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 [1]:
```
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 [2]:
```
# 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 $a_0, \ldots, a_d$ 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.

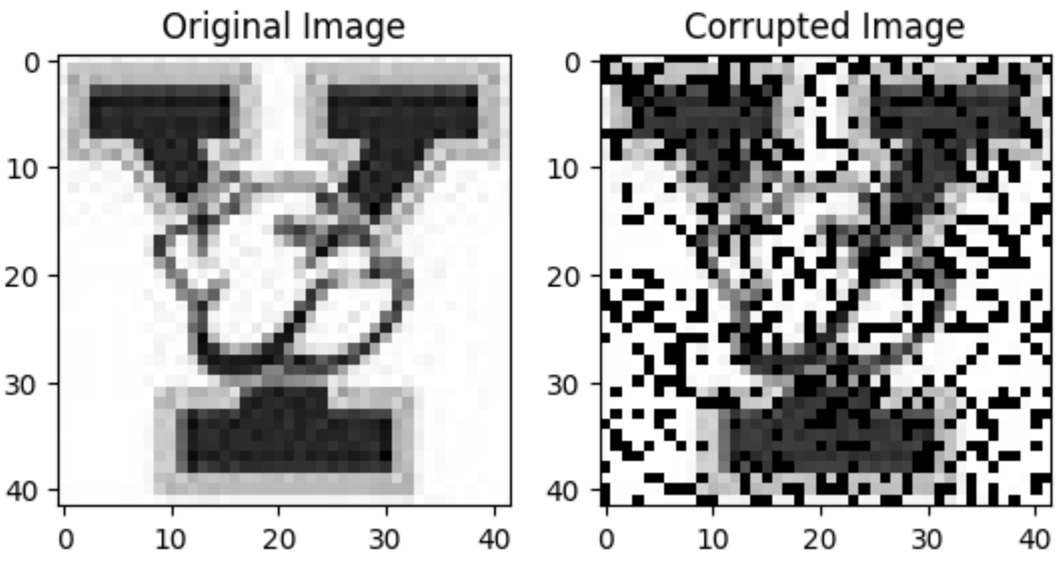
3. This polynomial filter has many applications. A handful of these applications are based on the fact that, given a graph with a signal x, the value of $x^T L x$ will be low in case the signal is smooth (i.e. smooth transitions of x between neighboring nodes). A large $x^T L x$ means that we have a 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 [4]:
```
# 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:

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

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:

your markdown here

To solve this optimization problem, we can use the method of Lagrange multipliers to derive a closed-form solution. The objective function is given by:

$$f(\mathbf{x}) = \|\mathbf{y}-\mathbf{M}\mathbf{x}\|_2^2 + \alpha\mathbf{x}^T\mathbf{P}\mathbf{x}$$

where (\textbf{M}) is a diagonal matrix that differentiates between missing and observed pixels. We start by expanding the objective function:

$$f(\mathbf{x}) = (\mathbf{y}-\mathbf{M}\mathbf{x})^T(\mathbf{y}-\mathbf{M}\mathbf{x}) + \alpha\mathbf{x}^T\mathbf{P}\mathbf{x}$$

Expanding and simplifying this, we get:

$$f(\mathbf{x}) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + \alpha\mathbf{x}^T\mathbf{P}\mathbf{x}$$

To find the optimal (\textbf{x}), we take the derivative with respect to (\textbf{x}) and set it to zero:

$$-2\mathbf{M}^T\mathbf{y} + 2\mathbf{M}^T\mathbf{M}\mathbf{x} + 2\alpha\mathbf{P}\mathbf{x} = 0$$

This simplifies to:

$$(\mathbf{M}^T\mathbf{M} + \alpha\mathbf{P})\mathbf{x} = \mathbf{M}^T\mathbf{y}$$

Since (\textbf{M}) is a diagonal matrix (with entries of either 0 or 1), (\textbf{M}^T \textbf{M} = \textbf{M}). Therefore, we can rewrite the equation as:

$$(\mathbf{M} + \alpha\mathbf{P})\mathbf{x} = \mathbf{M}\mathbf{y}$$

Thus, the closed-form solution for (\textbf{x}) is:

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

This expression provides the restored image by taking into account both the observed values in (\textbf{y}) and smoothing based on the polynomial filter (\textbf{P}). The regularization parameter (\alpha) controls the trade-off between fitting the observed data and promoting smoothness in the reconstructed image.

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:

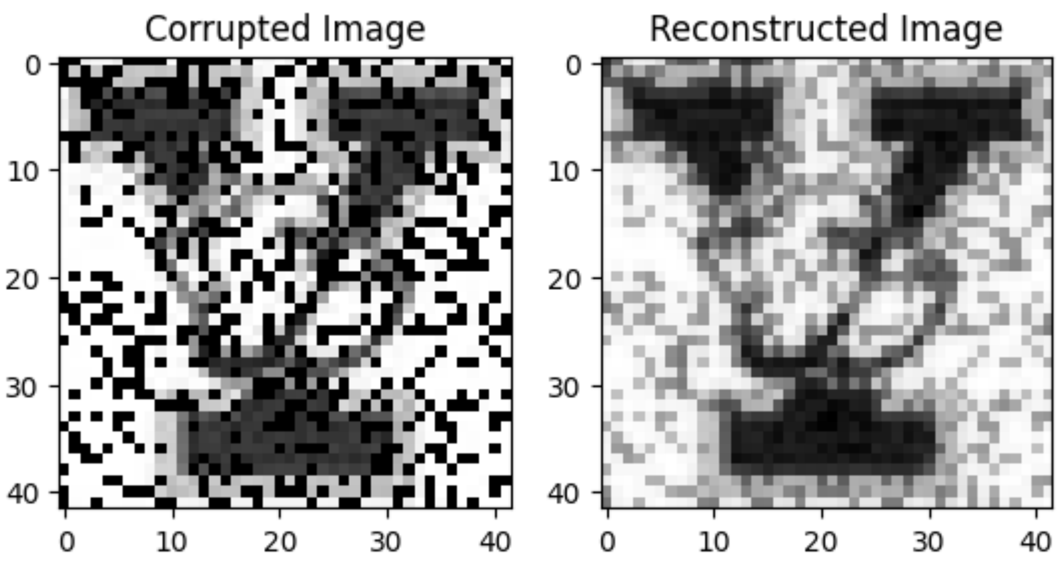
$$\mathbf{P} = \mathbf{L} + 0.05\,\mathbf{L}^2$$

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 [5]:
```
# Define the corrupted graph signal
y = corrupted_image.flatten()

# Define the diagonal matrix M based on observed pixels
M = np.diag(mask.flatten())

# Create the adjacency matrix A of the graph
A = grid_adj(height_img, width_img)

# Define the degree matrix D
D = np.diag(A.sum(axis=1))

# Define the graph Laplacian L
L = D - A

# Polynomial filter P (for example, using a simple form like P = L)
# You can adjust coefficients for higher-order terms if desired
a_0, a_1 = 1, 1
P = a_0 * np.eye(L.shape[0]) + a_1 * L
```

In [6]:
```
# closed form solution you derived above
# Solve for x using the closed-form solution with different alpha values
alpha = 0.1  # Adjust alpha for different smoothing effects
x = inv(M + alpha * P).dot(M).dot(y)
```

In [7]:
```
# Try to experiment with different alpha values
alpha = 0.1
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?

Influence of α:

1. **Very Large** α:
   When α is large, the regularization term $\alpha \mathbf{x}^T \mathbf{P} \mathbf{x}$ becomes dominant in the objective function. This means the solution will prioritize smoothness over fidelity to the original image data. As a result, the reconstructed image will appear overly smoothed, potentially losing important details and high-frequency information, which could make edges and textures in the image less distinguishable. In extreme cases, the reconstructed image might become nearly uniform, lacking any contrast or texture.

2. **Very Low** α:
   When α is close to zero, the data-fitting term $\|\mathbf{y} - \mathbf{M}\mathbf{x}\|_2^2$ becomes more influential. This leads to a solution that closely matches the observed values in $\mathbf{y}$ but might not enforce smooth transitions between pixels. Consequently, the reconstructed image will better retain original details but may exhibit abrupt changes or noise, especially in regions where data is missing. This could result in a "noisy" appearance in the reconstructed image, with high-frequency artifacts at the edges of missing data regions.

Influence of the Polynomial Degree of $\mathbf{P}$:

The degree of the polynomial function $\mathbf{P}$ influences the filter's ability to capture different frequencies in the graph signal, which affects how smooth the reconstructed image will be.

1. **Low Degree of** $\mathbf{P}$:

With a low degree (e.g., $d = 1$ or $d = 2$), $\mathbf{P}$ mainly captures low-frequency components of the graph signal. This degree will only enforce basic smoothness, resulting in a moderate level of smoothing, which might be insufficient for effectively diffusing information across larger gaps in missing data.

2. **High Degree of $\mathbf{P}$**:
   As the degree of $\mathbf{P}$ increases, the filter becomes more capable of capturing higher frequencies in the graph signal. This can help propagate information across more distant nodes in the graph, resulting in a stronger smoothing effect. However, if the degree is too high, it could lead to over-smoothing and loss of detail, similar to the effect of a very large $alpha$. Additionally, a high polynomial degree increases the computational cost and can lead to numerical instability, especially in large graphs.

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 $\sigma^2_\theta(s)$, two outputs of a neural network with input s and parameters θ.

Suppose the outputs of the neural network are given by

$$\mu_\theta(s) = \beta_1^T h(s)$$
$$\sigma^2_\theta(s) = \exp(\beta_2^T h(s))$$

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 $\nabla_{\beta_2} \log \pi_\theta(a \mid s)$.

Explain how these gradients and other gradient terms in $\nabla_\theta \log \pi_\theta(a \mid s)$ are used to estimate the policy.

2.2 Discrete action space with Softmax policy

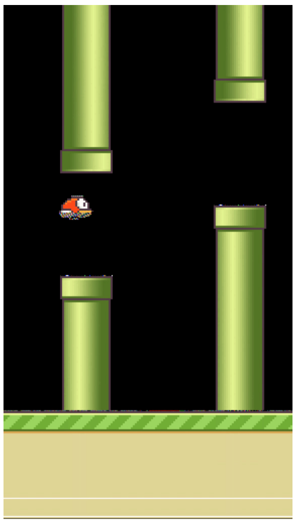
Suppose the action space is discrete with K possible actions, and the policy $\pi_\theta(a \mid s)$ is defined using a softmax function over preferences $u_\theta(s, a)$:

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

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 $\nabla_\beta \log \pi_\theta(a \mid s)$.

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 [31]:
```
# %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
```

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 [32]:
```
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.

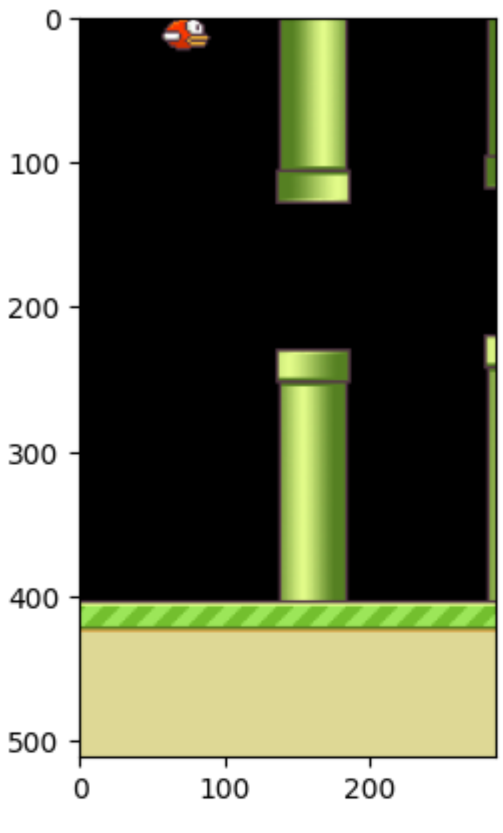
Let's take some random actions and see what happens:

In [33]:
```
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 [34]:
```
# show the image
import matplotlib.pyplot as plt
plt.imshow(image.transpose([1, 0, 2]))
plt.show()
plt.close()
```

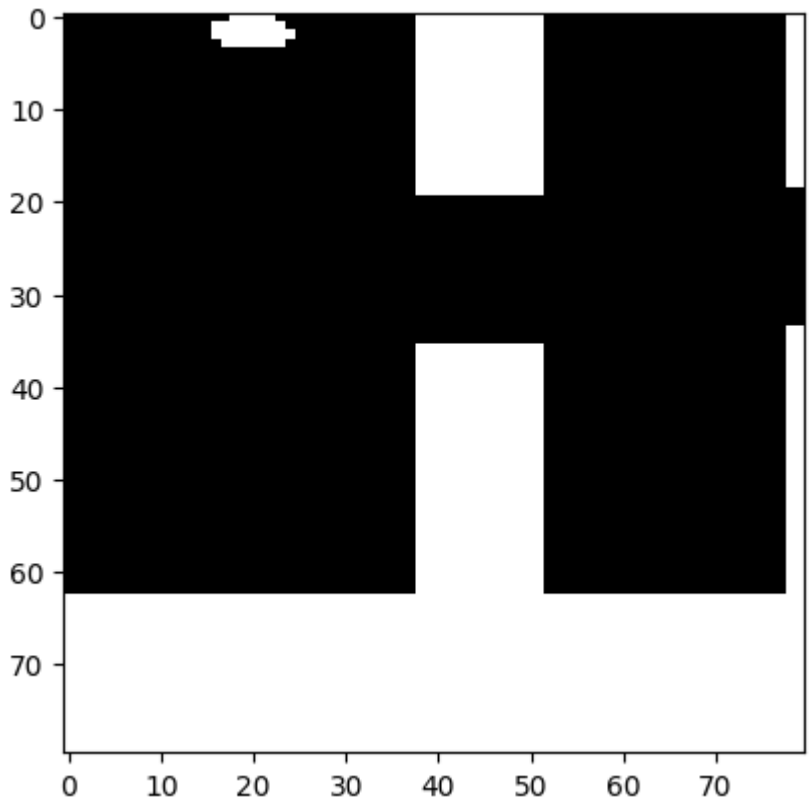


Preprocessing the images

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 [35]:
```
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 [36]:
```
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:

let agent have memory: to determine whether objects are moving closer or farther away, or to estimate the agent's velocity and acceleration relative to other objects.

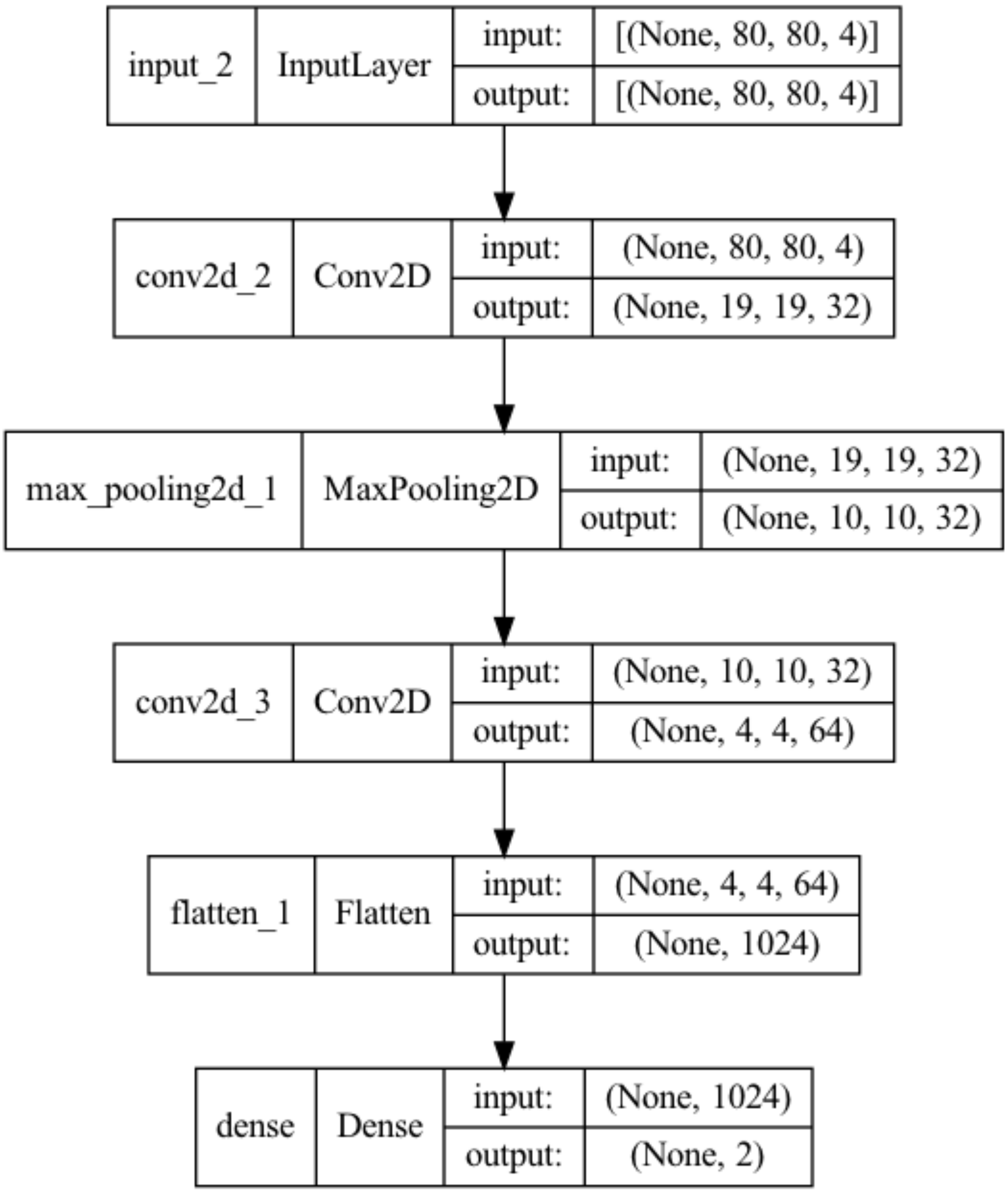
The state is chosen as a stack of four consecutive frames rather than a single frame to capture temporal information about the game environment. This is particularly useful in environments where movement and object dynamics are key, where the agent needs to understand the direction and speed of obstacles (like pipes) and itself.

If we use only a single frame, we lose the temporal context, making it difficult for the agent to determine whether objects are moving closer or farther away, or to estimate the agent's velocity and acceleration relative to other objects. By stacking four frames, the agent gains insight into these temporal aspects, allowing it to learn better strategies for navigation and obstacle avoidance based on the movement history contained in those frames. This temporal awareness is crucial for making accurate decisions in real-time.

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:


input_2 | InputLayer | input: | [(None, 80, 80, 4)]
output: | [(None, 80, 80, 4)]
conv2d_2 | Conv2D | input: | (None, 80, 80, 4)
output: | (None, 19, 19, 32)
max_pooling2d_1 | MaxPooling2D | input: | (None, 19, 19, 32)
output: | (None, 10, 10, 32)
conv2d_3 | Conv2D | input: | (None, 10, 10, 32)
output: | (None, 4, 4, 64)
flatten_1 | Flatten | input: | (None, 4, 4, 64)
output: | (None, 1024)
dense | Dense | input: | (None, 1024)
output: | (None, 2)

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 [42]:
```
from tensorflow.keras import layers, initializers
from tensorflow import keras

def create_q_model():
    # Input layer
    state = layers.Input(shape=(80, 80, 4,))

    # First Conv2D layer
    layer1 = layers.Conv2D(filters=32, kernel_size=8, strides=4, activati
                           kernel_initializer=initializers.TruncatedNorma
                           bias_initializer=initializers.Constant(0.01))(

    # MaxPooling layer
    layer2 = layers.MaxPool2D(pool_size=2, strides=2, padding="same")(lay

    # Second Conv2D layer
    layer3 = layers.Conv2D(filters=64, kernel_size=3, strides=2, activati
                           kernel_initializer=initializers.TruncatedNorma
                           bias_initializer=initializers.Constant(0.01))(

    # Flatten layer
    layer4 = layers.Flatten()(layer3)

    # Dense layer for output
    q_value = layers.Dense(units=2, activation="linear",
                           kernel_initializer=initializers.TruncatedNorma
                           bias_initializer=initializers.Constant(0.01))(

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

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

In [43]:
```
model = create_q_model()
print(model.summary())
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_9 (InputLayer)        [(None, 80, 80, 4)]       0         
                                                                 
 conv2d_8 (Conv2D)           (None, 19, 19, 32)        8224      
                                                                 
 max_pooling2d (MaxPooling2D  (None, 10, 10, 32)       0         
 )                                                               
                                                                 
 conv2d_9 (Conv2D)           (None, 4, 4, 64)          18496     
                                                                 
 flatten (Flatten)           (None, 1024)              0         
                                                                 
 dense (Dense)               (None, 2)                 2050      
                                                                 
=================================================================
Total params: 28,770
Trainable params: 28,770
Non-trainable params: 0
_________________________________________________________________
None
```

```
2024-11-15 00:05:55.212783: I tensorflow/core/platform/cpu_feature_guard.c
c:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network
Library (oneDNN) to use the following CPU instructions in performance-crit
ical operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
2024-11-15 00:05:55.213400: I tensorflow/core/common_runtime/process_uti
l.cc:146] Creating new thread pool with default inter op setting: 2. Tune
using inter_op_parallelism_threads for best performance.
```

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 [44]:
```
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 [45]:
```
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?

maintaining a fixed-size replay memory is a practical choice to keep memory requirements manageable, reduce temporal correlation in training data, adapt the learning to the latest policy, and prevent the agent from being influenced by outdated and less relevant data.

1. **Memory Constraints**: Storing all historical data can become computationally infeasible for practical problems, especially as training goes on for many steps. Deep reinforcement learning can require millions of time steps, and maintaining all that data would require an enormous amount of memory. Fixing the size of the replay memory helps keep memory usage manageable.

2. **Efficiency in Learning**: As the model learns, older data becomes less useful or

even counterproductive, especially in a non-stationary environment where the dynamics or optimal policy may change over time. Including all data would increase the training time while introducing potentially outdated data that could slow down learning or lead to instability.

3. **Avoiding Correlation**: Experience replay aims to break the temporal correlation between consecutive experiences by shuffling and randomly sampling from the replay memory. This helps reduce the chance that the neural network overfits to sequences in the data. A fixed replay memory size ensures that the data in memory is sufficiently diverse and does not include long, correlated sequences from the past.

4. **Better Adaptation to Current Dynamics**: Since the agent's policy is constantly evolving during training, the relevance of older experiences decreases over time. Using a fixed-size buffer with the most recent experiences allows the agent to adapt more effectively to its current policy and environment dynamics, rather than being influenced by potentially outdated and less relevant experiences.

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 [46]:
```
# 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

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

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

In [53]:
```
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 = np.argmax(q_value.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
            next_q_values = model(tf.convert_to_tensor(state_next_sample)
            max_next_q_values = np.max(next_q_values, axis=1)
            updated_q_values = reward_sample + (1 - terminal_sample) * ga

            # train the model on the states and updated Q-values
            with tf.GradientTape() as tape:
                # compute the current Q-values for the samples
                current_q_values = model(tf.convert_to_tensor(state_sampl
                current_q_values = tf.reduce_sum(current_q_values * tf.on

                # compute the loss
                loss = loss_function(updated_q_values, current_q_values)

            # backpropagation
            grads = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(grads, model.trainable_variable
        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}")
```

In [55]:
```
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
    loss_function = keras.losses.MeanSquaredError()

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

In [56]:
```
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 -1 | LOSS 0.01683366112411022
2
STEP 11000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0261604916304349
9
STEP 11500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0032830291893333
197
STEP 12000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0105573590844869
61
STEP 12500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1992755681276321
4
STEP 13000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.3192921280860901
STEP 13500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0055427094921469
69
STEP 14000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0120961070060729
98
STEP 14500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0934694781899452
2
STEP 15000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0077452268451452
255
STEP 15500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0255564786493778
23
STEP 16000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0023611634969711
304
STEP 16500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0032806308008730
41
STEP 17000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0156063009053468
7
STEP 17500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0031164716929197
31
STEP 18000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0058163255453109
74
STEP 18500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0053410101681947
71
STEP 19000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0084379604086279
87
STEP 19500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0225962251424789
43
STEP 20000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0354456529021263
1
STEP 20500 | PHASE train | ACTION 0 | REWARD 1 | LOSS 0.07592210173606873
STEP 21000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1060530617833137
5
STEP 21500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0451483316719532
STEP 22000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0233204104006290
44
STEP 22500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0368303284049034
1
STEP 23000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0492456592619419
1
STEP 23500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2079674005508422
9
STEP 24000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0197287499904632
57
STEP 24500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0295626968145370
5
STEP 25000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0230500735342502
6
STEP 25500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0272255428135395
05
STEP 26000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0161113664507865
9
STEP 26500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0409014299511909
5
STEP 27000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0776953771710395
8
STEP 27500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0688210725784301
8
STEP 28000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0387376807630062
1
STEP 28500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0679298043251037
6
STEP 29000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1000777035951614
4
STEP 29500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0998431742191314
7
STEP 30000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0406733676791191
1
STEP 30500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.5376922488212585
STEP 31000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.0564160719513893
1
STEP 31500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0664462447166442
9
STEP 32000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0997502505779266
4
STEP 32500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.4176612496376037
6
STEP 33000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1167316064238548
3
STEP 33500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3993814885616302
5
STEP 34000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1572796106338501
STEP 34500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.2133585810661316
STEP 35000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2799621820449829
STEP 35500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0748222470283508
3
STEP 36000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2792862951755523
7
STEP 36500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.1790128946304321
3
STEP 37000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3140020370483398
```

```
4
STEP 37500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1833860278129577
6
STEP 38000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 1.579838752746582
STEP 38500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0837092474102974
STEP 39000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2358281016349792
5
STEP 39500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.1793413311243057
3
STEP 40000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.6232079267501831
STEP 40500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2786513268947601
3
STEP 41000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1860232502222061
2
STEP 41500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2037529796361923
2
STEP 42000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.4208776950836181
6
STEP 42500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.7783730030059814
STEP 43000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 1.1934220790863037
STEP 43500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2332394123077392
6
STEP 44000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3275186419486999
5
STEP 44500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3040559887886047
4
STEP 45000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2225796580314636
2
STEP 45500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.3317279517650604
STEP 46000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.1620420515537262
STEP 46500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.8455294370651245
STEP 47000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.1906527727842331
STEP 47500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.2109429985284805
3
STEP 48000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3969378173351288
STEP 48500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2077260315418243
4
STEP 49000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.2027528136968612
7
STEP 49500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.152142733335495
STEP 50000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.3111290335655212
4
```

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.

Analysis of the Q-learning Training Process for Flappy Bird

1. Early Stage of Training (Within 2,000 Steps in the Observe Phase)

- **Behavior of Flappy Bird**: In the early phase (first 10,000 steps, specifically within 2,000 steps), the agent is primarily in the observation phase, where it takes random actions to gather diverse data about the environment. During this stage, the agent's actions are not guided by any understanding of the game or optimal strategies.
    - **Greedy Policy Estimate**: Since the Q-function is still largely untrained, the model's value estimates are quite arbitrary. The Q-values are essentially uninformative at this point, which means the agent is unlikely to have developed a coherent greedy policy. The actions taken are random, resulting in exploration without a clear strategy to avoid obstacles or navigate effectively. This is why the behavior appears inconsistent and non-optimal.

2. Behavior after Roughly 5,000 Training Steps

- **Improvement in Performance**: After 5,000 training steps, the model has started transitioning into the training phase and is updating its weights based on experiences sampled from the replay memory. The loss starts to decrease, indicating that the model is learning to approximate the optimal Q-values better.
    - **Comparison to Early Stages**: Compared to the early stages, where the agent was exploring randomly, there is now some evidence that the agent is learning to associate certain states with higher expected rewards. This results in more structured and purposeful actions that improve the game performance (e.g., avoiding obstacles more effectively or maintaining a better altitude).

3. Reasons for Improved Performance

- **Replay Memory**: The replay memory helps reduce the correlation between consecutive samples by shuffling experiences and using them for training in batches. This stabilizes learning because the agent is not overfitting to a specific sequence of events, and the model can learn from a diverse range of experiences. The fixed replay memory ensures that only relatively recent experiences are used, making the training data more relevant to the agent's current policy.

- **Exploration vs. Exploitation**: In the beginning, the agent explores the environment using random actions (`epsilon` is high). This allows the agent to gather varied experiences that cover different states of the environment. Over time, as `epsilon` decreases, the agent shifts from exploration to exploitation. This allows the agent to act according to the learned Q-values more frequently, thereby following a policy that is becoming increasingly optimal as training progresses. This balance between exploration and exploitation is critical for discovering an effective policy.

- **Batch Updates for Stability**: Using batch updates also contributes to stable learning. Instead of updating the model after every step (which could lead to unstable learning due to high variance in rewards), the agent samples a batch of experiences, effectively averaging the updates over a more comprehensive set of experiences. This helps in reducing variance and improving the stability of Q-value updates.

- **Q-learning Update Rule**: The Q-learning update rule incorporates the immediate reward and the maximum estimated future reward, weighted by the discount factor (`gamma`). Initially, the Q-values are arbitrary, but as training progresses, these values become more accurate estimates of the expected cumulative reward for each state-action pair. The combination of current rewards and future expected rewards helps the agent learn sequences of actions that yield high long-term gains rather than being myopically focused on short-term rewards.

Overall, the combination of replay memory, exploration-exploitation trade-off, and the Q-learning update mechanism allows the agent to learn effective policies over time, improving its performance in the game. This is evident from the observed reduction in loss and improvement in game behavior as the number of training steps increases.

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.