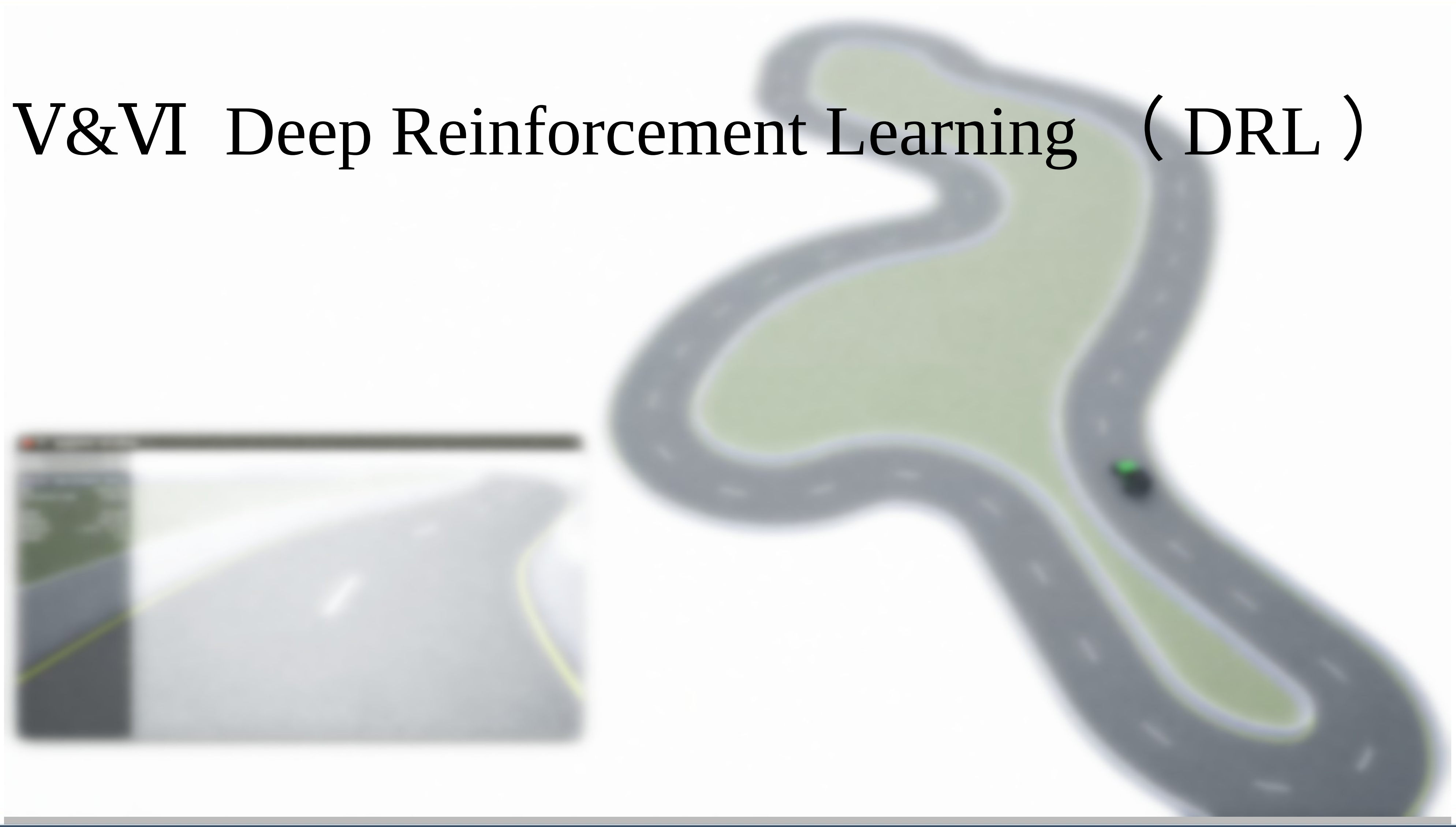


V&VI Deep Reinforcement Learning （DRL）

Outline

1. Introduce the Deep Reinforcement Learning
2. Double Deep Q Network (Double-DQN value-based)
3. Playing Carla with Double-DQN
4. Deep Deterministic Policy Gradient (DDPG Policy Gradient)
5. Playing Carla with DDPG

1. Deep Reinforcement Learning

- **Investigation:**
  ① Have you heard of reinforcement learning ?
  ② Know how RL works ?
  ③ RL algorithm: DQN, Double-DQN ,Dueling DQN, Actor – Critic, DDPG ?
  ④ Reproduce the algorithm by python ?

- **What is RL ?**

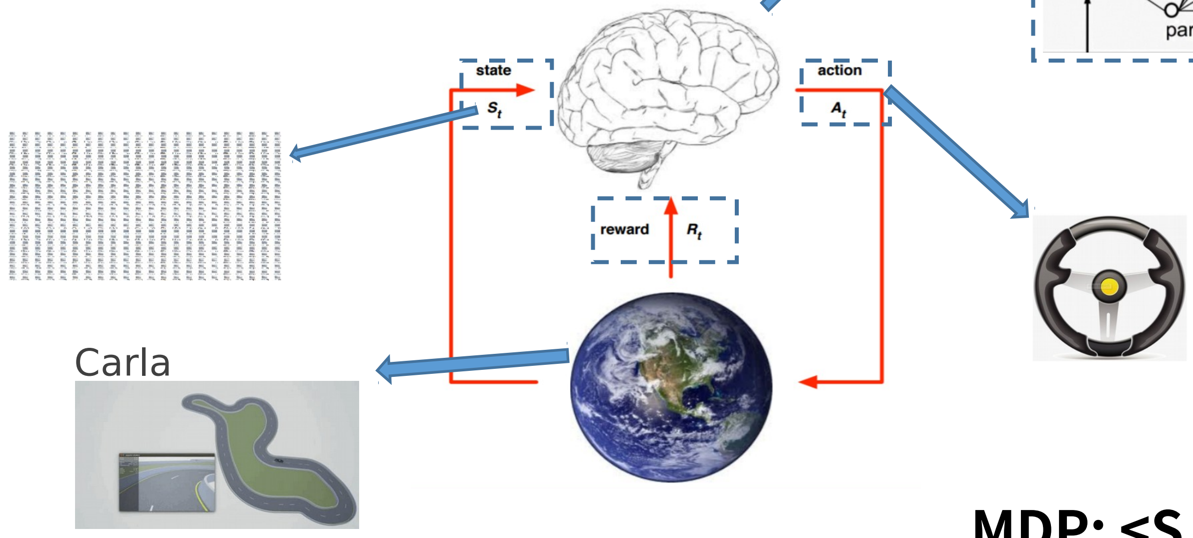
environment
agent
actions
rewards
observations

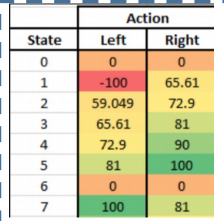
Let the robot learn **strategies** (**Maximum total reward**) by **interacting with the environment .**

- **Descript RL Process** :

Markov Decision Process

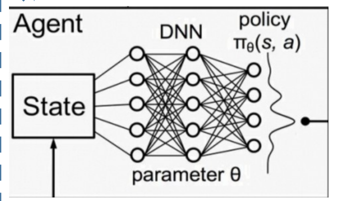
Agent





	Action	
State	Left	Right
0	0	0
1	-100	65.61
2	59.049	72.9
3	65.61	81
4	72.9	90
5	81	100
6	0	0
7	100	81

RL □ Deep RL



MDP: <S,A,P,R,γ>

1. Deep Reinforcement Learning

- **DRL algorithms:**

Value-based RL (Max the total reward)

Action discrete

Double-DQN :

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}[(Q^*(s,a|\theta) - y)^2]$$

$$y = r + \gamma \max_{a'} \overline{Q^*}\ (s',\ a')$$

Improved Algorithm：

① Fixed target
② Double DQN
③ Dueling DQN

Instance：
Action = [‘Up’ , ‘down’ , ‘left’ , ‘right’]

Policy Gradient (Strategies)

Action Continuous

PG: **PG:**

$$\nabla_\theta J(\theta) = \mathrm{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)r]$$

Basic PG algorithm:

① REINFORNCE: Monte Carlo ;
② Actor-Critic: TD-error ;

Improved Algorithm：

① More actors A2C, A3C
② Replay buffer PPO **DDPG**

Instance：
Action: torque = [-2, 2]

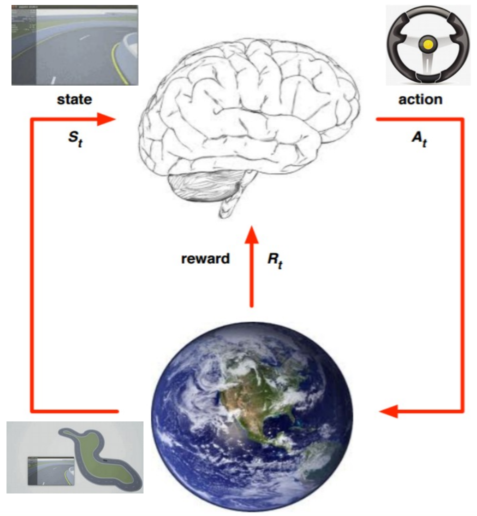
Way To Innovation

2. Value-based: DQN

- **Project Analysis：**

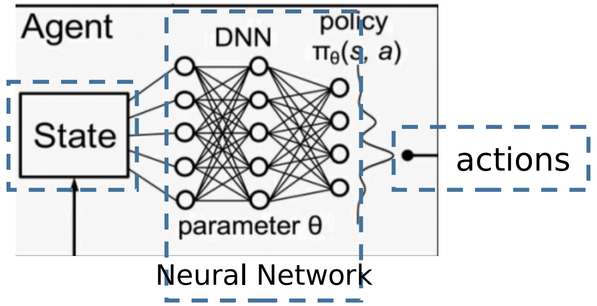
  Goal： Vehicle runs **on the race road** by **itself**.

  Two sub-project： Carla environment & Policy by Double DQN



- **Carla Env**

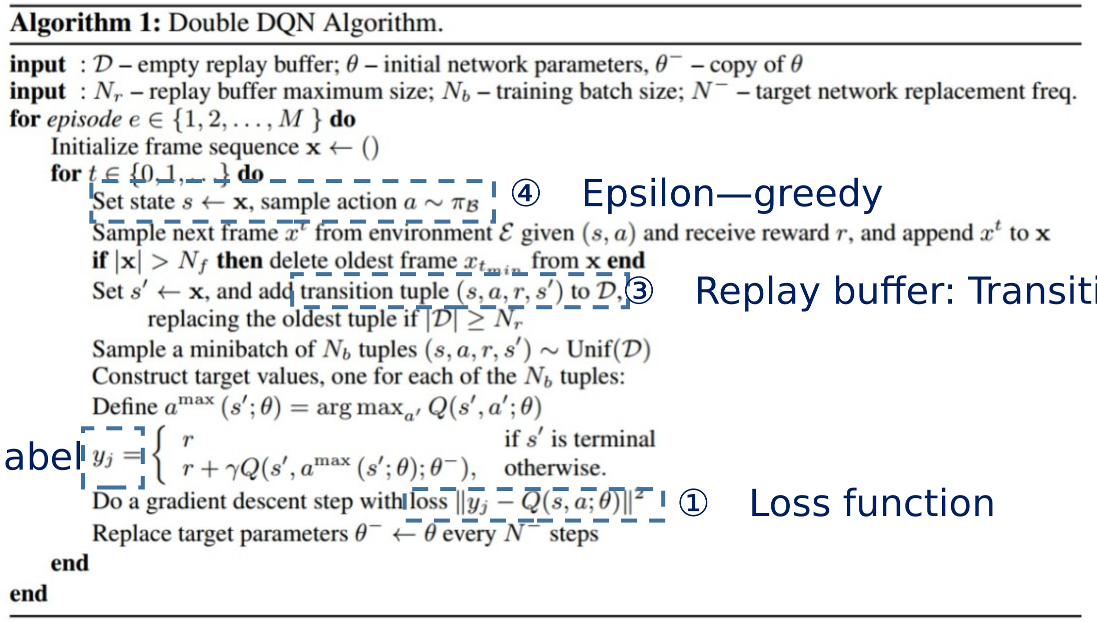
  ① Agent (Vehicle)



  ② State (rgb-image sensor)

  ③ Action (steer, brake, throttle)

- **Double-DQN (Intuitively)**



```
Algorithm 1: Double DQN Algorithm.
input : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
input : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
for episode $e \in \{1, 2, \ldots, M\}$ do
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \ldots\}$ do
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$    ④ Epsilon—greedy
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ then delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ end
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,    ③ Replay buffer: Transition
            replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
② Y label  $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
        Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$    ① Loss function
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
    end
end
```

② **Y label**

$$Q(s,a;\theta) = \underbrace{r}_{\text{Reward}} + \underbrace{\gamma Q(s', argmax_{a'} Q(s', a'; \theta); \theta')}_{\text{Next state Q-value}}$$

Double

2. Value-based: DQN

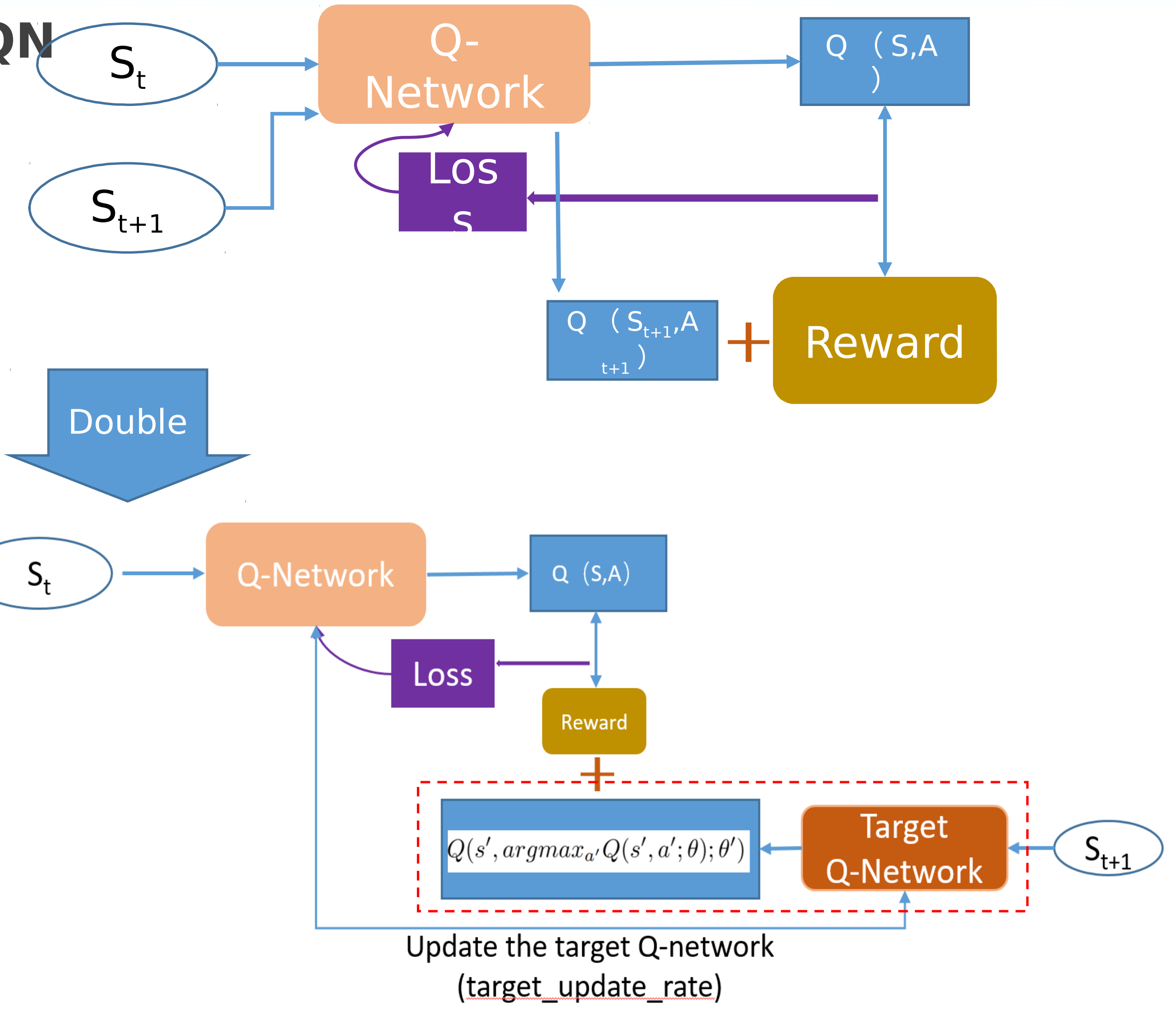
- **Naïve DQN – Fixed DQN – Double DQN**

① **Naïve DQN** ② **Fixed DQN**

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

③ **Double DQN**

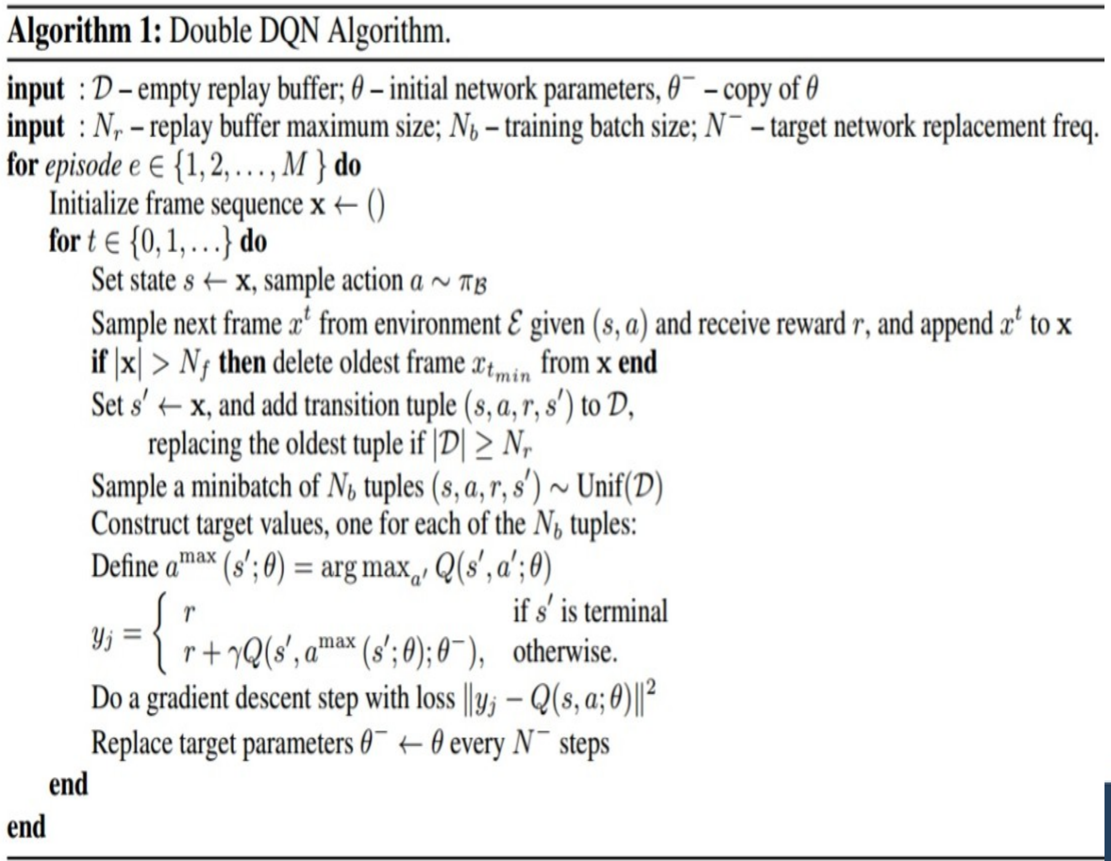
$$Q(s,a;\theta) = r + \gamma Q(s', argmax_{a'} Q(s',a';\theta);\theta')$$



2. Value-based: DQN

- **Double DQN: CartPole**

States:[pos, vel, theta, w_vel]
Action:[0: left, 1:right]
reward:[0: game over,
1: keep moving]



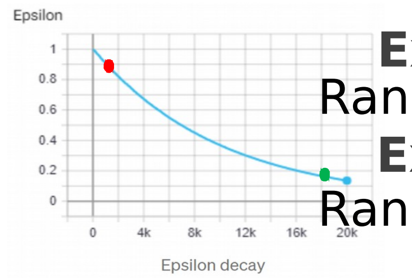
Algorithm 1: Double DQN Algorithm.

```
input : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
input : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
for episode $e \in \{1, 2, \ldots, M\}$ do
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ do
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ then delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ end
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
        replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the $N_b$ tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\| y_j - Q(s, a; \theta) \|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
  end
end
```

① **Replay buffer**

transition(s,a,r,s’), memory_size， mini-batch

② **Epsilon Greedy (Exploration-- Exploitation)**



Exploration
Random(0,1) < red point value
Exploitation
Random(0,1) > red point value

③ **Interact with environment**

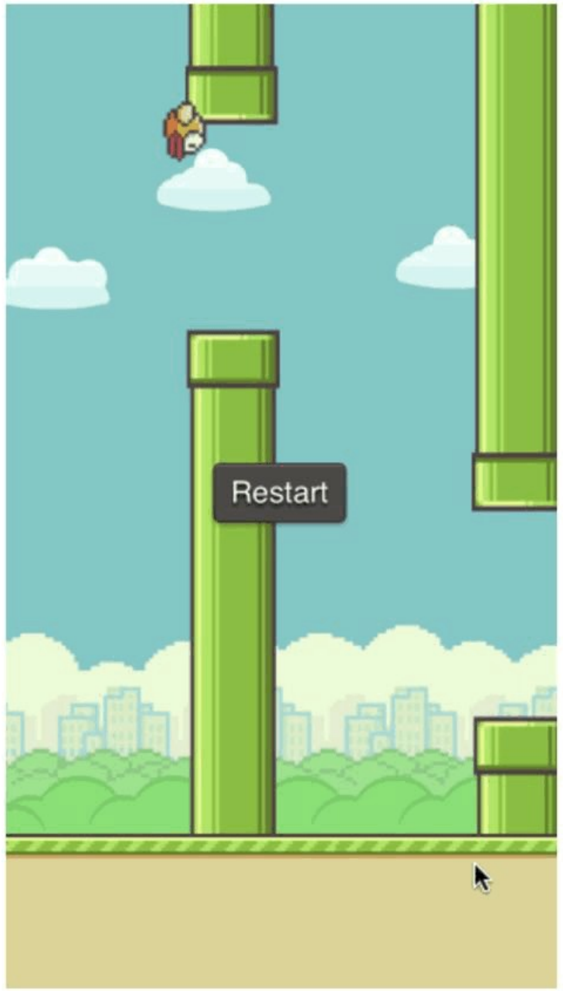
Action□ Next state □ reward □ Done

④ **Double NETWORK**

Q-network (Fully connected) predicts value & target va

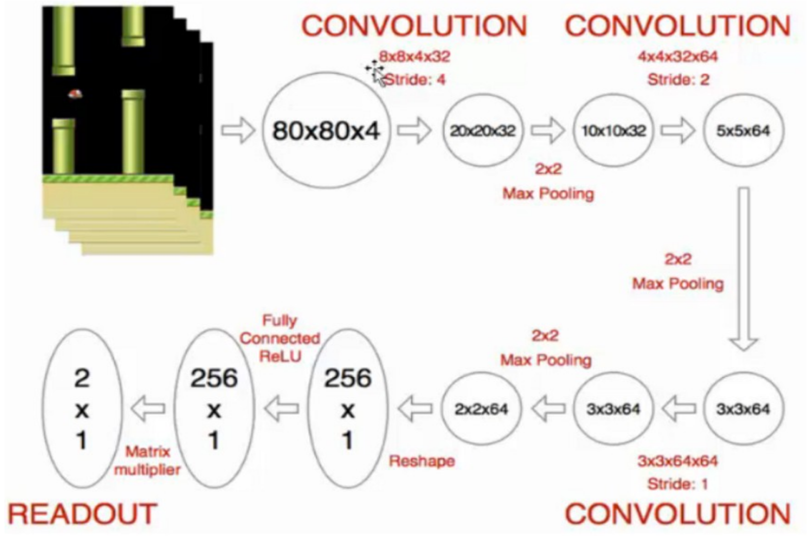
2. Value-based: DQN

- **Double DQN: Flappy bird (image)**

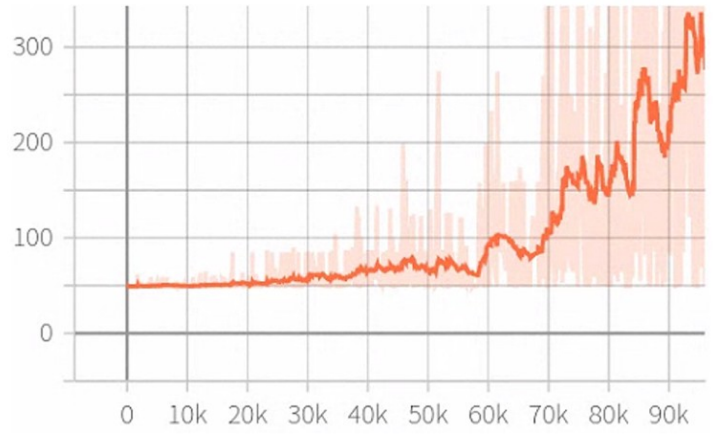


States ? Action ? Reward ? Done ?

DQN: CNN



4 frames CNN CNN CNN FC FC



Episode reward

Tricks:

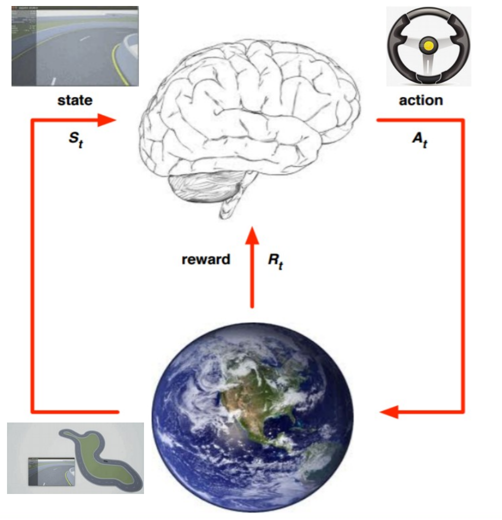
① Rgb image gray image;
② Resize image;
③ 4 frame;

HyperParameter:

① Max episode;
② Memory size;
③ Batch size;
④ Epsilon decay;
⑤ Target network update
⑥ Learning rate;

2. Value-based: DQN

- **Carla step by step：**



① Create a Carla Environment;
  a) Spawn a vehicle
  b) Spawn a rgb-camera attaching to vehicle

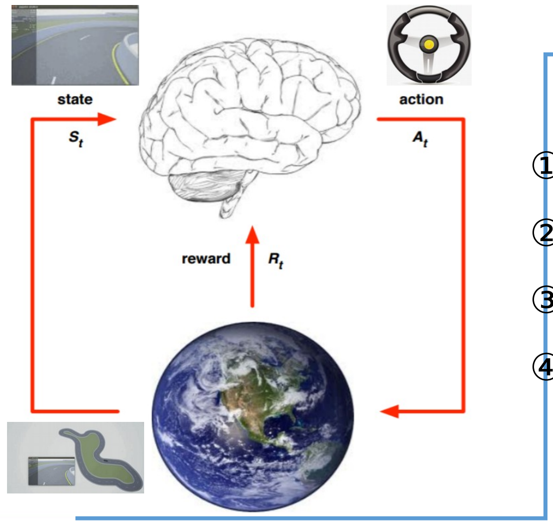
② Build a CNN network (input: image);
  a) CNN network
  b) State dimension and action dimension

③ Achieve Double DQN algorithm to train CNN network;
  a) Epsilon-greedy policy – action;
  b) Interact with env
  c) Transition (s, a, r, s’)
  d) Training network by replay buffer

④ Use the CNN (Policy) to control the vehicle;
  a) Load the CNN params;
  b) Interact with Carla env;

2. Value-based: DQN

- **Create a Carla Environment :**



CarlaEnv Class:

① Reset： spawn vehicle and sensor for each episode ;

② Collision_data: Vehicle have a collision and this episode is done;

③ Process_image: Record a env state;

④ Step: agent interacts with the Carla simulator;

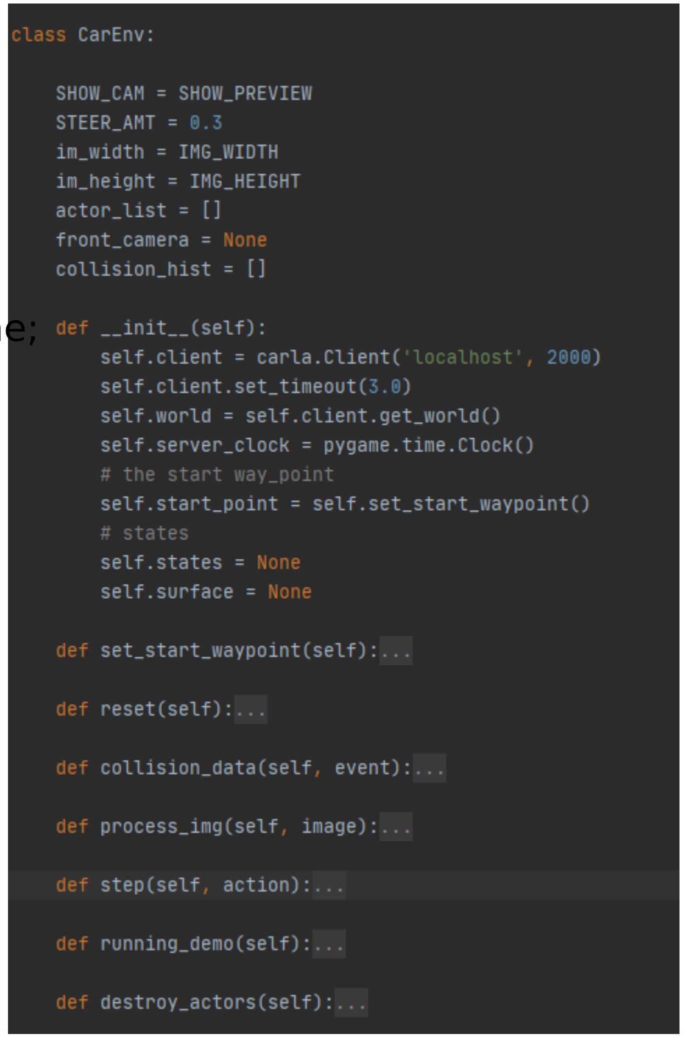
  a) Apply a control based on the action;

  b) Design a reward for the action;

  c) The episode is done or not;

⑤ Find the startpoint;

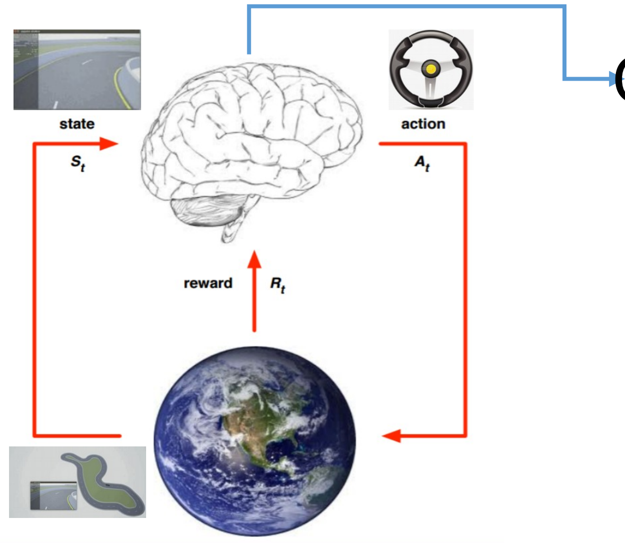
⑥ Destroy the actors (vehicle and sensor);



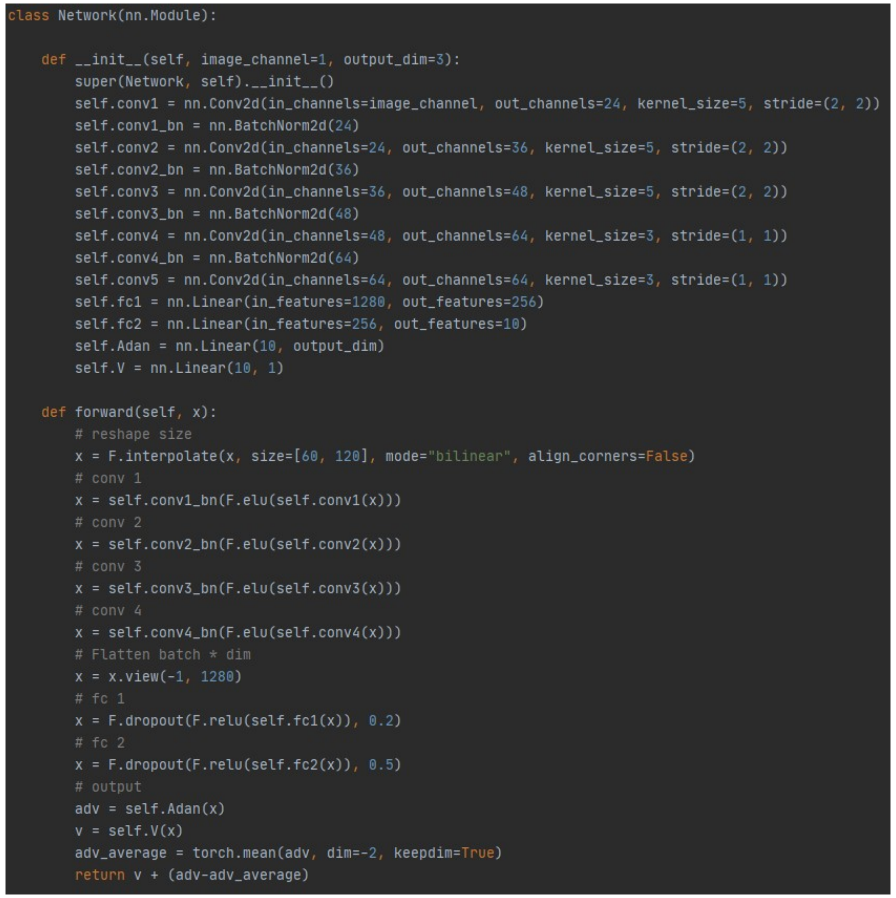
```
class CarEnv:

    SHOW_CAM = SHOW_PREVIEW
    STEER_AMT = 0.3
    im_width = IMG_WIDTH
    im_height = IMG_HEIGHT
    actor_list = []
    front_camera = None
    collision_hist = []

    def __init__(self):
        self.client = carla.Client('localhost', 2000)
        self.client.set_timeout(3.0)
        self.world = self.client.get_world()
        self.server_clock = pygame.time.Clock()
        # the start way_point
        self.start_point = self.set_start_waypoint()
        # states
        self.states = None
        self.surface = None

    def set_start_waypoint(self):...

    def reset(self):...

    def collision_data(self, event):...

    def process_img(self, image):...

    def step(self, action):...

    def running_demo(self):...

    def destroy_actors(self):...
```

2. Value-based: DQN

- **Build a CNN network :**



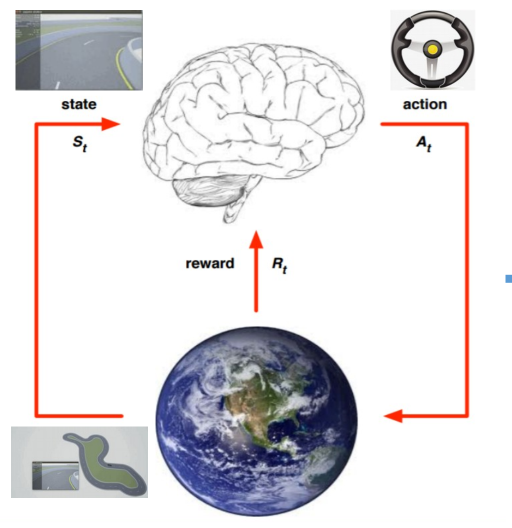
CNN + Fully Connected



```python
class Network(nn.Module):

    def __init__(self, image_channel=1, output_dim=3):
        super(Network, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=image_channel, out_channels=24, kernel_size=5, stride=(2, 2))
        self.conv1_bn = nn.BatchNorm2d(24)
        self.conv2 = nn.Conv2d(in_channels=24, out_channels=36, kernel_size=5, stride=(2, 2))
        self.conv2_bn = nn.BatchNorm2d(36)
        self.conv3 = nn.Conv2d(in_channels=36, out_channels=48, kernel_size=5, stride=(2, 2))
        self.conv3_bn = nn.BatchNorm2d(48)
        self.conv4 = nn.Conv2d(in_channels=48, out_channels=64, kernel_size=3, stride=(1, 1))
        self.conv4_bn = nn.BatchNorm2d(64)
        self.conv5 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=(1, 1))
        self.fc1 = nn.Linear(in_features=1280, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=10)
        self.Adan = nn.Linear(10, output_dim)
        self.V = nn.Linear(10, 1)

    def forward(self, x):
        # reshape size
        x = F.interpolate(x, size=[60, 120], mode="bilinear", align_corners=False)
        # conv 1
        x = self.conv1_bn(F.elu(self.conv1(x)))
        # conv 2
        x = self.conv2_bn(F.elu(self.conv2(x)))
        # conv 3
        x = self.conv3_bn(F.elu(self.conv3(x)))
        # conv 4
        x = self.conv4_bn(F.elu(self.conv4(x)))
        # Flatten batch * dim
        x = x.view(-1, 1280)
        # fc 1
        x = F.dropout(F.relu(self.fc1(x)), 0.2)
        # fc 2
        x = F.dropout(F.relu(self.fc2(x)), 0.5)
        # output
        adv = self.Adan(x)
        v = self.V(x)
        adv_average = torch.mean(adv, dim=-2, keepdim=True)
        return v + (adv-adv_average)
```

2. Value-based: DQN

- **Training Network by Double DQN:**



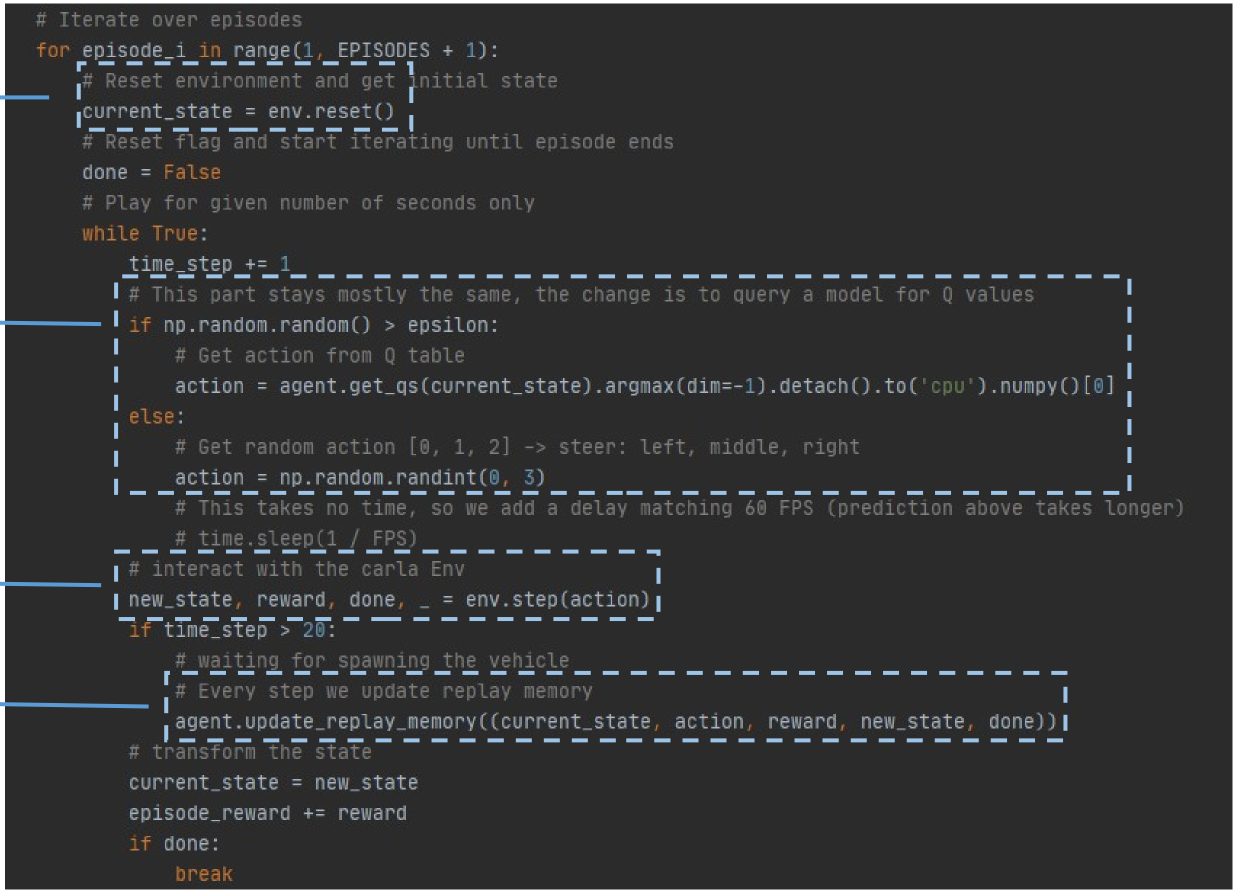
Collect data

Obtain the current state (rbg-image)

Epsilon-greedy

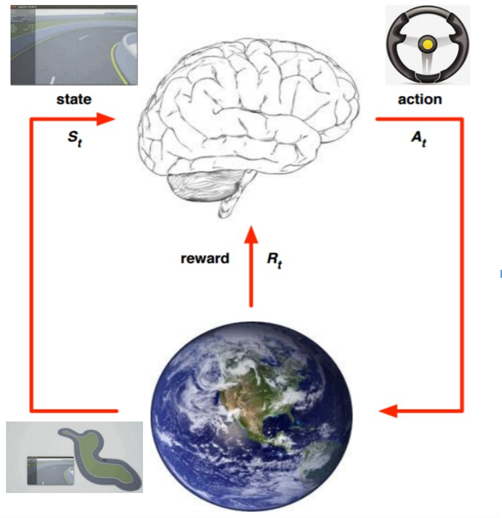
Agent interacts with Carla Env

Replay Buffer

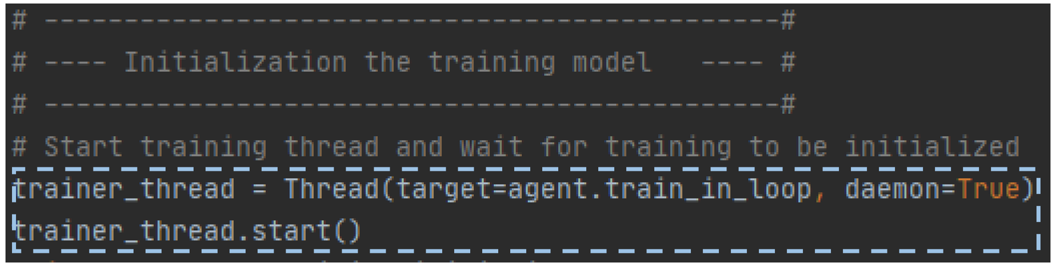

```
# Iterate over episodes
for episode_i in range(1, EPISODES + 1):
    # Reset environment and get initial state
    current_state = env.reset()
    # Reset flag and start iterating until episode ends
    done = False
    # Play for given number of seconds only
    while True:
        time_step += 1
        # This part stays mostly the same, the change is to query a model for Q values
        if np.random.random() > epsilon:
            # Get action from Q table
            action = agent.get_qs(current_state).argmax(dim=-1).detach().to('cpu').numpy()[0]
        else:
            # Get random action [0, 1, 2] -> steer: left, middle, right
            action = np.random.randint(0, 3)
            # This takes no time, so we add a delay matching 60 FPS (prediction above takes longer)
            # time.sleep(1 / FPS)
        # interact with the carla Env
        new_state, reward, done, _ = env.step(action)
        if time_step > 20:
            # waiting for spawning the vehicle
            # Every step we update replay memory
            agent.update_replay_memory((current_state, action, reward, new_state, done))
        # transform the state
        current_state = new_state
        episode_reward += reward
        if done:
            break
```

2. Value-based: DQN

- **Training Network by Double DQN:**

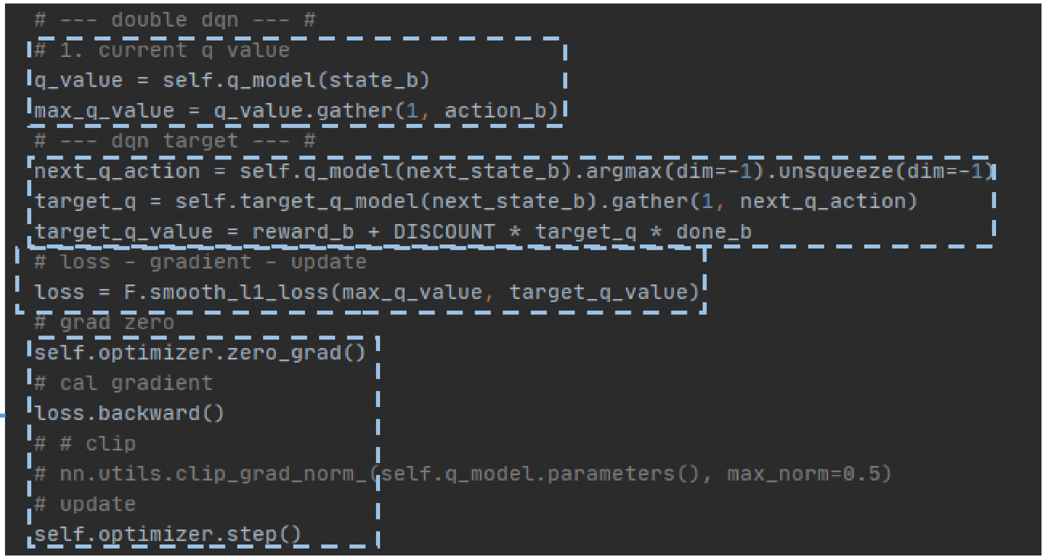


Training Model



```
# ---------------------------------------------#
# ---- Initialization the training model   ---- #
# ---------------------------------------------#
# Start training thread and wait for training to be initialized
trainer_thread = Thread(target=agent.train_in_loop, daemon=True)
trainer_thread.start()
```

Multi Threads



```
# --- double dqn --- #
# 1. current q value
q_value = self.q_model(state_b)
max_q_value = q_value.gather(1, action_b)
# --- dqn target --- #
next_q_action = self.q_model(next_state_b).argmax(dim=-1).unsqueeze(dim=-1)
target_q = self.target_q_model(next_state_b).gather(1, next_q_action)
target_q_value = reward_b + DISCOUNT * target_q * done_b
# loss - gradient - update
loss = F.smooth_l1_loss(max_q_value, target_q_value)
# grad zero
self.optimizer.zero_grad()
# cal gradient
loss.backward()
# # clip
# nn.utils.clip_grad_norm_(self.q_model.parameters(), max_norm=0.5)
# update
self.optimizer.step()
```

Q-value(s, a)

Target Q-value

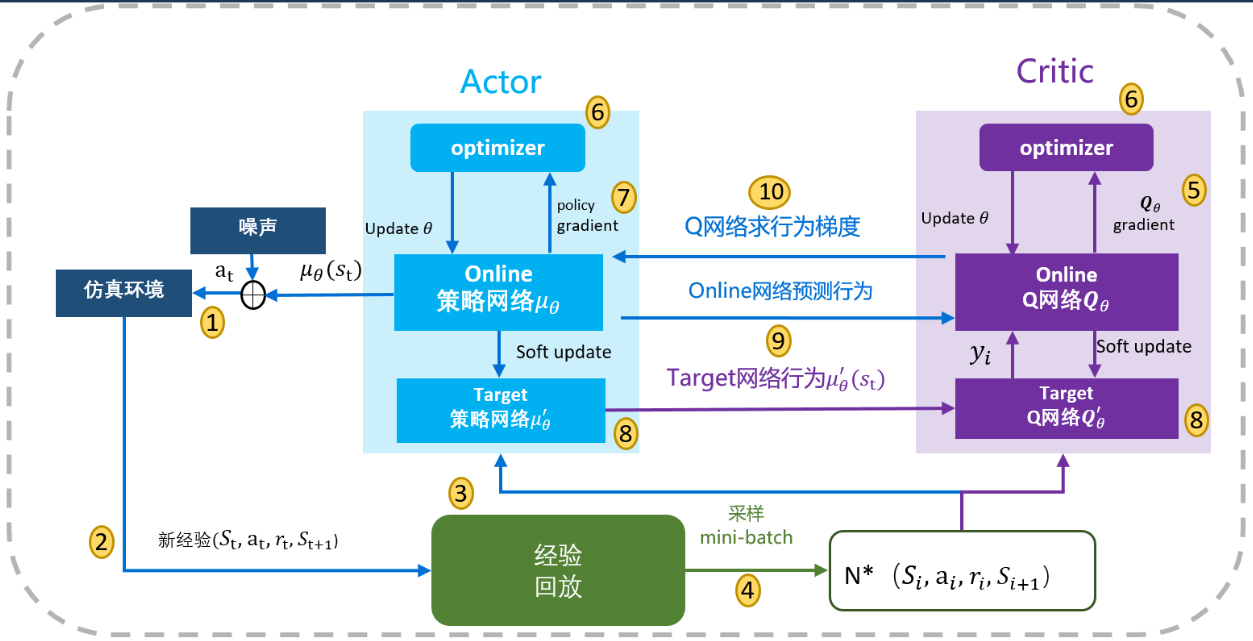
Loss function

Pytorch Gradient Descent

Way To Innovation

Deep Deterministic Policy Gradient (DDPG Policy Gradient)

- **DDPG**



① Action;
② Environment interaction new transition ;
③ Replay buffer ;
④ Mini-batch ;
⑤ Critic DQN gradient descent ;
⑥ Adam optimizer ;
⑦ Actor gradient ascent ;
⑧ Update the target network ;
⑨ Critic input： actor predicts action ;
⑩ Critic update actor parameter θ^μ;

Critic DQN

MSE：

$$L(\theta^Q) = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i \mid \theta^Q)\right)^2$$

y_i: target DQN label

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$$

Actor Policy Network

=

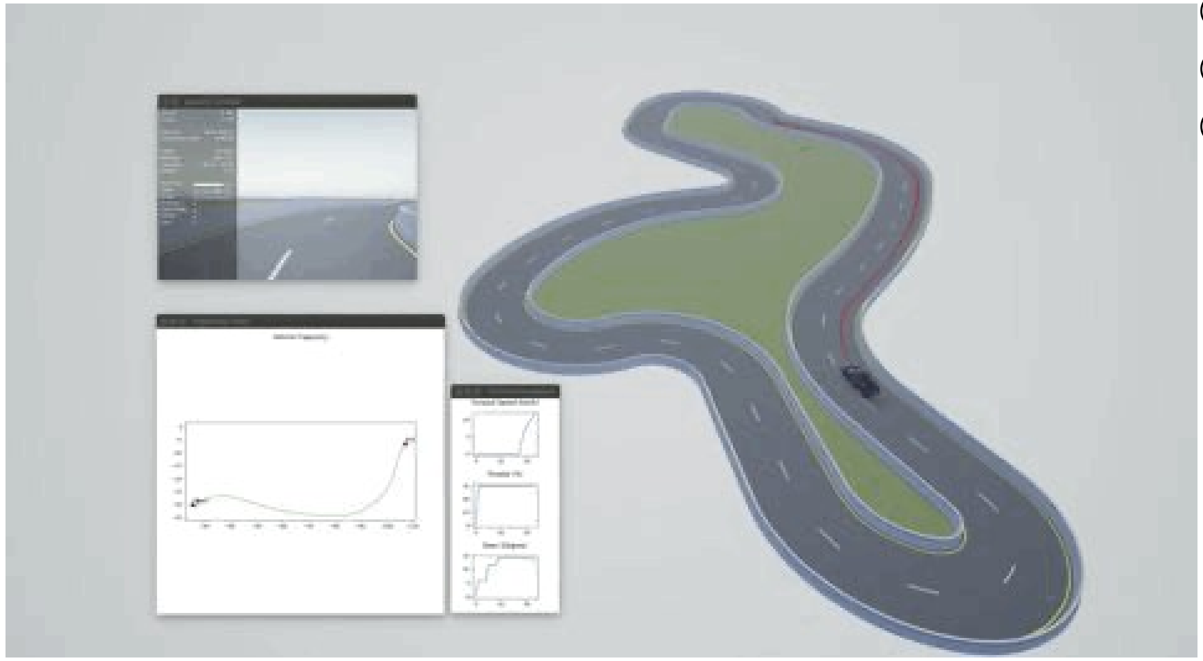
$$\nabla_{\theta^\mu} J = \frac{1}{N}\sum_i [\nabla_{\theta^\mu}\mu(s_i \mid \theta^\mu) * \nabla_a Q(s, a \mid \theta^Q)|_{a=\mu(s), s=s_i}]$$

$$L(\theta^\mu) = -\frac{1}{N}\sum_i Q(s, a)$$

$$\boldsymbol{a} = \boldsymbol{\mu}(\boldsymbol{s})$$

Project

Goal



Control Method

① Keyboard;
② PID;
③ Behavior Cloning;
④ Reinforcement learning (DQN);

Sensor

① RGB-image;
② Depth-image;
③ Lidar;

Just the beginning

- Improve the RL control
  - ✓ Steer, Throttle, brake and more ;
  - ✓ Policy gradient ;

- Autonomous driving license
  - ✓ Side parking ;
  - ✓ Reversing into the garage ;
  - ✓ Right angle bend;

- Multi-agent
  - ✓ V2V ;
  - ✓ V2X ;