PHENIKAA UNIVERSITY

FACULTY OF ELECTRICAL AND ELECTRONIC ENGINEERING

SUBJECT: ADVANCED REINFORCEMENT LEARNING

MID TERM PROJECT

Topic: Using DQN to play Flappy Bird

Student: Tran Van Do

ID: 20010737

Advisor: Vu Hoang Dieu

Hanoi, May 21th, 2024

Table of Contents

I. INTRODUCTION

1. Reinforcement learning

Reinforcement Learning (RL) is a subfield of machine learning where an agent learns to make decisions by interacting with an environment to achieve a goal. Unlike supervised learning, where the model learns from a dataset of labeled examples, RL involves learning from the consequences of actions, using rewards as feedback to guide the learning process. This introduction covers the key concepts, components, and techniques in RL, providing a foundation for understanding how RL works and its applications.

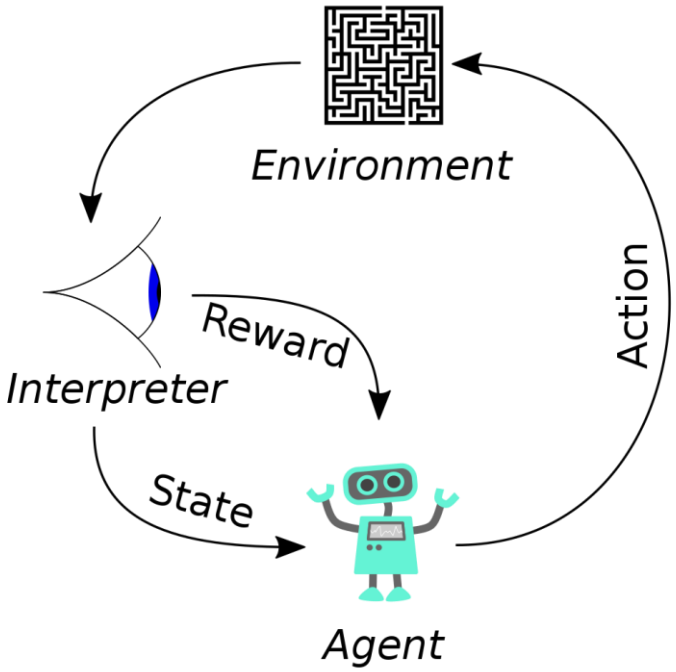


Figure 1: Reinforcement learning diagram (Reinforvement learning - Wikimedia)

Agent and Environment:

Agent: The learner or decision maker.

Environment: Everything the agent interacts with and seeks to influence.

State: A representation of the current situation the agent is in. It captures all necessary information to make a decision.

Action: A set of all possible moves the agent can take. Actions affect the state of the environment.

Reward: A scalar feedback signal indicating how good or bad the agent's action was in a given state. The goal of the agent is to maximize cumulative reward over time.

Policy: A strategy used by the agent to determine the next action based on the current state. It can be deterministic or stochastic.

Value Function: A prediction of future rewards used to evaluate states or state-action pairs. It helps the agent to anticipate long-term benefits rather than just immediate rewards.

Q-Function (Action-Value Function): A specific type of value function that evaluates the expected return of taking an action in a given state and then following a policy.

Model of the Environment: Some RL algorithms use a model of the environment to simulate its behavior, which can be particularly useful for planning.

2. Q-Learning

2.1. Q-Function

The Q-function, also known as the action-value function, is a fundamental concept in reinforcement learning (RL). It represents the expected return (cumulative future rewards) of taking a particular action in a given state and subsequently following a specific policy. This function plays a critical role in many RL algorithms, particularly in Q-learning and other value-based methods.

The Q-function, denoted as Q(s,a), evaluates the expected return starting from state ss, taking action aa, and thereafter following a policy π. Mathematically, the Q-function can be expressed as:

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a, \pi\right]$$

where:

- S_t is the state at time t.
- A_t is the action taken at time t.
- R_{t+1} is the reward received after taking action A_t.
- γ is the discount factor ($0 \leq \gamma < 1$), which determines the importance of future rewards.
- π is the policy being followed.

2.2. Q-learning

Q-learning is a popular model-free reinforcement learning algorithm used to find the optimal action-selection policy for any given finite Markov decision process (MDP). It does not require a model of the environment (i.e., it does not need to know the transition probabilities or the reward functions) and can handle problems with stochastic transitions and rewards without explicitly learning the model.

The Q-Learning Algorithm

Q-learning updates the Q-values iteratively based on the agent's experiences. The core update rule for Q-learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

where:

- s_t is the current state.
- a_t is the action taken in state s_t.
- r_{t+1} is the reward received after taking action a_t.
- s_{t+1} is the next state.
- α is the learning rate ($0 < \alpha \leq 1$), which determines the extent to which new information overrides the old information.
- γ is the discount factor ($0 \leq \gamma < 1$), which represents the importance of future rewards.
- $\max_{a'} Q(s_{t+1}, a')$ represents the maximum Q-value for the next state, indicating the best possible action from that state.

3. Deep Q-Networks (DQN)

A Deep Q-Network (DQN) is an advanced algorithm that combines Q-learning with deep neural networks. It was developed to handle complex environments with large state spaces where traditional Q-learning would be infeasible due to the size of the Q-table. DQNs were popularized by their success in training agents to play Atari 2600 games directly from pixel inputs, a milestone achieved by researchers at DeepMind.

In DQN, Deep Neural Networks are used in DQNs to approximate the Q-function, $Q(s,a;\theta)$, where θ represents the parameters of the neural network. They are capable of handling high-dimensional input spaces, such as raw pixels from images.

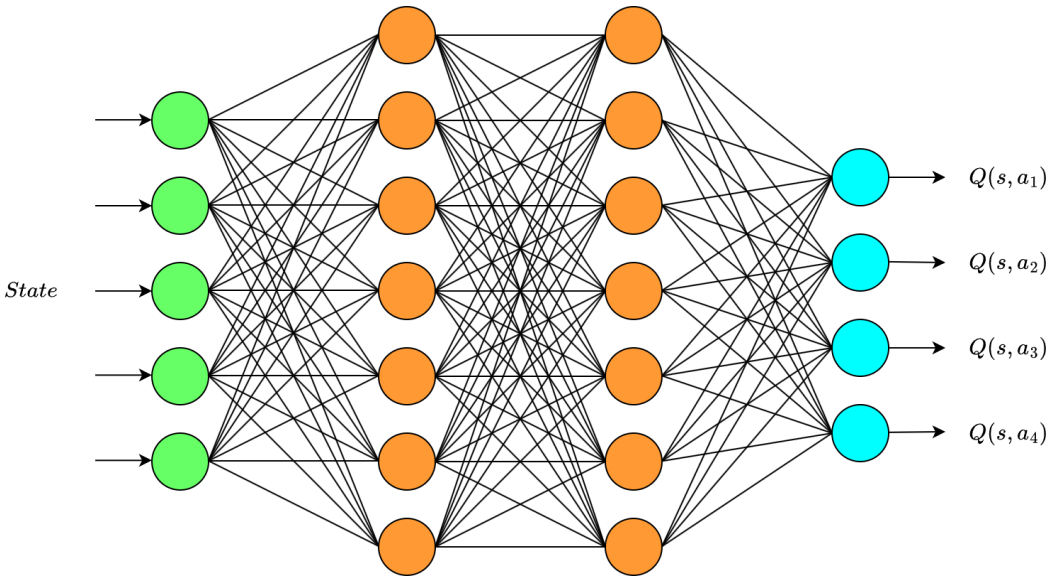


Figure 2: Q-Network (Book: Deep Learning Bible - 5. Reinforcement Learning)

4. Flappy Bird

Flappy Bird is a 2013 casual game developed by Vietnamese video game artist and programmer Dong Nguyen (Vietnamese: Nguyễn Hà Đông), under his game development company .Gears. The game is a side-scroller where the player controls a bird attempting to fly between columns of green pipes without hitting them. The player's score is determined by the number of pipes they pass. Nguyen created the game over a period of several days, using the bird from a cancelled game made in 2012. (Wikipedia)

Figure 3: App icon

Flappy Bird is an arcade-style game in which the player controls the bird Faby, which moves persistently to the right. The player is tasked with navigating Faby through pairs of pipes that have equally sized gaps placed at random heights. Faby automatically descends and only ascends when the player taps the touchscreen. Each successful pass through a pair of pipes awards the player one point. Colliding with a pipe or the ground ends the gameplay. During the game over screen, the player is awarded a bronze medal if they reached ten or more points, a silver medal from twenty points, a gold medal from thirty points, and a platinum medal from forty points.

II. ENVIRONMENT

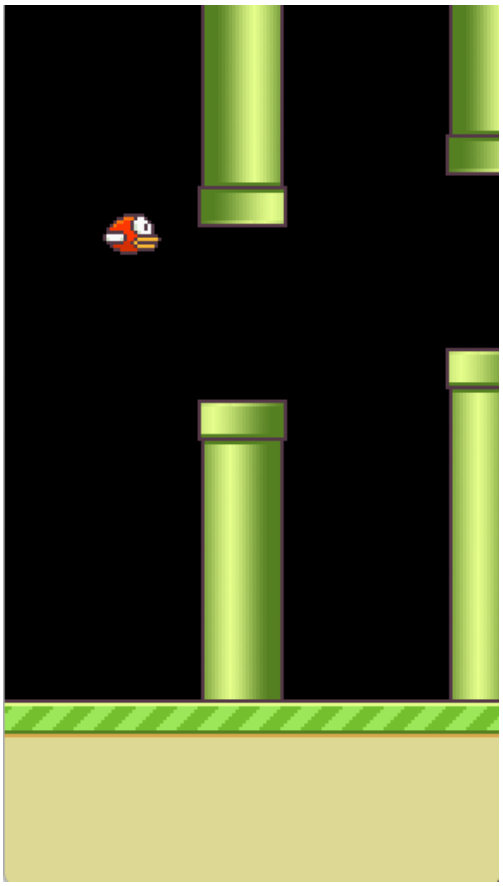


Figure 4: Flappy bird game

Action Space: 2 actions

- 0: NOOP
- 1: flap wings

Observation Space:

The observation will be the RGB image that is displayed to a human player with observation space Box(low=0, high=255, shape=(512, 288, 3), dtype=np.uint8).

Rewards: -1, 0, 1

You get +1 every time you pass a pipe, otherwise +0.0 for each frame where you don't collide against the top and bottom bounds, or against a pipe and -1 when you collide against the top and bottom bounds, or against a pipe.

III. IMPLEMENTATION

1. Deep Q-Network

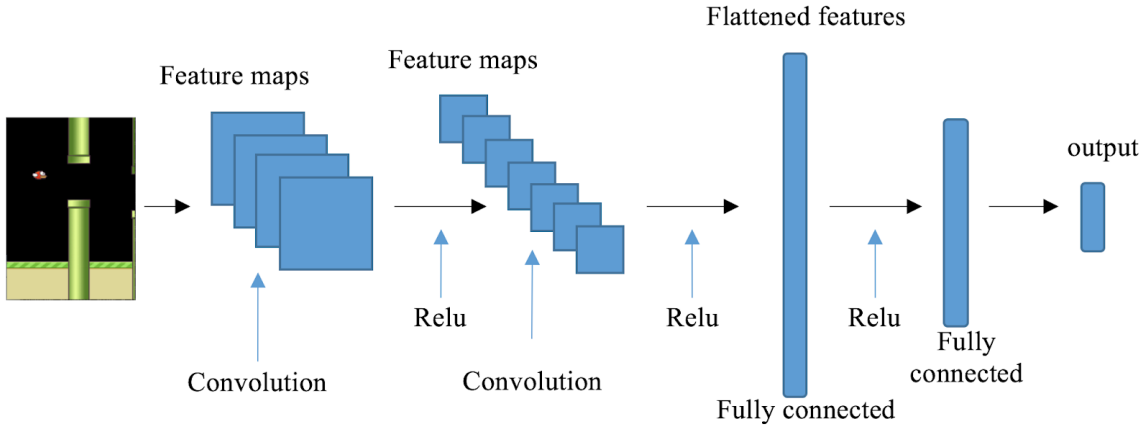


Figure 5: Deep Network to play FlappyBird with DQN

DQN network: includes 3 2D Convolution layers and 2 Fully Connected layers with activation as ReLU.

```
class DQN(nn.Module):
    def __init__(self, input_shape=[4,84,84], nactions=2):
        super(DQN, self).__init__()
        self.nactions = nactions
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0],32,kernel_size=4,stride=2),
            nn.ReLU(),
            nn.Conv2d(32,64,kernel_size=3,stride=2),
            nn.ReLU(),
            nn.Conv2d(64,64,kernel_size=2,stride=1),
            nn.ReLU()
        )
        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, nactions)
        )
    def _get_conv_out(self,shape):
        o = self.conv( torch.zeros(1,*shape) )
        return int(np.prod(o.size()))
    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        x = self.fc(conv_out)
        return x
```

2. Double DQN

Double DQN Network: Online Network and Target Network both use the same model as DQN Network

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 6: Double DQN algorithm "Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015)

3. Dueling DQN

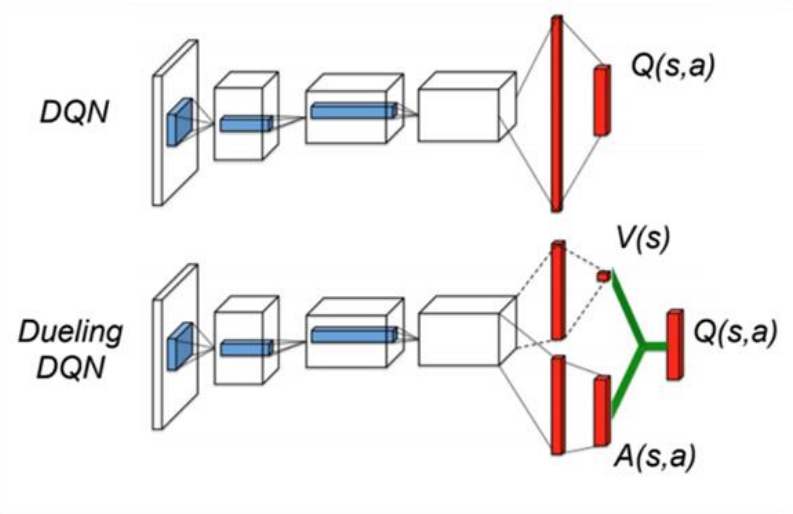


Figure 7: Single stream Q-network (top) and the dueling Q-Networking (bottom)

DQN Dueling Network: includes 3 2D Convolution layers and 2 Fully Connected layers for state value (V) and 2 Fully Connected layers for advantage value (A).

The last module uses the following mapping:

$$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + \left(A(s,a,\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

```
class DDQN(nn.Module):
  def __init__(self,input_shape,nactions):
    super(DDQN,self).__init__()
    self.nactions = nactions
    self.conv = nn.Sequential(
      nn.Conv2d(input_shape[0],32,kernel_size=4,stride=2),
      nn.ReLU(),
```

```
            nn.Conv2d(32,64,kernel_size=3,stride=2),
            nn.ReLU(),
            nn.Conv2d(64,64,kernel_size=2,stride=1),
            nn.ReLU()
        )
        conv_out_size = self._get_conv_out(input_shape)

        self.fca = nn.Sequential(
            nn.Linear( conv_out_size, 512),
            nn.ReLU(),
            nn.Linear( 512, nactions )
        )
        self.fcv = nn.Sequential(
            nn.Linear(conv_out_size,512),
            nn.ReLU(),
            nn.Linear(512,1)
        )
    def _get_conv_out(self,shape):
        o = self.conv( torch.zeros(1,*shape) )
        return int(np.prod(o.size()))

    def forward(self,x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        action_v = self.fca(conv_out)
        value_v = self.fcv(conv_out).expand(x.size(0), self.nactions)
        return value_v + action_v - action_v.mean(1).unsqueeze(1).expand(x.size(0), self.nactions)
```

IV. HYPERPARAMETER TUNING

1. Hyperparametter

```
# Action of Agent
ACTIONS = [0,1]
# Size of buffer experience
EXPERIENCE_BUFFER_SIZE = 2000
# Number of images in one train
STATE_DIM = 4
# Gamma
GAMMA = 0.99
# Epsilon, decreasing with each epochs
EPSILON_START = 1
EPSILON_FINAL = 0.001
EPSILON_DECAY_FRAMES = (10**4)/3
# Number of goals to achieve
MEAN_GOAL_REWARD = 10
# Batch_size: Number of buffers per training session
BATCH_SIZE = 32
# Minimum amount of Buffer Experience memory to be able to train
MIN_EXP_BUFFER_SIZE = 500
# Frame frequency to synchronize Target Network for Online Network
SYNC_TARGET_FRAMES = 30
# learning rate
LEARNING_RATE = 1e-4
# skip_frame: number of frames performed with the previous action
```

```
SKIP_FRAME = 2
# Policy for each game's restart.
INITIAL_SKIP = [0,1,0,1,0,1,0,0,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,0,1]
# Maximum number of training epochs
EPOCHS_MAX = 5000
```

2. Function

2.1. Preprocess state for training

Cropping image regions does not affect training.

Resize the image to 84x84 size

Use a layer of Filter2D to sharpen the image

Convert to black and white image, bring pixel value to about 0-1.

```
KERNEL = np.array([[-1,-1,-1], [-1, 9,-1],[-1,-1,-1]])
IMAGE_SIZE = 84
def processFrame(image):
    # env.SCREENWIDTH = 288
    # env.BASEY = 404
    frame = image[55:288, :404] # crop image ->
    frame = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY) # convert image to black and white
    frame = cv2.resize(frame,(IMAGE_SIZE,IMAGE_SIZE),interpolation=cv2.INTER_AREA)
    _ , frame = cv2.threshold(frame,50,255,cv2.THRESH_BINARY)
    frame = cv2.filter2D(frame,-1,KERNEL)
    frame = frame.astype(np.float64)/255.0
    return frame
```

2.2. Creat Agent

```
class Agent():
    def __init__(self,env,buffer,t_model="DQN",state_buffer_size = STATE_DIM):
        self.env = env  # game Flappy Bird
        self.exp_buffer = buffer    # memory buffer
        self.t_model = t_model      # choose Agent for model: DQN or DDQN with PER
        self.state_buffer_size = state_buffer_size  # Number of images in the buffer one train
        self.state = collections.deque(maxlen = self.state_buffer_size)
        self.next_state= collections.deque(maxlen = self.state_buffer_size)
        self._reset()

    # Restart game with fix start policy in INITIAL_SKIP
    def _reset(self):
        self.total_rewards = 0
        self.state.clear()
        self.next_state.clear()

        for i in INITIAL_SKIP[:-7]:
            image,reward,done = self.env.frame_step(i)
            self.total_rewards+=reward
```

```
            if done:
                self._reset()
        frame = processFrame(image)
        self.state.append(frame)
        self.next_state.append(frame)

        for i in INITIAL_SKIP[-7:-5]:
            image,reward,done = self.env.frame_step(i)
            self.total_rewards+=reward
            if done:
                self._reset()
        frame = processFrame(image)
        self.state.append(frame)
        self.next_state.append(frame)

        for i in INITIAL_SKIP[-5:-3]:
            image,reward,done = self.env.frame_step(i)
            self.total_rewards+=reward
            if done:
                self._reset()
        frame = processFrame(image)
        self.state.append(frame)
        self.next_state.append(frame)

        for i in INITIAL_SKIP[-3:-1]:
            image,reward,done = self.env.frame_step(i)
            self.total_rewards+=reward
            if done:
                self._reset()
        frame = processFrame(image)
        self.state.append(frame)
        self.next_state.append(frame)

    # Agent takes a action -> Return reward.
    def step(self,net,tgt_net,epsilon=0.9,device='cpu'):
        # reward in this step
        self.total_rewards = 0
        # Exploration - Exploitation
        if np.random.random() < epsilon:
            action = np.random.choice(ACTIONS)
        else:
            state_v = torch.tensor(np.array([self.state],copy=False),dtype=torch.float32).to(device)
            action = int(torch.argmax(net(state_v)))
        # Take action
        image,reward,done = self.env.frame_step(action)
        self.total_rewards += reward
        # Take with same previous action for SKIP_FRAME next frame.
        for _ in range(SKIP_FRAME):
                image,reward,done =  self.env.frame_step(action)
                self.total_rewards += reward
                if done:
                    break
```

```
        frame = processFrame(image)
        self.next_state.append(frame)
        # Add buffer to memory Experience Buffer
        # If DQN: (state, action, reward, done, next_state)
        # If PER: ((state, action, reward, done, next_state), p)
        if len(self.next_state)==self.state_buffer_size and len(self.state)==self.state_buffer_size:
            if self.t_model == "DQN":
                self.exp_buffer.append((self.state.copy(),action,int(self.total_rewards),done,self.next_state.copy()))
            else:
                #PER - Prioritized Experience Replay
                # o =
                o = net( torch.tensor(
np.array([self.state]),dtype=torch.float32).to(device)).to('cpu').detach().numpy()[0][action]
                e = float(torch.max(tgt_net( torch.tensor(
np.array([self.next_state]),dtype=torch.float32).to(device))))
                p = abs(o-e)+0.0001
                self.exp_buffer.append((self.state.copy(),action,int(self.total_rewards),done,self.next_state.copy()),p)

        self.state.append(frame)

        end_reward = int(self.total_rewards)
        if done:
            self._reset()

        return end_reward
```

2.3. ExperienceBuffer

```
class ExperienceBuffer():
    # initial memory buffer
    def __init__(self,capacity):
        self.buffer = collections.deque(maxlen=capacity)
    # clear memory buffer
    def clear(self):
        self.buffer.clear()

    def __len__(self):
        return len(self.buffer)

    def append(self,exp):
        self.buffer.append(exp)
    # get sample in memory buffer with size=batch_size
    def sample(self,batch_size):
        indices = np.random.choice( range(len(self.buffer)), batch_size)
        states,actions,rewards,dones,next_states = zip(*[ self.buffer[idx] for idx in indices ])
        return np.array(states), np.array(actions), np.array(rewards, dtype=np.float32),\
    np.array(dones,dtype=np.uint8), np.array(next_states)
```

2.4. ExperienceBuffer for PER

```
class ExperienceBufferPER():
    # initial memory buffer and priority
    def __init__(self,capacity):
        self.buffer = collections.deque(maxlen=capacity)
        self.priority = collections.deque(maxlen=capacity)
    # clear memory buffer and priority
    def clear(self):
        self.buffer.clear()
        self.priority.clear()

    def __len__(self):
        return len(self.buffer)

    def append(self,exp,p):
        self.buffer.append(exp)
        self.priority.append(p)
    # get sample in memory buffer with size=batch_size with probability based on priority
    def sample(self,batch_size):
        probs = np.array(self.priority)/sum(np.array(self.priority))
        indices = np.random.choice( range(len(self.buffer)), batch_size, p = probs)
        states,actions,rewards,dones,next_states = zip(*[ self.buffer[idx] for idx in indices ])
        return np.array(states), np.array(actions), np.array(rewards, dtype=np.float32),\
    np.array(dones,dtype=np.uint8), np.array(next_states)
```

2.5. Calculate Loss for DQN

```
def calc_loss_DQN(batch,net,device='cpu'):
    # get data from batch (choosed in memory) and conver to torch Tensor and device
    states,actions,rewards,dones,next_states = batch
    states_v = torch.tensor(states,dtype=torch.float32).to(device)
    actions_v = torch.tensor(actions,dtype=torch.long).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    dones_v = torch.ByteTensor(dones).to(device)
    next_states_v = torch.tensor(next_states,dtype=torch.float32).to(device)
    # Predict Q values for state
    state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
    # Predict values for next state
    with torch.no_grad():
        next_state_action_values = net(next_states_v)
    # Q values with Bellman Equation
    next_state_action_values = rewards_v + GAMMA * ((1-
dones_v)*torch.max(next_state_action_values,dim=1)[0])
    expected_values = next_state_action_values.detach()
    # retun loss
    return nn.MSELoss()(state_action_values,expected_values)
```

2.6. Calculate Loss for DDQN

```
def calc_loss_DDQN(batch,net,tgt_net,device='cpu'):
    # get data from batch (choosed in memory) and conver to torch Tensor and device
```

```
    states,actions,rewards,dones,next_states = batch
    states_v = torch.tensor(states,dtype=torch.float32).to(device)
    actions_v = torch.tensor(actions,dtype=torch.long).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    dones_v = torch.ByteTensor(dones).to(device)
    next_states_v = torch.tensor(next_states,dtype=torch.float32).to(device)
    # Predict Q values for state by online network
    state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
    # Predict Q values for state by target network
    next_state_action_values = tgt_net(next_states_v).max(1)[0]
    # next_state_action_values[dones_v] = 0.0
    next_state_action_values = next_state_action_values.detach()
    expected_values = rewards_v +  next_state_action_values * GAMMA
    # retun loss
    return nn.MSELoss()(state_action_values,expected_values)
```

V.TRAINING

1. Deep Q-Network with experience replay

```
# Memory to save data log (loss, epsilons and rewards) during training
all_losses = []
all_epsilons = []
all_rewards = []
# Define network
net = DQN((STATE_DIM,IMAGE_SIZE,IMAGE_SIZE), len(ACTIONS)).to(device)
# Initialize the environment, Agent
env = game.GameState()
buffer = ExperienceBuffer(EXPERIENCE_BUFFER_SIZE) # Using ExperienceBuffer with DQN
agent = Agent(env,buffer,t_model="DQN") # Using Agent with DQN
epsilon = EPSILON_START
optimizer = optim.Adam(net.parameters(),lr=LEARNING_RATE)
total_rewards = []  # List of rewards during training
best_mean_reward = float('-inf')    # value best_mean_reward
last_mean = float('-inf')   # the last mean_reward is saved
game_id = 0 # number of games played
while game_id < EPOCHS_MAX:
    # Set epsilon, decreasing with each epochs
    epsilon = max( EPSILON_FINAL , EPSILON_START - game_id/EPSILON_DECAY_FRAMES )
    # take action
    reward = agent.step(net,0,epsilon,device=device)
    if reward != 0: # when die or overcome obstacles
        game_id += 1
        total_rewards.append(reward)
        mean_reward = np.mean(total_rewards[-100:]) # get mean reward from the last 100 reward changes
        if game_id%10 == 0:
            print("GAME : {} | EPSILON : {:.4f} | MEAN REWARD : {}".format( game_id, epsilon,
mean_reward ))
        if best_mean_reward < mean_reward:  # save best_mean_reward
            best_mean_reward = mean_reward
```

```
            if best_mean_reward - last_mean >= 0.1: # save last mean_reward and model parameter when
mean_reward increase by at least 0.1
                torch.save(net.state_dict(),'checkpoints/best_DQN.dat')
                print("REWARD {} -> {}. Model Saved".format(last_mean,mean_reward))
                last_mean = best_mean_reward
        # Stop training when achieve the set target number
        if mean_reward >= MEAN_GOAL_REWARD:
            print("Learned in {} Games.".format(game_id))
            break
    # Save the minimum number of buffers to be able to perform training
    if len(buffer) < MIN_EXP_BUFFER_SIZE:
        continue
    # Training
    optimizer.zero_grad()
    batch = buffer.sample(BATCH_SIZE)   # get sample with BATCH_SIZE buffers
    loss_t = calc_loss_DQN(batch,net,device=device)
    all_losses.append(float(loss_t))
    all_epsilons.append(float(epsilon))
    all_rewards.append(mean_reward)
    loss_t.backward()
    optimizer.step()

# dictionary of lists
dict = {'Loss': all_losses, 'Epsilon': all_epsilons, 'Reward': all_rewards}
# saving the dataframe
df = pd.DataFrame(dict)
# save to .csv
df.to_csv('log/DQN.csv')
```

2. Double DQN

```
all_losses = []
all_epsilons = []
all_rewards = []

net = DQN( (STATE_DIM,84,84), len(ACTIONS) ).to(device)
tgt_net = DQN( (STATE_DIM,84,84), len(ACTIONS) ).to(device) # have Target Network

env = game.GameState()
buffer = ExperienceBuffer(EXPERIENCE_BUFFER_SIZE) # Using ExperienceBuffer with DQN
agent = Agent(env,buffer) # Using Agent with DQN
epsilon = EPSILON_START
optimizer = optim.Adam(net.parameters(),lr=LEARNING_RATE)

total_rewards = []
best_mean_reward = float('-inf')
last_mean = float('-inf')
game_id = 0
while game_id < EPOCHS_MAX:
    epsilon = max( EPSILON_FINAL , EPSILON_START - game_id/EPSILON_DECAY_FRAMES )
```

```
    reward = agent.step(net,tgt_net,epsilon,device=device)
    if reward != 0:
        game_id += 1
        total_rewards.append(reward)
        mean_reward = np.mean(total_rewards[-100:])
        if game_id%5 == 0:
            print("GAME : {} | EPSILON : {:.4f} | MEAN REWARD : {}".format( game_id, epsilon,
mean_reward ))
        if best_mean_reward < mean_reward:
            best_mean_reward = mean_reward

            if best_mean_reward - last_mean >= 0.1:
                torch.save(net.state_dict(),'checkpoints/best_DDQN.dat')
                print("REWARD {} -> {}. Model Saved".format(last_mean,mean_reward))
                last_mean = best_mean_reward

        if game_id % SYNC_TARGET_FRAMES == 0:
            tgt_net.load_state_dict(net.state_dict())

        if mean_reward >= MEAN_GOAL_REWARD:
            print("Learned in {} Games.".format(game_id))
            break

    if len(buffer) < MIN_EXP_BUFFER_SIZE:
        continue

    optimizer.zero_grad()
    batch = buffer.sample(BATCH_SIZE)
    loss_t = calc_loss_DDQN(batch,net,tgt_net,device=device)    # Using calc_loss_DDQN with DDQN
    all_losses.append(float(loss_t))
    all_epsilons.append(float(epsilon))
    all_rewards.append(mean_reward)
    loss_t.backward()
    optimizer.step()

# dictionary of lists
dict = {'Loss': all_losses, 'Epsilon': all_epsilons, 'Reward': all_rewards}
# saving the dataframe
df = pd.DataFrame(dict)
# save to .csv
df.to_csv('log/DDQN.csv')
```

3. Dueling DDQN with Prioritized Experience Replay

```
all_losses = []
all_epsilons = []
all_rewards = []

net = DDQN( (STATE_DIM,84,84), len(ACTIONS) ).to(device)
tgt_net = DDQN( (STATE_DIM,84,84), len(ACTIONS) ).to(device)

env = game.GameState()
```

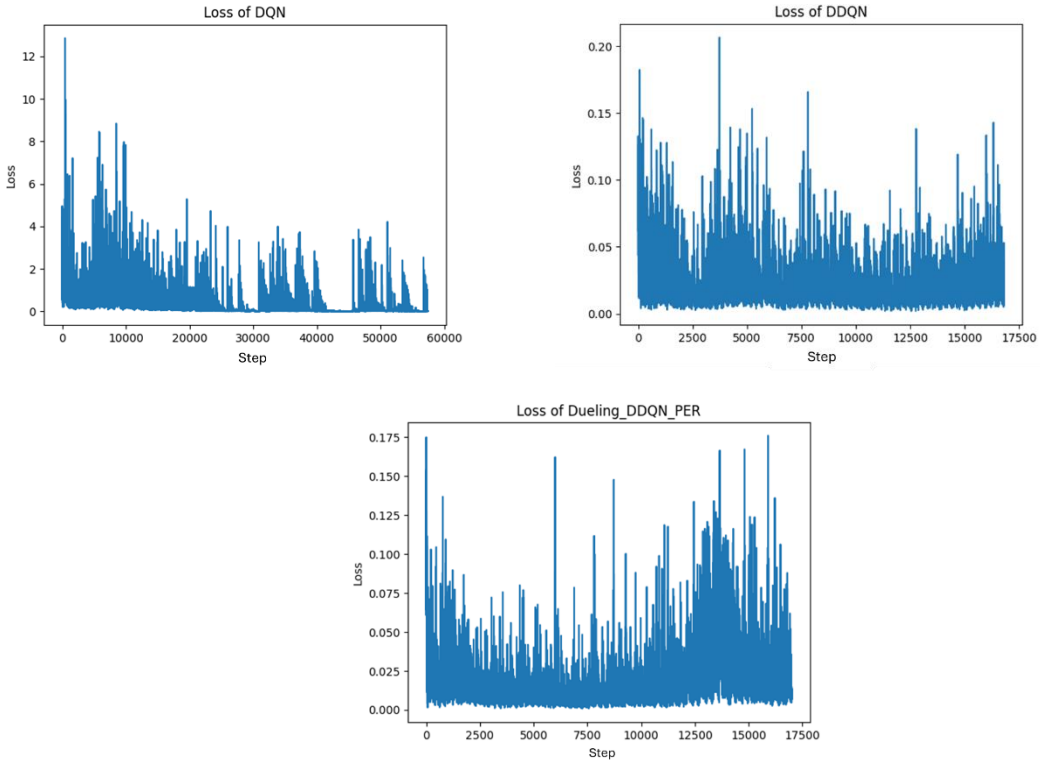
```
buffer = ExperienceBufferPER(EXPERIENCE_BUFFER_SIZE)    # Using ExperienceBufferPER with PER
agent = Agent(env,buffer, t_model="PER")    # Using Agent with PER
epsilon = EPSILON_START
optimizer = optim.Adam(net.parameters(),lr=LEARNING_RATE)

total_rewards = []
best_mean_reward = float('-inf')
last_mean = float('-inf')
game_id = 0
while game_id < EPOCHS_MAX:
    epsilon = max( EPSILON_FINAL , EPSILON_START - game_id/EPSILON_DECAY_FRAMES )

    reward = agent.step(net,tgt_net,epsilon,device=device)
    if reward != 0:
        game_id += 1
        total_rewards.append(reward)
        mean_reward = np.mean(total_rewards[-100:])
        if game_id%5 == 0:
            print("GAME : {} | EPSILON : {:.4f} | MEAN REWARD : {}".format( game_id, epsilon,
mean_reward ))
        if best_mean_reward < mean_reward:
            best_mean_reward = mean_reward

            if best_mean_reward - last_mean >= 0.1:
                torch.save(net.state_dict(),'checkpoints/best_DuelingDQN_PER.dat')
                print("REWARD {} -> {}. Model Saved".format(last_mean,mean_reward))
                last_mean = best_mean_reward

        if game_id % SYNC_TARGET_FRAMES == 0:
            tgt_net.load_state_dict(net.state_dict())

        if mean_reward >= MEAN_GOAL_REWARD:
            print("Learned in {} Games.".format(game_id))
            break

    if len(buffer) < MIN_EXP_BUFFER_SIZE:
        continue

    optimizer.zero_grad()
    batch = buffer.sample(BATCH_SIZE)
    loss_t = calc_loss_DDQN(batch,net,tgt_net,device=device)    # Using calc_loss_DDQN with DDQN
    all_losses.append(float(loss_t))
    all_epsilons.append(float(epsilon))
    all_rewards.append(mean_reward)
    loss_t.backward()
    optimizer.step()

# dictionary of lists
dict = {'Loss': all_losses, 'Epsilon': all_epsilons, 'Reward': all_rewards}
# saving the dataframe
df = pd.DataFrame(dict)
# save to .csv
df.to_csv('log/Dueling_DDQN.csv')
```

VI. EVALUATION AND COMPARISON

Due to insufficient resources for the training process, Google Colab was used to use T4 GPU.

Due to limitations in GPU usage time, the first time the 3 models were trained with several epochs of 2000. We get the results as shown:

Figure 8: Loss values of DQN, DDQN, Dueling DDQN with PER



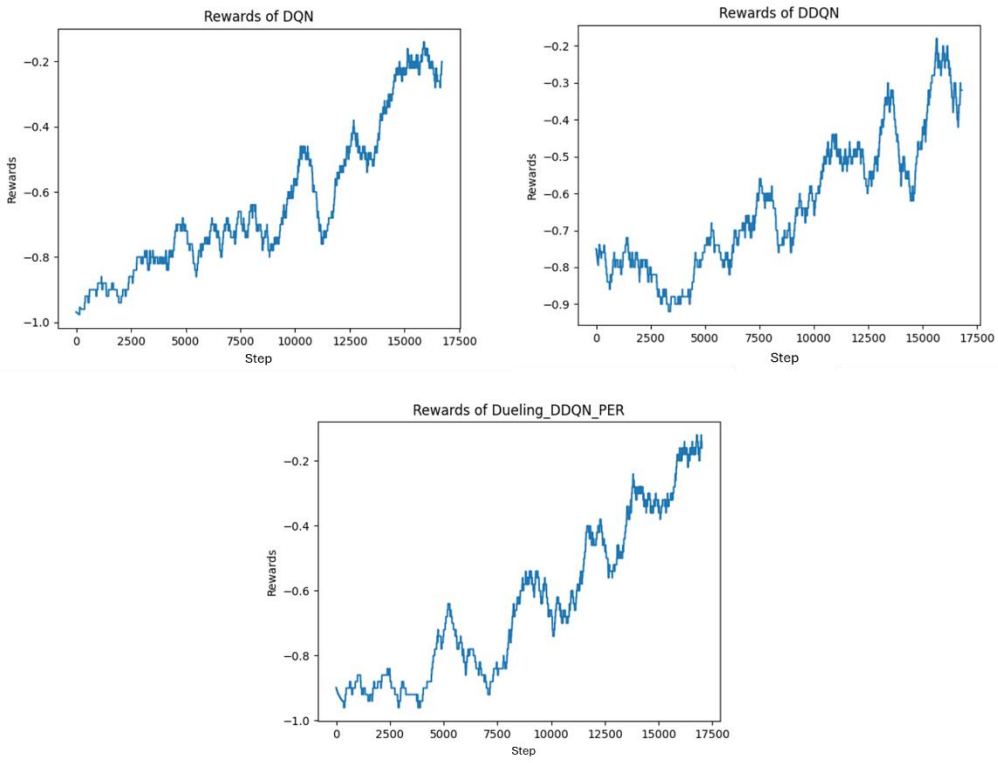


Figure 9: Reward of DQN, DDQN and Dueling DDQN with PER (train 2000 epochs)

After completing training for 2000 epochs, continue training for another 5000 epochs. Due to limitations in GPU usage time on Google Colab, only Deep Q-Network can be trained.

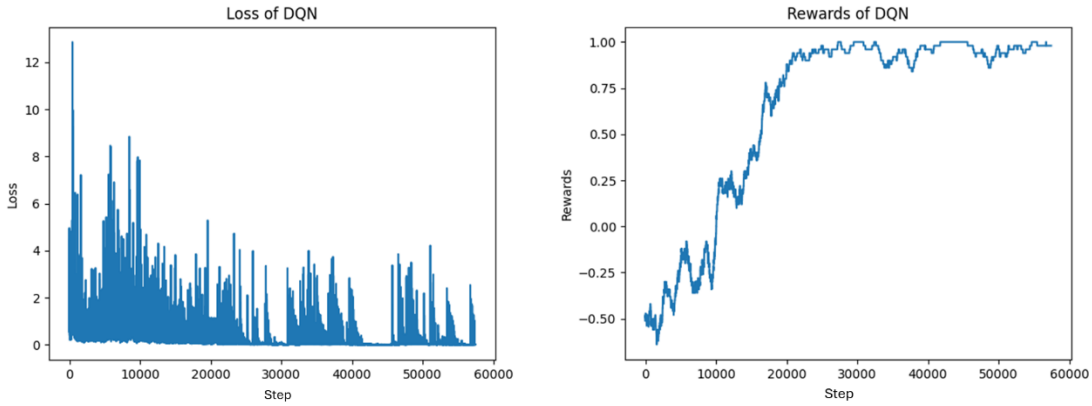


Figure 10: Loss and rewards of DQN after traing 5000 epochs

With a training amount of 2000 – 5000 epochs, the Agent can only achieve a score of 1. But, as the results show, the model training process is likely to improve with continued training.

Challenges encountered and how they were addressed.

Challenges encountered:

- Select hyperparameters:
  + Decrease of epsilon after each game
  + Number of epochs for training: due to limited computational resources.

How we were addressed:

- Use multiple Google accounts to get computing resources.
- Training tests to choose a reasonable epsilon reduction.

Github link: https://github.com/dotvision-f/PhenikaaUni_DRL/tree/main/Midterm

REFERENCES

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. "*Playing Atari with Deep Reinforcement Learning*".2013.

Hado van Hasselt, Arthur Guez, David Silver. *"Deep Reinforcement Learning with Double Q-learning"*. 2015

Ziyu Wang, Tom Schaul, Matteo Hessel, et al. *"Dueling Network Architectures for Deep Reinforcement Learning"*. 2016

Tom Schaul, John Quan, Ioannis Antonoglou, David Silver. *"Prioritized Experience Replay"*. 2016

FlappyBird Enviroment: [sourabhv/FlapPyBird] (https://github.com/sourabhv/FlapPyBird)