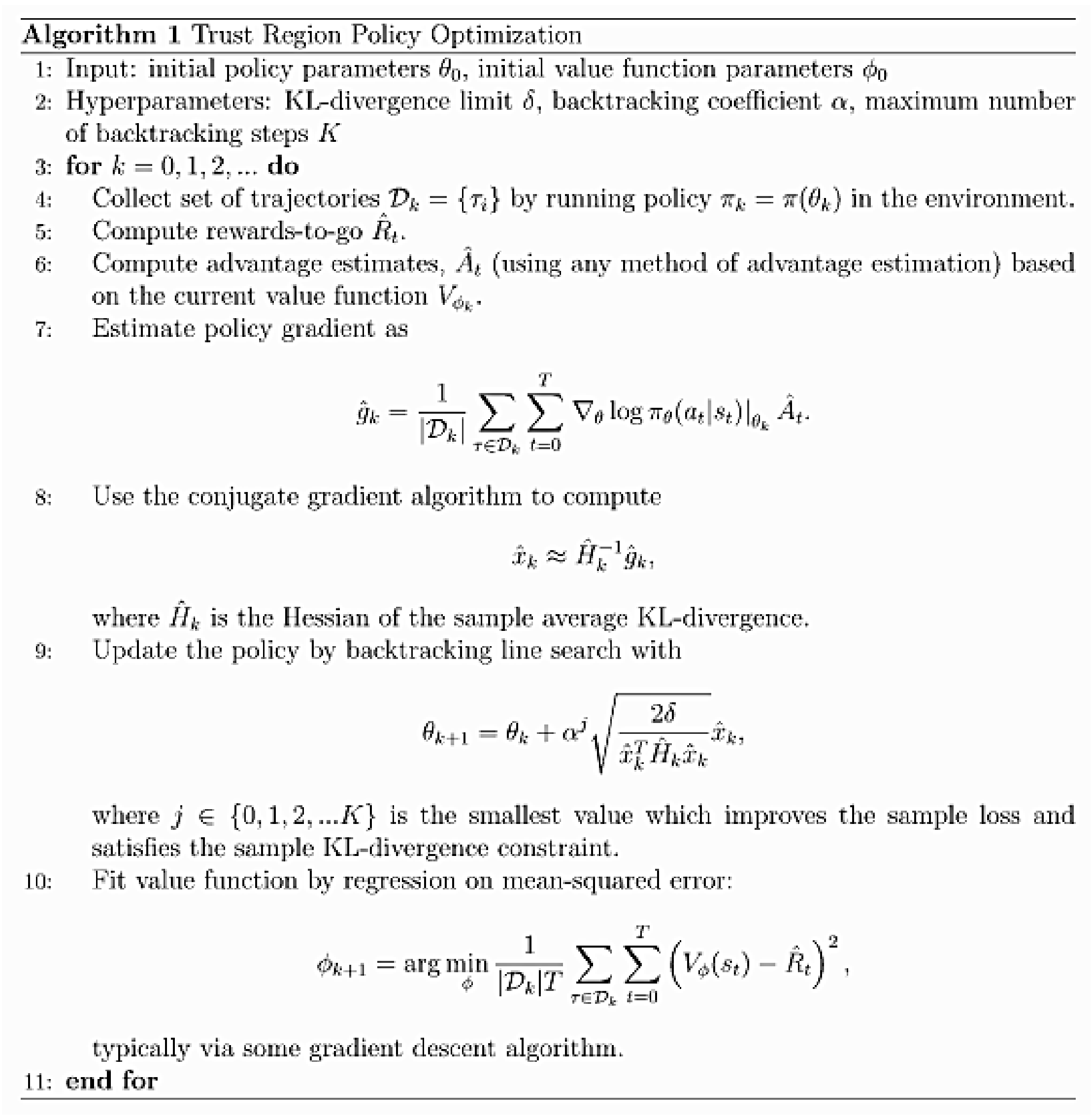
$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T F^{-1} g}} F^{-1} g$$

where $\alpha \in (0,1)$ is the backtracking coefficient, and j is the smallest nonnegative integer such that $\pi_{\theta_{k+1}}$ satisfies the KL constraint and produces a positive surrogate advantage.

Computing and storing the matrix inverse, F^{-1}, is painfully expensive when dealing with neural network policies with thousands or millions of parameters. TRPO sidesteps the issue by using the conjugate gradient algorithm to solve $Fx = g$ for $x = F^{-1}g$, requiring only a function which can compute the matrix-vector product Fx instead of computing and storing the whole matrix F directly. This is not too hard to do, the function's defined by:

$$Fx = \nabla_\theta((\nabla_\theta D_{KL}(\theta \| \theta_k))^T x)$$

which gives us the correct output without computing the whole matrix.



Algorithm 1 Trust Region Policy Optimization

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number
   of backtracking steps $K$
3: for $k = 0, 1, 2, ...$ do
4:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5:     Compute rewards-to-go $\hat{R}_t$.
6:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
7:     Estimate policy gradient as
           $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$
8:     Use the conjugate gradient algorithm to compute
           $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$
       where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9:     Update the policy by backtracking line search with
           $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$
       where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and
       satisfies the sample KL-divergence constraint.
10:    Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
11: end for
```

Proximal policy optimization (PPO):

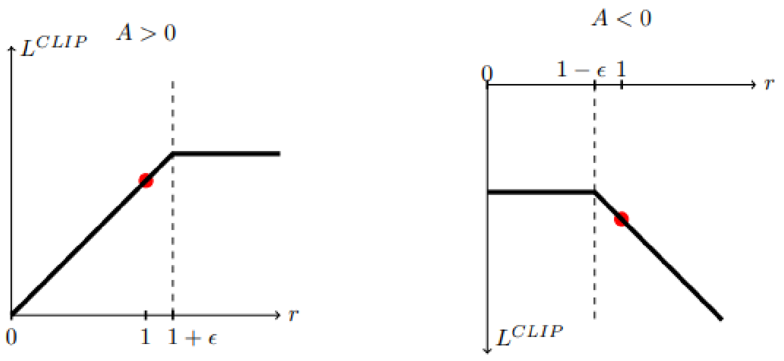
TRPO tries to solve RL problems with a complex second-order method, in the other hand, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old. The main idea behind PPO is to ensure that the policy updates do not deviate too far from the previous policy, which can cause instability or slow convergence. This is achieved by introducing a clipping objective that restricts the maximum change in the probability ratio between the new and old policies.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

$$Where, r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, and, \hat{A}\ is\ the\ advantage\ function.$$

Stochastic gradient descent can be used to optimize this objective function.

The hyperparameter ε is applied within the clip function to restrict the movement of r_t outside the interval $[1-\varepsilon, 1+\varepsilon]$. This is intended to prevent r_t from moving too far from its current value and helps to ensure stable optimization.



PPO also uses a value function to estimate the expected reward of the current policy, which is used to compute the advantage function. The advantage function measures how much better or worse each action is compared to the expected reward, which is used to update the policy parameters.

```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

PPO has several advantages over other policy optimization algorithms, such as faster convergence, better sample efficiency, and improved stability. It also has a number of hyperparameters that can be tuned to balance the trade-off between exploration and exploitation (entropy). PPO is more popular then TRPO due to its simplicity and better performance on various tasks.

Some Resources:

https://www.mathworks.com/content/dam/mathworks/ebook/gated/reinforcement-learning-ebook-all-chapters.pdf book

deep reinforcement learning hands-on by maxim lapan book

foundation of deep reinforcement learning book

https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

https://www.javatpoint.com/reinforcement-learning

https://www.synopsys.com/ai/what-is-reinforcement-learning.html

Deep Q learning paper : https://arxiv.org/pdf/1312.5602.pdf

https://link.springer.com/content/pdf/10.1007/BF00992696.pdf (reinforce)

https://towardsdatascience.com/policy-gradients-in-reinforcement-learning-explained-ecec7df94245 reinforce gradient source

https://spinningup.openai.com/en/latest/algorithms/ddpg.html

https://arxiv.org/pdf/1509.02971.pdf DDPG

https://spinningup.openai.com/en/latest/algorithms/ppo.html

https://spinningup.openai.com/en/latest/algorithms/trpo.html

https://arxiv.org/pdf/1707.06347.pdf (PPO research-paper)

https://arxiv.org/pdf/1502.05477.pdf (TRPO research paper)

https://www.andrew.cmu.edu/course/10-703/slides/Lecture_NaturalPolicyGradientsTRPOPPO.pdf (Natural policy gradient - TRPO and PPO)

https://homes.cs.washington.edu/~sham/papers/rl/natural.pdf (natural policy gradient)