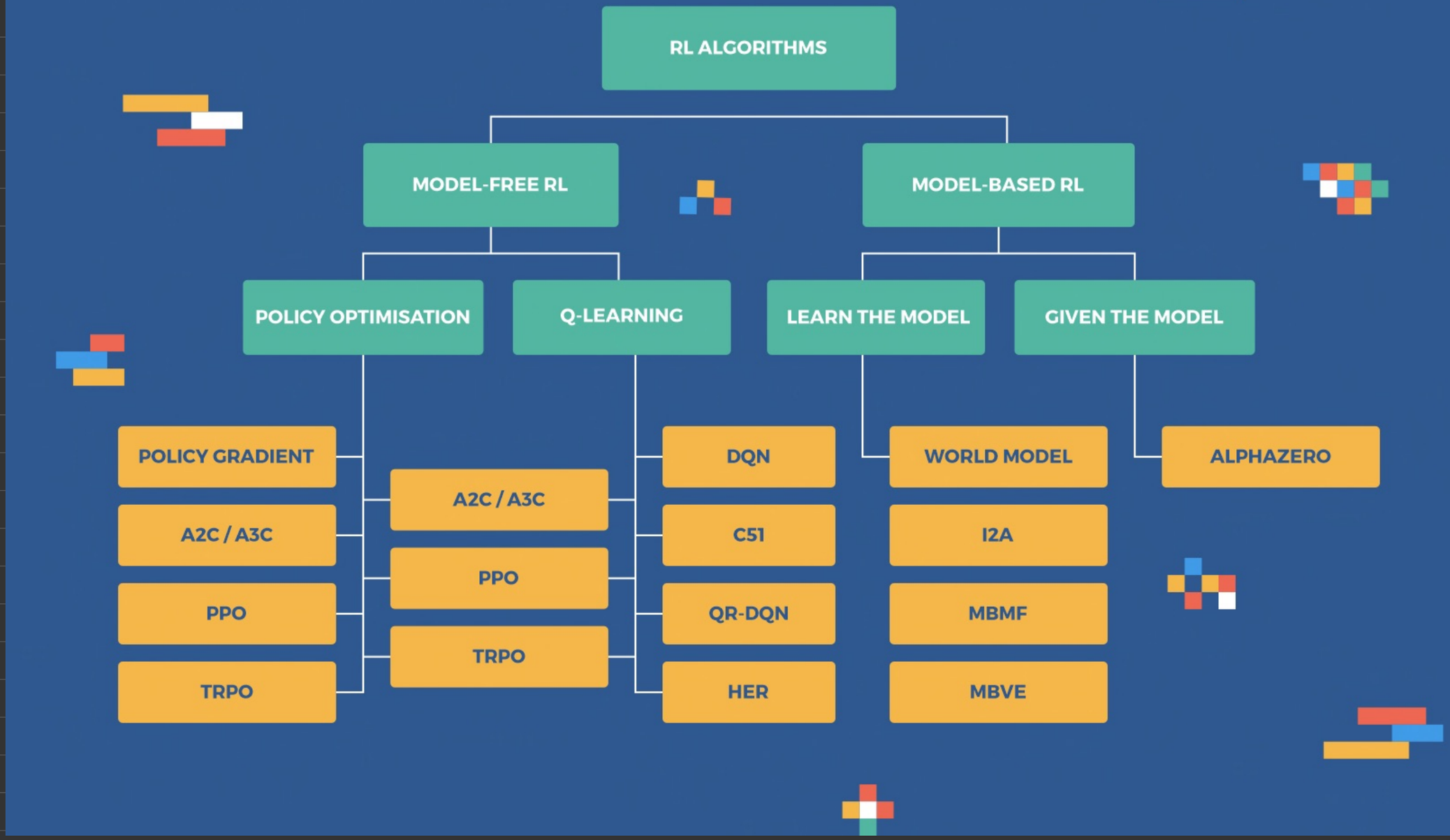
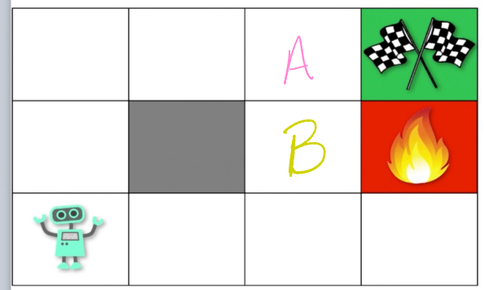
REINFORCEMENT LEARNING ALGORITHMS MAP



Fundamentals: Q-Learning



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 $TD_t(s_t, a_t)$:

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

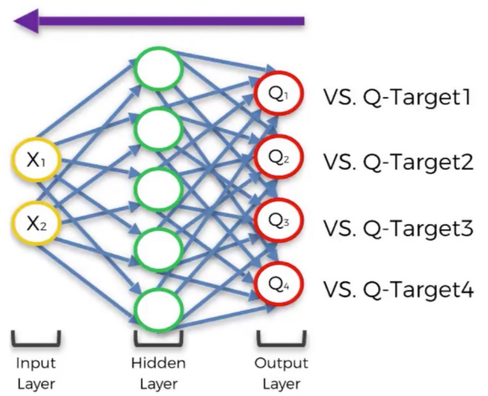
5. We update the Q-value by applying the Bellman equation:

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

Deep Reinforcement Learning 2.0 © SuperDataSCience

It will learn the Q values which is based on Q learning and it will

Fundamentals: Deep Q-Learning



$$L = \sum (Q\text{-}Target - Q)^2$$

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.

Deep Reinforcement Learning 2.0 © SuperDataSCience

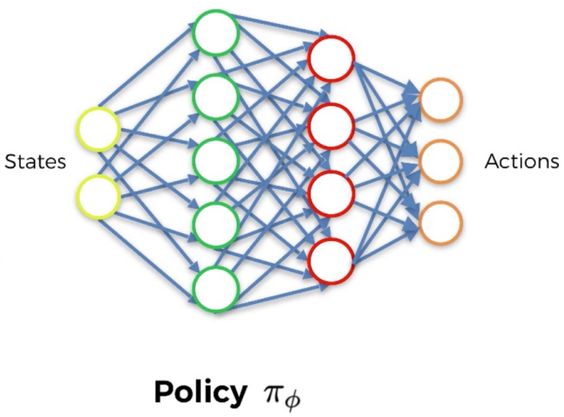
- TD3 will learn two values at same time
  - The Q-value
  - the parameters to Policy, so that make better choices

- The Q-value Represents the quality of being in a state S and take an action a.

- The Q-value of being in state A and take the action Right will be the highest of all others, because it move the agent to the goal.

- In contrast the Q-value of being in state B and took action Right will be the lowest, because the agent will loose the game.

Fundamentals: Policy Gradient



Policy π_ϕ

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

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

Policy Gradient:

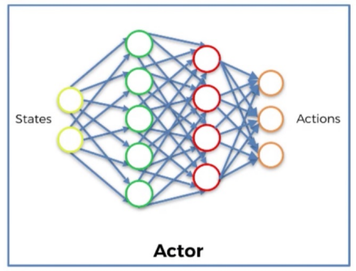
- We compute the gradient of the expected return with respect to the parameters ϕ

$$\nabla_\phi J(\phi)$$

- We update the parameters through gradient ascent:

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

Fundamentals: Actor-Critic



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:

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

- In the previous model we try to estimate the Q-value.
- Given the Q-value the Agent apply some policy to choose the best action.
- The policy may be pick the action that gives the higher reward $\arg\max_a [Q(s,a)]$
- Now this kind of algo try to learn the best policy, not the best Q-value.

- In this model we have two networks
  1. Actor "Policy" get the states and output the possible actions
  2. The critic that get States, Actions and produces the Q-value
- Actor try to find the best action.
- Critic try to find the best Q-values.

Fundamentals: Taxonomy of AI models

Model-Free vs. Model-Based

Value-Based vs. Policy Based

Off-Policy vs. On-Policy

Deep Reinforcement Learning 2.0 © SuperDataScience

Model Free

- Do not try to learn how the env work.
- Just take the env output, do some process and choose the action.
- Q-learning, Actor-Critic and most of the Algo.

Model Based

- Try to understand the env and create a model to represent it
- The model is trying to capture the transition func and the reward func.

If, after learning the Agent can make predictions about what the next state/Reward is before it do the action than it's a MODEL-BASED, otherwise if it can't predict what next state/Reward would, it's a MODEL FREE

Value Based

- Agent calculates the Q-value and choose the action that results in the highest Reward.
- the agent doesn't learn the action, it learn the value, and the policy will be choose the highest action value. $\text{Argmax}_a[Q(s,a)]$

Policy Based

- Instead of predict the Q-value to later choose the action, It predict the action directly from observed state.

Off-Policy

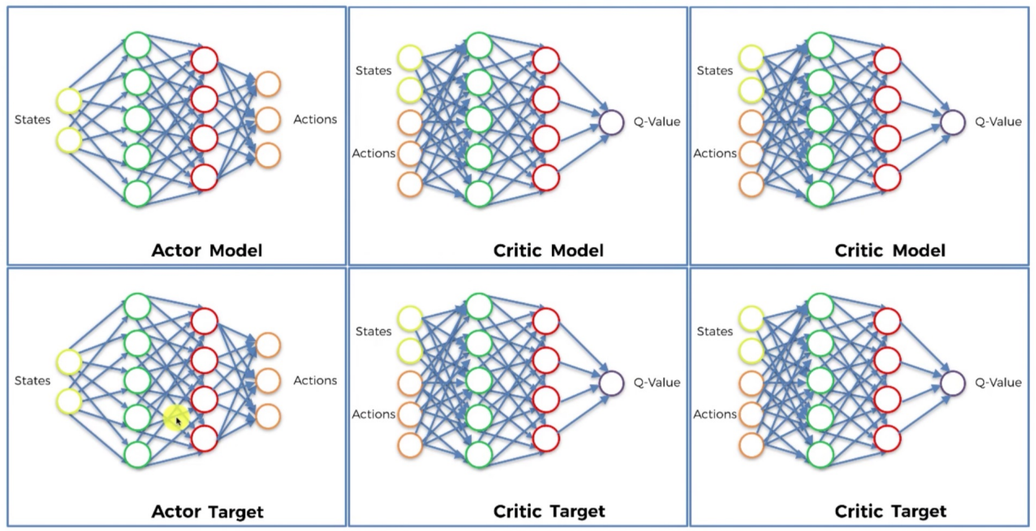
- use a experience Replay Memory to store past transitions so that it can sample and learn from the history

On-Policy

- Do not store past transitions. Learn just from new observations.

	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)



Deep Reinforcement Learning 2.0 © SuperDataScience

- This algo produces continuous outputs, instead of discret output actions as Q_learning
- Remember the $\max_a [Q(s,a)]$ on Q_learning. This only works in discrete number of actions a

- The Actor predicts the Policy
- the Critic predicts the Q-values

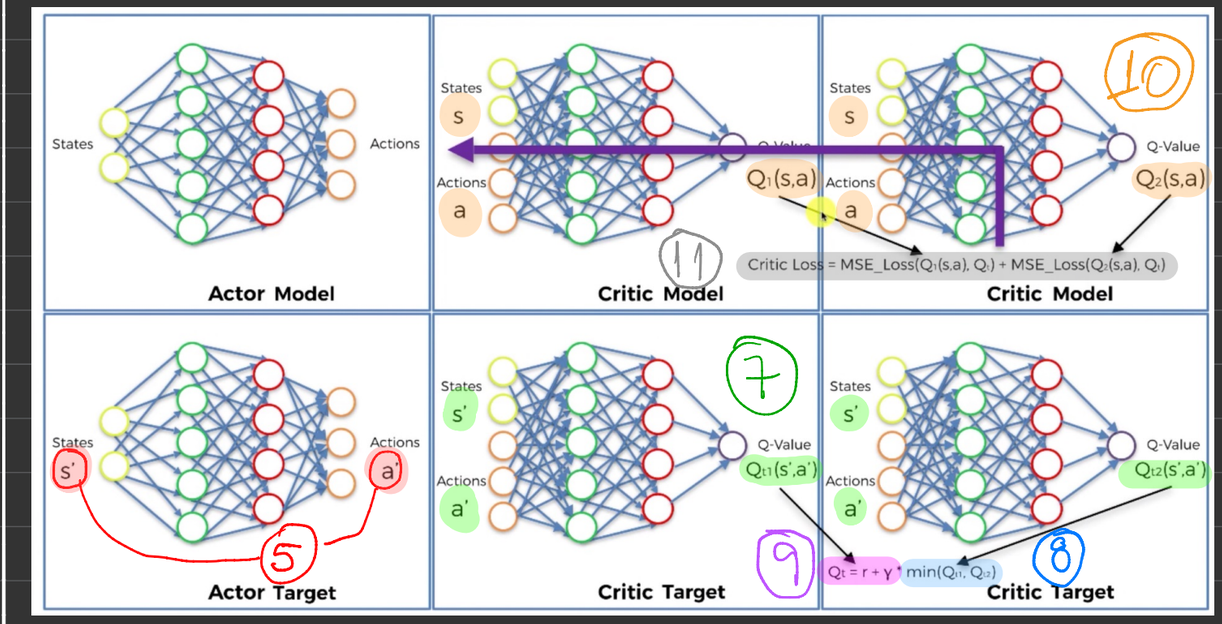
Initialization:
Step 1: We initialize the Experience Replay memory, with a size of 20000. We will populate it with each new transition.
Step 2: We build one neural network for the Actor model and one neural network for the Actor target.
Step 3: We build two neural networks for the two Critic models and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

Q-Learning Step

Twin-Delayed DDPG (TD3)

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 = MSE_Loss($Q_1(s,a)$, Q_t) + 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.



8)
- Taking the $\min(Q_{t1}, Q_{t2})$ is the same as the $\max_a [Q(s',a)]$ of Q_learning

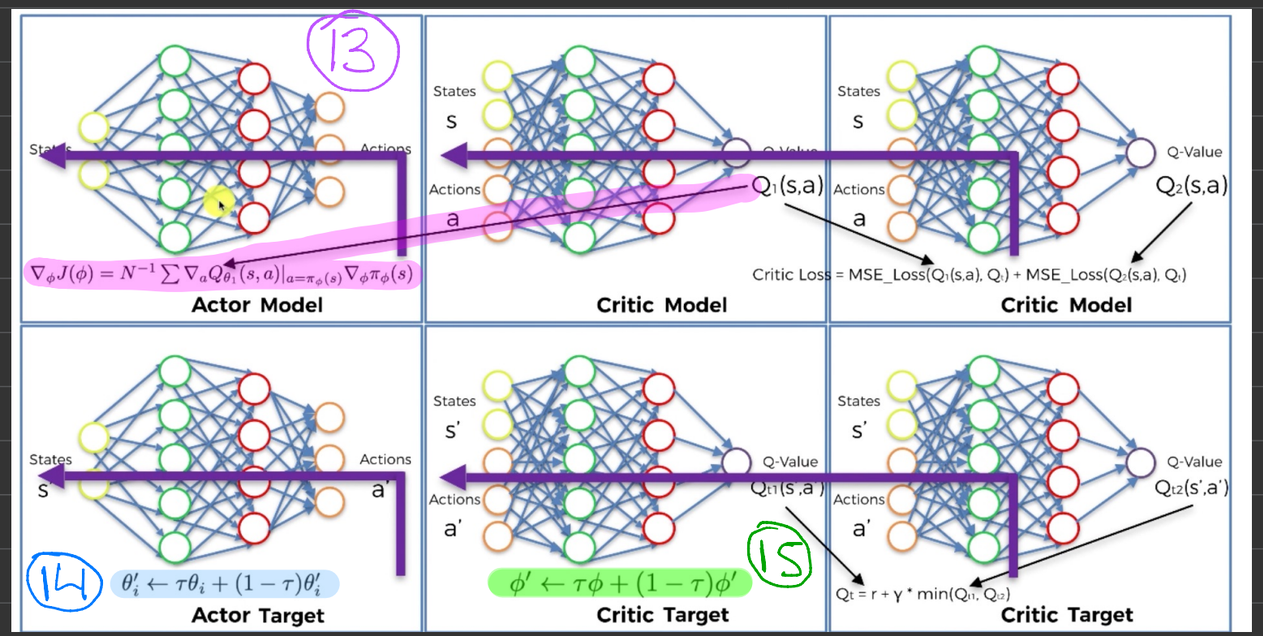
$$\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$$

$$\tilde{a} \leftarrow \text{clip}(\tilde{a})$$

- Also taking the min() helps to AVOID over estimate the target Q_values.

Policy learning Step

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: $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$



- Polyak uses a small number τ 0.8. This means we update the target network with a very small portion of Model weights, instead of Replace the entire target weights by the Target.

- Gradient Ascent to maximize the value

- In Step 13 we do:

```
a = Actor(state)
critic_out = Critic_1(state, a)
critic_out = - critic_out.mean()
critic_out.backward()
Actor_opt.step()
```

→ Reduce to a single value

→ Perform Grad. Asc. to update the actor weights so that it produces an Action that Results in a higher $Q_{1a}(s,a)$

```python
def train(self, iterations, batch_size=100, gamma=0.99, tau=0.005,
          policy_noise=0.2, noise_clip=0.5, policy_freq=2):
    if self.replay_buffer.mem_cntr < batch_size:
        return

    for i in range(iterations):
        # 4. Sample from memory
        batch_state, batch_action, batch_reward, batch_state_, batch_done = self.replay_buffer.sample(batch_size)
        state = torch.tensor(batch_state, dtype=torch.float, device=device)
        action = torch.tensor(batch_action, dtype=torch.float, device=device)
        reward = torch.tensor(batch_reward, dtype=torch.float, device=device).view(-1, 1)
        state_ = torch.tensor(batch_state_, dtype=torch.float, device=device)
        done = torch.tensor(batch_done, dtype=torch.bool, device=device).view(-1, 1)

        # 5. Actor target play state_
        action_ = self.actor_target(state_)

        # 6. Add gaussian noise the same shape as action (bs, n_actions)
        noise = torch.Tensor(batch_action).data.normal_(0, policy_noise).clamp(-noise_clip, noise_clip).to(device)
        action_ = (action_ + noise).clamp(-self.max_action, self.max_action)

        # 7. Critic target play with state_, action_
        q1_target, q2_target = self.critic_target(state_, action_)

        # 8. Min of q1_target and q2_target
        min_q = torch.min(q1_target, q2_target)  # remove from backpropagation

        # 9. Calculate bellman equation
        q_target = reward + (gamma * min_q).detach()
        q_target[done] = 0.0

        # 10. Critic play with state, action
        q1, q2 = self.critic(state, action)

        # 11. calculate critic loss of each Critic and the q_target
        critic_loss_1 = torch.nn.functional.mse_loss(q1, q_target)
        critic_loss_2 = torch.nn.functional.mse_loss(q2, q_target)
        critic_loss = critic_loss_1 + critic_loss_2

        # 12. backpropagation
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # 13. Delayed update of Actor model by perform gradient ascent trying to
        if i % policy_freq == 0:
            actor_loss = - self.critic.q1(state, self.actor(state)).mean()
            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            # 14. Update actor_target using Polyak Average
            for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
                target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)

            # 15. Update critic_target using Polyak Average
            for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
                target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```