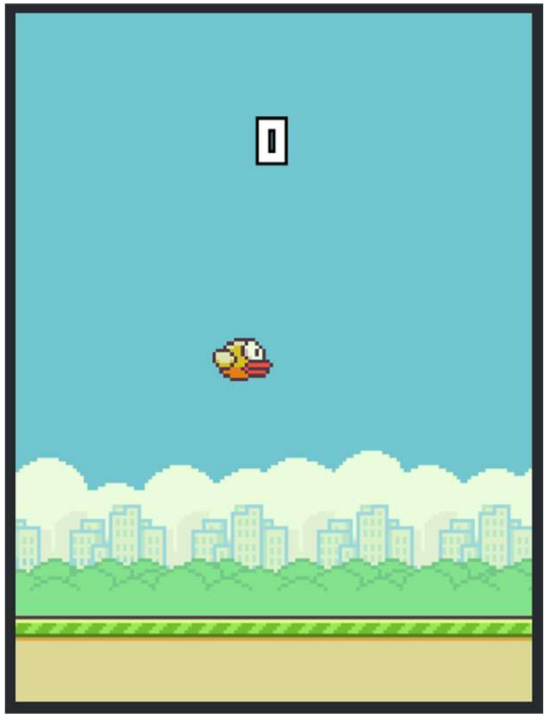
Practice: Training Agents to Play Games

Flappy Bird

- A very simple game with a single action 'flap'.

《**플래피 버드**》(영어: Flappy Bird)는 베트남의 게임 개발자 응우옌하동(베트남어: Nguyễn Hà Đông)이 2013년 개발한 모바일 게임이다. 2013년 5월 24일 정식으로 공개되었으나, 2014년 2월 10일 개발자의 요청으로 삭제되었다. 개발자는 자신의 게임이 몇분정도 즐길 수 있게 하는 목적이었으나 많은사람들이 몇시간씩 중독되어서 삭제를 요청했다고 한다.

- You control the bird and go past the pipes without touching them.
- You should also not touch the ground.
- The bird constantly falls to the ground, so you must 'flap' to keep the bird up.

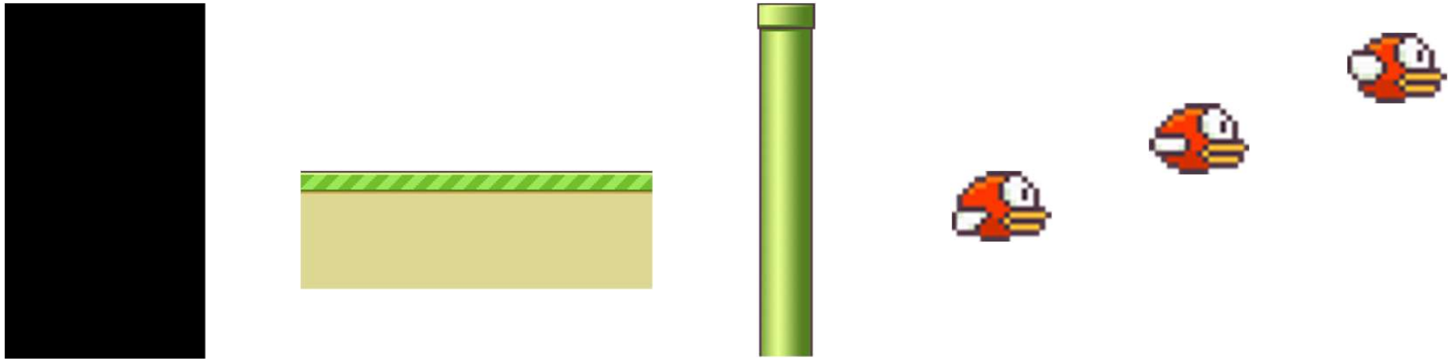


- First, you try it.
  - http://flappybird.io/

A pygame Implementation of FlappyBird

- The game is so simple, we can implement the game with less than a few hundred lines.

- First we need to prepare sprites.
  - 6 image files (png format)
  - You can use more sprites for better graphic



- If you'd like, you can also use audio clips for sound effects.

A pygame Implementation of FlappyBird

- The "original" FlappyBird should take keyboard inputs for user interaction.
- The version here is used to train and evaluate RL agents.
- The pygame library is a set of python modules for writing video games.
  - https://www.pygame.org

```python
# https://github.com/uvipen/Flappy-bird-deep-Q-learning-pytorch

from itertools import cycle
from numpy.random import randint
from pygame import Rect, init, time, display
from pygame.event import pump
from pygame.image import load
from pygame.surfarray import array3d, pixels_alpha
from pygame.transform import rotate
import numpy as np
```

A pygame Implementation of FlappyBird

- Class "FlappyBird" is the class that defines the game.
  - Initialize display and load all sprites
  - The hitmask defines the object area within the image.


```
class FlappyBird(object):

    init()
    fps_clock = time.Clock()
    screen_width = 288
    screen_height = 512
    screen = display.set_mode((screen_width, screen_height))
    display.set_caption('Deep Q-Network Flappy Bird')
    base_image = load('assets/sprites/base.png').convert_alpha()
    background_image = load('assets/sprites/background-black.png').convert()

    pipe_images = [rotate(load('assets/sprites/pipe-green.png').convert_alpha(), 180),
                   load('assets/sprites/pipe-green.png').convert_alpha()]
    bird_images = [load('assets/sprites/redbird-upflap.png').convert_alpha(),
                   load('assets/sprites/redbird-midflap.png').convert_alpha(),
                   load('assets/sprites/redbird-downflap.png').convert_alpha()]

    bird_hitmask = [pixels_alpha(image).astype(bool) for image in bird_images]
    pipe_hitmask = [pixels_alpha(image).astype(bool) for image in pipe_images]
```

A pygame Implementation of FlappyBird

- Parameters used for the game
- The bird_index_generator is used for movement of the bird


```python
fps = 30
pipe_gap_size = 100
pipe_velocity_x = -4

# parameters for bird
min_velocity_y = -8
max_velocity_y = 10
downward_speed = 1
upward_speed = -9

bird_index_generator = cycle([0, 1, 2, 1])
```

A pygame Implementation of FlappyBird

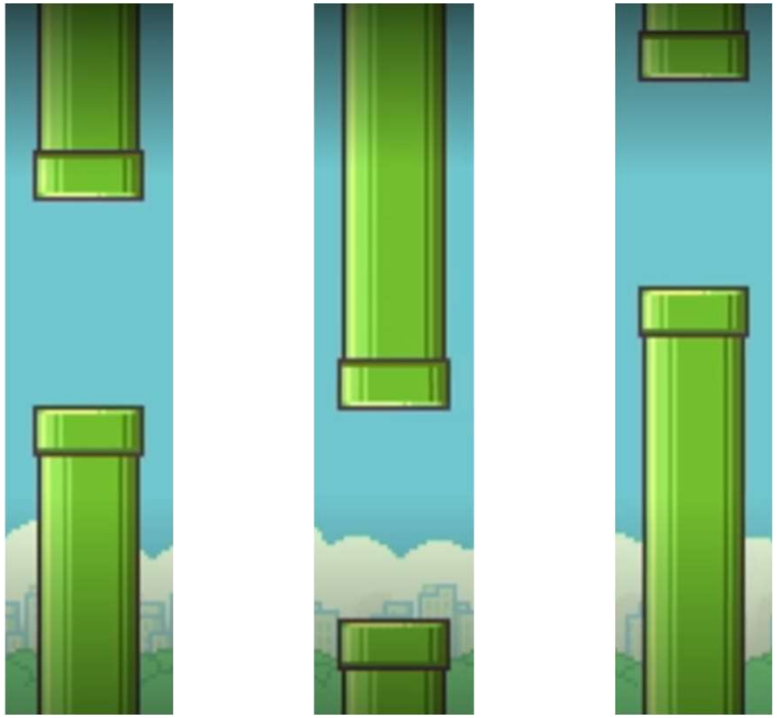
- Set the game to the initial state
  - Set the position of the bird and the base (ground).
  - Initially, two set of pipes are created. Their positions are out of the screen.

```
def __init__(self):

    self.iter = self.bird_index = self.score = 0

    self.bird_width = self.bird_images[0].get_width()
    self.bird_height = self.bird_images[0].get_height()
    self.pipe_width = self.pipe_images[0].get_width()
    self.pipe_height = self.pipe_images[0].get_height()

    self.bird_x = int(self.screen_width / 5)
    self.bird_y = int((self.screen_height - self.bird_height) / 2)

    self.base_x = 0
    self.base_y = self.screen_height * 0.79
    self.base_shift = self.base_image.get_width() - self.background_image.get_width()

    pipes = [self.generate_pipe(), self.generate_pipe()]
    pipes[0]["x_upper"] = pipes[0]["x_lower"] = self.screen_width
    pipes[1]["x_upper"] = pipes[1]["x_lower"] = self.screen_width * 1.5
    self.pipes = pipes

    self.current_velocity_y = 0
    self.is_flapped = False
```

A pygame Implementation of FlappyBird

- Pipes are generated with random gap positions between the two pipes.

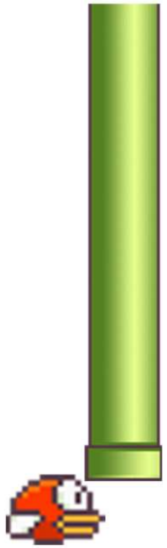
```
def generate_pipe(self):
    x = self.screen_width + 10
    gap_y = randint(2, 10) * 10 + int(self.base_y / 5)
    return {"x_upper": x, "y_upper": gap_y - self.pipe_height, "x_lower": x, "y_lower": gap_y + self.pipe_gap_size}
```



A pygame Implementation of FlappyBird

- Check whether the bird has collided with the ground or the pipes
  - First we check whether the bird touches the ground.
  - Then, we check whether the bounding boxes of the bird and the pipes overlap.
  - Even if they overlap, the bird lives unless their hitmasks actually overlap.

```python
def is_collided(self):
    # Check if the bird touch ground
    if self.bird_height + self.bird_y + 1 >= self.base_y:
        return True
    bird_bbox = Rect(self.bird_x, self.bird_y, self.bird_width, self.bird_height)
    pipe_boxes = []
    for pipe in self.pipes:
        pipe_boxes.append(Rect(pipe["x_upper"], pipe["y_upper"], self.pipe_width, self.pipe_height))
        pipe_boxes.append(Rect(pipe["x_lower"], pipe["y_lower"], self.pipe_width, self.pipe_height))
        # Check if the bird's bounding box overlaps to the bounding box of any pipe
        if bird_bbox.collidelist(pipe_boxes) == -1:
            return False
        for i in range(2):
            cropped_bbox = bird_bbox.clip(pipe_boxes[i])
            min_x1 = cropped_bbox.x - bird_bbox.x
            min_y1 = cropped_bbox.y - bird_bbox.y
            min_x2 = cropped_bbox.x - pipe_boxes[i].x
            min_y2 = cropped_bbox.y - pipe_boxes[i].y
            if np.any(self.bird_hitmask[self.bird_index][min_x1:min_x1 + cropped_bbox.width,
                      min_y1:min_y1 + cropped_bbox.height] * self.pipe_hitmask[i][min_x2:min_x2 + cropped_bbox.width,
                                                                                   min_y2:min_y2 + cropped_bbox.height]):
                return True
    return False
```



no collision

A pygame Implementation of FlappyBird

- The next_frame() is equivalent of step() in gym.
  - In this version, a reward of 0.1 is given if the bird lives through the frame.
    - You can design rewards differently.
  - There are two actions: 'flap' is 1 and 'do nothing' is 0.
  - If the bird flaps, then we set the y velocity to the upward speed.
    - The upward speed is -9. It is negative because going up means the position value is decreased.

```python
def next_frame(self, action):
    pump()
    reward = 0.1
    terminal = False

    # if the bird 'flaps' (action == 1), it will move up
    if action == 1:
        self.current_velocity_y = self.upward_speed
        self.is_flapped = True
```

A pygame Implementation of FlappyBird

- If the bird goes through the x center of the pipes, the score is incremented.
- The agent also gets +1 reward.

```python
# if the bird moves past pipes, it is rewarded with 1 point
bird_center_x = self.bird_x + self.bird_width / 2
for pipe in self.pipes:
    pipe_center_x = pipe["x_upper"] + self.pipe_width / 2
    if pipe_center_x < bird_center_x < pipe_center_x + 5:
        self.score += 1
        reward = 1
        break
```

A pygame Implementation of FlappyBird

- Update y position of the bird
  - The bird's y position is calculated based on the current position and its y velocity.
  - The bird's x position does not move.

```
# update position of the bird
if self.current_velocity_y < self.max_velocity_y and not self.is_flapped:
    self.current_velocity_y += self.downward_speed
if self.is_flapped:
    self.is_flapped = False
self.bird_y += min(self.current_velocity_y, self.bird_y - self.current_velocity_y - self.bird_height)
if self.bird_y < 0:
    self.bird_y = 0
```

A pygame Implementation of FlappyBird

- Update x position of the pipes
  - The pipes constantly move left.
  - We have multiple pipes, and they all move at a constant rate.

```
# update position of the pipes
for pipe in self.pipes:
    pipe["x_upper"] += self.pipe_velocity_x
    pipe["x_lower"] += self.pipe_velocity_x
```

A pygame Implementation of FlappyBird

- Generate new pipes and delete old pipes
  - If the leftmost pipe is very much to the left, we generate a new pipe.
  - If the leftmost pipe goes out of the screen, we delete the pipe.

```
# generate new pipes and delete old pipes
if 0 < self.pipes[0]["x_lower"] < 5:
    self.pipes.append(self.generate_pipe())
if self.pipes[0]["x_lower"] < -self.pipe_width:
    del self.pipes[0]
```

A pygame Implementation of FlappyBird

- Check if the bird has collided with the ground or the pipes
  - If the bird has collided, then the agent is rewarded -1.
    - Again, you can design the reward function differently.
  - Also, we move back to the initial state.

```python
# if the bird has collided, we reset to the initial state
if self.is_collided():
    terminal = True
    reward = -1
    self.__init__()
```

A pygame Implementation of FlappyBird

- We redraw the screen to reflect the updated positions of the sprites.

```
# draw the sprites on the display
self.screen.blit(self.background_image, (0, 0))
self.screen.blit(self.base_image, (self.base_x, self.base_y))
self.screen.blit(self.bird_images[self.bird_index], (self.bird_x, self.bird_y))
for pipe in self.pipes:
    self.screen.blit(self.pipe_images[0], (pipe["x_upper"], pipe["y_upper"]))
    self.screen.blit(self.pipe_images[1], (pipe["x_lower"], pipe["y_lower"]))
image = array3d(display.get_surface())
display.update()
```

A pygame Implementation of FlappyBird

- We advance the frame and return values to the agent.
  - The agent uses the whole image to determine states.
  - reward is needed to calculate the Q values.
  - For terminal states, the target value does not include the next state value.

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2 \qquad y_i = \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_\theta(s_i', a') & \text{if } s' \text{ is not terminal} \end{cases}$$

```
self.fps_clock.tick(self.fps)
return image, reward, terminal
```

Training Agents to Play FlappyBird

- Now we will implement an RL agent to play FlappyBird.

- The reward function was embedded in the FlappyBird class definition.
  - +1 when the bird passes through the pipes
  - -1 when the bird collides
  - +0.1 for all other timesteps

- The actions are also given.
  - 0: do nothing
  - 1: flap

- The states will be calculated from the current snapshot of the game.
  - Preprocessing is done to reduce the input size.

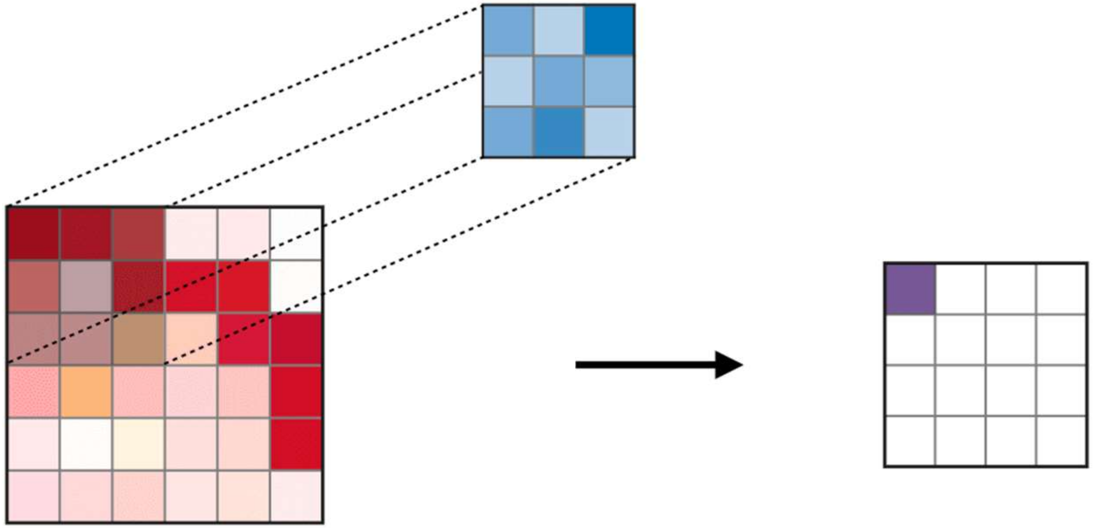
Training Agents to Play FlappyBird

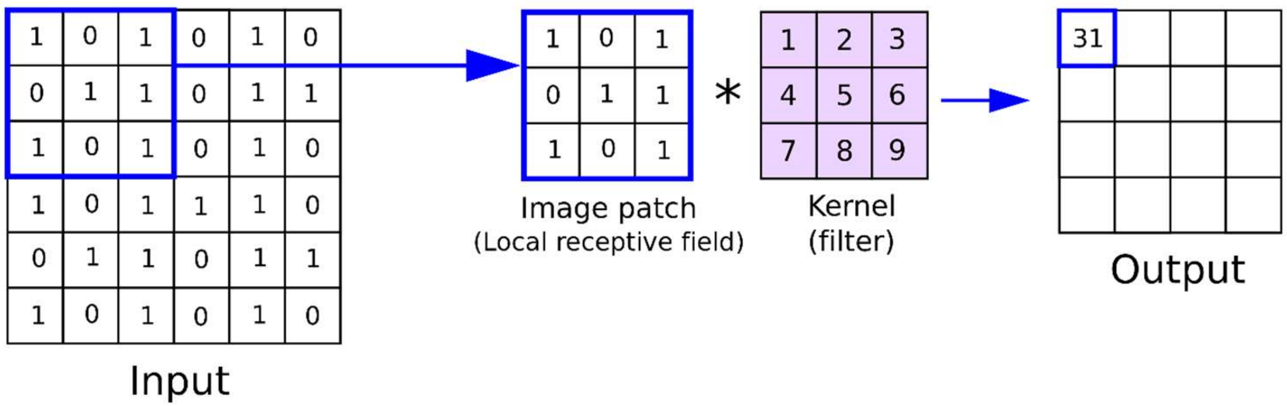
- The DQN model
  - We use a CNN (Convolutional Neural Network) to extract features from an image.

```
class DeepQNetwork(nn.Module):
    def __init__(self):
        super(DeepQNetwork, self).__init__()

        self.conv1 = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4), nn.ReLU(inplace=True))
        self.conv2 = nn.Sequential(nn.Conv2d(32, 64, kernel_size=4, stride=2), nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, stride=1), nn.ReLU(inplace=True))

        self.fc1 = nn.Sequential(nn.Linear(7 * 7 * 64, 512), nn.ReLU(inplace=True))
        self.fc2 = nn.Linear(512, 2)
        self._create_weights()

    def _create_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.uniform_(m.weight, -0.01, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, input):
        output = self.conv1(input)
        output = self.conv2(output)
        output = self.conv3(output)
        output = output.view(output.size(0), -1)
        output = self.fc1(output)
        output = self.fc2(output)

        return output
```

Training Agents to Play FlappyBird

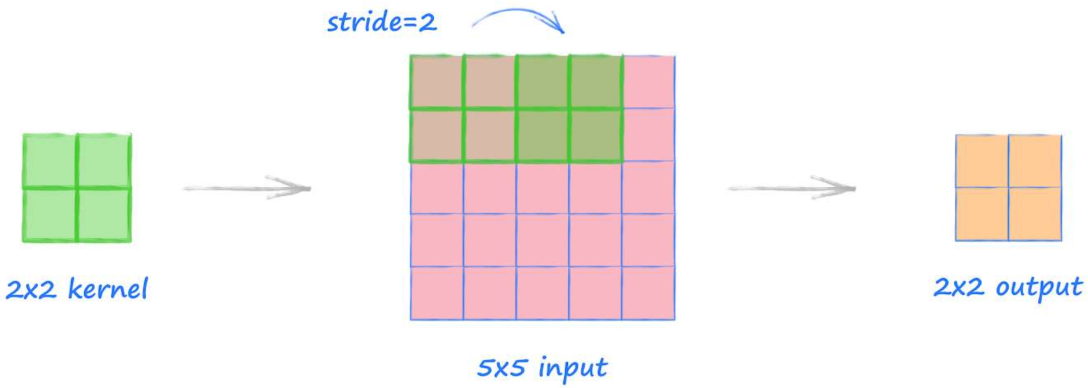
- The convolutional layer





Training Agents to Play FlappyBird

- Kernel size and stride of the convolution layers
  - After passing through convolution layer, the size of the image is reduced.




```
self.conv1 = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4), nn.ReLU(inplace=True))
self.conv2 = nn.Sequential(nn.Conv2d(32, 64, kernel_size=4, stride=2), nn.ReLU(inplace=True))
self.conv3 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, stride=1), nn.ReLU(inplace=True))

self.fc1 = nn.Sequential(nn.Linear(7 * 7 * 64, 512), nn.ReLU(inplace=True))
self.fc2 = nn.Linear(512, 2)
```

Training Agents to Play FlappyBird

- Initializing parameters
  - weights: sampled from uniform random distribution
  - biases: initialized to zero

```python
def _create_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
            nn.init.uniform_(m.weight, -0.01, 0.01)
            nn.init.constant_(m.bias, 0)
```

Training Agents to Play FlappyBird

- The forward path
  - The input dimension is (1, 4, 84, 84)
  - After 1st conv layer: (1, 32, 20, 20)
  - After 2nd conv layer: (1, 64, 9, 9)
  - After 3rd conv layer: (1, 64, 7, 7)
  - After flattening: (1, 3136)
  - After 1st fc layer: (1, 512)
  - After 2nd fc layer: (1, 2) → Q values of the two actions

```python
def forward(self, input):
    output = self.conv1(input)
    output = self.conv2(output)
    output = self.conv3(output)
    output = output.view(output.size(0), -1)
    output = self.fc1(output)
    output = self.fc2(output)

    return output
```

Training Agents to Play FlappyBird

- Beginning of the program: getting command-line arguments

```
if __name__ == "__main__":
    opt = get_args()
    train(opt)
```

```
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("--image_size", type=int, default=84, help="The common width and height for all images")
    parser.add_argument("--batch_size", type=int, default=32, help="The number of images per batch")
    parser.add_argument("--optimizer", type=str, choices=["sgd", "adam"], default="adam")
    parser.add_argument("--lr", type=float, default=1e-6)
    parser.add_argument("--gamma", type=float, default=0.99)
    parser.add_argument("--initial_epsilon", type=float, default=0.1)
    parser.add_argument("--final_epsilon", type=float, default=1e-4)
    parser.add_argument("--num_iters", type=int, default=2000000)
    parser.add_argument("--replay_memory_size", type=int, default=50000, help="Number of epoches between testing phases")
    parser.add_argument("--saved_path", type=str, default="trained_models")

    args = parser.parse_args()
    return args
```

Training Agents to Play FlappyBird

- The function train()
  - Set a random number generator seed
  - Create a DQN model
  - Get an optimizer and a loss function
  - Create an instance of the game

```
def train(opt):
    if torch.cuda.is_available():
        torch.cuda.manual_seed(123)
    else:
        torch.manual_seed(123)
    model = DeepQNetwork()

    optimizer = torch.optim.Adam(model.parameters(), lr=opt.lr)
    criterion = nn.MSELoss()
    game_state = FlappyBird()
```

Training Agents to Play FlappyBird

- To start learning, we first perform action 0 (do nothing) and get the video frame.

- The pre_processing function resizes the image to 84x84 and also change the pixel colors to black & white.

- The state is computed by concatenating 4 images.
  - Because this is the initial state, we just concatenate the same images.
  - As we move on, the state is the history of the 4 recent image frames.

```
image, reward, terminal = game_state.next_frame(0)
image = pre_processing(image[:game_state.screen_width, :int(game_state.base_y)], opt.image_size, opt.image_size)
image = torch.from_numpy(image)
if torch.cuda.is_available():
    model.cuda()
    image = image.cuda()
state = torch.cat(tuple(image for _ in range(4)))[None, :, :, :]
```

Training Agents to Play FlappyBird

- Preprocessing of the image produced by the game.
  - We use the opencv library (cv2) for this.
  - We resize the image to our desired width and height. (84x84)
  - We change the color image into a grayscale image.
    - This reduces number of color channels from 3 to 1.
  - We then use thresholding to change all pixel values to 255 (white) except pixels with value 0 (black).

```
def pre_processing(image, width, height):
    image = cv2.cvtColor(cv2.resize(image, (width, height)), cv2.COLOR_BGR2GRAY)
    _, image = cv2.threshold(image, 1, 255, cv2.THRESH_BINARY)
    return image[None, :, :].astype(np.float32)
```

Training Agents to Play FlappyBird

- Now we are ready for the training loop
  - We first use the DQN model to obtain Q values for the two potential actions.
  - The epsilon will be decreased from 0.1 to 0.0001 as the iteration advances.
  - Now we pick a random number from 0 to 1.
  - If the random number is less than epsilon, we choose a random action.
  - Otherwise, we choose an action with higher Q value.

```
replay_memory = []
iter = 0
while iter < opt.num_iters:
    prediction = model(state)[0]
    # Exploration or exploitation
    epsilon = opt.final_epsilon + (
            (opt.num_iters - iter) * (opt.initial_epsilon - opt.final_epsilon) / opt.num_iters)
    u = random()
    random_action = u <= epsilon
    if random_action:
        print("Perform a random action")
        action = randint(0, 1)
    else:
        action = torch.argmax(prediction)
```

Training Agents to Play FlappyBird

- We perform action and get the next image as well as the reward and whether we reached a terminal state.
- The image is preprocessed.
- Then, the image is concatenated with the previous 3 images.
- The state is the history of 4 image frames.

```
next_image, reward, terminal = game_state.next_frame(action)
next_image = pre_processing(next_image[:game_state.screen_width, :int(game_state.base_y)], opt.image_size,
                            opt.image_size)
next_image = torch.from_numpy(next_image)
if torch.cuda.is_available():
    next_image = next_image.cuda()
next_state = torch.cat((state[0, 1:, :, :], next_image))[None, :, :, :]
```

Training Agents to Play FlappyBird

- We store the transition in the replay buffer.
- Then, we sample a batch of transitions from the replay buffer.
  - If the replay buffer does not have enough amount of transitions, just take all transitions.

```
replay_memory.append([state, action, reward, next_state, terminal])
if len(replay_memory) > opt.replay_memory_size:
    del replay_memory[0]
batch = sample(replay_memory, min(len(replay_memory), opt.batch_size))
state_batch, action_batch, reward_batch, next_state_batch, terminal_batch = zip(*batch)

state_batch = torch.cat(tuple(state for state in state_batch))
action_batch = torch.from_numpy(
    np.array([[1, 0] if action == 0 else [0, 1] for action in action_batch], dtype=np.float32))
reward_batch = torch.from_numpy(np.array(reward_batch, dtype=np.float32)[:, None])
next_state_batch = torch.cat(tuple(state for state in next_state_batch))
```

Training Agents to Play FlappyBird

- We calculate the target Q value and the predicted Q value of the state-action pairs in the transition.

$$y_i = r_i + \gamma \max_{a'} Q_\theta(s_i', a')$$

$$\hat{y}_i = Q_\theta(s_i, a_i)$$

```
if torch.cuda.is_available():
    state_batch = state_batch.cuda()
    action_batch = action_batch.cuda()
    reward_batch = reward_batch.cuda()
    next_state_batch = next_state_batch.cuda()
current_prediction_batch = model(state_batch)
next_prediction_batch = model(next_state_batch)

y_batch = torch.cat(
    tuple(reward if terminal else reward + opt.gamma * torch.max(prediction) for reward, terminal, prediction in
          zip(reward_batch, terminal_batch, next_prediction_batch)))

q_value = torch.sum(current_prediction_batch * action_batch, dim=1)
```

Training Agents to Play FlappyBird

- We calculate gradient and update parameters.

```
optimizer.zero_grad()
loss = criterion(q_value, y_batch)
loss.backward()
optimizer.step()
```

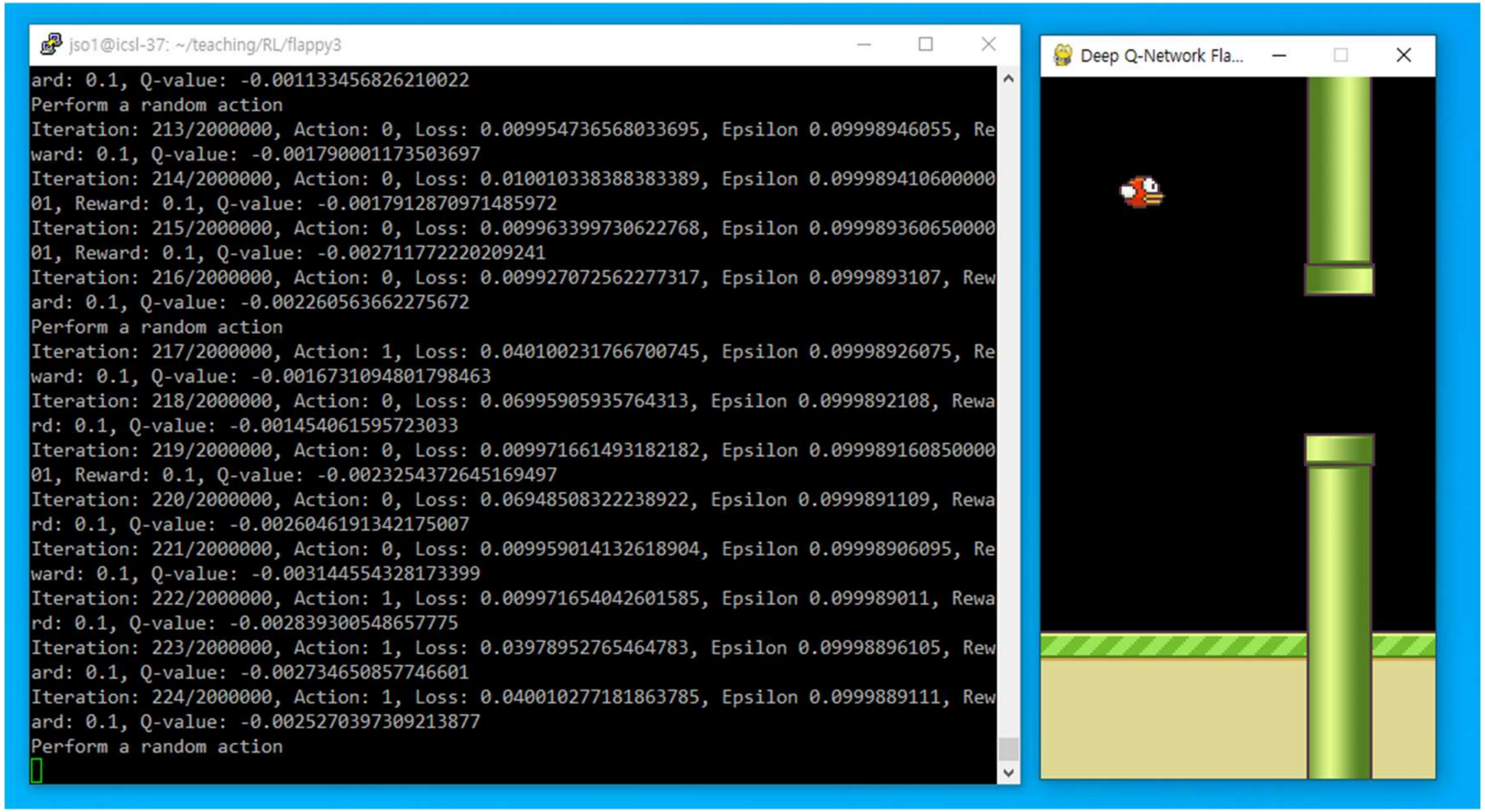
Training Agents to Play FlappyBird

- We move on to the next time step
- We print out the train information in each time step
- We save the model after every N steps.

```
    state = next_state
    iter += 1
    print("Iteration: {}/{}, Action: {}, Loss: {}, Epsilon {}, Reward: {}, Q-value: {}".format(
        iter + 1,
        opt.num_iters,
        action,
        loss,
        epsilon, reward, torch.max(prediction)))
    if (iter+1) % 100000 == 0:
        torch.save(model, "{}/flappy_bird_{}".format(opt.saved_path, iter+1))
torch.save(model, "{}/flappy_bird".format(opt.saved_path))
```

Training Agents to Play FlappyBird

- We are done!
- Now we train the agent for a few hours (or a few days) and the agent will learn to play FlappyBird very well.

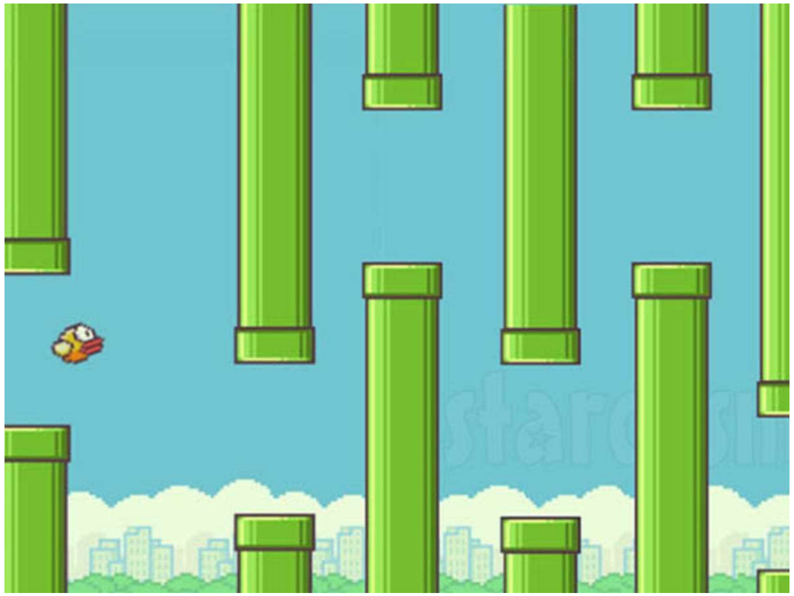


Try Other DRL Algorithms

- The code shown here is a very basic implementation of DRL using DQN.
- It doesn't even use a target network for stable learning.

- You are encouraged to implement other DRL algorithms like Policy Gradient, Actor-Critic, Proximal Policy Optimization to see if they can do better.

- You can also change the game design to make the game easier or harder!



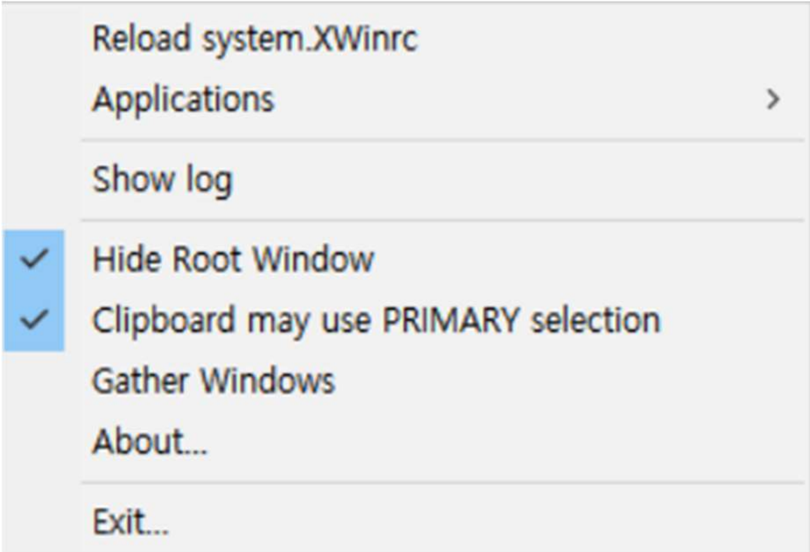
Tips: Display Forwarding

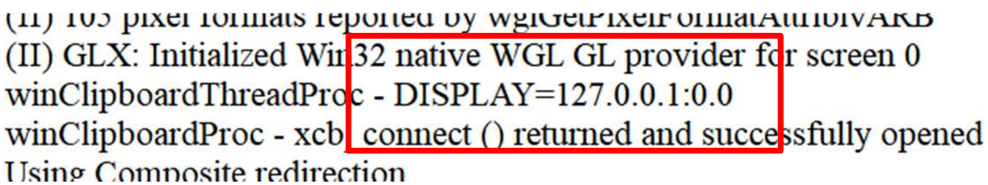
- If you are working remotely using SSH, you need to forward display.
- Here, the assumption is that your local machine is running Windows, and you are running the program from a remote Linux machine (Ubuntu 20.04).

- First, you should install an X server (e.g. Xming or VcXsrv)
- Download and install VcXsrv.
  – https://sourceforge.net/projects/vcxsrv/

Tips: Display Forwarding

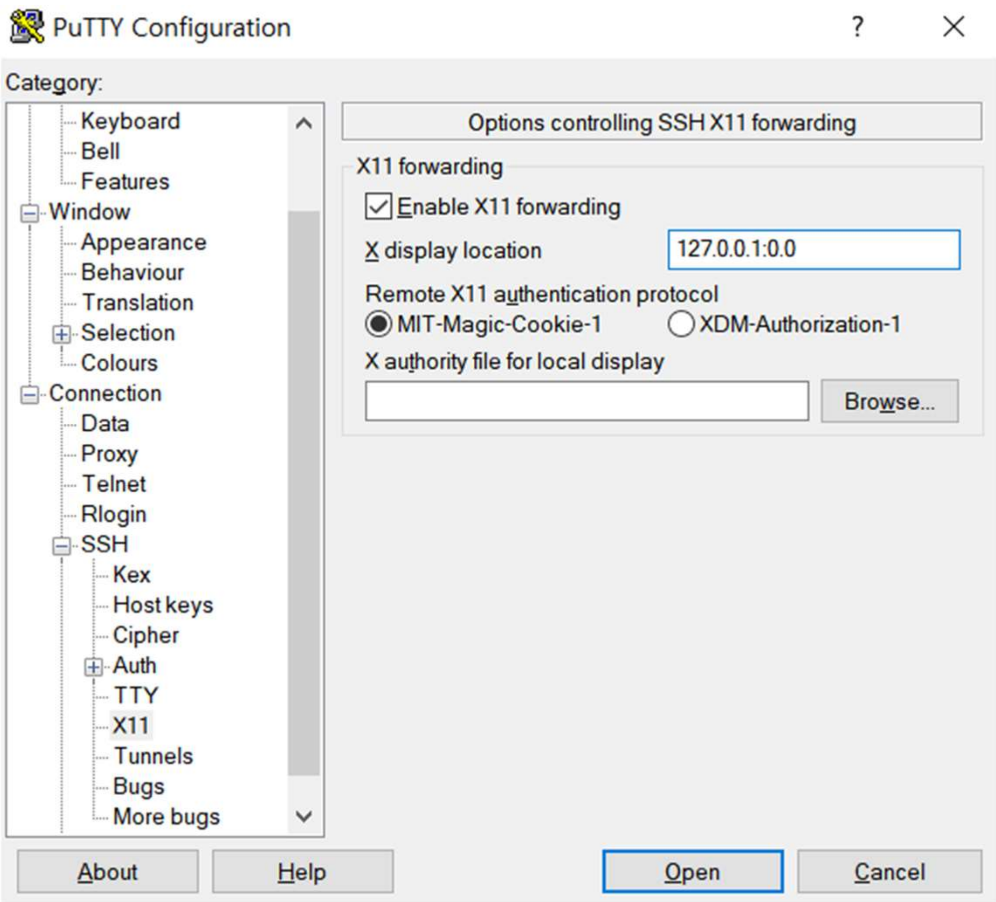
- Run the program, and you will see an icon in the tray.
- Right-click on the icon and select "Show log".
- Look for the line "winClipboardThreadProc - DISPLAY=127.0.0.1:0.0".





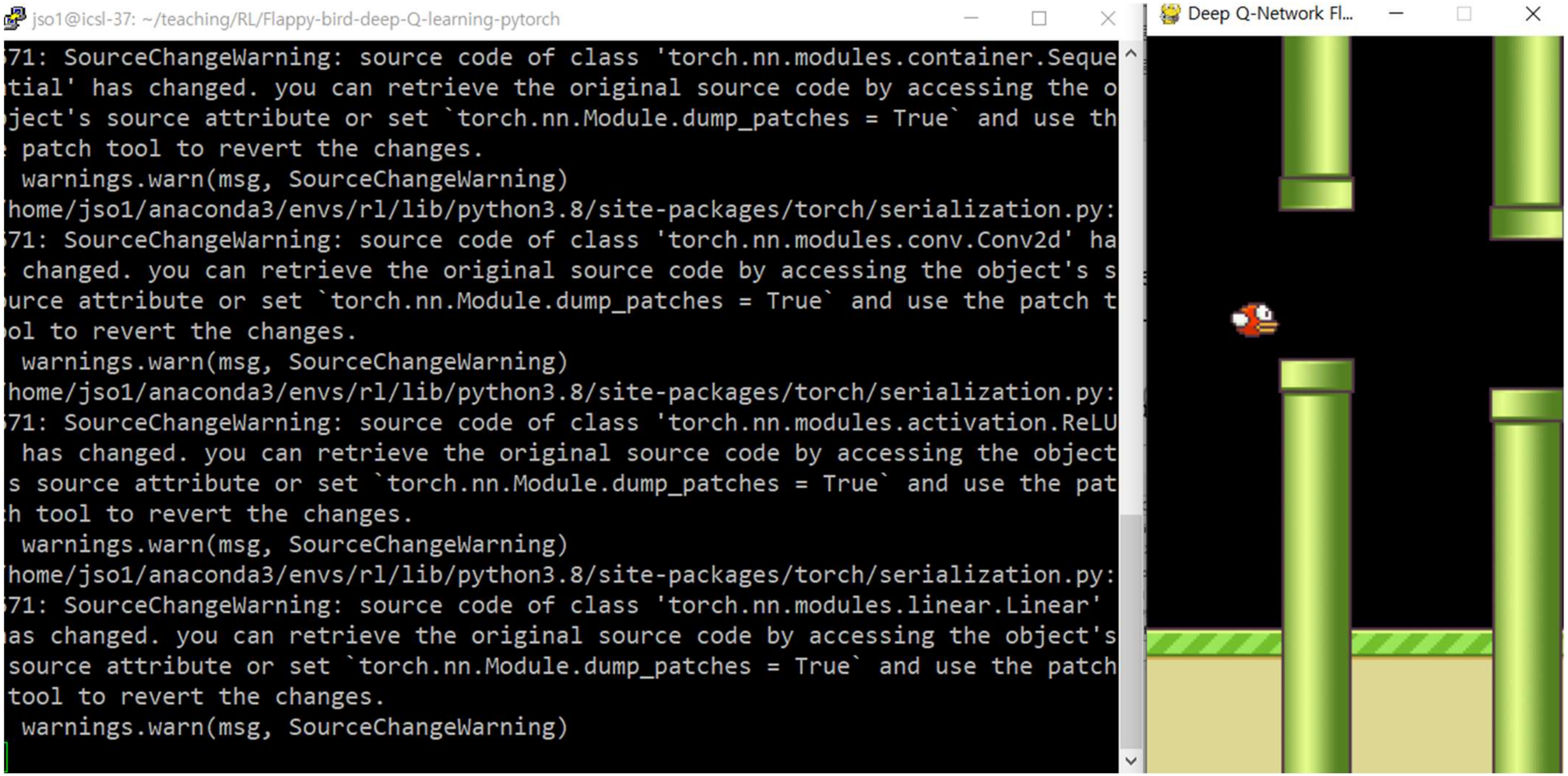
Tips: Display Forwarding

- Run putty to connect to your remote machine.
- In the menu, select Connection - SSH - X11.
- Check "Enable X11 Forwarding"
- In "X display location", write the address you saw in the VcXsrv log.



Tips: Display Forwarding

- Now you will be able to see the display.



Final Remarks

- In the course we have learned the fundamentals of reinforcement learning and a few reinforcement learning algorithms
  - The Markov Decision Process
  - Bellman Equation and Dynamic Programming
  - Monte Carlo Methods
  - Temporal Difference Learning
  - Deep Q Networks
  - Policy Gradient Algorithms
  - Actor-Critic Methods
  - Deep Deterministic Policy Gradient
  - Twin Delayed DDPG
  - Soft Actor-Critic
  - Proximal Policy Optimization

Final Remarks

- There are much more space to explore in the world of reinforcement learning
  - Multi-Agent reinforcement learning
  - Distributional reinforcement learning
  - Inverse reinforcement learning
  - Meta reinforcement learning
  - Hierarchical reinforcement learning
  - Interpretable reinforcement learning
  - Curiosity-driven exploration
  - Imagination-augmented agents
  - and still going!

End of Class

Questions?

Email: jso1@sogang.ac.kr