Intelligent Systems
Project Presentation

Erfan Panahi | Amirhussain Birzhandi

Project Overview

Introduction to
RL-GANs

Implementing the Connect4 Using
Deep Q-Learning

PROJECT B

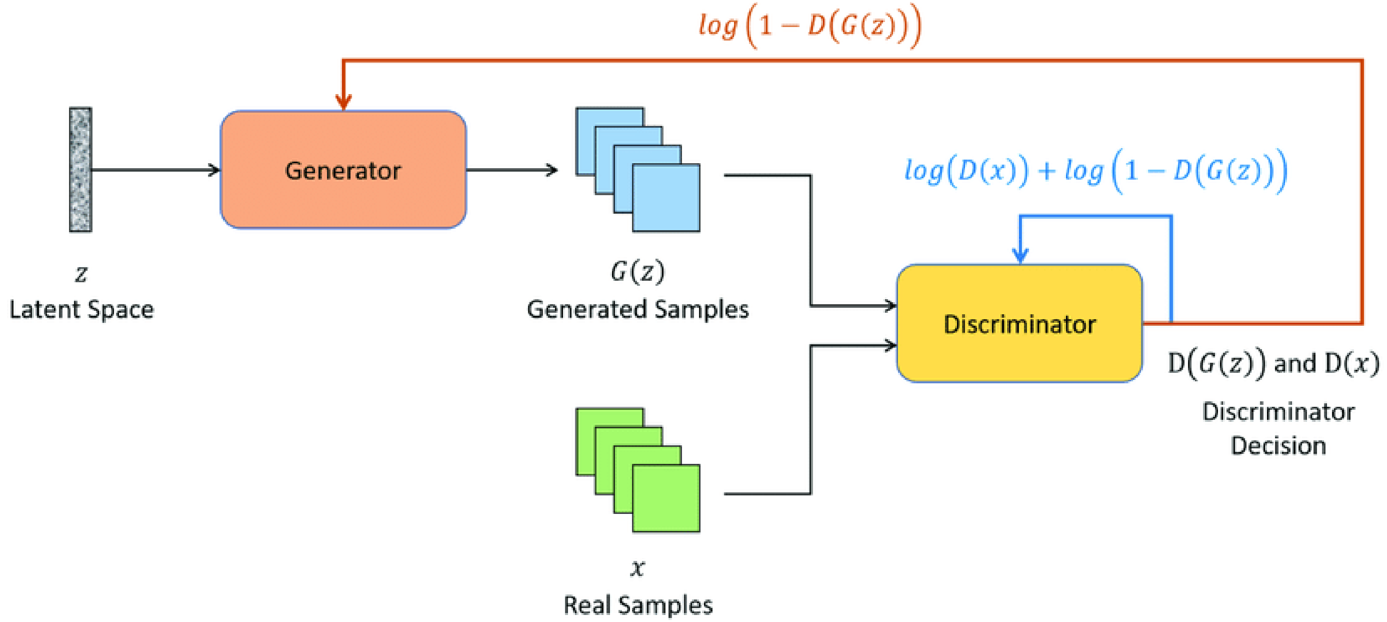
- GANs
- Architecture of RL-GANS
- Algorithm of RL-GAN

- Deep Q-Learning
- Connect 4
- Implementation & Results

Part 1 Introduction to RL-GANs

GANs

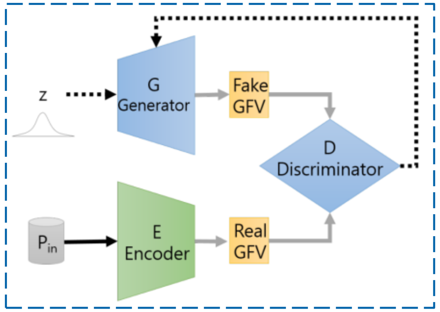
Generative
Adversarial
Network

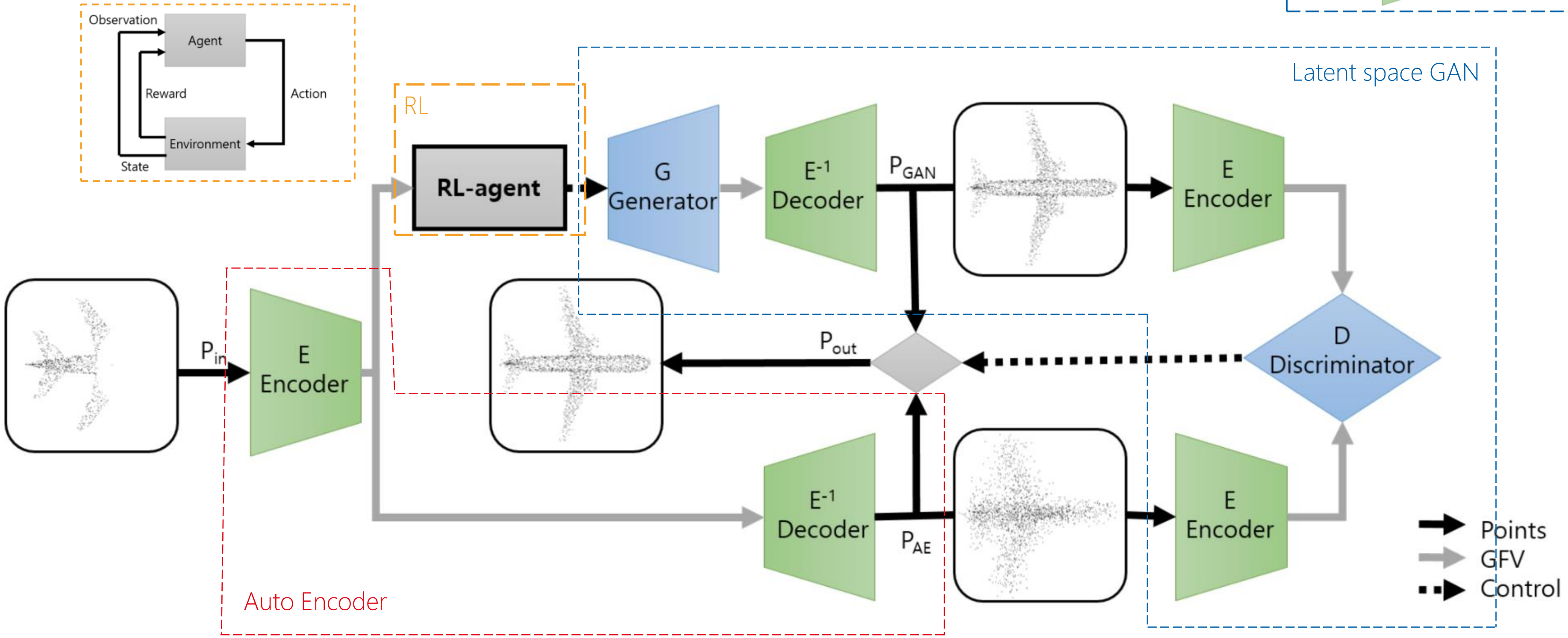


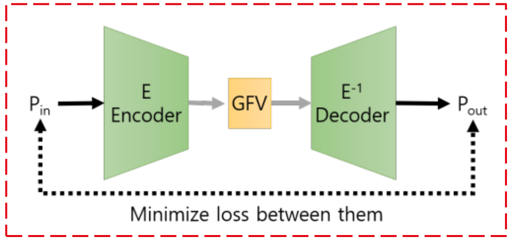
$$\min_G \max_D V(D,G)$$

$$V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log(D(x)] + \mathbb{E}_{z\sim p_Z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

Architecture of RL-GANS







RL-GAN-Net: A Reinforcement Learning Agent Controlled GAN Network for Real-Time Point Cloud Shape Completion

Architecture of RL-GANS

$$d_{CH}(P_1, P_2) = \sum_{a \in P_1} \min_{b \in P_2} \| a - b \|_2^2 + \sum_{b \in P_2} \min_{a \in P_1} \| a - b \|_2^2$$

Chamfer Loss: $L_{CH} = d_{CH}(P_{in}, E^{-1}(G(z))$

GFV Loss: $L_{GFV} = \| G(z) - E(P_{in}) \|_2^2$

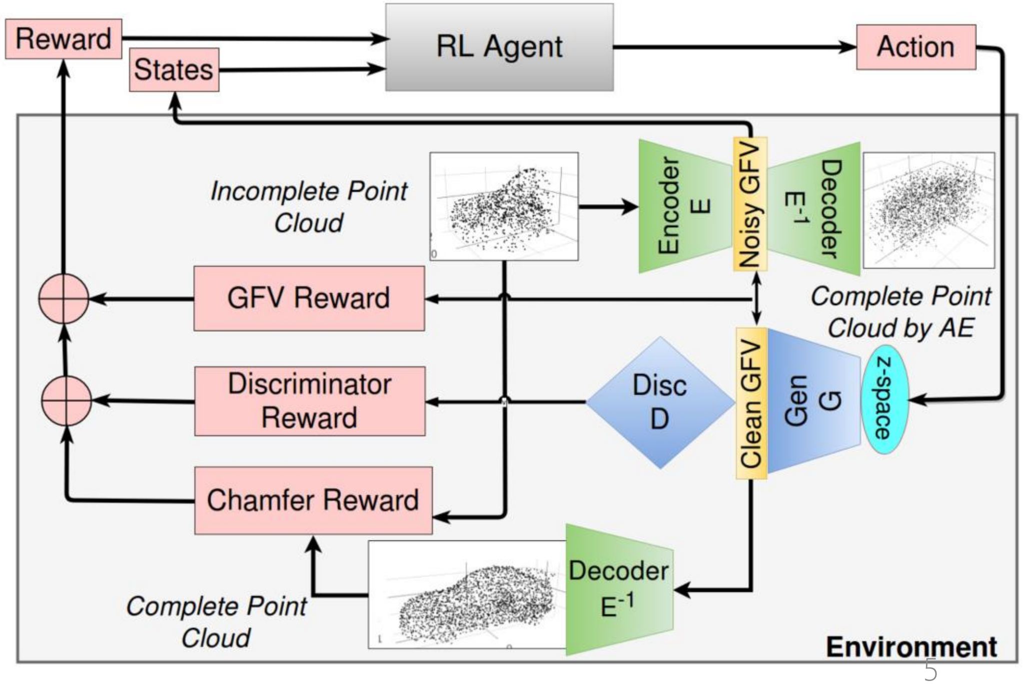
Discriminator Loss: $L_{CH} = -D(G(z))$

$$r_{CH} = -L_{CH}$$

$$r_{GFV} = -L_{GFV}$$

$$r_D = -L_D$$

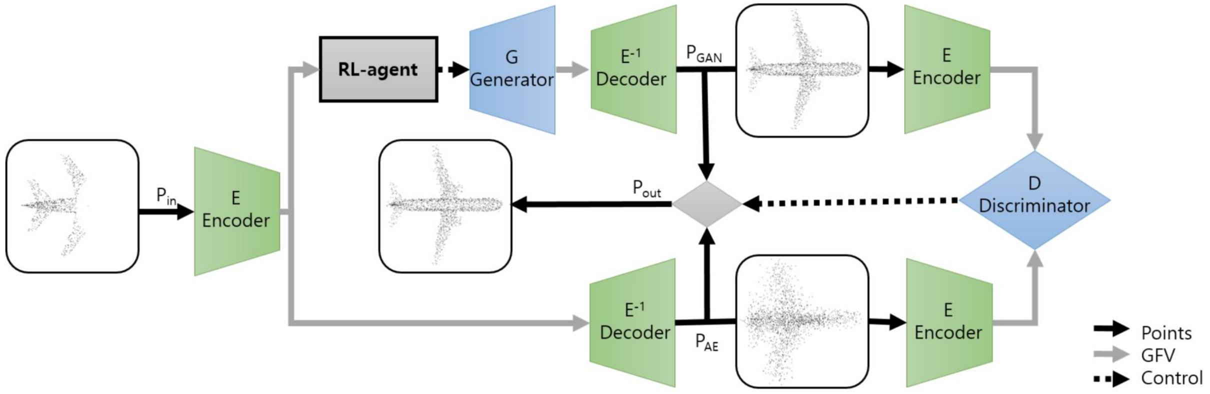
$$\boldsymbol{r = \omega_{CH}.r_{CH} + \omega_{GFV}.r_{GFV} + \omega_D.r_D}$$



Algorithm of RL-GAN

Deep Deterministic Policy Gradient

$$\nabla_{\theta^\mu} J(\theta) = \mathbb{E}_{s_t\sim\rho^\beta}\left[\underbrace{\nabla_\alpha Q(s,a|\theta^Q)}_{critic\ network}\ \underbrace{\nabla_{\theta^\mu}\mu(s|\theta^\mu)}_{actor\ network}\right]_{s=s_t,a=\mu(s_t)}$$



Algorithm 1 Training RL-GAN-Net

Agent Input:

State (s_t): $s_t = GFV_n = \mathbf{E}(P_{in})$; Sample pointcloud P_{in} from dataset into the pre-trained encoder **E** to generate noisy latent representation GFV_n.

Reward (r_t): Calculated using Eq. (5)

Agent Output:

Action (a_t): $a_t = z$

Pass z-vector to the pre-trained generator **G** to form clean latent vector GFV_c=$\mathbf{G}(z)$

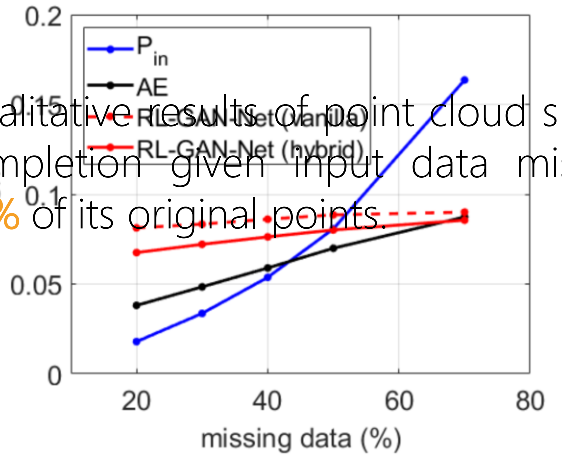
Final Output:

$P_{out} = \mathbf{E}^{-1}(GFV_c)$; Pass GFV_c into decoder E^{-1} to generate output point cloud P_{out}.

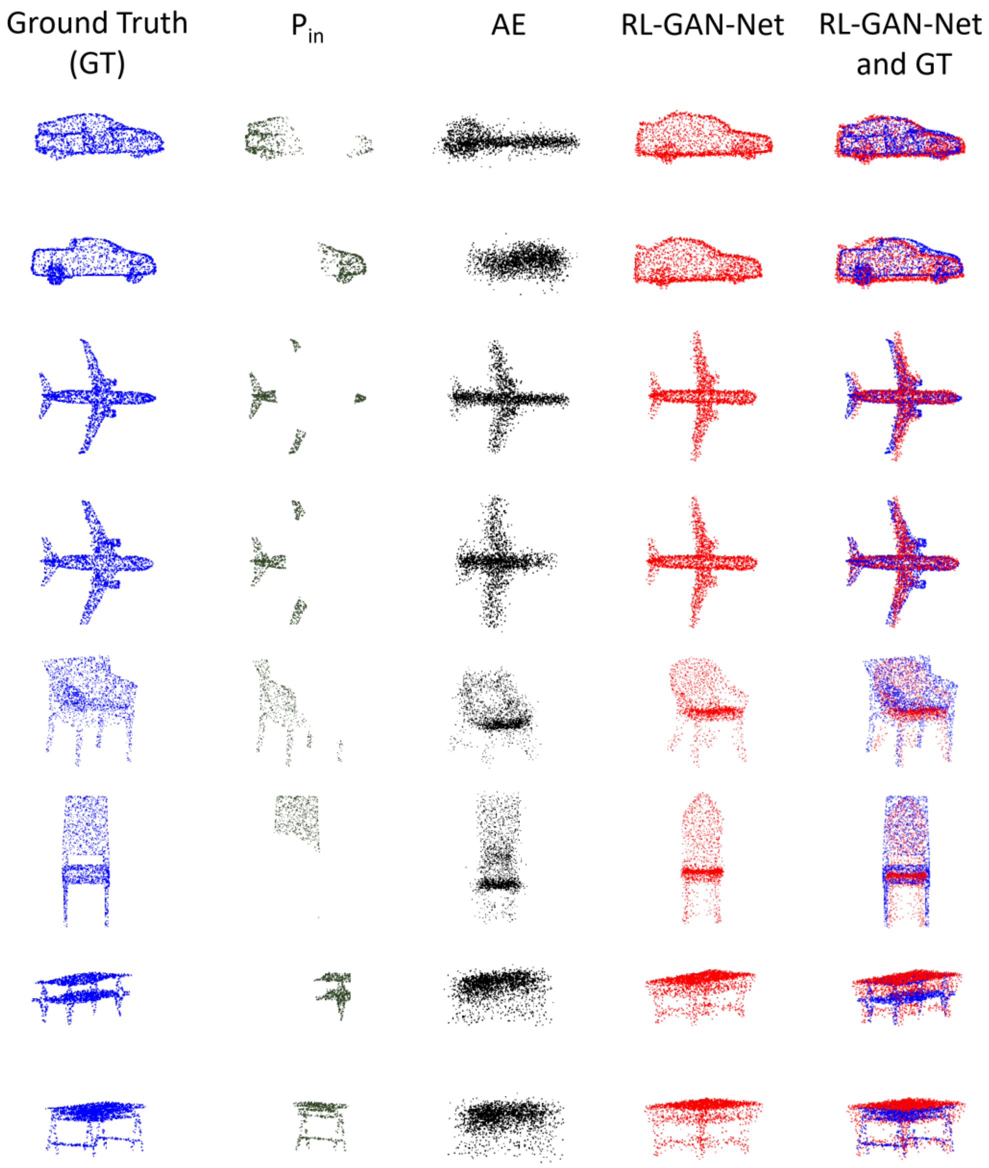
```
Initialize procedure Env with pre-trained generator G, discriminator D, encoder E and decoder E^{-1}
Initialize policy π with DDPG, actor A, critic C, and replay buffer R
for t_steps < maxsteps do
    Get P_in
    if t_steps > 0 then
        Train A and C with R
    if t_LastEvaluation > f_EvalFrequency then
        Evaluate π
    GFV_n ← E(P_in)
    if t_steps > t_StartTime then
        Random Action a_t
    if t_steps < t_StartTime then
        Use a_t ← A ← GFV_n
    (s_t, a_t, r_t, s_{t+1}) ←Env ←a_t
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
endfor
procedure ENV(P_in, a_t)
    Get State (s_t) : GFV_n ← E(P_in)
    Implement Action : GFV_c ← G (a_t = z)
    Calculate reward r_t using Eq. (5)
    Obtain point cloud : P_out ← E^{-1} (GFV_c)
```

Performance Analysis

Qualitative results of point cloud shape completion given input data missing 70% of its original points.



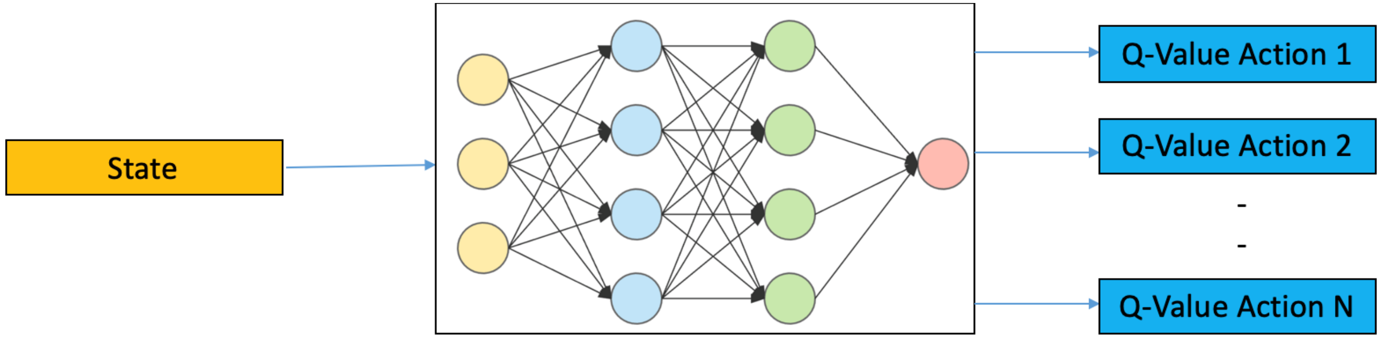
(a) Chamfer distance to GT



Part 2 Implementing the Connect4 Using Deep Q-Learning

Deep Q-Learning

Deep Q-Learning uses the Q-learning idea and takes it one step further. Instead of using a Q-table, we use a Neural Network that takes a state and approximates the Q-values for each action based on that state.



Deep Q Learning

Reference: https://www.assemblyai.com/blog/reinforcement-learning-with-deep-q-learning-explained/

Deep Q-Learning

Choosing the Architecture of the Network

Choosing the Best Action

Storing the Reward, Action, and Observation

Updating the Weights of the Network at the End of each Episodes

Changing the Hyperparameters and going to Next Episode

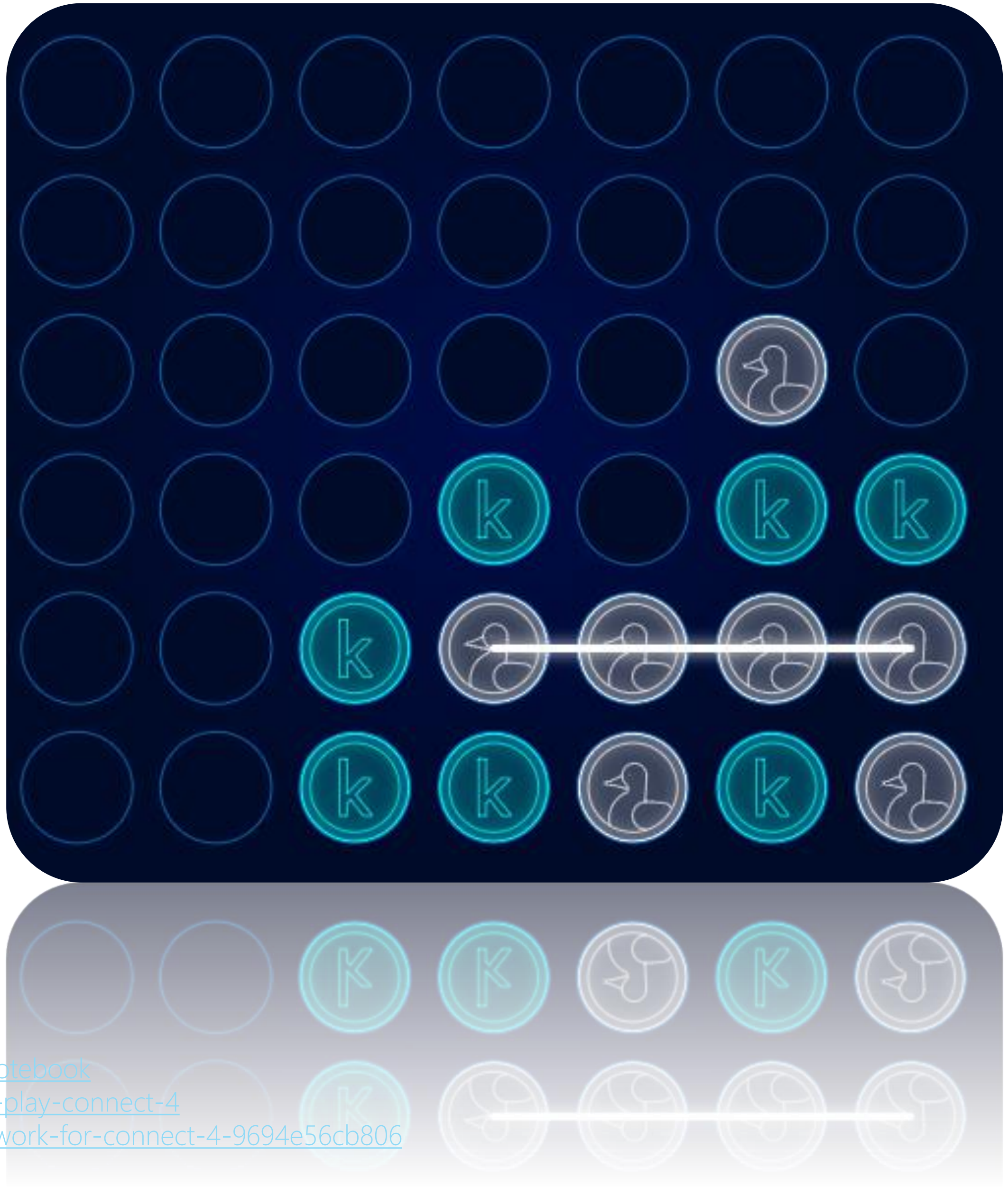
$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha \left(\overbrace{\underbrace{R_t + \gamma \max Q(s_{t+1}, a_{t+1})}_{Target} - \underbrace{Q_{old}(s_t, a_t)}_{Prediction}}^{TD\ error} \right)$$

$$TD\ error = R_t + \gamma \max Q(s_{t+1}, a_{t+1}) - Q_{old}(s_t, a_t)$$

Connect4

Environment:

kaggle-environment



References:
https://www.kaggle.com/code/ajeffries/connectx-getting-started/notebook
https://www.kaggle.com/code/gordotron85/teaching-an-agent-to-play-connect-4
https://medium.com/@louisdhulst/training-a-deep-q-learning-network-for-connect-4-9694e56cb806

Implementation & Results

- Choosing the Architecture of the Network
- Choosing the Best Action
- Storing the Reward, Action, and Observation
- Updating the Weights of the Network at the End of each Episodes
- Changing the Hyperparameters and Going to Next Episode

```
def DQN_Training(model, opt, EPSILONE_RATE, BOARD_SIZE, NUM_ACTIONS, NUM_EPISODE):
  env = make("connectx", debug=True)
  epsilon = 1; win_num = 0; win_list = []
  Reward_list = np.zeros(NUM_EPISODE)
  Exp = Experience()
  for episode in range(NUM_EPISODE):
```

```
def Training(model, optimizer, observations, actions, rewards):
  with tf.GradientTape() as tape:
    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits = model(np.array(observations)), labels = np.array(actions)) + rewards)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

```
      Act = chooseAction(model, observation, epsilon, NUM_ACTIONS)
      New_observation, winner, done, info = trainer.step(Act)
      reward = Reward(winner, done)
      Reward_list[episode] += reward
      Exp.store(observation, Act, reward)
      observation = np.array(New_observation['board']).reshape(BOARD_SIZE)
      if done:
        if winner == 1: win_num += 1
        win_list.append(win_num)
        Training(model, opt, Exp.observations, Exp.actions, Exp.rewards)
        break
    Exp.clear()
    epsilon = np.exp(-episode * EPSILONE_RATE)
  return model, win_num, win_list, Reward_list, epsilon
```

Implementation & Results

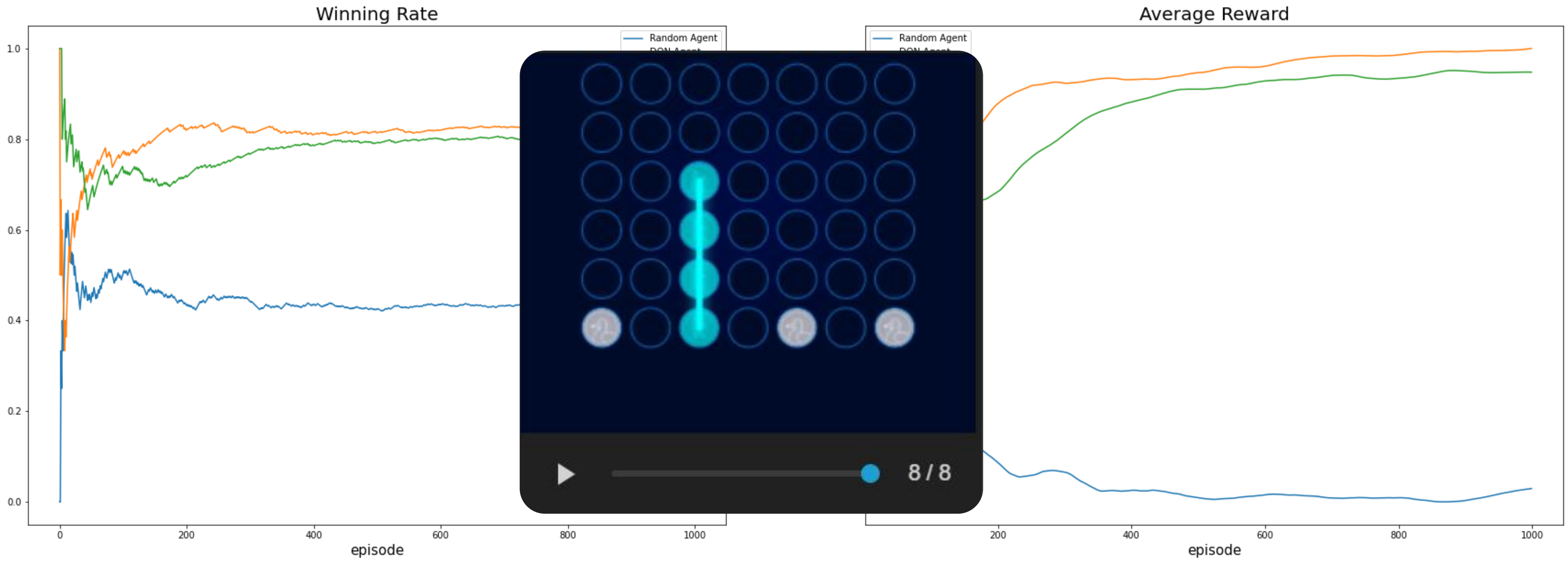
Comparing the Model with Random Agent

```
def RandomVsRandom(BOARD_SIZE, NUM_ACTIONS, NUM_EPISODE):
  env = make("connectx", debug=True)
  Reward_list = np.zeros(NUM_EPISODE)
  win_num = 0; win_list = []
  for episode in range(NUM_EPISODE):
    trainer = env.train([None, 'random'])
    observation = np.array(trainer.reset()['board']).reshape(BOARD_SIZE)
    done = False
    while True:
      Act = random.randint(0, NUM_ACTIONS-1)
      New_observation, winner, done, info = trainer.step(Act)
      observation = np.array(New_observation['board']).reshape(BOARD_SIZE)
      reward = Reward(winner, done)
      Reward_list[episode] += reward
      if done:
        if winner == 1: win_num += 1
        win_list.append(win_num)
        break
  return win_num, win_list, Reward_list
```

Implementation & Results

Comparing the Model with Random Agent

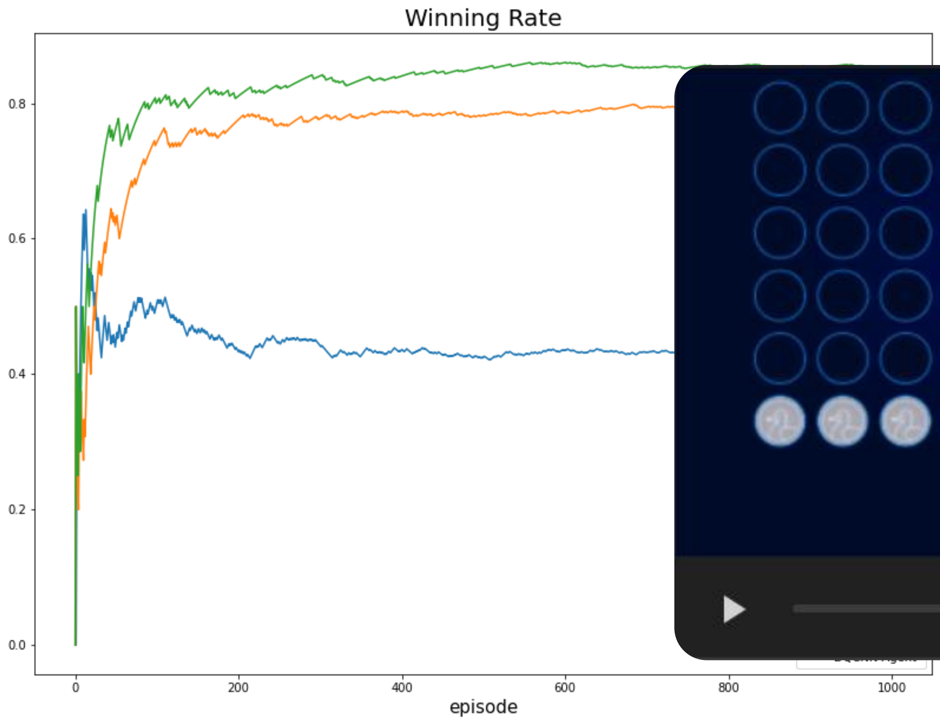
```python
def Reward(winner, done):
  if done:
    if winner == 1: reward = 50
    else: reward = -50
  else:
    reward = 1
  return reward
```

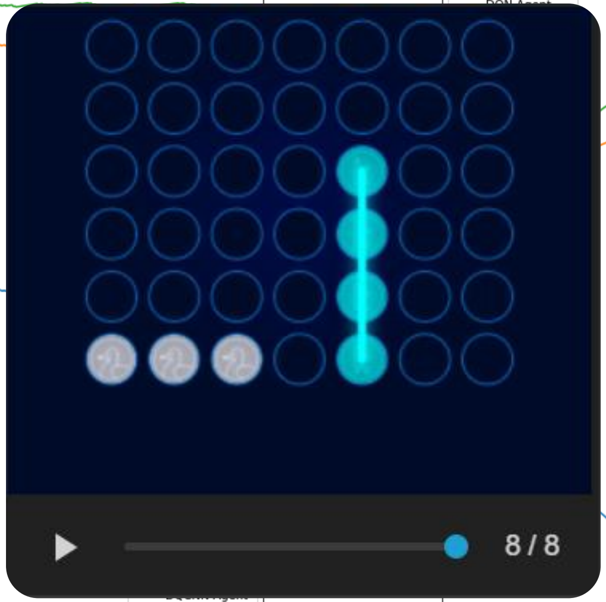


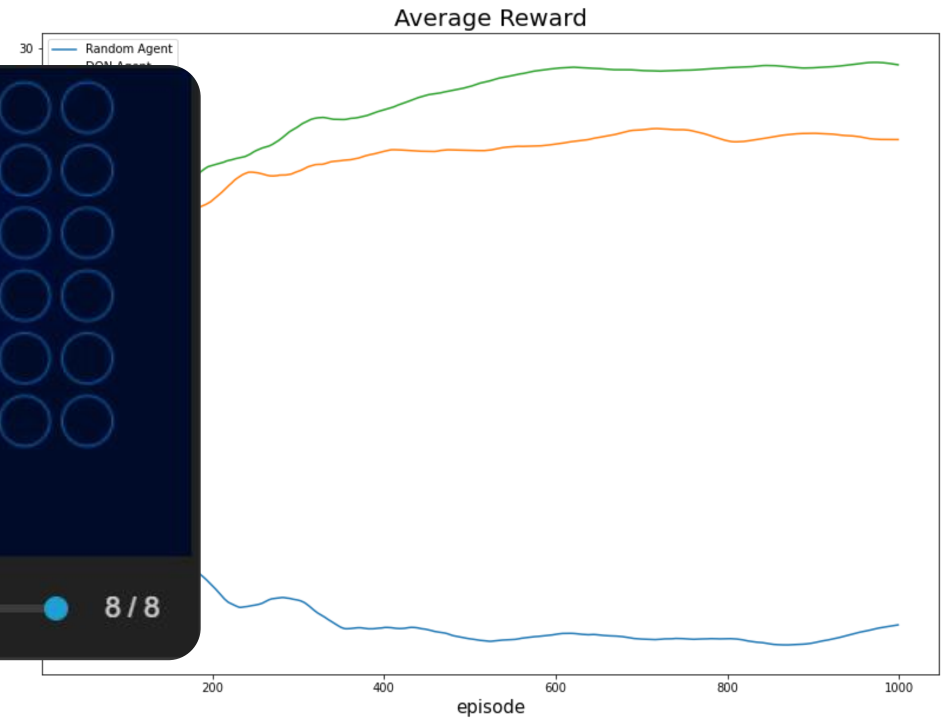
Implementation & Results

Comparing the Model with Random Agent

```python
def Reward(winner, done):
  if done:
    if winner == 1: reward = 50
    else: reward = -50
  else:
    reward = -1
  return reward
```



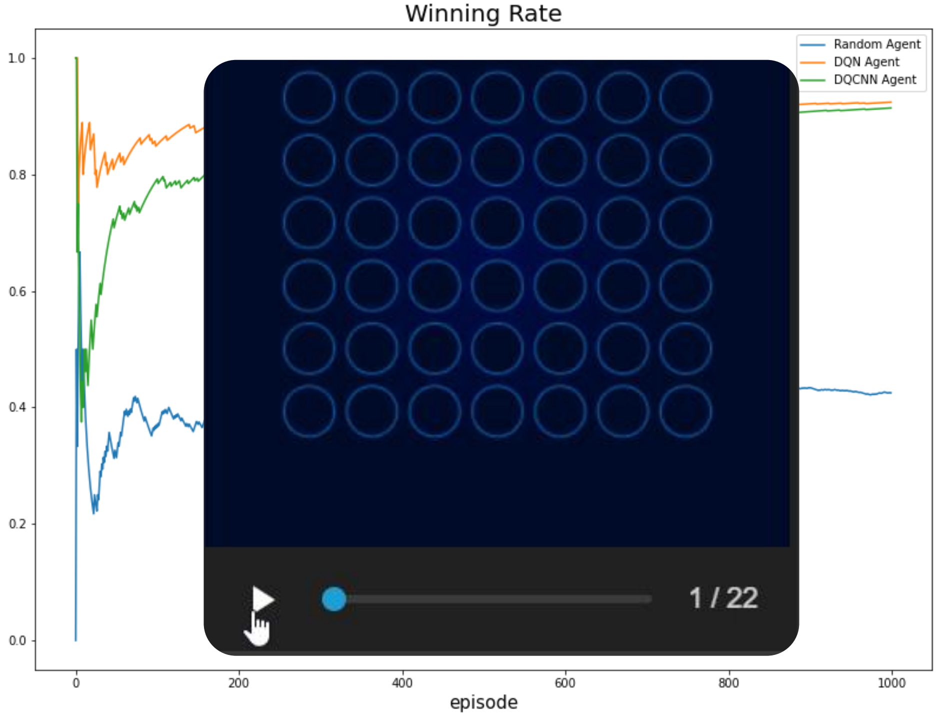


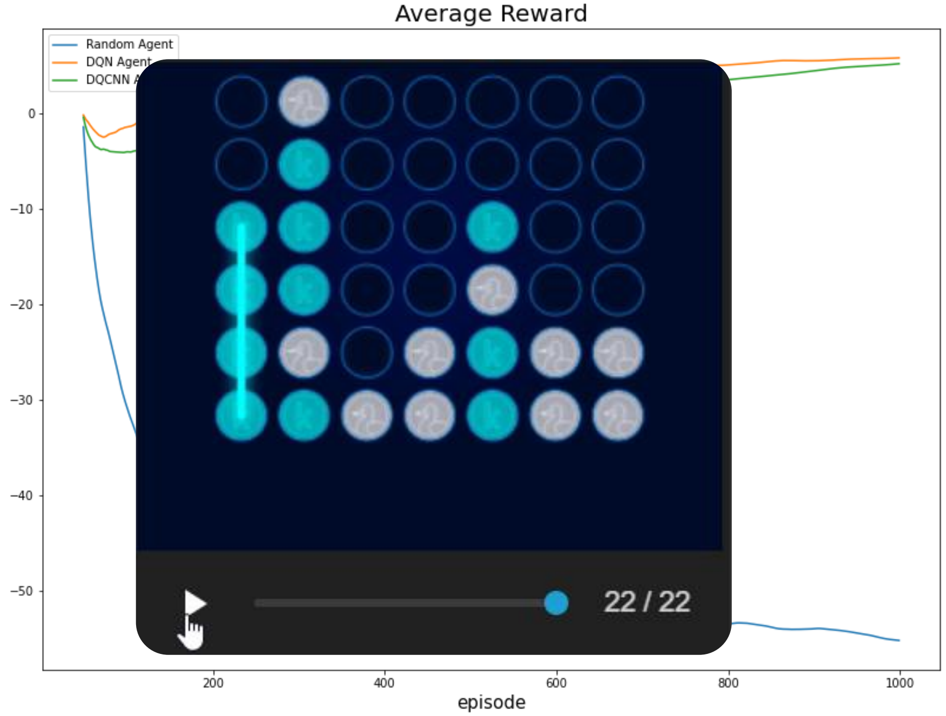


Implementation & Results

Comparing the Model with Random Agent

```python
def Reward(winner, done):
  if done:
    if winner == 1: reward = 20
    else: reward = -100
  else:
    reward = -1
  return reward
```





Thank You