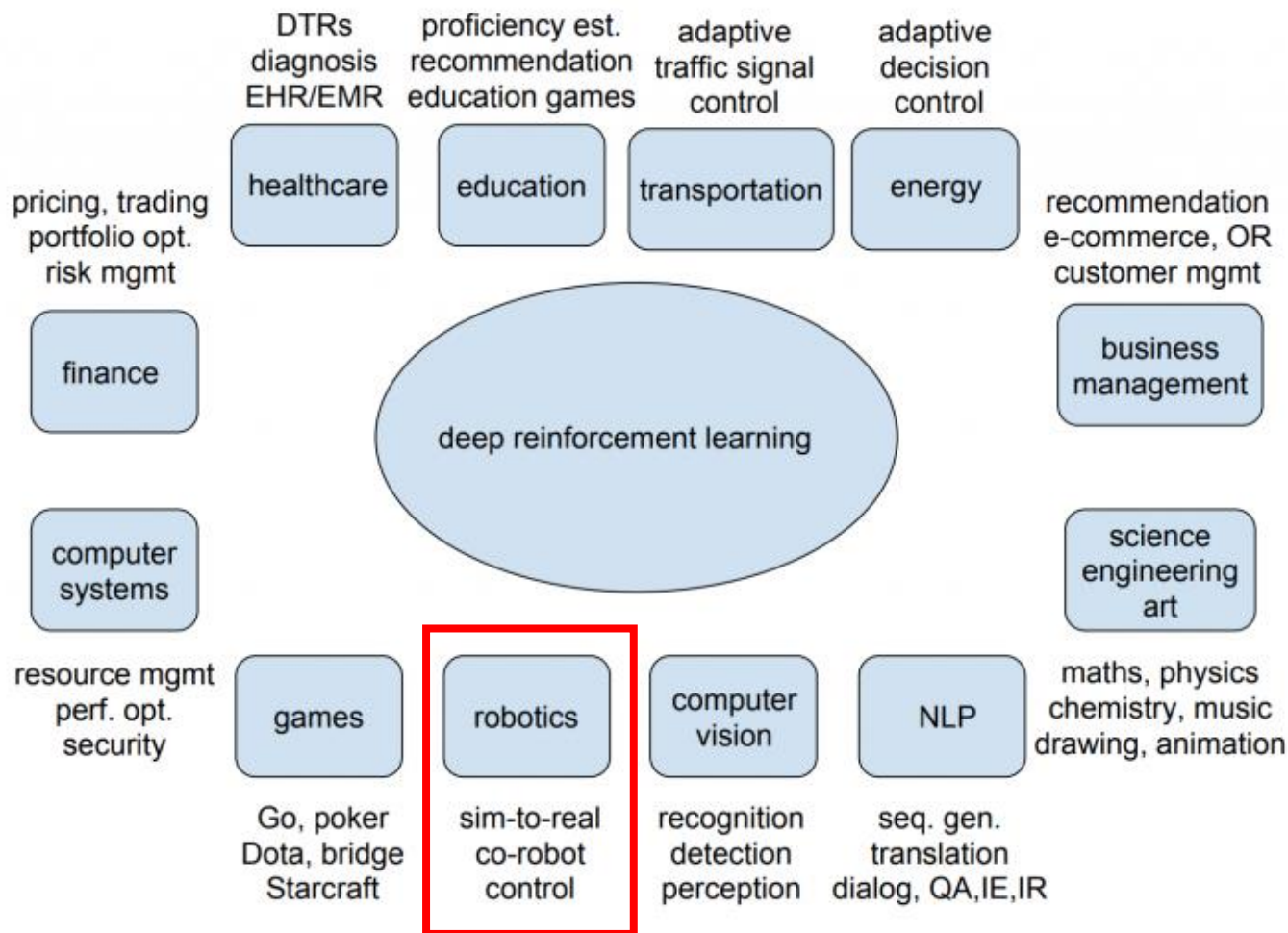




DEEP REINFORCEMENT LEARNING: DDPG + LEARNING FROM HUMAN EXPERIENCES

Usefulness of Deep RL



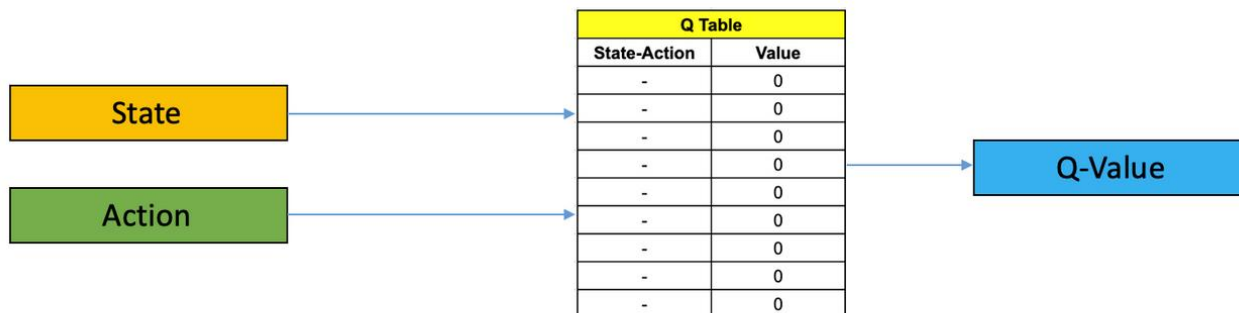
Using Q-Learning as an example:

- Non-deep methods not practical for environments with numerous states and possible actions per state (e.g. 10,000 states and 1,000 actions per state, which require a table of 10 million cells). Problems will arise:
 - Required memory will increase as no. of states increases
 - Unrealistic amount of time required to explore each state to fill in the Q-table
- Can utilize neural networks to approximate these Q-values → Deep Q-Learning

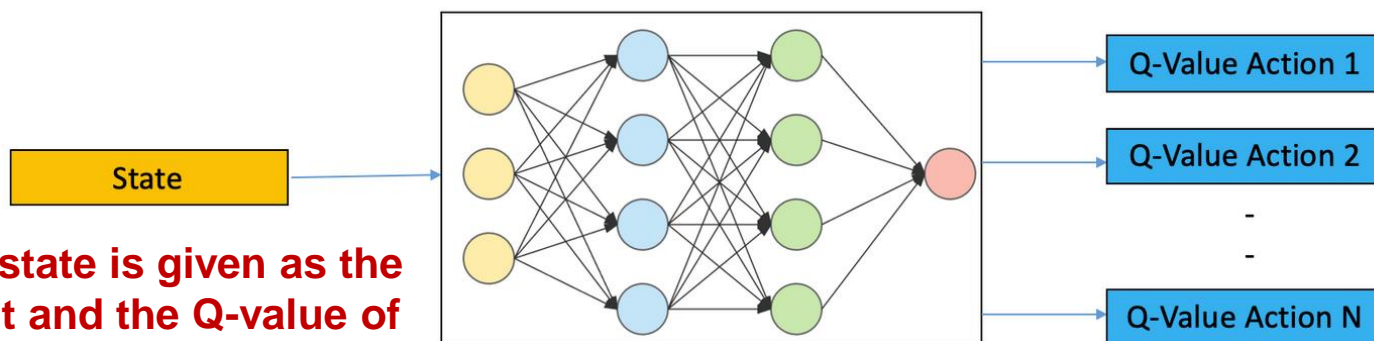


Non-Deep vs Deep RL

Using Q-Learning as an example:



Q Learning



The state is given as the input and the Q-value of all possible actions is generated as the output

Deep Q Learning



DDPG (Original)



- **Deep Deterministic Policy Gradient (DDPG)** = An algorithm which **concurrently learns a Q-function and a deterministic policy**
- Relies on the actor-critic architecture with two elements: **Actor and Critic**
- **The Critic estimates the value function** (i.e. action-value: the Q value for DDPG case), whereas **the Actor updates the policy distribution** in the direction suggested by the Critic (i.e. with policy gradients for DDPG case)
- **Uses off-policy¹ data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy**
- Can only be used for **environments with continuous action spaces** (e.g.TB3)
- **DDPG training performance can be improved with the help of human experiences**

1 – An off-policy learner learns the value of the optimal policy independently of the agent's actions; it figures out the optimal policy regardless of the agent's motivation; evaluate or improve a policy different from that used to generate the data; the policy that is used for updating the policy and the policy used for acting is NOT the same; e.g. Q-learning

In contrast, on-policy methods attempt to evaluate or improve the policy that is used to make decisions. The policy that is used for updating the policy and the policy used for acting is the same, unlike in Q-learning.



DDPG (Original)



Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' 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



Why DDPG for TB3?



- When there are a finite number of discrete actions (e.g. the robot in the MDP problem), the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them; this also immediately gives us the action which maximizes the Q-value
- But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. **Using a normal optimization algorithm is NOT practical as it would make calculating the optimal policy a painful and expensive process**
- Because the action space is continuous for the TB3, we need to **set up an efficient, gradient-based learning rule for a policy to approximate the optimal policy (as opposed to computing the true optimal policy)**



Self-Learning Autonomous Vehicles



Wayve; Drive by self-learning

Alex Kendall, Co-Founder & CTO of Wayve says:

“Our cars learn to drive from data with machine learning. Every time a safety driver intervenes and takes over, the car learns to drive better. We don’t tell the car how to drive, rather **it learns to drive from experience, example and feedback, just like a human.** This is **more safe and scalable than any other approach today.**”



Self-Learning Autonomous Vehicles



Wayve; Drive by self-learning



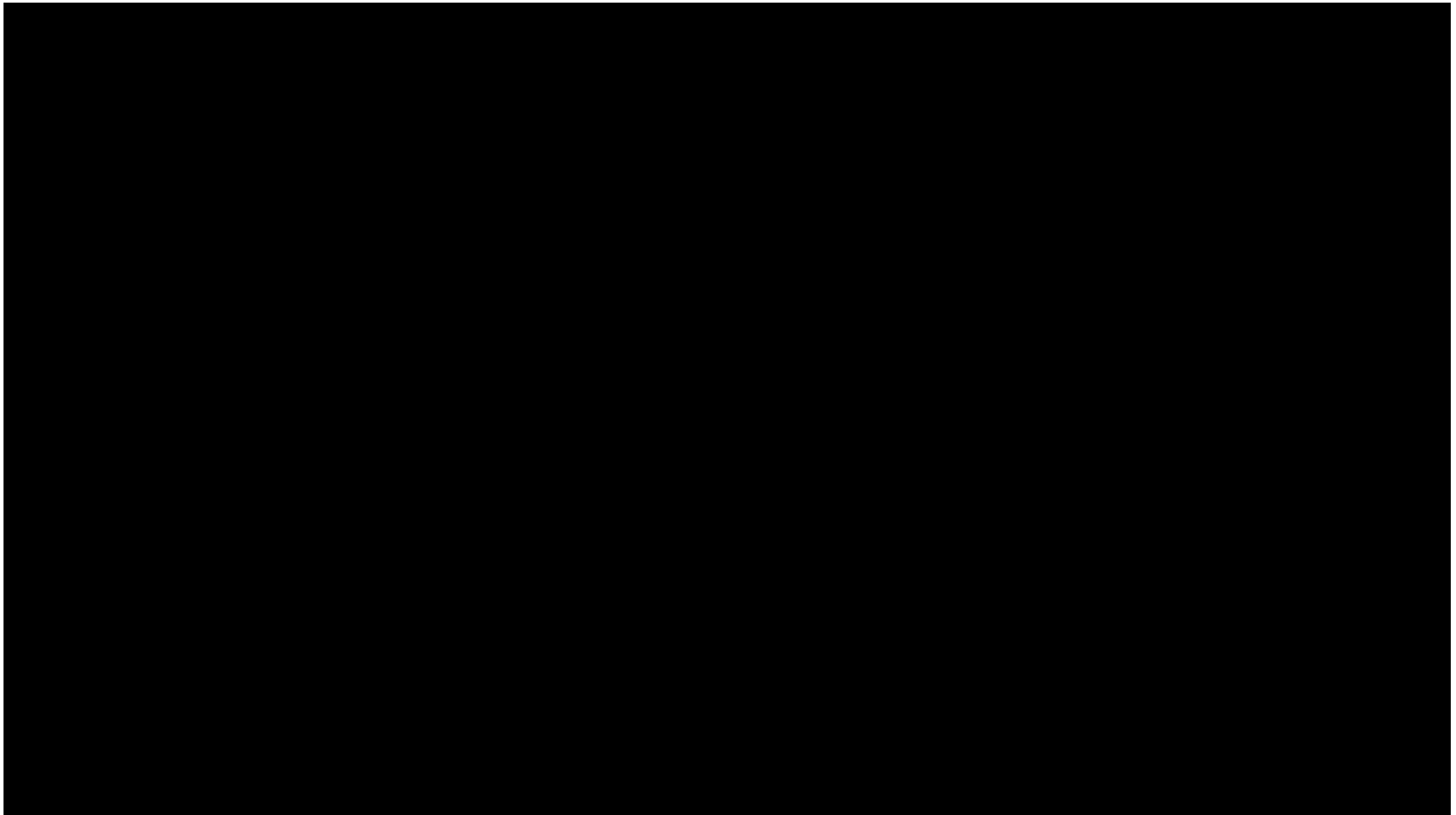
Source: <https://techcrunch.com/2019/04/03/wayve-claims-world-first-in-driving-a-car-autonomously-with-only-its-ai-and-a-satnav/>

Challenges?

1. **Danger** to public during physical training
2. **Depreciation** of AV during physical training
3. **Still a L4**



Wayve Example: Learning from Human Experiences



Source: <https://www.youtube.com/watch?v=eRwTbRtnT1I>



TB3 Example: Learning from Human Experiences

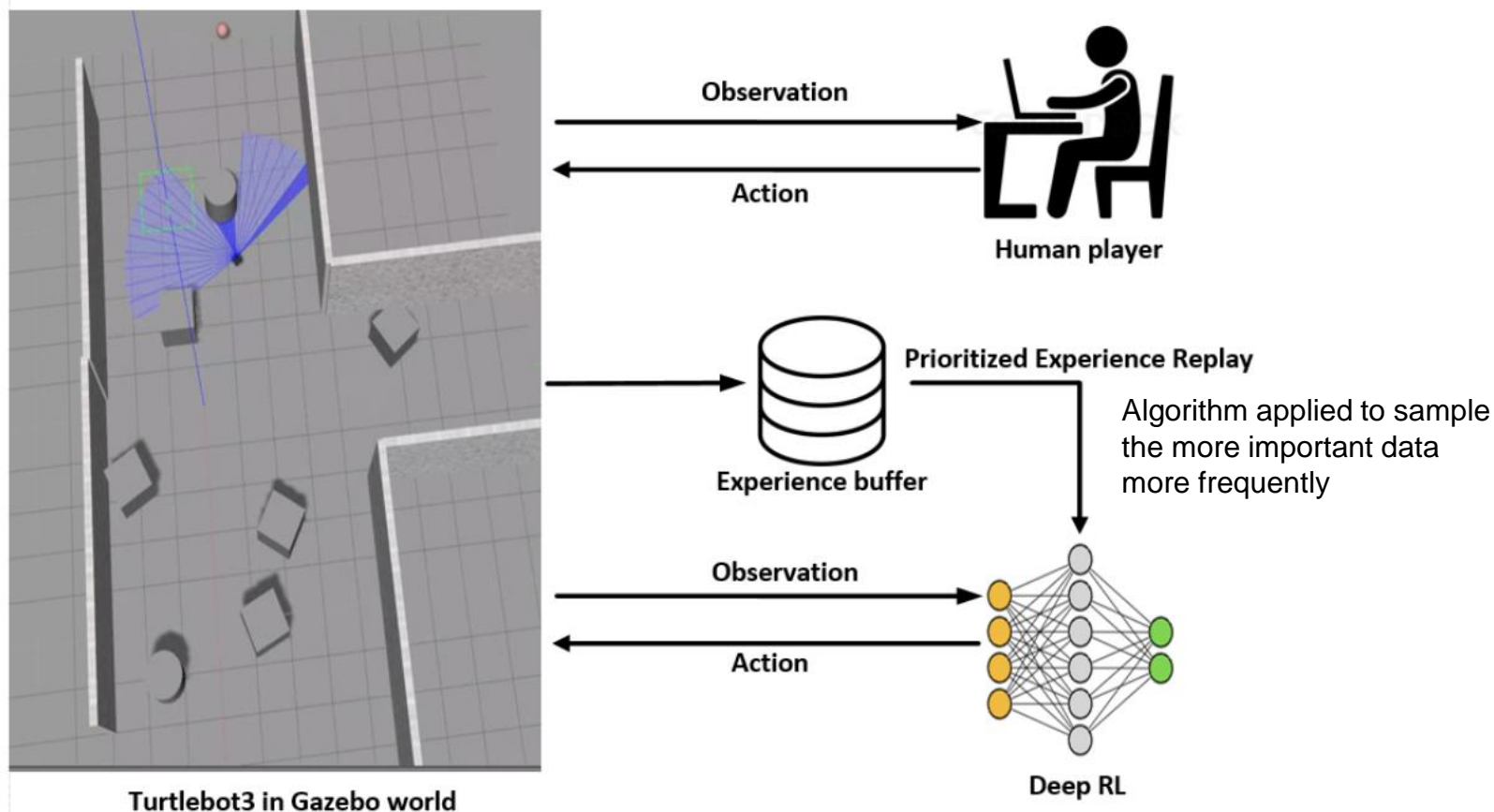


Concept and codes adapted from:

- Accelerated Sim-to-Real Deep Reinforcement Learning: Learning Collision Avoidance from Human Player (given in Luminus; link below: <https://arxiv.org/pdf/2102.10711.pdf>)
- https://github.com/hanlinniu/turtlebot3_ddpg_collision_avoidance



Sensor-level Mapless Collision Avoidance Algorithm

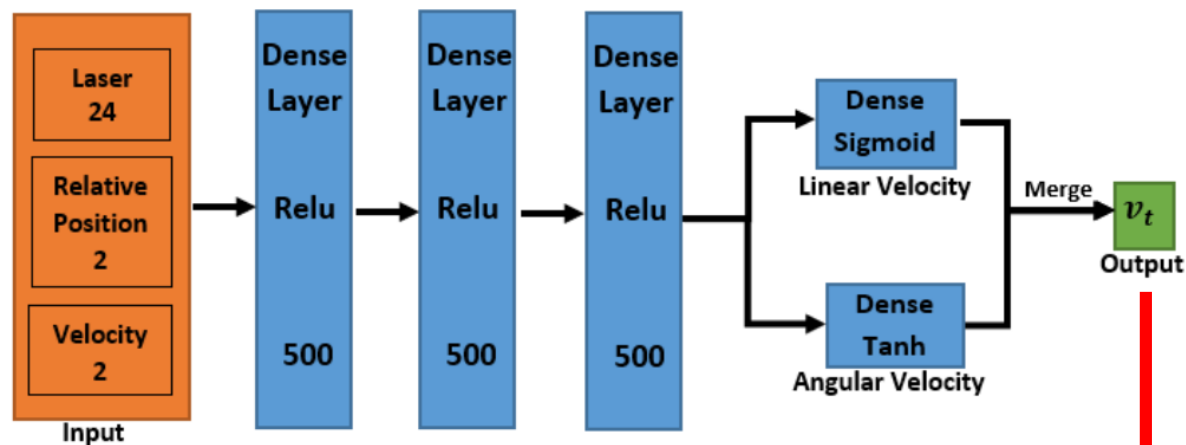


A learning-based mapless collision avoidance algorithm and training strategy were proposed by utilizing human demonstration data and simulated data

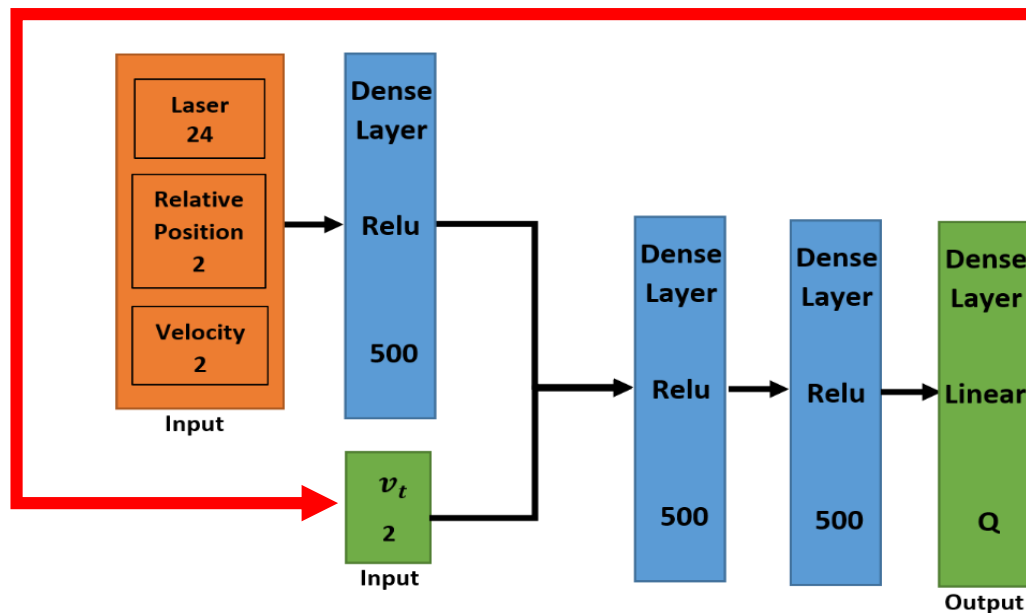


Network Architectures

Actor Network Architecture



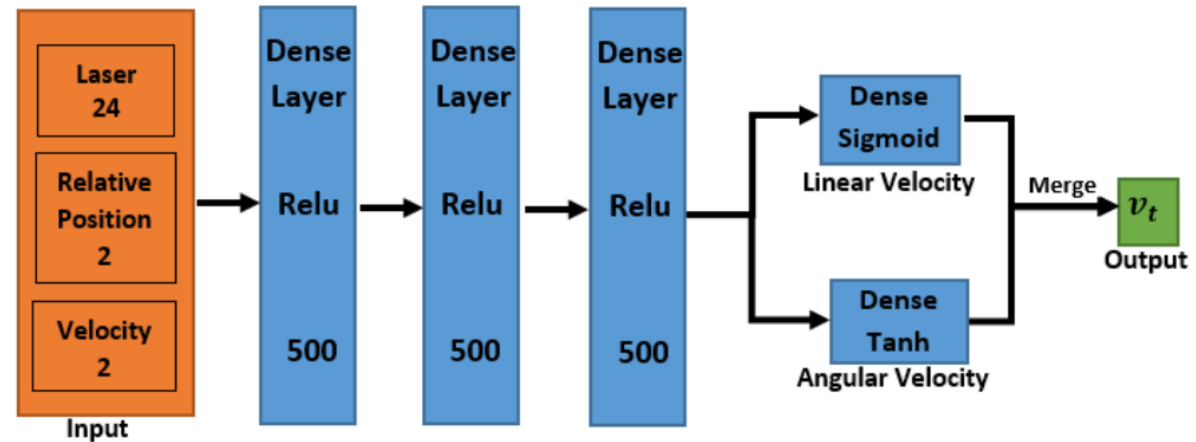
Critic Network Architecture



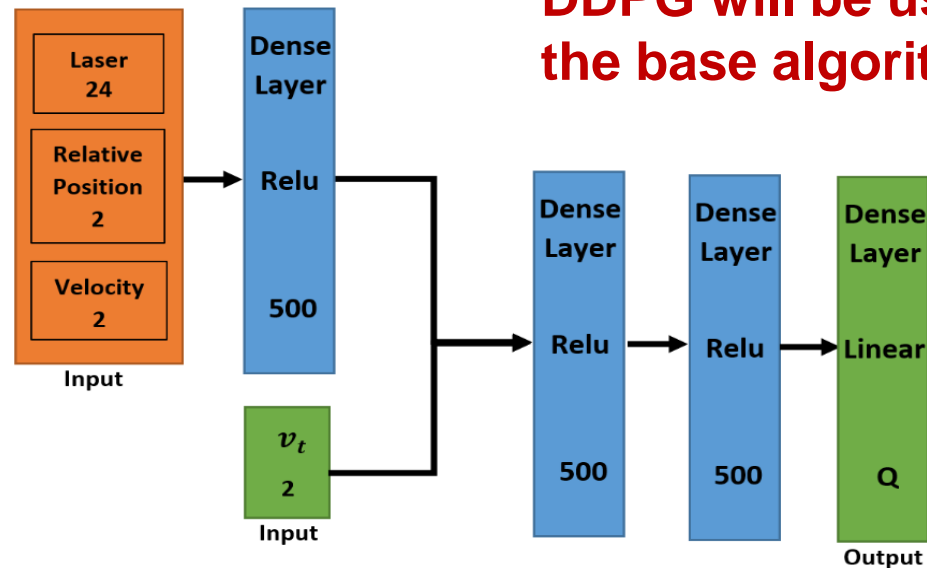


Network Architectures

Actor Network Architecture



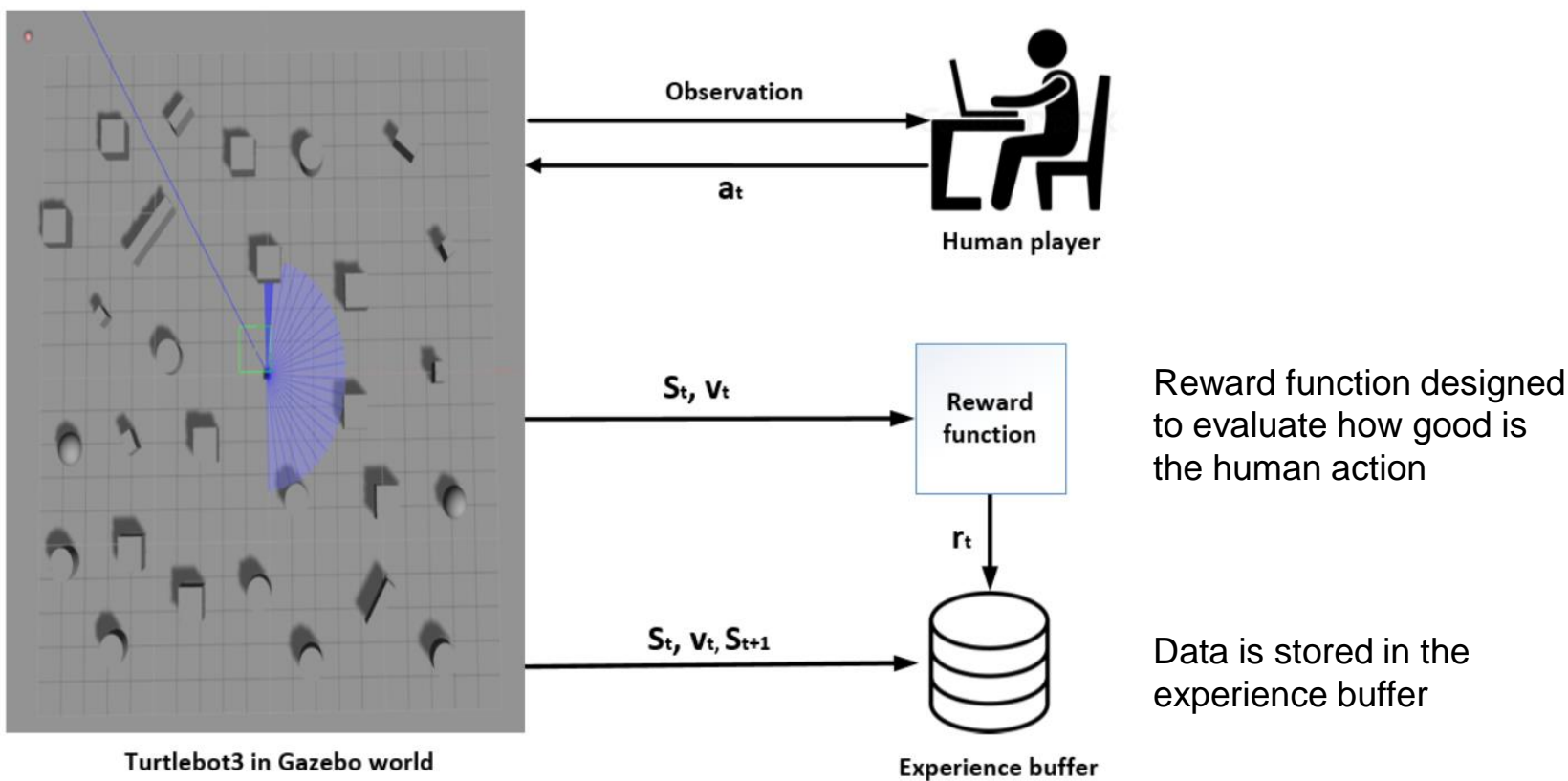
Critic Network Architecture



DDPG will be used as the base algorithm!



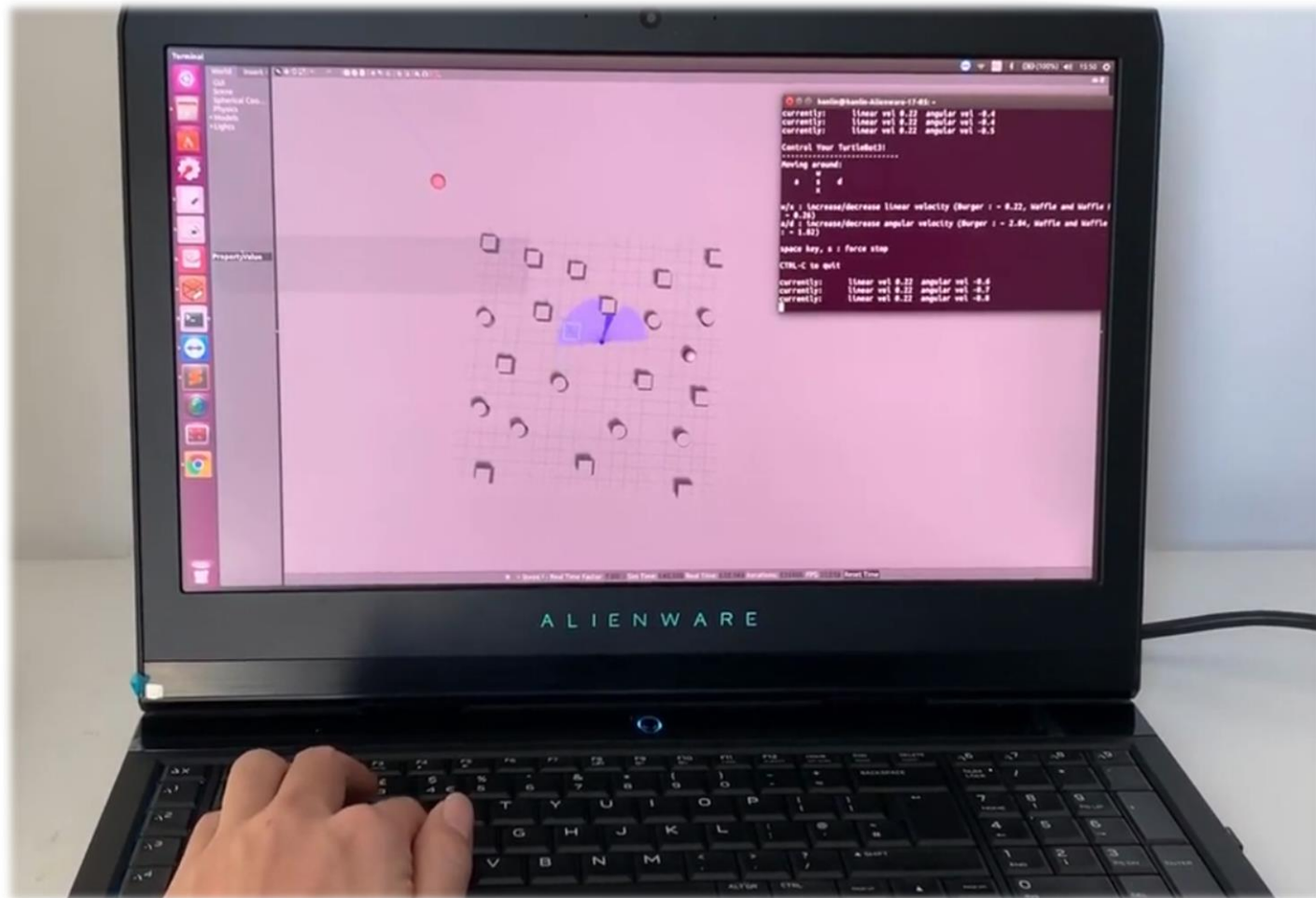
Human Player Data Collection



Collected human player data will be fed to the networks to speed up model training



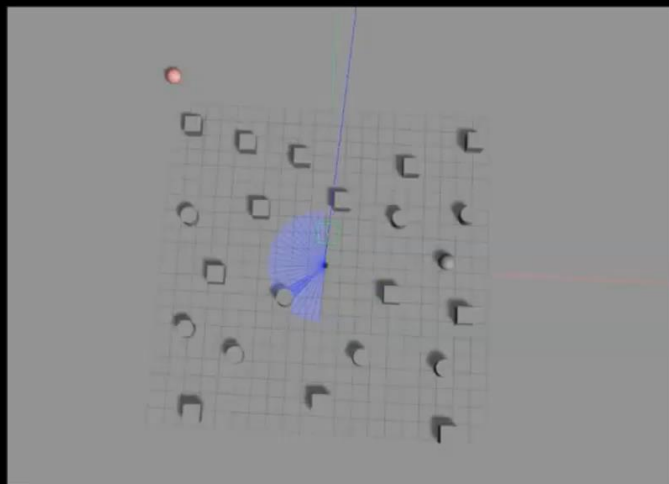
Human Player Data Collection



Human tele-operation data is collected before model training

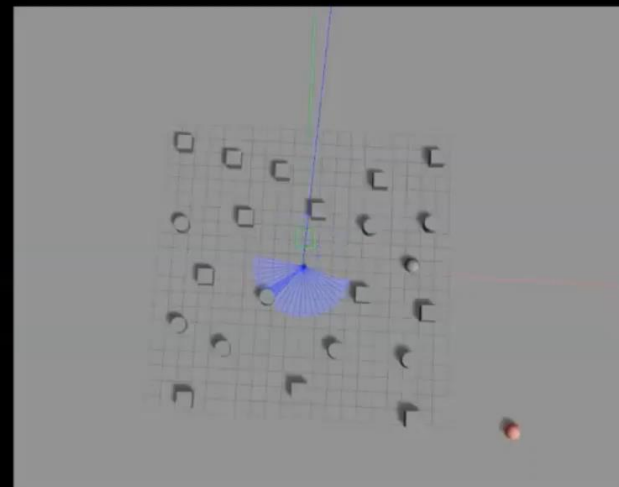


Comparison between DDPG & DDPG with Human Experiences



Before training

Our method



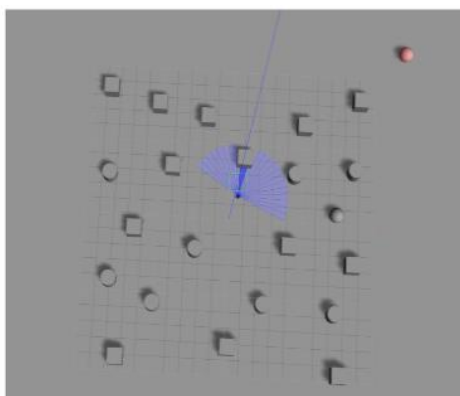
Before training

DDPG

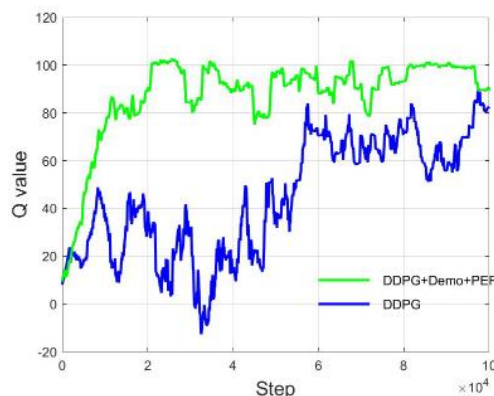
Source: <https://www.youtube.com/watch?v=BmwxevgdGc&feature=youtu.be>



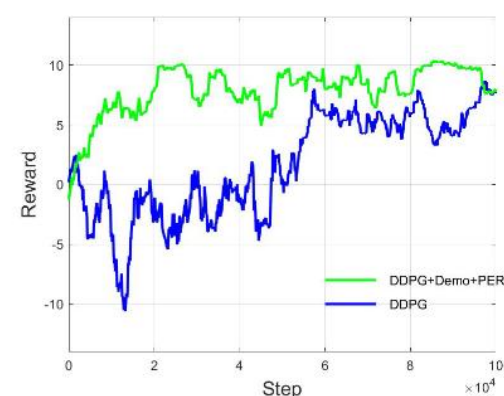
Comparison between DDPG & DDPG with Human Experiences



(a)

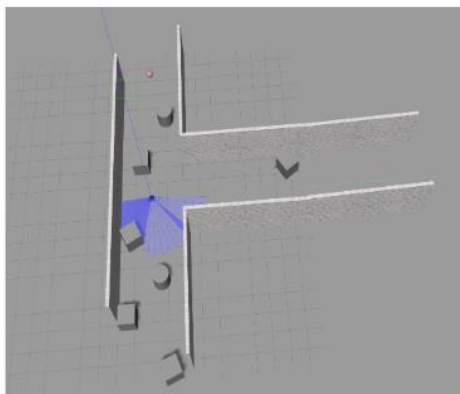


(b)

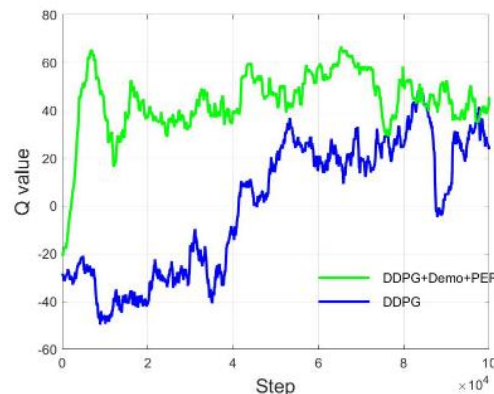


(c)

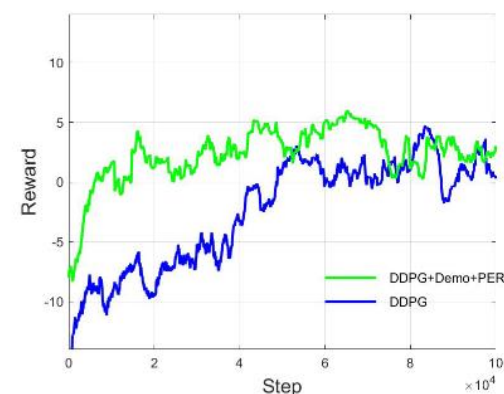
Fig. 6: (a) Environment 1 in Gazebo, (b) Q-value comparison and (c) reward comparison of the proposed method with DDPG in Environment 1.



(a)



(b)



(c)

Fig. 7: (a) Environment 2 in Gazebo, (b) Q-value comparison and (c) reward comparison of the proposed method with DDPG in Environment 2.



Application to Real-Life



The proposed method is also
compared with DDPG on the real robot

Source: <https://www.youtube.com/watch?v=BmwxevgdGc&feature=youtu.be>



TRY OUTS

OPTIONAL; IF YOU HAVE AMPLE TIME



Download & Install hrse_ddpg Package

Refer to README files in the hrse_ddpg package

- 1. Change to the source space directory of the catkin workspace:**

```
$ cd ~/catkin_ws/src
```

- 2. Git Clone the relevant packages:**

```
$ git clone https://github.com/nicholashojunhui/hrse\_ddpg.git
```

- 3. Build the packages in the catkin workspace:**

```
$ cd ~/catkin_ws && catkin_make
```

- 4. Go to:**

**/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/scripts/original_ddpg/
and**

**/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/scripts/fd_replay/play_h
uman_data/, and make all python files executable**



Steps to install dependencies to run hrse_ddpg package

Refer to README files in the hrse_ddpg package

1. Go to

https://emanual.robotis.com/docs/en/platform/turtlebot3/machine_learning and follow the steps to install Ananconda, ROS dependency packages, tensorflow and Keras

2. Follow the below commands to uninstall and install numpy

```
$ pip uninstall numpy
$ pip show numpy
$ pip uninstall numpy
$ pip show numpy
$ pip install numpy pyqtgraph
```

3. Install gym:

```
$ pip install gym==0.10.5
```



Steps to install dependencies to run hrse_ddpg package

Refer to README files in the hrse_ddpg package

4. Navigate to path:

"~/catkin_ws/src/turtlebot3/turtlebot3_description/urdf"

- Under "turtlebot3_waffle_pi.gazebo.xacro" file, right click and open with gedit
- Set visual for laser = true:

```
<xacro:arg name="laser_visual" default="true"/>
```

- Set state -> LaserScan setting to 24 samples instead of 360

```
<gazebo reference = "base_scan">  
  <material>Gazebo/FlatBlack</material>  
  <sensor type="ray" name="lds_lfcd_sensor">  
    <pose>0 0 0 0 0 0</pose>  
    <visualize>$(arg laser_visual)</visualize>  
    <update_rate>5</update_rate>  
    <ray>  
      <scan>  
        <horizontal>  
          <samples>24</samples>
```



Steps to Train & Run Models (for original DDPG & DDPG with Human Experiences)

**Steps to Train Model and to Run Trained
Model can be found in the README files
under paths:**

`/hrse_ddpg/turtlebot_ddpg/scripts/original_ddpg/`

`/hrse_ddpg/turtlebot_ddpg/scripts/fd_replay/play_human_data/`



Steps to Train Model (DDPG Original)



Perform training from scratch WITHOUT any human inputs by:

1. Launch the TB3 in the Gazebo Corridor World (rmb to change file path to the correct one!)

```
$ roslaunch turtlebot_ddpg turtlebot3_empty_world.launch  
world_file:= '/home/correct_username/catkin_ws/src/hrse_ddpg/t  
urtlebot_ddpg/worlds/turtlebot3_modified_corridor2.world'
```

2. Under `ddpg_network_turtlebot3_original_ddpg.py` file,
Change `train_indicator` to '1'
3. Run following command to train the network WITHOUT any human inputs

```
$ rosrunc turtlebot_ddpg  
ddpg_network_turtlebot3_original_ddpg.py
```



Steps to Run Trained Model (DDPG Original)



Once training is done, test the model:

1. Launch the TB3 in the Gazebo Corridor World (rmb to change file path to the correct one!)

```
$ roslaunch turtlebot_ddpg turtlebot3_empty_world.launch  
world_file:='/home/correct_username/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/worlds/turtlebot3_modified_corridor2.world'
```

2. Under `ddpg_network_turtlebot3_original_ddpg.py` file,
Change `train_indicator` to '0'
3. Under if `train_indicator==0` condition,
Change the paths of `actor_critic.actor_model.load_weights` and `actor_critic.critic_model.load_weights` to the correct ones; depends on your username and ideal model
4. Run following command to test the network based on the trained model

```
$ rosrun turtlebot_ddpg  
ddpg_network_turtlebot3_original_ddpg.py
```



Steps to Train Model (DDPG + Human Experiences)



First, undergo the human player data collection process:

1. Launch the TB3 in the Gazebo Corridor World (rmb to change file path to the correct one!)

```
$ roslaunch turtlebot_ddpg turtlebot3_empty_world.launch  
world_file:='/home/correct_username/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/worlds/turtlebot3_modified_corridor2.world'
```

2. Run following command first to prepare human player data collection process

```
$ rosrun turtlebot_ddpg ddp_g_record_data.py
```

3. Run following teleop key command to manually move the robot to the destination without colliding with any obstacles; repeat the process for at least 30 mins

```
$ rosrun turtlebot3_teleop turtlebot3_teleop_key
```



Steps to Train Model (DDPG + Human Experiences)



Once done with the human player data collection, perform model training by:

1. Launch the TB3 in the Gazebo Corridor World (rmb to change file path to the correct one!)

```
$ roslaunch turtlebot_ddpg turtlebot3_empty_world.launch  
world_file:= '/home/correct_username/catkin_ws/src/hrse_ddpg/t  
urtlebot_ddpg/worlds/turtlebot3_modified_corridor2.world'
```

2. Under `ddpg_network_turtlebot3_amcl_fd_replay_human.py` file, Change `train_indicator` to '1'
3. Run following command to train the network based on the human inputs

```
$ rosrun turtlebot_ddpg  
ddpg_network_turtlebot3_amcl_fd_replay_human.py
```



Steps to Run Trained Model (DDPG + Human Experiences)



Once training is done, test the trained model:

1. Launch the TB3 in the Gazebo Corridor World (rmb to change file path to the correct one!)

```
$ roslaunch turtlebot_ddpg turtlebot3_empty_world.launch  
world_file:='/home/correct_username/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/worlds/turtlebot3_modified_corridor2.world'
```

2. Under `ddpg_network_turtlebot3_amcl_fd_replay_human.py` file,
Change `train_indicator` to '0'
3. Under if `train_indicator==0` condition,
Change the paths of `actor_critic.actor_model.load_weights` and `actor_critic.critic_model.load_weights` to the correct ones; depends on your username and ideal model
4. Run following command to test the network based on the trained model

```
$ rosrun turtlebot_ddpg  
ddpg_network_turtlebot3_amcl_fd_replay_human.py
```



Functions in Code Files that you can modify



ddpg_turtlebot_turtlebot3_original_ddpg.py

ddpg_turtlebot_turtlebot3_amcl_fd_replay_human.py

```
def __init__(self):
```

```
    # Create a Twist message and add linear x and angular z values
```

```
    self.move_cmd = Twist()
```

```
    self.move_cmd.linear.x = 0.6 #linear_x
```

```
    self.move_cmd.angular.z = 0.2 #angular_z
```

You can set the initial velocity values for the TB3

```
    # set target position
```

```
    self.target_x = 4.0
```

```
    self.target_y = 0.0
```

You can change the initial target position; currently set to [4.0, 0.0] position

```
def reset(self):
```



Functions in Code Files that you can modify



ddpg_turtlebot_turtlebot3_original_ddpg.py

ddpg_turtlebot_turtlebot3_amcl_fd_replay_human.py

```
def reset(self):
```

```
# for corridor
'''self.target_x = (np.random.random()-0.5)*5 + 12*index_x
self.target_y = (np.random.random()-0.5)*3
random_turtlebot_y = (np.random.random())*5 #+ index_turtlebot_y'''
```

```
# Target position = 4m below origin
self.target_x = 4.0          #specific x coordinate
self.target_y = 0.0          #specific y coordinate
random_turtlebot_y = (np.random.random())*5 #+ index_turtlebot_y
```

If you want the target positions to be changed randomly, uncomment this chunk

And comment this chunk



Functions in Code Files that you can modify



ddpg_turtlebot_turtlebot3_original_ddpg.py

ddpg_turtlebot_turtlebot3_amcl_fd_replay_human.py

```
def turtlebot_is_crashed(self, laser_values, range_limit):  
    self.laser_crashed_value = 0  
    self.laser_crashed_reward = 0  
  
    for i in range(len(laser_values)):  
        if (laser_values[i] < 2*range_limit):  
            self.laser_crashed_reward = -80  
        if (laser_values[i] < range_limit):  
            self.laser_crashed_value = 1  
            self.laser_crashed_reward = -200  
            self.reset()  
            time.sleep(1)  
            break  
    return self.laser_crashed_reward
```

You can change the
crash penalty values
under this chunk



Functions in Code Files that you can modify



ddpg_turtlebot_turtlebot3_original_ddpg.py

ddpg_turtlebot_turtlebot3_amcl_fd_replay_human.py

```
def game_step(self, time_step=0.1, linear_x=0.8, angular_z=0.3):
```

```
    # make distance reward
    (self.position, self.rotation) = self.get_odom()
    turtlebot_x = self.position.x
    turtlebot_y = self.position.y
    #distance_turtlebot_target_previous = math.sqrt((self.target_x - turtlebot_x_previous)**2 + (self.target_y -
    #current_distance_from_origin = math.sqrt((turtlebot_x)**2 + (turtlebot_y)**2)

    distance_turtlebot_target = math.sqrt((self.target_x - turtlebot_x)**2 + (self.target_y - turtlebot_y)**2)
    print("self.target_x is %s" %self.target_x)
    print("self.target_y is %s" %self.target_y)
    print("turtlebot_x is %s" %turtlebot_x)
    print("turtlebot_y is %s" %turtlebot_y)

    distance_reward = 10.0 - abs(distance_turtlebot_target)

    self.laser_crashed_reward = self.turtlebot_is_crashed(laser_values, range_limit=0.25)
    self.laser_reward = sum(normalized_laser)-24
    self.collision_reward = self.laser_crashed_reward + self.laser_reward
```

You can change the
reward structures
under this chunk

Func



Functions in Code Files that you can modify



ddpg_turtlebot_turtlebot3_original_ddpg.py

ddpg_turtlebot_turtlebot3_amcl_fd_replay_human.py

```
def game_step(self, time_step=0.1, linear_x=0.8, angular_z=0.3):

    self.angular_punish_reward = 0
    self.linear_punish_reward = 0

    if angular_z > 0.8:
        self.angular_punish_reward = -10
    if angular_z < -0.8:
        self.angular_punish_reward = -10

    if linear_x < 0.2:
        self.linear_punish_reward = -2

    self.arrive_reward = 0
    if distance_turtlebot_target<1:
        self.arrive_reward = 100
        print("REACHED TARGET!!!")
        self.reset()
        time.sleep(1)

    #reward = distance_reward*(5/time_step)*1.2*7 + self.arrive_reward + self.collusion_reward + self.angular_punish_reward + self.lin
    reward = distance_reward*10 + self.arrive_reward + self.collusion_reward + self.angular_punish_reward + self.linear_punish_reward
    print("laser_reward is %s" %self.laser_reward)
    print("laser_crashed_reward is %s" %self.laser_crashed_reward)
    print("arrive_reward is %s" %self.arrive_reward)
    #print("distance reward is : %s", distance_reward*(5/time_step)*1.2*7)
    print("distance reward is: %s" %(distance_reward*10))
```

Cont:
You can change the
reward structures
under this chunk



Functions in Code Files that you can modify

ddpg_network_turtlebot3_original_ddpg.py:

ddpg_network_turtlebot3_amcl_fd_replay_human.py:

```
def create_actor_model(self):  
    state_input = Input(shape=self.env.observation_space.shape)  
    h1 = Dense(500, activation='relu')(state_input)  
    #h2 = Dense(1000, activation='relu')(h1)  
    h2 = Dense(500, activation='relu')(h1)  
    h3 = Dense(500, activation='relu')(h2)  
    delta_theta = Dense(1, activation='tanh')(h3)  
    speed = Dense(1, activation='sigmoid')(h3) # sigmoid makes the output to be range [0, 1]  
  
    #output = Dense(self.env.action_space.shape[0], activation='tanh')(h3)  
    #output = Concatenate()([delta_theta])#merge([delta_theta, speed],mode='concat')  
    output = Concatenate()([delta_theta, speed])  
    model = Model(input=state_input, output=output)  
    adam = Adam(lr=0.0001)  
    model.compile(loss="mse", optimizer=adam)  
    return state_input, model
```

You can redefine
the actor model



Functions in Code Files that you can modify

ddpg_network_turtlebot3_original_ddpg.py:

ddpg_network_turtlebot3_amcl_fd_replay_human.py:

```
def create_critic_model(self):
    state_input = Input(shape=self.env.observation_space.shape)
    state_h1 = Dense(500, activation='relu')(state_input)
    #state_h2 = Dense(1000)(state_h1)

    action_input = Input(shape=self.env.action_space.shape)
    action_h1 = Dense(500)(action_input)

    merged = Concatenate()([state_h1, action_h1])
    merged_h1 = Dense(500, activation='relu')(merged)
    merged_h2 = Dense(500, activation='relu')(merged_h1)
    output = Dense(1, activation='linear')(merged_h2)
    model = Model(input=[state_input, action_input], output=output)

    adam = Adam(lr=0.0001)
    model.compile(loss="mse", optimizer=adam)
    return state_input, action_input, model
```

You can redefine
the critic model



Functions in Code Files that you can modify



ddpg_network_turtlebot3_original_ddpg.py:

ddpg_network_turtlebot3_amcl_fd_replay_human.py:

```
def main():
```

```
    sess = tf.Session()
```

```
    K.set_session(sess)
```

```
#####
```

```
game_state= ddpq_turtlebot_turtlebot3_original_ddpg.GameState()
```

```
actor_critic = ActorCritic(game_state, sess)
```

```
#####
```

```
num_trials = 10000
```

```
trial_len = 500
```

```
train_indicator = 0
```

You can change the training parameters (i.e. num_trials, train_len, train_indicator)



Functions in Code Files that you can modify



ddpg_network_turtlebot3_original_ddpg.py:

ddpg_network_turtlebot3_amcl_fd_replay_human.py:

```
if train_indicator==0:
    for i in range(num_trials):
        print("trial:" + str(i))
        current_state = game_state.reset()

        # Minimal model to reach target: model-30-500.h5
        # Optimal models: model-50-500.h5
        actor_critic.actor_model.load_weights("/home/nicho/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/scripts/original_ddpg/actormodel-50-500.h5")
        actor_critic.critic_model.load_weights("/home/nicho/catkin_ws/src/hrse_ddpg/turtlebot_ddpg/scripts/original_ddpg/criticmodel-50-500.h5")
        print("models loaded")

        #actor_critic.actor_model.load_weights("actormodel-160-500.h5")
        #actor_critic.critic_model.load_weights("criticmodel-160-500.h5")
        #####
        total_reward = 0
```

When running trained models (i.e. `train_indicator == 0`), edit the file paths to load your desired models



THANK YOU

Email: nicholas.ho@nus.edu.sg