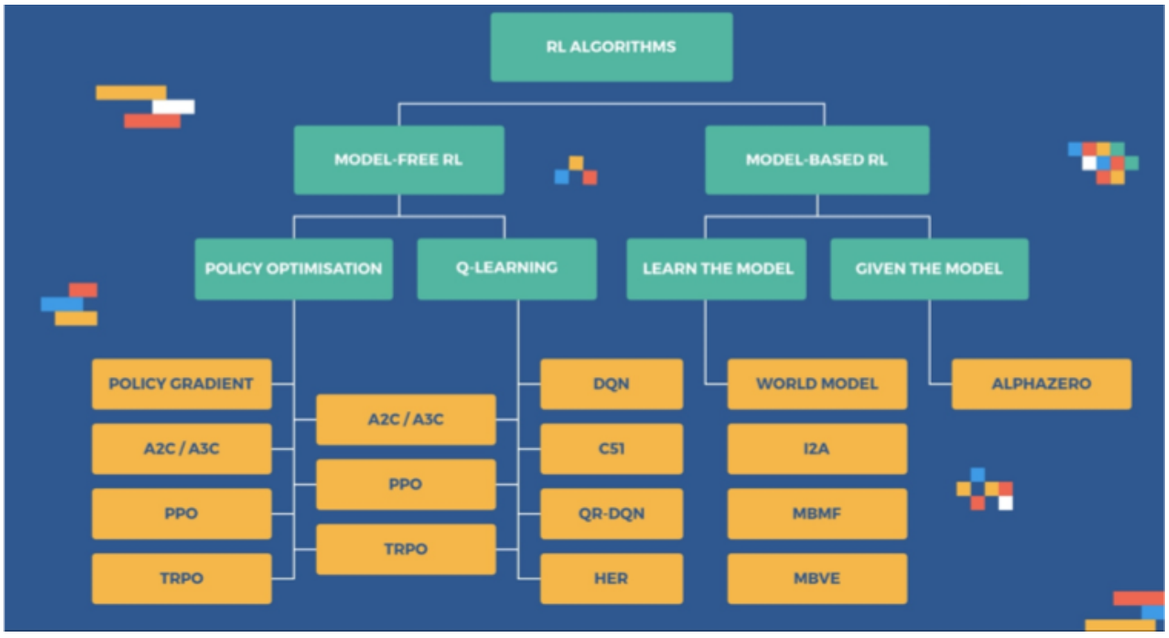
Deep Reinforcement Learning 2.0 from Udemy



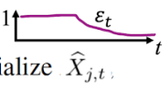
Multi-Armed Bandits

From: https://www.youtube.com/watch?v=XxTgX8FIDII and https://www.youtube.com/watch?v=FgmMK6RPU1c

ε-greedy:

Notice that ε would start to go down after a period of time.

Input : number of rounds n, number of arms m, a constant k , sequence $\{\varepsilon_t\}_{t=1}^{n} = \min\{1, \frac{km}{t}\}$



Initialization: play all arms once and initialize $\widehat{X}_{j,t}$

for $t = m + 1$ **to** n **do**

With probability ε_t play an arm uniformly at random (each arm has probability $\frac{1}{m}$ of being selected), otherwise (with probability $1 - \varepsilon_t$) play ("best") arm j such that

$$\widehat{X}_{j,t-1} \geq \widehat{X}_{i,t-1} \; \forall i.$$

Get reward $X_j(t)$;

Update $\widehat{X}_{j,t}$;

end

UCB (Upper Confidence Bound):

Input : number of rounds n, number of arms m

Initialization: play all arms once and initialize $\widehat{X}_{j,t}$

for $t = m + 1$ **to** n **do**

play arm j with the highest upper confidence bound on the mean estimate:

$$\widehat{X}_{j,t-1} + \sqrt{\frac{2\,\log(t)}{T_j(t-1)}};$$

Number of times arm j was played up to time t-1

Get reward X_j;

Update $\widehat{X}_{j,t}$;

end

Bellman Equation

s' is the next State

R(s, a) is the instant Reward by taking an Action a at State s (e.g. 1, -1, or 0)

The Bellman Equation $V(s) = \max_a(R(s,a) + \gamma V(s'))$

$\gamma = 0.9$

V=0.81	V=0.9	V=1	
V=0.73		V=0.9	
V=0.66	V=0.73	V=0.81	V=0.73

Markov Decision Process

Use the Expected Value of the next State, because it is a Markov Decision Process, and it is unknown in which State will Action a result.

$$V(s) = \max_a(R(s,a) + \gamma \overbrace{V(s')}^{0.8 * V(s'_1) + 0.1 * V(s'_2) + 0.1 * V(s'_3)})$$

and therefore

$$V(s) = \max_a\left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')\right)$$

P(s, a, s')V(s') is the probably of "at State s, taking Action a, and get to State s' " times the Value of s'

V=0.71	V=0.74	V=0.86	
V=0.63		V=0.39	
V=0.55	V=0.46	V=0.36	V=0.22

Q-Learning

While V(s) function is about a State s, Q(s, a) function is about an Action a.

$$V(s) = \max_a\left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')\right)$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'}\left(P(s,a,s')V(s')\right) = R(s,a) + \gamma \sum_{s'}\left(P(s,a,s') \max_{a'} Q(s',a')\right)$$

Temporal Difference TD(a, s):

Before:	After:
$Q(s,a)$	$R(s,a) + \gamma \max_{a'} Q(s',a')$

$$TD(a,s) = R(s,a) + \gamma \max_{a'} Q(s',a') - Q_{t-1}(s,a)$$

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha TD_t(a,s) = Q_{t-1}(s,a) + \alpha\left(R(s,a) + \gamma \max_{a'} Q(s',a') - Q_{t-1}(s,a)\right)$$

From: https://www.youtube.com/watch?v=0iqz4tcKN58

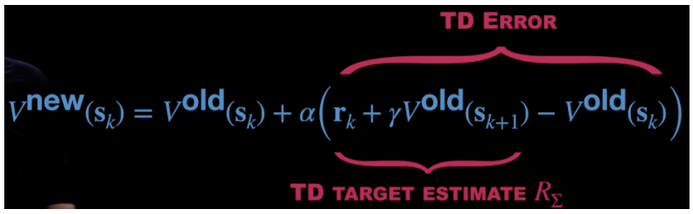
Q(s, a) = quality of a State-Action pair

Value function at State s i.e. V(S) = max Q(s, a) over all possible Action a

Value function at time k State i.e. $V(\mathbf{s}_k) = \mathbb{E}\left(\mathbf{r}_k + \gamma V(\mathbf{s}_{k+1})\right)$ In other words, Expected(Reward at time k + discounted Value at time k+1)

The following function shows how to update the Value function. α is the learning rate. (Me: similar to gradient decent)

V_old is the old best estimate. The $\mathbf{r}_k + \gamma V^{\mathbf{old}}(\mathbf{s}_{k+1})$ is what actually happened.

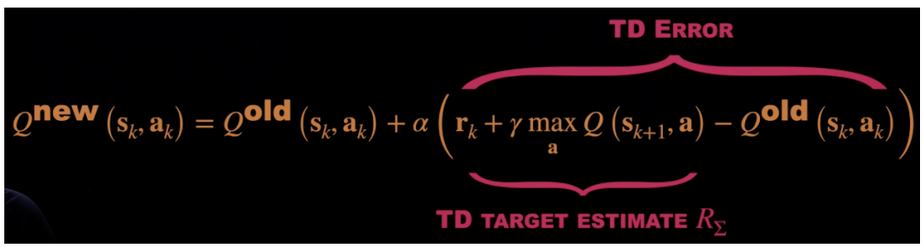


Q-Learning is the temporal difference learning on the Q function.

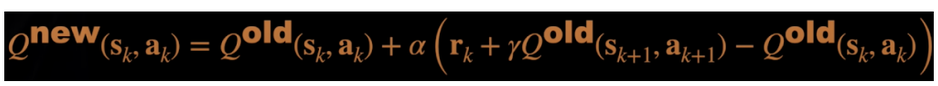
Q function is about, given State s and Action a, and assuming doing the best possible in the future, what is the Quality of being at that State s and taking that Action a.

If Q function is known, in State s we can check all Action a and pick an Action a that results in the best Quality.

Compared to SARSA, since Q-Learning assumes doing the best possible in the future, we can learn from taking a sub-optimal current step (i.e. explore "Off Policy" experiences).



SARSA:



Q-Learning process in below,

Initialization (First iteration):

For all couples of states s and actions a, the Q-values are initialized to 0.

Next iterations:

At each iteration t ≥ 1, we repeat the following steps:

1. We select a random state s_t from the possible states.
2. From that state, we play a random action a_t.
3. We reach the next state s_{t+1} and we get the reward $R(s_t, a_t)$.
4. We compute the Temporal Difference $TD_t(s_t, a_t)$:

   $$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. We update the Q-value by applying the Bellman equation:

   $$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

Deep Q-Learning

Action a is selected based on the NN's output softmax probability

From: https://www.youtube.com/watch?v=wDVteayWWvU

$$Q^{\text{new}}(\mathbf{s}_k, \mathbf{a}_k) = Q^{\text{old}}(\mathbf{s}_k, \mathbf{a}_k) + \alpha \left(\mathbf{r}_k + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{k+1}, \mathbf{a}) - Q^{\text{old}}(\mathbf{s}_k, \mathbf{a}_k) \right)$$

$$Q(\mathbf{s}, \mathbf{a}) \approx Q(\mathbf{s}, \mathbf{a}, \boldsymbol{\theta}) \qquad \textbf{PARAMETERIZE Q FUNCTION WITH NN}$$

$$\mathscr{L} = \mathbb{E}\left[\left(\mathbf{r}_k + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{k+1}, \mathbf{a}_{k+1}, \boldsymbol{\theta}) - Q(\mathbf{s}_k, \mathbf{a}_k, \boldsymbol{\theta}) \right)^2 \right]$$

From: https://www.youtube.com/watch?v=MQ6pP65o7OM

Given a transition < *s, a, r, s'* >, the Q-table update rule in the previous algorithm must be replaced with the following:

- Do a feedforward pass for the current state *s* to get **predicted Q-values for all actions**
- Do a feedforward pass for the next state *s'* and calculate maximum overall network outputs ***max*** $_{a'}$ ***Q(s', a')***
- Set Q-value target for action to ***r + γmax*** $_{a'}$ ***Q(s', a')*** (use the max calculated in step 2).
  - For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
- Update the weights using backpropagation.

- Experience Replay
  - Stores experiences (actions, state transitions, and rewards) and creates mini-batches from them for the training process
- Fixed Target Network
  - Error calculation includes the target function depends on network parameters and thus changes quickly. Updating it only every 1,000 steps increases stability of training process.

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha \left[r_{t+1} + \gamma \max_{p} Q(s_{t+1}, p) - Q(s_t, a) \right]$$

target Q function in the red rectangular is fixed

- Reward Clipping
  - To standardize rewards across games by setting all positive rewards to +1 and all negative to -1.
- Skipping Frames
  - Skip every 4 frames to take action

Deep Q-Learning process in below,

Initialization:

1. The memory of the Experience Replay is initialized to an empty list M.
2. We choose a maximum size of the memory.

At each time t, we repeat the following process, until the end of the epoch:

1. We predict the Q-values of the current state s_t.
2. We play the action that has the highest Q-value: $a_t = \underset{a}{\text{argmax}}\{Q(s_t, a)\}$
3. We get the reward $R(s_t, a_t)$.
4. We reach the next state s_{t+1}.
5. We append the transition (s_t, a_t, r_t, s_{t+1}) in the memory M.
6. We take a random batch $B \subset M$ of transitions. For all the transitions $(s_{t_B}, a_{t_B}, r_{t_B}, s_{t_B+1})$ of the random batch B:

   We get the predictions: $Q(s_{t_B}, a_{t_B})$

   We get the targets: $R(s_{t_B}, a_{t_B}) + \gamma \max_a(Q(s_{t_B+1}, a))$

   We compute the loss between the predictions and the targets over the whole batch B:

$$\text{Loss} = \frac{1}{2}\sum_B \left(R(s_{t_B}, a_{t_B}) + \gamma \max_a(Q(s_{t_B+1}, a)) - Q(s_{t_B}, a_{t_B})\right)^2 = \frac{1}{2}\sum_B TD_{t_B}(s_{t_B}, a_{t_B})^2$$

   We backpropagate this loss error back into the neural network, and through stochastic gradient descent we update the weights according to how much they contributed to the loss error.

Policy Gradient and Actor-Critic

A Twin Delayed DDPG model has two networks, Critic network for computing Q values, and Actor (Policy) network for computing Actions.

A Policy network takes States and outputs Actions.

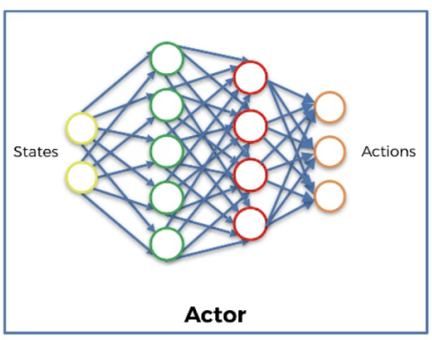
Return R_t is the sum of reward (across the time). R_0 is when t =0.

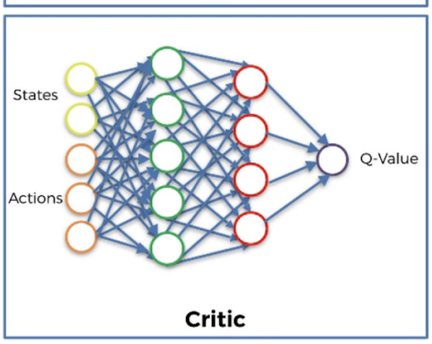
J(Φ) is the Expected return: State s follows a certain probability distribution; Action a follows what's returned by the Policy network π.

Φ is the parameters/weights in the Policy network π.

Compute the gradient of the expected return with respect to the parameters Φ.

The goal of the gradient ascent is to update Φ so J is maximized.





Return: $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$

Goal: Maximize the expected return $J(\phi) = \mathbb{E}_{s_i \sim p_\pi, a_i \sim \pi}[R_0]$

Deterministic Policy Gradient:

- In actor-critic methods, the policy, known as the actor, can be updated through the deterministic policy gradient algorithm:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s \sim p_\pi}\left[\nabla_a Q^\pi(s, a)|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)\right]$$

- We update the policy parameters through gradient ascent:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi J(\pi_\phi)|_{\phi_t}$$

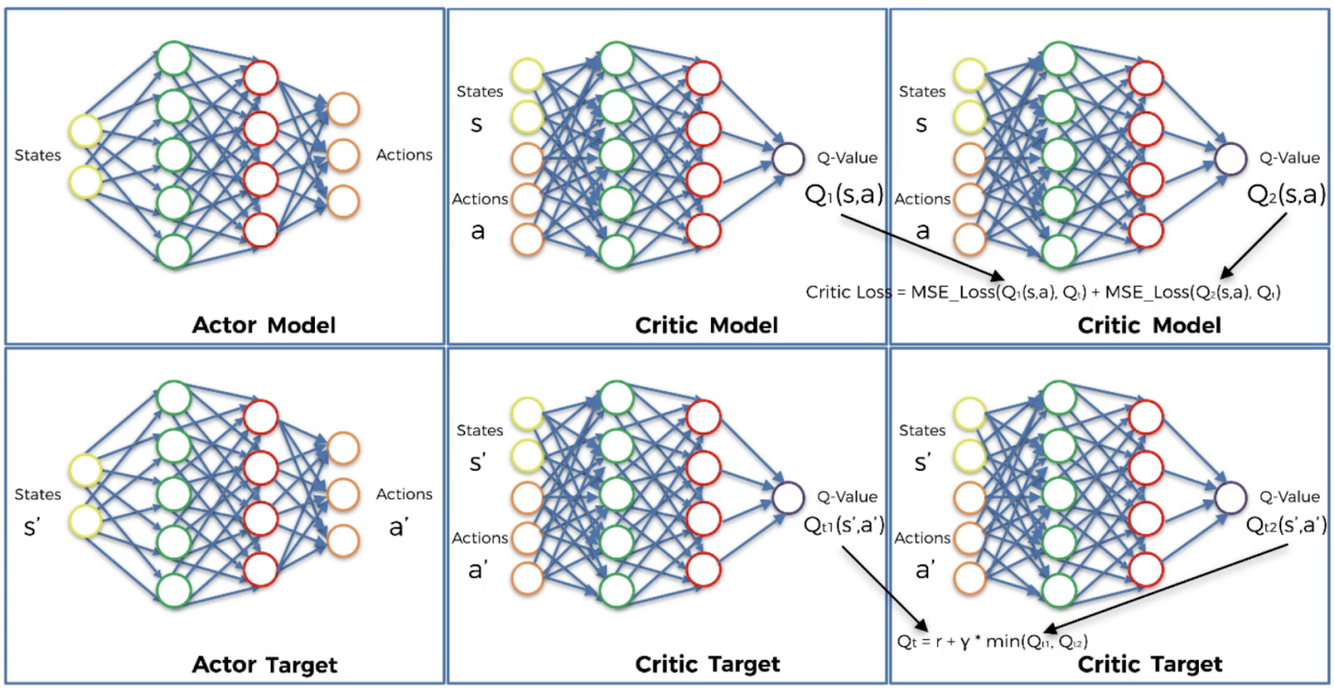
	Deep Q-Learning	Policy Gradient	Actor-Critic	World Models
Model-Free vs. Model-Based	Model-Free	Model-Free	Model-Free	Model-Based
Value-Based vs. Policy-Based	Value-Based	Policy-Based	Both	Policy-Based
Off-Policy vs. On-Policy	Off-Policy	On-Policy	Off-Policy	On-Policy

Twin-Delayed DDPG (TD3)

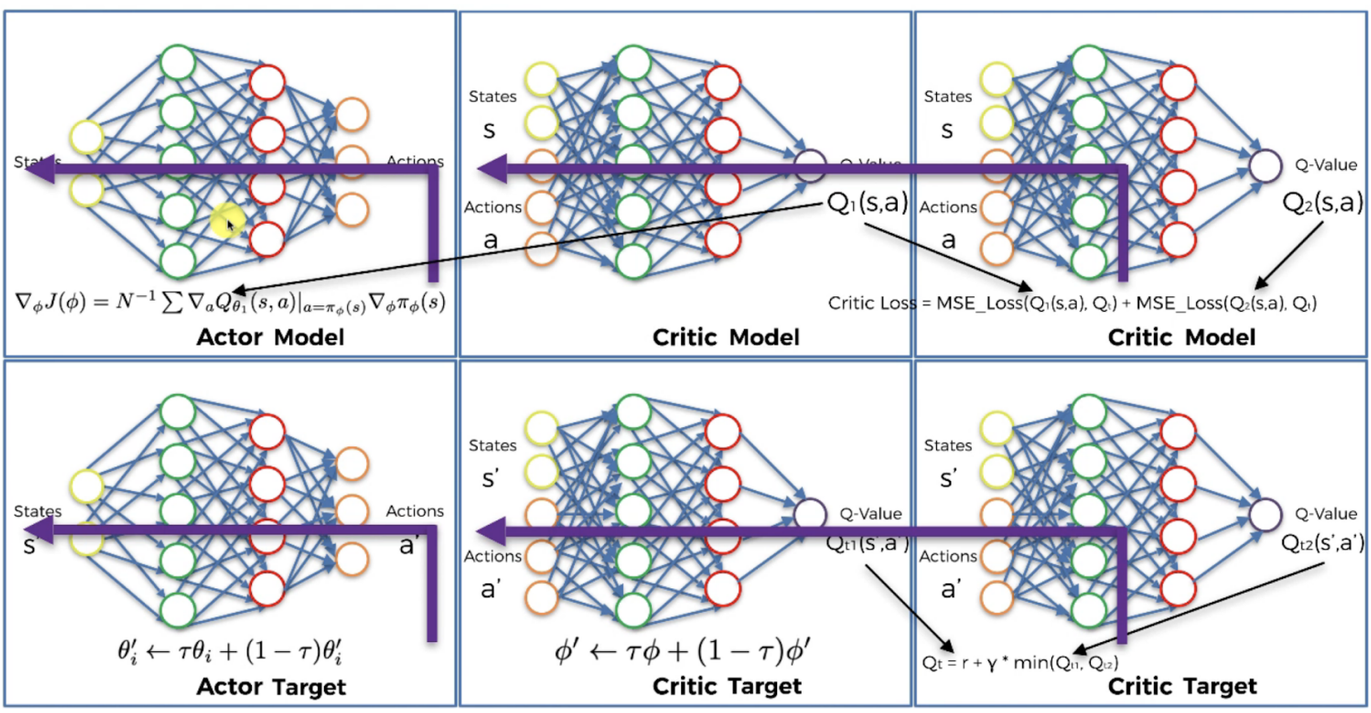
DDPG stands for Deep Deterministic Policy Gradient.

Two Actor Models learn about the Policy.

Four Critic Models learn about the Q value.



In below, the purple arrows stand for back propagation



Initialization:
Step 1: We initialize the Experience Replay memory, with a size of 20000. We will populate it with each new transition.
Step 2: We build one neural network for the Actor model and one neural network for the Actor target.
Step 3: We build two neural networks for the two Critic models and two neural networks for the two Critic targets.

Training Process - We run a full episode with first 10,000 actions played randomly, and then with actions played by the Actor model. Then we repeat the following steps:
Step 4: We sample a batch of transitions (s, s', a, r) from the memory. Then for each element of the batch:
Step 5: From the next state s', the Actor target plays the next action a'.
Step 6: We add Gaussian noise to this next action a' and we clamp it in a range of values supported by the environment.
Step 7: The two Critic targets take each the couple (s', a') as input and return two Q-values $Q_{t1}(s',a')$ and $Q_{t2}(s',a')$ as outputs.
Step 8: We keep the minimum of these two Q-values: $\min(Q_{t1}, Q_{t2})$. It represents the approximated value of the next state.
Step 9: We get the final target of the two Critic models, which is: $Q_t = r + \gamma * \min(Q_{t1}, Q_{t2})$, where γ is the discount factor.
Step 10: The two Critic models take each the couple (s, a) as input and return two Q-values $Q_1(s,a)$ and $Q_2(s,a)$ as outputs.
Step 11: We compute the loss coming from the two Critic models: Critic Loss = $\text{MSE\_Loss}(Q_1(s,a), Q_t) + \text{MSE\_Loss}(Q_2(s,a), Q_t)$
Step 12: We backpropagate this Critic loss and update the parameters of the two Critic models with a SGD optimizer.
Step 13: Once every two iterations, we update our Actor model by performing gradient ascent on the output of the first Critic model: $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s,a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$, where ϕ and θ_1 are resp. the weights of the Actor and the Critic.
Step 14: Still once every two iterations, we update the weights of the Actor target by polyak averaging: $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
Step 15: Still once every two iterations, we update the weights of the Critic target by polyak averaging: $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$