Intermediate Machine Learning: Assignment 4

Deadline

Assignment 4 is due Monday, November 18 by 11:59pm. Late work will not be accepted as per the course policies (see the Syllabus and Course policies on Canvas).

Directly sharing answers is not okay, but discussing problems with the course staff or with other students is encouraged.

You should start early so that you have time to get help if you're stuck. The drop-in office hours schedule can be found on Canvas. You can also post questions or start discussions on Ed Discussion. The assignment may look long at first glance, but the problems are broken up into steps that should help you to make steady progress.

Submission

Submit your assignment as a pdf file on Gradescope, and as a notebook (.ipynb) on Canvas. You can access Gradescope through Canvas on the left-side of the class home page. The problems in each homework assignment are numbered. Note: When submitting on Gradescope, please select the correct pages of your pdf that correspond to each problem. This will allow graders to more easily find your complete solution to each problem.

To produce the .pdf, please do the following in order to preserve the cell structure of the notebook:

Go to "File" at the top-left of your Jupyter Notebook Under "Download as", select "HTML (.html)" After the .html has downloaded, open it and then select "File" and "Print" (note you will not actually be printing) From the print window, select the option to save as a .pdf

Topics

- Graph kernels
- Reinforcement learning

This assignment will also help to solidify your Python skills.

Problem 1: Graph kernels (20 points)

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as img
import sklearn
import random
from numpy.linalg import inv
%matplotlib inline
```

In [22]:
```
# Helper functions for third part of exercise

def rgb2gray(rgb):
    """Function to turn RGB images in greyscale images."""
    return np.dot(rgb[..., :3], [0.2989, 0.5870, 0.1140])


def grid_adj(rows, cols):
    """Function that creates the adjacency matrix of
    a grid graph with predefined amount of rows and columns."""
    M = np.zeros([rows*cols, rows*cols])
    for r in np.arange(rows):
        for c in np.arange(cols):
            i = r*cols + c
            if c > 0:
                M[i-1, i] = M[i, i-1] = 1
            if r > 0:
                M[i-cols, i] = M[i, i-cols] = 1
    return M
```

The graph Laplacian for a weighted graph on n nodes is defined as

$$L = D - W$$

where W is an $n \times n$ symmetric matrix of positive edge weights, with $W_{ij} = 0$ if (i, j) is not an edge in the graph, and D is the diagonal matrix with $D_{ii} = \sum_{j=1}^{n} W_{ij}$. This generalizes the definition of the Laplacian used in class, where all of the edge weights are one.

1. Show that L is a Mercer kernel, by showing that L is symmetric and positive-semidefinite.

2. In graph neural networks we define polynomial filters of the form

$$P = a_0 I + a_1 L + a_2 L^2 + \cdots a_d L^d$$

where L is the Laplacian and $a_0, \dots, a_d$ are parameters, corresponding to the filter parameters in standard convolutional neural networks.

If each $a_i \geq 0$ is non-negative, show that P is also a Mercer kernel.

3. This polynomial filter has many applications. A handful of these applications are based on the fact that, given a graph with a signal x, the value of $x^T Lx$ will be low in case the signal is smooth (i.e. smooth transitions of x between neighboring nodes). A large $x^T Lx$ means that we have a rough graph signal (i.e. a lot of jumps in x between neighboring nodes).

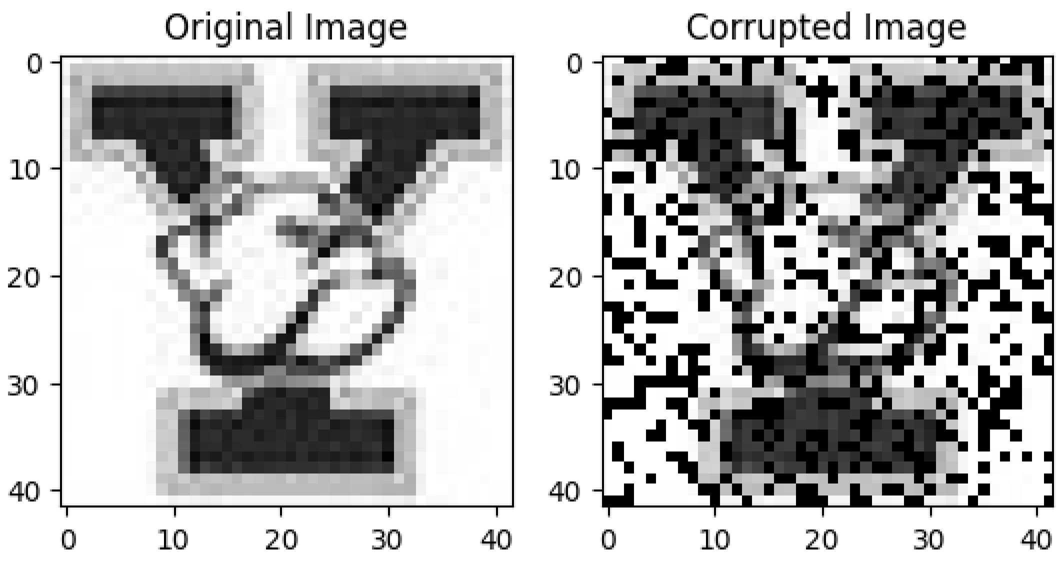
   An intersting application that uses this property is the so-called image inpainting process, where an image is seen as grid graph. Image inpainting tries to restore a corrupted image by smoothing out the neighboring pixel values. In this problem we corrupt an image by turning off (i.e. making the pixel value equal to zero) a certain portion of the pixels. Your goal will be to restore the corrupted image and hence recreate the original image.

   First, let's corrupt an image by turning off a portion of the pixels. For this exercise, we choose to turn off 30% of the pixels. The result is shown below. Try to understand the code, as some variables might be interesting for your work.

   The image "Yale_Bulldogs.jpg" can be found in Canvas under assn4 folder, and also in the GitHub repo https://github.com/YData123/sds365-fa24/tree/main/assignments/assn4

In [3]:
```
# Normalize the pixels of the original image
image = img.imread("Yale_Bulldogs.jpg") / 255
# Turn picture into greyscale
gray_image = rgb2gray(image)
height_img = gray_image.shape[0]
width_img = gray_image.shape[1]

# Turn off (value 0) certain pixels
fraction_off = int(0.30*height_img*width_img)
mask = np.ones(height_img*width_img, dtype=int)
```

```
# Set the first fraction of pixels off
mask[:fraction_off] = 0
# Shuffle to create randomness
np.random.shuffle(mask)
# Multiply the original image by the reshapes mask
mask = np.reshape(mask, (height_img, width_img))
corrupted_image = np.multiply(mask, gray_image)

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(gray_image, cmap=plt.get_cmap('gray'))
ax1.set_title("Original Image")
ax2.imshow(corrupted_image, cmap=plt.get_cmap('gray'))
ax2.set_title("Corrupted Image")

plt.show()
```



Inpainting missing pixel values can be formulated as the following optimization problem:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left\{ \|\mathbf{y}-\mathbf{Mx}\|_2^2 + \alpha\mathbf{x}^T\mathbf{Px} \right\}$$

where $\mathbf{y} \in \mathbb{R}^n$ (n being the total amount of pixels) is the corrupted graph signal (with missing pixel values being 0) and α is a regularization (smoothing) parameter that controls for smoothness of the graph. $\mathbf{P}$ is the polynomial filter based on the laplacian $\mathbf{L}$. Finally, $\mathbf{M} \in \mathbb{R}^{n\times n}$ is a diagonal matrix that satisfies:

$$\mathbf{M}(i,i) = \begin{cases} 1, \text{ if } \mathbf{y}(\text{i}) \text{ is observed} \\ \\ 0, \text{ if } \mathbf{y}(\text{i}) \text{ is corrupted} \end{cases}$$

The optimization problem tries to find an $\mathbf{x}$ that matches the observed values in $\mathbf{y}$, and at the same time tries to be smooth on the graph. Start with deriving a closed form solution of this optimization problem:

Answer to Problem 1 (1. 2. 3.)

Problem 1 (1.)

Proof: We wish to that L is symmetric and positive semidefinite, and thus a Mercer Kernel.

Symmetry:

Since W is a symmetric matrix (by definition, $W_{ij} = W_{ji}$), the diagonal matrix D is also symmetric because $D_{ii} = \sum_{j=1}^{n} W_{ij}$ only involves sums of symmetric terms. Therefore, $L = D - W$ is symmetric as the sum or difference of symmetric matrices is symmetric.

Positive Semidefinite:

Consider a vector $x \in \mathbb{R}^n$. The quadratic form associated with L is:

$$x^\top L x = x^\top (D - W) x = x^\top D x - x^\top W x.$$

Expanding each term:

$$x^\top D x = \sum_{i=1}^{n} D_{ii} x_i^2 = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} W_{ij} \right) x_i^2,$$
$$x^\top W x = \sum_{i=1}^{n} \sum_{j=1}^{n} W_{ij} x_i x_j.$$

Combining these, we have:

$$x^\top Lx = \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i^2 - x_i x_j).$$

From **Lemma 1.** we can the equivalent expression:

$$x^\top Lx = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i - x_j)^2.$$

Since $W_{ij} \geq 0$ and $(x_i - x_j)^2 \geq 0$, it follows that $x^\top Lx \geq 0$ for all $x \in \mathbb{R}^n$. Therefore, L is positive semidefinite. $\square$

Lemma 1.

To go from

$$x^\top Lx = \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i^2 - x_i x_j)$$

to

$$x^\top Lx = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i - x_j)^2,$$

we proceed as follows:

Expand $(x_i - x_j)^2 = x_i^2 - 2x_i x_j + x_j^2$ and substitute:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i - x_j)^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i^2 - 2x_i x_j + x_j^2).$$

Group terms:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_i^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_i x_j + \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_j^2.$$

Use symmetry of W ($W_{ij} = W_{ji}$) to combine terms:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_i^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_j^2 = 2\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}x_i^2.$$

Divide by 2 to account for redundancy:

$$x^\top Lx = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(x_i - x_j)^2.$$

This completes the derivation. □

Problem 1. (2.)

Proof We wish to show that the polynomial filter P is symmetric and positive semidefinite, and thus a Mercer Kernel. P is defined as:

$$P = a_0 I + a_1 L + a_2 L^2 + \cdots + a_d L^d,$$

where $a_0, \ldots, a_d \geq 0$ and L is the Laplacian matrix.

Symmetry

Since L is symmetric, all powers of L, i.e., $L^2, L^3, \ldots, L^d$, are also symmetric. I (the identity matrix) is symmetric. Thus, P, is symmetric as a linear combination of symmetric matrices is symmetric.

Positive Semidefinite

For any $x \in \mathbb{R}^n$, consider the quadratic form associated with P:

$$x^\top Px = a_0 x^\top Ix + a_1 x^\top Lx + a_2 x^\top L^2 x + \cdots + a_d x^\top L^d x.$$

Notice that each term is non-negative:

1. $x^\top Ix = \|x\|^2 \geq 0$.
2. $x^\top Lx \geq 0$ since L is positive semidefinite.
3. For $k \geq 2$, $x^\top L^k x \geq 0$ because L^k is the product of positive semidefinite matrices, which remains positive semidefinite.

Since $a_0, \ldots, a_d \geq 0$, the sum $x^\top P x \geq 0$. Thus, P is positive semidefinite. $\square$

Problem 1 (3.)

We wish to optimize:

$$L(x) = \min_{\mathbf{x} \in \mathbb{R}^n} \left\{ \|\mathbf{y} - \mathbf{M}\mathbf{x}\|_2^2 + \alpha \mathbf{x}^T \mathbf{P} \mathbf{x} \right\}$$

Let us first expand the expression:

$$\begin{aligned} L(x) &= (y - Mx)^T (y - Mx) + \alpha x^T P x && (1) \\ &= (y^T - x^T M^T)(y - Mx) + \alpha x^T P x && (2) \\ &= y^T y - y^T M x - x^T M^T y + x^T M^T M X + \alpha x^T P x && (3) \end{aligned}$$

Take the **FOC** w.r.t x to arrive at $\hat{x}$ and noticing that $M^T = M$ and $M^T M = M$:

$$L'(x) = 2M^T M x - 2M^T y + 2\alpha P x = 0$$

$$M^T M x + \alpha P x = M^T y$$

$$\hat{x} = (M^T M + \alpha P)^{-1} M^T y = (M + \alpha P)^{-1} M y \square$$

Next, let's restore our image. To keep things simple, let's say we already trained the polynomial filter $\mathbf{P}$ of degree 2 and we found the following weights:

$$\mathbf{P} = \mathbf{L} + 0.05\, \mathbf{L}^2$$

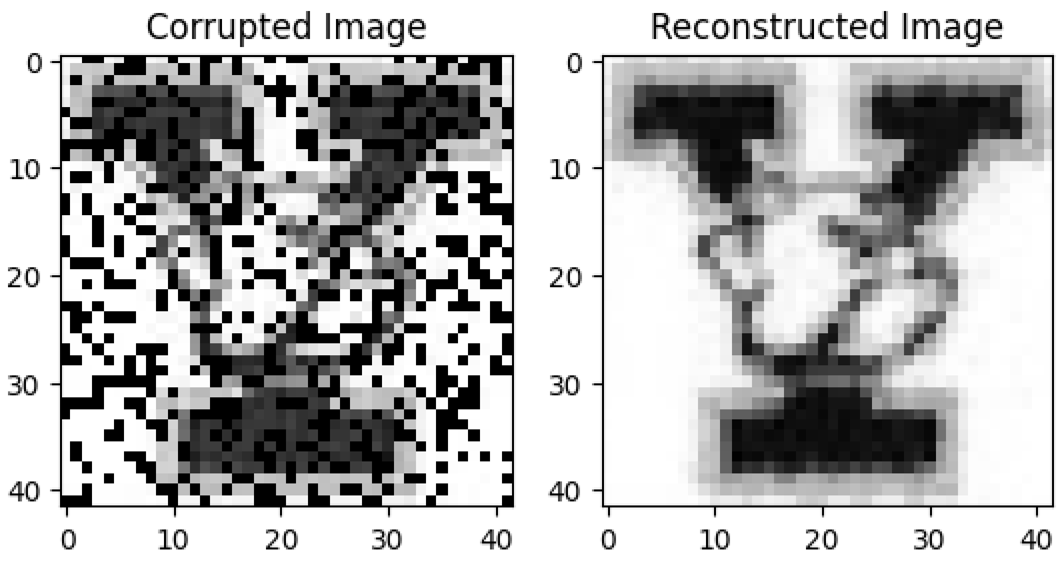
Fill in the following lines of code and show your reconstructed images next to the corrupted image. Assume that the weights on the graph edges are equal to 1.

In [50]:
```
# Corrupted graph signal
y = corrupted_image.flatten()

# Diagonal matrix defined as above
M = np.diag(mask.flatten())

# Adjacency matrix of the graph (by using the helper function)
```

```python
A = grid_adj(height_img, width_img)

# Diagonal matrix defined as above
D = np.diag(A.sum(axis=1))

# Graph Laplacian defined as above
L = D - A

# Polynomial filter defined as above
P = L + 0.05 * (L @ L)
```

In [51]:
```python
# Imported this function for Problem 1 (3.)
from scipy.sparse.linalg import cg

# Try to experiment with different alpha values
alpha = 0.1

# closed form solution you derived above
I = np.eye(L.shape[0])
objective_matrix = M + alpha * P
b = M @ y
x, _ = cg(objective_matrix, b)
```

In [52]:
```python
reconstructed_image = np.reshape(x, (height_img, width_img))

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(corrupted_image, cmap=plt.get_cmap('gray'))
ax1.set_title("Corrupted Image")
ax2.imshow(reconstructed_image, cmap=plt.get_cmap('gray'))
ax2.set_title("Reconstructed Image")

plt.show()
```



4. Discuss the influence of the smoothing parameter α in the optimization problem above. What happens for very large and very low values of α? Finally, discuss the degree of our polynomial function $\mathbf{P}$. What happens if we would choose a large degree?

Answer to Problem 1 (4.)

When $\alpha = 0$, the Reconstructed Image R is the exact same as the Corrupted Image. For small α, R is clearly reconstructed with distinct contrast in pixels. As $\alpha \to \infty$, R gets smoother due to the enhanced smoothing properties of P.

P is a degree 2 polynomial. If we increase the degree k, R will exhibit smoother traits. As increasing k, we increase the "kernel size", allow for long-range dependencies between nodes to have more effects - leading to more smoothing.

Problem 2: Positive reinforcement (10 points)

As discussed in class, reinforcement learning using policy gradient methods is based on maximizing the expected total reward

$$J(\theta) = \mathbb{E}_\theta[R(\tau)],$$

where the expectation is over the probability distribution over sequences τ through a choice of actions using the policy. This can be rewritten as

$$\nabla_\theta J(\theta) = \mathbb{E}_\theta \left[R(\tau) \nabla_\theta \log p(\tau \mid \theta) \right].$$

Approximating this gradient involves computing $\nabla_\theta \log \pi_\theta(a \mid s)$ where π_θ is the policy.

2.1 Continuous action space with Gaussian policy

Suppose that the action space is continuous and $\pi_\theta(a \mid s)$ is a normal density with mean $\mu_\theta(s)$ and variance $\sigma_\theta^2(s)$, two outputs of a neural network with input s and parameters θ.

Suppose the outputs of the neural network are given by

$$\begin{aligned} \mu_\theta(s) &= \beta_1^T h(s) \\ \sigma_\theta^2(s) &= \exp(\beta_2^T h(s)) \end{aligned}$$

where $h(s)$ is the vector of neurons in the last layer, immediately before the outputs. Derive explicit expressions for $\nabla_{\beta_1} \log \pi_\theta(a \mid s)$ and $\nabla_{\beta_2} \log \pi_\theta(a \mid s)$.

Answer 2.1

Express $\pi_\theta(a \mid s)$ out:

$$\begin{aligned} \pi_\theta(a \mid s) &= \frac{1}{2\pi\sigma_\theta^2(s)} \exp\left(\frac{1}{2\sigma_\theta^2(s)} (a - \mu_\theta(s))^2 \right) \qquad (4) \\ &= \frac{1}{\left(2\pi \exp(\beta_2^T h(s))\right)^{1/2}} \exp\left(\frac{1}{2\exp(\beta_2^T h(s))} \left(a - \beta_1^T h(s)\right)^2 \right) \qquad (5) \end{aligned}$$

Express $\log(\pi_\theta(a \mid s))$ out:

$$\log(\pi_\theta(a \mid s)) = -\frac{1}{2}\log(2\pi) - \frac{1}{2}\beta_2^T h(s) + \frac{1}{2\exp(\beta_2^T h(s))} \left(a - \beta_1^T h(s)\right)^2 \qquad (6)$$

Solve for $\nabla_{\beta_1} \log \pi_\theta(a \mid s)$:

$$\nabla_{\beta_1} \log \pi_\theta(a \mid s) = -\frac{1}{2\exp(\beta_2^T h(s))} \times -2h(s)\left(a - \beta_1^T h(s)\right) \tag{7}$$

$$= \frac{1}{\exp(\beta_2^T h(s))}\left(a - \beta_1^T h(s)\right) h(s) \tag{8}$$

Solve for $\nabla_{\beta_2} \log \pi_\theta(a \mid s)$:

$$\nabla_{\beta_2} \log \pi_\theta(a \mid s) = -\frac{1}{2} h(s) - \frac{1}{2\exp(\beta_2^T h(s))}\left(a - \beta_1^T h(s)\right)^2 h(s) \tag{9}$$

Using the log-likelihood trick, the gradient is depedent on β_1 and β_2:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a \mid s) \cdot R\right]$$

The gradients are used to update the parameters as follows:

$$\beta_1 \leftarrow \beta_1 + \eta \cdot \mathbb{E}_{\pi_\theta}\left[\nabla_{\beta_1} \log \pi_\theta(a \mid s) \cdot R\right],$$

$$\beta_2 \leftarrow \beta_2 + \eta \cdot \mathbb{E}_{\pi_\theta}\left[\nabla_{\beta_2} \log \pi_\theta(a \mid s) \cdot R\right],$$

where η is the learning rate.

2.2 Discrete action space with Softmax policy

Suppose the action space is discrete with K possible actions, and the policy $\pi_\theta(a \mid s)$ is defined using a softmax function over preferences $u_\theta(s, a)$:

$$\pi_\theta(a \mid s) = \frac{\exp(u_\theta(s, a))}{\sum_a \exp(u_\theta(s, a))},$$

where $u_\theta(s, a) = \beta^T h(s, a)$, and $h(s, a)$ is a feature vector for state-action pair (s, a). Derive the expression for $\nabla_\beta \log \pi_\theta(a \mid s)$.

Answer 2.2

Express $\beta \log \pi_\theta(a \mid s)$ out:

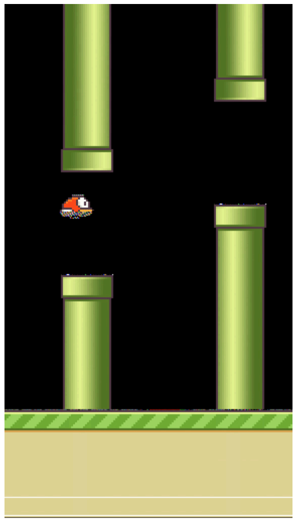
$$\begin{aligned}\log \pi_\theta(a \mid s) &= u_\theta(s,a) - \log \sum_a \exp(u_\theta(s,a)) && (10)\\ &= \beta^T h(s,a) - \log \sum_a \exp(\beta^T h(s,a)). && (11)\end{aligned}$$

Express $\nabla_\beta \log \pi_\theta(a \mid s)$ out:

$$\begin{aligned}\nabla_\beta \log \pi_\theta(a \mid s) &= \nabla_\beta \left(\beta^T h(s,a)\right) - \nabla_\beta \log \sum_a \exp(\beta^T h(s,a)) && (12)\\ &= h(s,a) - \frac{\nabla_\beta \sum_a \exp(\beta^T h(s,a))}{\sum_a \exp(\beta^T h(s,a))} && (13)\\ &= h(s,a) - \frac{\sum_a \nabla_\beta \exp(\beta^T h(s,a))}{\sum_a \exp(\beta^T h(s,a))} && (14)\\ &= h(s,a) - \frac{\sum_a \exp(\beta^T h(s,a)) h(s,a)}{\sum_a \exp(\beta^T h(s,a))} && (15)\\ &= h(s,a) - \sum_a \pi_\theta(a \mid s) h(s,a). && (16)\end{aligned}$$

Problem 3: Deep Q-Learning for Flappy Bird (25 points)

Deep Q-learning was proposed (and patented) by DeepMind and made a big splash when the same deep neural network architecture was shown to be able to surpass human performance on many different Atari games, playing directly from the pixels. In this problem, we will walk you through the implementation of deep Q-learning to learn to play the Flappy Bird game.



The implementation is based these references:

- DeepLearningFlappyBird
- Deep Q-Learning for Atari Breakout

We use the `pygame` package to visualize the interaction between the algorithm and the game environment. However, *pygame* is not well supported by Google Colab; we recommend you to run the code for this problem locally. A window will be popped up that displays the game as it progress in real-time (as for the Cartpole demo from class).

This problem is structured as follows:

- Load necessary packages
- Test the visualization of the game, to make sure everything's working
- Process the images to reduce the dimension
- Setup the game history buffer
- Implement the core Q-learning function
- Run the learning algorithm
- Interpret the results

Introduction

The Flappy Bird game is requires a few Python packages. Please install these *as soon as possible*, and notify us of any issues you experience so that we can help. The Python files can also be found on Canvas and in our GitHub repo at https://github.com/YData123/sds365-fa24/tree/main/assignments/assn4 .

In [1]:
```
# %pip install pygame
# %pip install opencv-python
import numpy as np
import cv2
import wrapped_flappy_bird as flappy_bird
from collections import deque
import random
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, initializers
```

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

The Flappy Bird environment

Interaction with the game environment is carried out through calls of the form

`(image, reward, terminal) = game.frame_step(action)`

where the meaning of these variables is as follows:

- `action` : $\binom{1}{0}$ for doing nothing, $\binom{0}{1}$ for "flapping the bird's wings"
- `image` : the image for the next step of the game, of size $(288, 512, 3)$ with three RGB channels
- `reward` : the reward received for taking the action; -1 if an obstacle is hit, 0.1 otherwise.
- `terminal` : `True` if an obstacle is hit, otherwise `False`

Now let's take a look at the game interface. First, initiate the game:

In [2]:
```
num_actions = 2

# initiate a game
game = flappy_bird.GameState()
```

```
# get the first state by doing nothing
do_nothing = np.zeros(num_actions)
do_nothing[0] = 1
image, reward, terminal = game.frame_step(do_nothing)

print('shape of image:', image.shape)
print('reward: ', reward)
print('terminal: ', terminal)
```

```
shape of image: (288, 512, 3)
reward:  0.1
terminal:  False
```

After running the above cells, a window should pop up, and you can watch the game being played in that window.

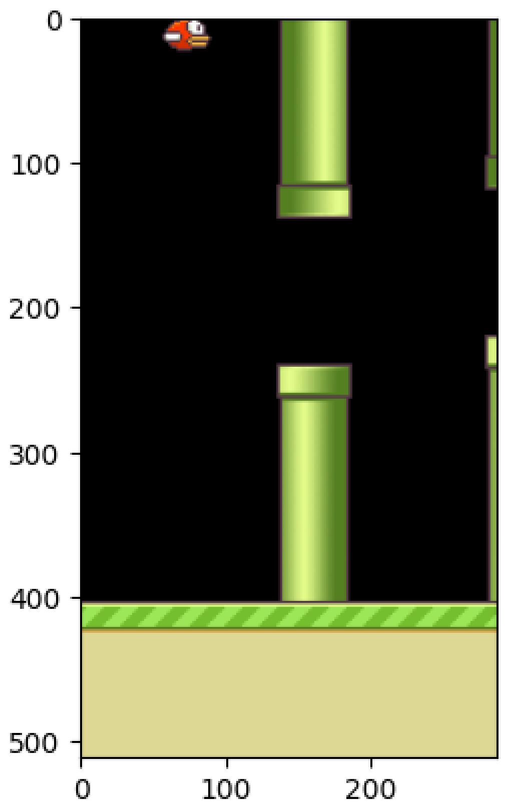
Let's take some random actions and see what happens:

In [3]:
```
for i in range(587):

    # choose a random action
    action = np.random.choice(num_actions)

    # create the corresponding one-hot vector
    action_vec = np.zeros(num_actions)
    action_vec[action] = 1

    # take the action and observe the reward and the next state
    image, reward, terminal = game.frame_step(action_vec)
```

Are you able to see Flappy moving across the window and crashing into things? Great! If you're having any issues, post to EdD and we'll do our best to help you out.

Here is how we can visualize a frame of the game as an image within a cell.

In [4]:
```
# show the image
import matplotlib.pyplot as plt
plt.imshow(image.transpose([1, 0, 2]))
plt.show()
plt.close()
```



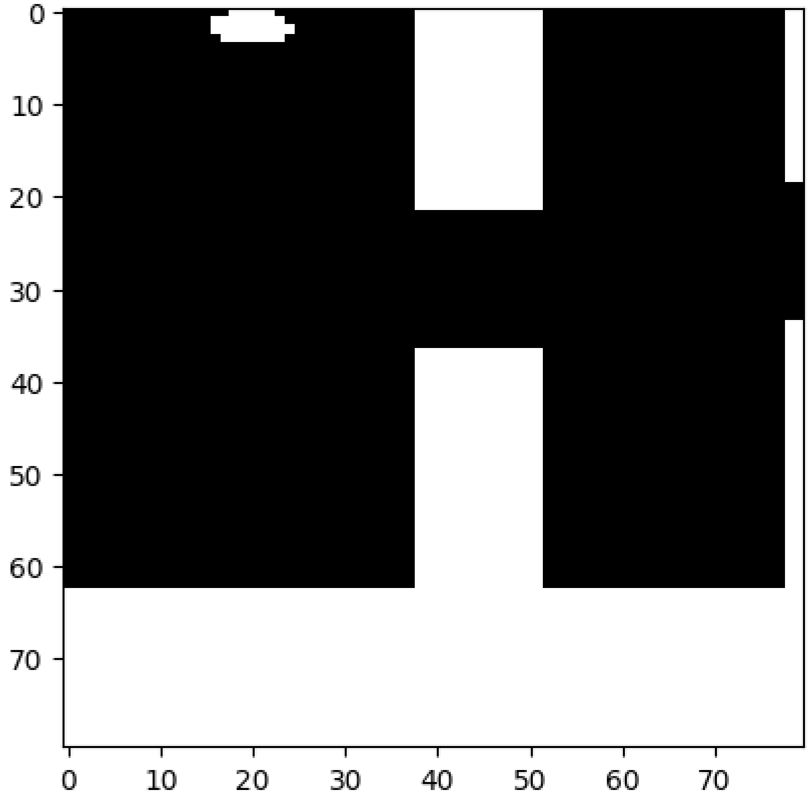
Preprocessing the images

Alright, next we need to prepocess the images by converting them to grayscale and resizing them to 80×80 pixels. This will help to reduce the computation, and aid learning. Besides, Flappy is "color blind." (Fun fact: The instructor of this course is also color vision deficient.)

In [6]:
```
def resize_gray(frame):
    frame = cv2.cvtColor(cv2.resize(frame, (80, 80)), cv2.COLOR_BGR2GRAY)
    ret, frame = cv2.threshold(frame, 1, 255, cv2.THRESH_BINARY)
    return np.reshape(frame, (80, 80, 1))

image_transformed = resize_gray(image)
print('Shape of the transformed image:', image.shape)
```

```
# show the transformed image
_ = plt.imshow(image_transformed.transpose((1, 0, 2)), cmap='gray')
```

```
Shape of the transformed image: (288, 512, 3)
```



This shows the preprocessed image for a single frame of the game. In our implementation of Deep Q-Learning, we encode the state by stacking four consecutive frames, resulting in a tensor of shape (80,80,4).

Then, given the `current_state`, and a raw image `image_raw` of size $288 \times 512 \times 3$, we convert the raw image to a $80 \times 80 \times 1$ grayscale image using the code in the previous cell. The , we remove the first frame of `current_state` and add the new frame, giving again a stack of images of size (80, 80, 4).

In [7]:
```
def preprocess(image_raw, current_state=None):
    # resize and convert to grayscale
    image = resize_gray(image_raw)
    # stack the frames
    if current_state is None:
        state = np.concatenate((image, image, image, image), axis=2)
    else:
        state = np.concatenate((image, current_state[:, :, :3]), axis=2)
    return state
```

3.1 Explain the game state

Why is the state chosen to be a stack of four consecutive frames rather than a single frame? Give an intuitive explanation.

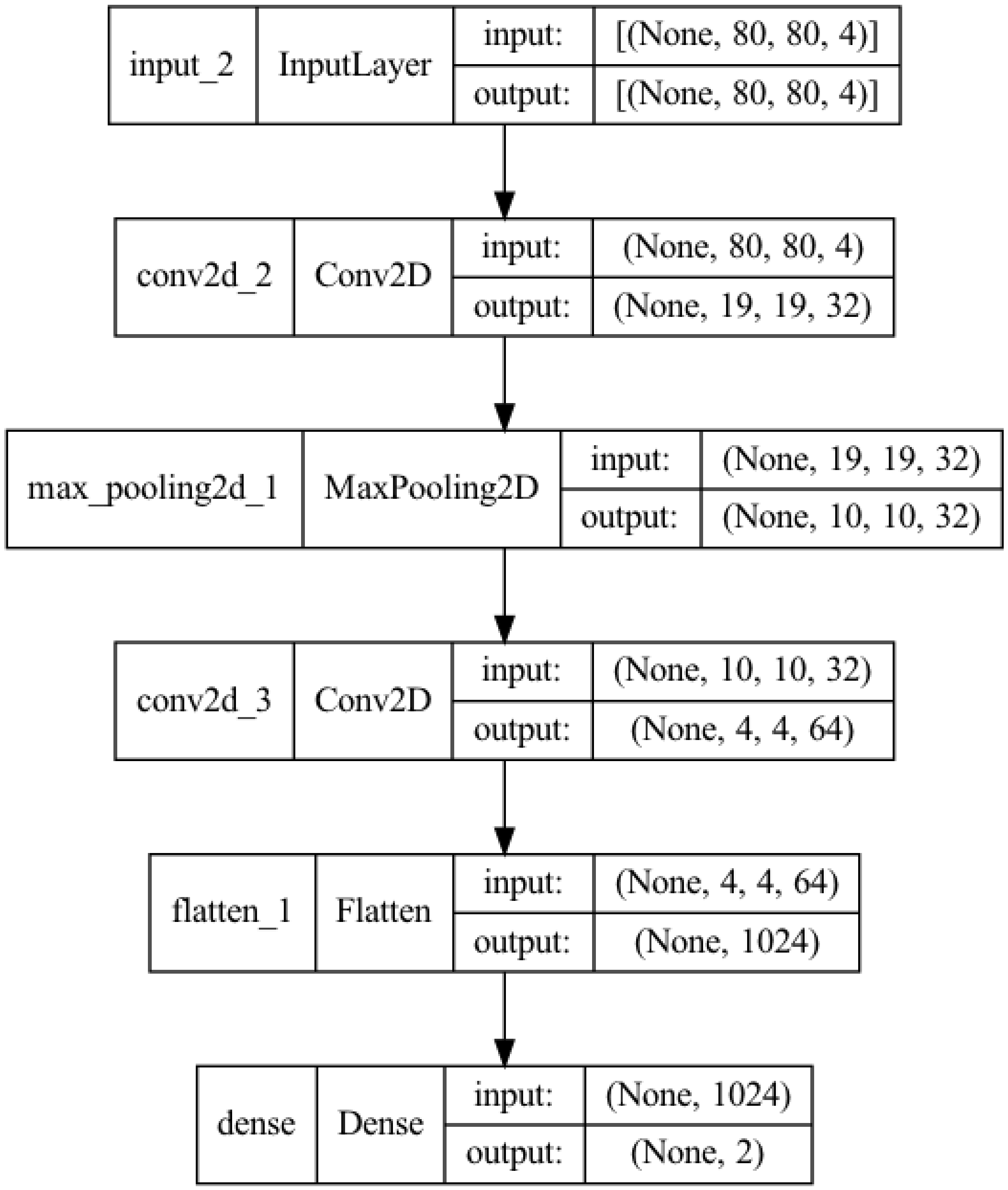
Answer 3.1

Flappy Bird is inherently a temporal game, where the bird's position at timestep t influences what happens at $t + 1$. Feeding the model only a single frame at t prevents it from leveraging this critical temporal relationship. For instance, if the bird is flying at t and an obstacle appears at $t + 1$, the model would lack the context needed to predict and act appropriately. Without the prior frames, the model cannot recognize trends like the bird's trajectory or the impending obstacle, leading to failures in decision-making.

Constructing the neural network

Now we are ready to construct the neural network for approximating the Q function. Recall that, given input s which is of size $80 \times 80 \times 4$ due to the previous preprocessing, the output of the network should be of size 2, corresponding to the values of $Q(s, a_1)$ and $Q(s, a_2)$ respectively.

Here is the summary of the model we'd like to build:



3.2 Initialize the network

Complete the code in the next cell so that your model architecture matches that in the above picture. Here we specify the initialization of the weights by using `keras.initializers`. Note that we haven't talked about the `strides` argument for CNNs; you can read about stride here: https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/. It's not important to understand this in detail, you just need to choose the number and sizes of the filters to get the shapes to match the specification.

In [8]:
```
from tensorflow.keras import initializers
def create_q_model():
    state = layers.Input(shape=(80, 80, 4,))

    layer1 = layers.Conv2D(filters=32, kernel_size=8, strides=4, activation="relu",
                           kernel_initializer=initializers.TruncatedNormal(mean=0., stddev=0.01),
                           bias_initializer=initializers.Constant(0.01))(state)
    layer2 = layers.MaxPool2D(2, strides=2, padding="SAME")(layer1)
    layer3 = layers.Conv2D(filters=64, kernel_size=4, strides=2, activation="relu",
                           kernel_initializer=initializers.TruncatedNormal(mean=0., stddev=0.01),
                           bias_initializer=initializers.Constant(0.01))(layer2)
    layer4 = layers.Flatten()(layer3)
    q_value = layers.Dense(units=2, activation="linear",
                           kernel_initializer=initializers.TruncatedNormal(mean=0., stddev=0.01),
                           bias_initializer=initializers.Constant(0.01))(layer4)

    return keras.Model(inputs=state, outputs=q_value)
```

Plot the model summary to make sure that the network is the same as expected.

In [10]:
```
model = create_q_model()
print(model.summary())
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 80, 80, 4)]       0         
                                                                 
 conv2d_2 (Conv2D)           (None, 19, 19, 32)        8224      
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 10, 10, 32)       0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (None, 4, 4, 64)          32832     
                                                                 
 flatten_1 (Flatten)         (None, 1024)              0         
                                                                 
 dense_1 (Dense)             (None, 2)                 2050      
                                                                 
=================================================================
Total params: 43,106
Trainable params: 43,106
Non-trainable params: 0
_________________________________________________________________
None
```

Deep Q-learning

We're now ready to implement the Q-learning algorithm. There are some subtle details in the implementation that you need to sort out. First, recall that the update rule for Q learning is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r(s,a) + \gamma \cdot \max_{a'} Q(\text{next}(s,a), a') - Q(s,a))$$

where γ is the discount factor and α can be viewed as the step size or learning rate for gradient ascent.

We'll set these as follows:

In [11]:
```
gamma = 0.99            # decay rate of past observations
step_size = 1e-4        # step size
```

Estimation with experience replay

At the beginning of training, we spend 10,000 steps taking random actions, as a means of observing the environment.

We build a replay memory of length 10,000 steps, and every time we update the weights of the network, we sample a batch of size 32 and perform a Q-learning update on this batch.

After we have collected 10,000 steps of new data, we discard the old data, and replace it with the new "experiences."

In [14]:
```
observe = 10000             # timesteps to observe before training
replay_memory = 10000       # number of previous transitions to remember
batch_size = 32             # size of each batch
```

3.3 Justify the data collection

Why does it make sense to maintain the replay memory of a fixed size instead of including all of the historical data?

Answer 3.3

From a practical standpoint, limiting the replay memory prevents excessive memory usage and computational burden that would arise from storing and processing all historical data. In other words, a fixed memory size ensures that the training process remains scalable and computationally feasible.

From a learning standpoint, older data may no longer represent the current policy or the agent's understanding of the environment. Including excessively outdated data could introduce bias and reduce the relevance of sampled experiences, thereby hindering the learning process. This mimics humans, where we often weigh recent intertactions over past events (except for trauma) when learning and growing.

Exploration vs exploitation

When performing Q-learning, we face the tradeoff between exploration and exploitation. To encourage exploration, a simple strategy is to take a random action at each step with certain probability.

More precisely, for each time step t and state s_t, with probability ϵ, the algorithm takes a random action (wing flap or do nothing), and with probability $1 - \epsilon$ the algorithm takes a greedy action according to $a_t = \arg\max_a Q_\theta(s_t, a)$. Here θ refers to the parameters of our CNN.

In [15]:
```
# value of epsilon
epsilon = 0.05
```

3.4 Complete the Q-learning algorithm

Next you will need to complete the Q-learning algorithm by filling in the missing code in the following function. The missing parts include

- Taking a greedy action
- Given a batch of samples $\{(s_t, a_t, r_t, s_{t+1}, \text{terminal}_t)\}_{t\in B}$, computing the corresponding $Q_\theta(s_t, a_t)$.
- Given a batch of samples $\{(s_t, a_t, r_t, s_{t+1}, \text{terminal}_t)\}_{t\in B}$, computing the corresponding updated Q-values

$$\hat{y}(s_t, a_t) = \begin{cases} r_t + \gamma \max_a Q_\theta(s_{t+1}, a), & \text{if terminal}_t = 0, \\ r_t, & \text{otherwise.} \end{cases}$$

Then, the mean squared error loss for the batch is

$$\frac{1}{|B|}\sum_{t\in B}(\hat{y}(s_t, a_t) - Q_\theta(s_t, a_t))^2.$$

In [34]:
```
def dql_flappy_bird(model, optimizer, loss_function):

    # initiate a game
    game = flappy_bird.GameState()

    # store the previous state, action and transitions
    history_data = deque()

    # get the first observation by doing nothing and preprocess the image
    do_nothing = np.zeros(num_actions)
    do_nothing[0] = 1
    image, reward, terminal = game.frame_step(do_nothing)

    # preprocess to get the state
    current_state = preprocess(image_raw=image)

    # training
    t = 0

    while t < 50000:
        if epsilon > np.random.rand(1)[0]:
            # random action
            action = np.random.choice(num_actions)
        else:
            # compute the Q function
            current_state_tensor = tf.convert_to_tensor(current_state)
            current_state_tensor = tf.expand_dims(current_state_tensor, 0)
            q_value = model(current_state_tensor, training=False)

            # greedy action
            action = tf.argmax(q_value[0]).numpy()

        # take the action and observe the reward and the next state
        action_vec = np.zeros([num_actions])
        action_vec[action] = 1
        image_raw, reward, terminal = game.frame_step(action_vec)
        next_state = preprocess(current_state=current_state,
                                image_raw=image_raw)

        # store the observation
        history_data.append((current_state, action, reward, next_state,
                            terminal))
        if len(history_data) > replay_memory:
            history_data.popleft()  # discard old data

        # train if done observing
        if t > observe:

            # sample a batch
            batch = random.sample(history_data, batch_size)
            state_sample = np.array([d[0] for d in batch])
            action_sample = np.array([d[1] for d in batch])
            reward_sample = np.array([d[2] for d in batch])
            state_next_sample = np.array([d[3] for d in batch])
            terminal_sample = np.array([d[4] for d in batch])

            # compute the updated Q-values for the samples
            # Compute predicted Q-values in evaluation mode
            next_q_value = model(tf.convert_to_tensor(state_next_sample), training=False)
            # Choose max Q-values
            max_next_q_value = tf.reduce_max(next_q_value, axis=1)
            # Apply Bellman Equation where 1 - terminal_sample = 0 if terminal_sample
            updated_q_value = reward_sample + (1 - terminal_sample) * gamma * max_next_q_value

            # train the model on the states and updated Q-values
            with tf.GradientTape() as tape:
                # compute the current Q-values for the samples
                # Compute predicted Q-values in training model
                current_q_value = model(tf.convert_to_tensor(state_sample))
                # Get the Q-values of the current action
                indices = tf.stack([tf.range(batch_size, dtype=tf.int64), tf.convert_to_tensor(action_sample, dtype=t
                current_q_value = tf.gather_nd(current_q_value, indices)

                # compute the loss
                # (i.e., the current q value (LHS) should be as close as possible to the future q value (RHS))
                loss = loss_function(updated_q_value, current_q_value)

            # backpropagation
            grads = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
        else:
            loss = 0

        # update current state and counter
        current_state = next_state
        t += 1

        # print info every 500 steps
        if t % 500 == 0:
            print(f"STEP {t} | PHASE {'observe' if t<=observe else 'train'}",
                  f"| ACTION {action} | REWARD {reward} | LOSS {loss}")
```

You're now ready to play the game! Just run the cell below; do not change the code.

In [35]:
```
def playgame(start_from_ckpt=False, ckpt_path=None):

    #! DO NOT change the random seed !
    np.random.seed(4)

    if start_from_ckpt:
        # if you want to start from a checkpoint
        model = keras.models.load_model('ckpt_path')
    else:
        model = create_q_model()

    # specify the optimizer and loss function
    optimizer = keras.optimizers.Adam(learning_rate=step_size, clipnorm=1.0)
    loss_function = keras.losses.MeanSquaredError()

    # play the game
    dql_flappy_bird(model=model, optimizer=optimizer, loss_function=loss_function)
```

In [36]:
```
playgame()
```

```
STEP 500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 1000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 1500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 2000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 2500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 3000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 3500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 4000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 4500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 5000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 5500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 6000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 6500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 7000 | PHASE observe | ACTION 0 | REWARD 0.1 | LOSS 0
STEP 7500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 8000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 8500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 9000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 9500 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 10000 | PHASE observe | ACTION 1 | REWARD 0.1 | LOSS 0
STEP 10500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.28295496106147766
STEP 11000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.015218841843307018
STEP 11500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.009220397099852562
STEP 12000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.00467890128493309
STEP 12500 | PHASE train | ACTION 1 | REWARD -1 | LOSS 0.006331755314022303
STEP 13000 | PHASE train | ACTION 1 | REWARD -1 | LOSS 0.008049175143241882
STEP 13500 | PHASE train | ACTION 0 | REWARD -1 | LOSS 0.009237512946128845
STEP 14000 | PHASE train | ACTION 1 | REWARD -1 | LOSS 0.0023635243996977806
STEP 14500 | PHASE train | ACTION 1 | REWARD -1 | LOSS 0.017527198418974876
STEP 15000 | PHASE train | ACTION 0 | REWARD -1 | LOSS 0.002077145269140601
STEP 15500 | PHASE train | ACTION 0 | REWARD -1 | LOSS 0.0022251643240451813
STEP 16000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.044338420033454895
STEP 16500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0010258664842694998
STEP 17000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0011961793061345816
STEP 17500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 6.2455644607543945
STEP 18000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.01515561155974865
STEP 18500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0033726878464221954
STEP 19000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.001314115826971829
STEP 19500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0012079196749255061
STEP 20000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.010786657221615314
STEP 20500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0008329328848049045
STEP 21000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0037328407634049654
STEP 21500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.001217487035319209
STEP 22000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.038400426506996155
STEP 22500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.002085252432152629
STEP 23000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.007639157585799694
STEP 23500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.011697839014232159
STEP 24000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.006360439117997885
STEP 24500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.031184522435069084
STEP 25000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.00884314440190792
STEP 25500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.02862236276268959
STEP 26000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.011431178078055382
STEP 26500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.07754367589950562
STEP 27000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.030263269320130348
STEP 27500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.037599116563797
STEP 28000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.01077241636812687
STEP 28500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.038344718515872955
STEP 29000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.03754967451095581
STEP 29500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.023081833496689796
STEP 30000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.03964317589998245
STEP 30500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.07027740776538849
```

```
KeyboardInterrupt
```

3.5 Describe the training

Describe what you see by answering the following questions:

- In the early stage of training (within 2,000 steps in the *explore* phase), describe the behavior of the Flappy Bird. What do you think is the greedy policy given by the estimation of the Q-function in this stage?
- Describe what you see after roughly 5,000 training steps. Do you see any improvement? In particular, compare Flappy's behavior with their behavior in the early stages of training.
- Explain why the performance has improved, by relating to the model design such as the replay memory and the exploration.

Answer 3.5

1. At this stage, Flappy Bird exhibits a behavior of flapping continuously to fly to the top until it hits a barrier. Occasionally, the bird seems to pause its flapping briefly, allowing it to descend. This behavior aligns with the greedy policy, where the Q-function estimates that flapping prevents the bird from hitting the ground, which results in a small positive reward (e.g., 0.1) for

staying alive. Since the Q-function is still in the early stages of learning, the policy predominantly favors actions that avoid immediate termination (flapping) without yet optimizing for long-term survival.

2. By this point, the bird's behavior shows noticeable improvement. Instead of flapping indiscriminately to the top, the bird now demonstrates a more controlled pattern of flapping, pausing at times to descend. This change occurs because the agent has experienced episodes where choosing not to flap allowed it to navigate partially through the tubes, resulting in higher rewards. Compared to the early stages, the bird’s actions now reflect a understanding of how to survive longer. However, at this stage it has not passed through a tube yet.

3. During exploration, the agent randomly takes actions (e.g., choosing "no flap" or action 0) that it might not otherwise select based on its current policy. This randomness allows the agent to discover that descending at specific times can lead to higher rewards, even if it initially seems counterintuitive. Replay memory stores past experiences (state, action, reward, next state) for training. This mechanism allows the model to learn from a variety of scenarios, including rare successful actions, rather than only relying on the most recent transitions. As training progresses, older experiences where the bird immediately flew to the top are replaced by more recent, informative episodes, such as those where the bird navigated closer to passing through the tubes. This prioritization helps the model focus on learning effective strategies for longer survival. Interestingly, we observe a spike in the loss function (e.g., at Step 17500), highlighting that the model is actively learning. This indicates that the model has identified passing through the tube as yielding unexpectedly higher rewards in the Q-value function. As a result, it adjusts its policy, recognizing that strategic flapping with pauses to navigate through the tube is more rewarding than simply flying to the top.

It takes a long time to fully train the network, so you're not required to complete the training. Here's a video showing the performance of a well trained DQN.