# 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 [ ]:
```
# 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

$L_{ii} = D_{ii} - w_{ii} = D_{ii} = \sum_j^n w_{ij}$

$L = D - W \rightarrow L_{ij}\ (i \neq j) = D_{ij} - w_{ij} = -w_{ij}$ = weight

1. WTS $L$ is a Mercer kernel

a) WTS $L$ is symmetric: $L_{ij} = L_{ji}$

i) case 1: $i = j$ so $L_{ii} = L_{ii}$ ✓

ii) case 2: $i \neq j$ so $L_{ij} = D$

$L_{ij} = D_{ij} - w_{ij} = -w_{ij}$

$-w_{ij} = -\text{weight}(i_j) = -\text{weight}(j_i) = -w_{ji}$ ✓

$\therefore L_{ij} = L_{ji}$ so $L$ is symmetric

b) WTS $L$ is positive-semidefinite: WTS for all $x \in \mathbb{R}^n$, $x^T L x \geq 0$

A diagonally dominant real positive symmetric matrix is positive semidefinite. (textbook)

WTS: $|L_{ii}| \geq \sum_{j \neq i} |L_{ij}|$ for all $i$

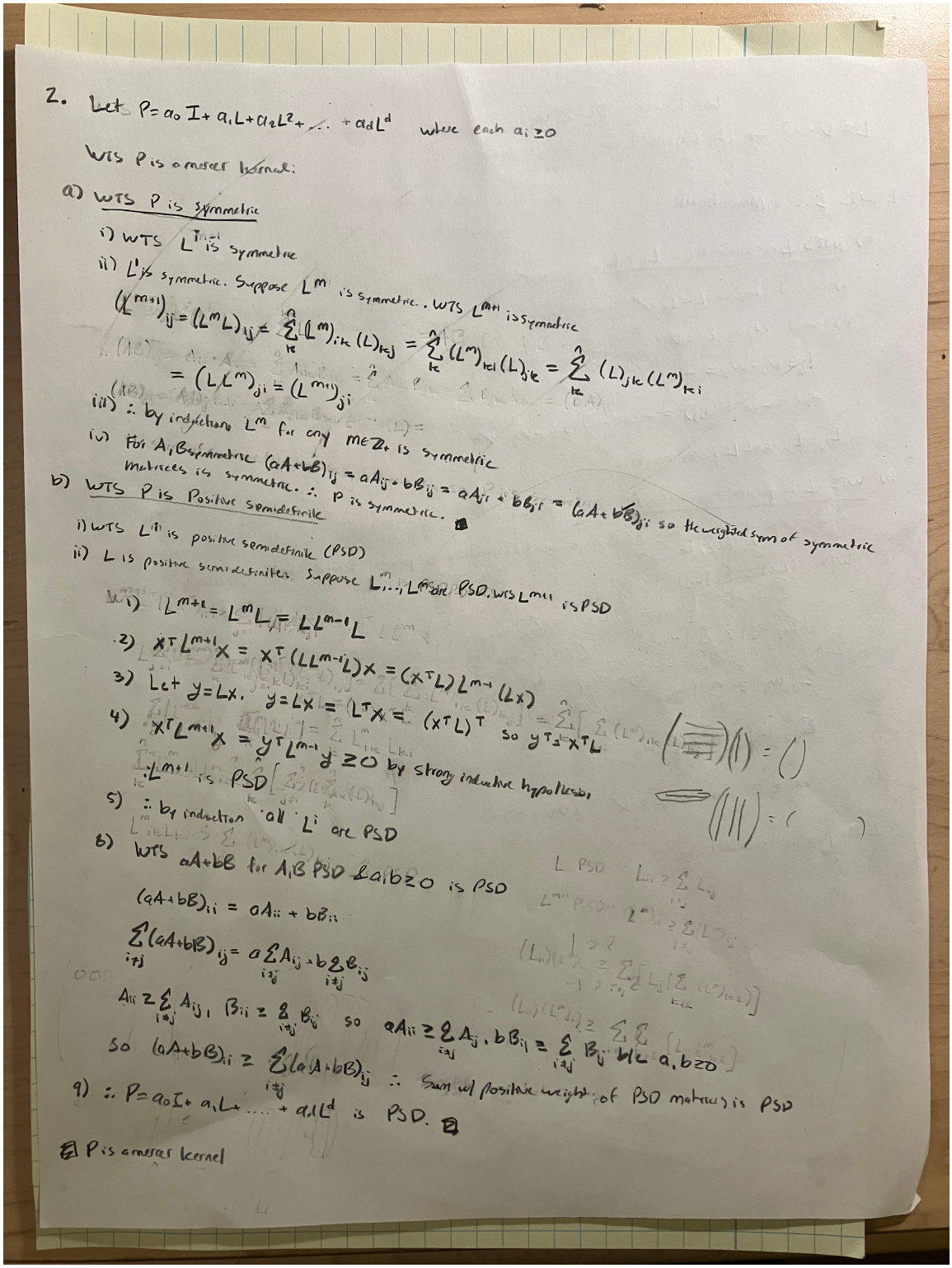
$|\sum_j^n w_{ij}| \stackrel{?}{\geq} \sum_{j \neq i} |-w_{ij}|$ (definition of $L$)

$\sum_j^n |w_{ij}| \stackrel{?}{\geq} \sum_{j \neq i} |w_{ij}|$ ($w_{ij} \geq 0$)

$\sum_{j \neq i}^n w_{ij} + w_{ii} \stackrel{?}{\geq} \sum_{j \neq i} w_{ij}$

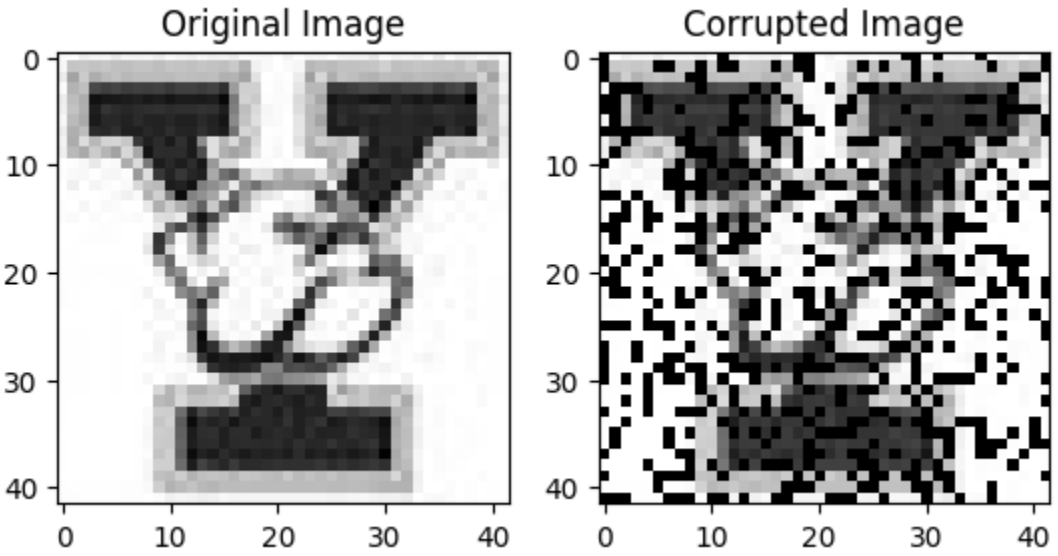
$w_{ii} \stackrel{?}{\geq} 0 \rightarrow 0 \geq 0$ ✓ ($w_{ii} = 0$)

$\therefore L$ is positive semidefinite




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()
```



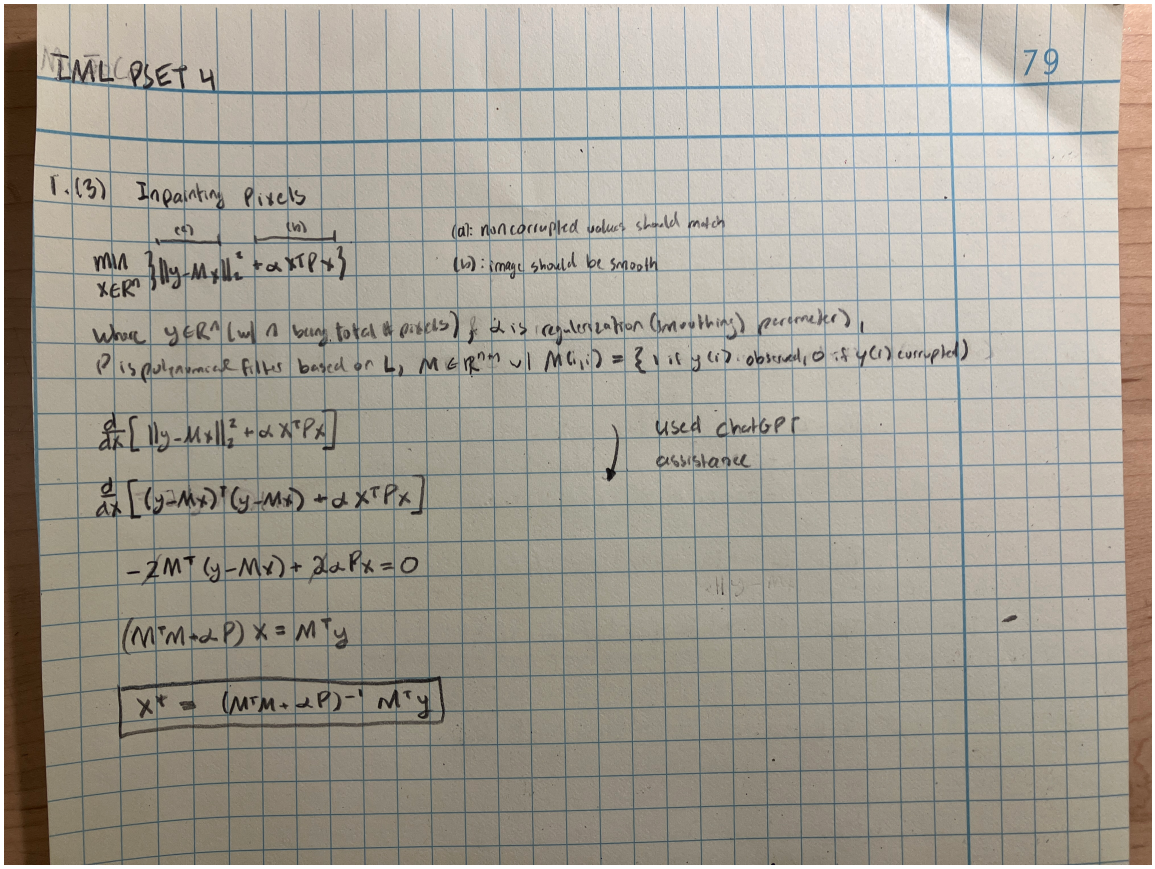

Inpainting missing pixel values can be formulated as the following optimization problem:

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

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 $\alpha$ 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:

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

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:




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:

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

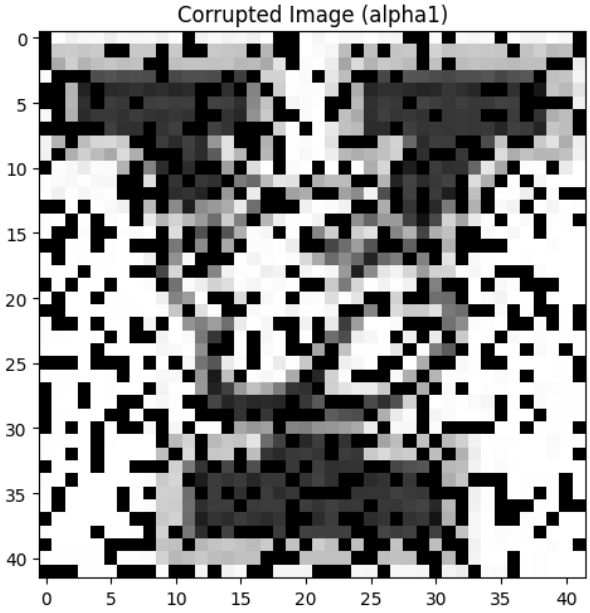
Fill in the following lines of code and show your reconstructed images next to the corrupted image. Assume that the weights on the graph edges are equal to 1.

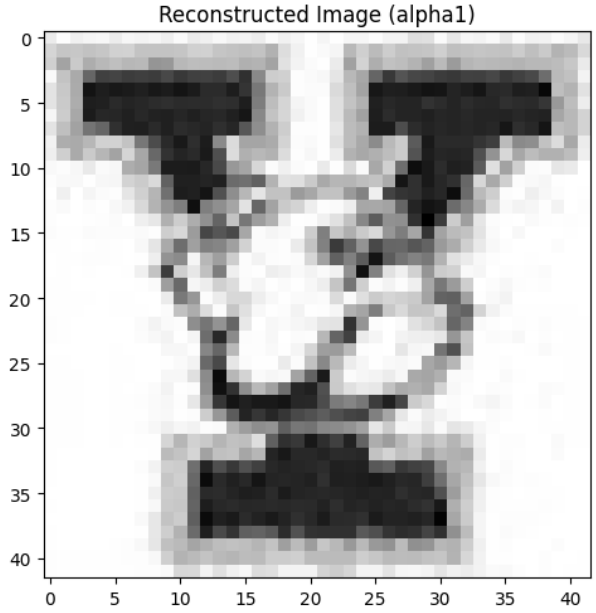
In [ ]:
```
# Corrupted graph signal
y = np.reshape(corrupted_image, (height_img*width_img,))

# Diagonal matrix defined as above
M = np.diag(np.reshape(mask, (height_img*width_img,)))

# Adjacency matrix of the graph (by using the helper function)
A = grid_adj(height_img, width_img)

# Diagonal matrix defined as above
D = np.diag([np.sum(A[i, :]) for i in range(height_img*width_img)])

# Graph Laplacian defined as above
L = D - A

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

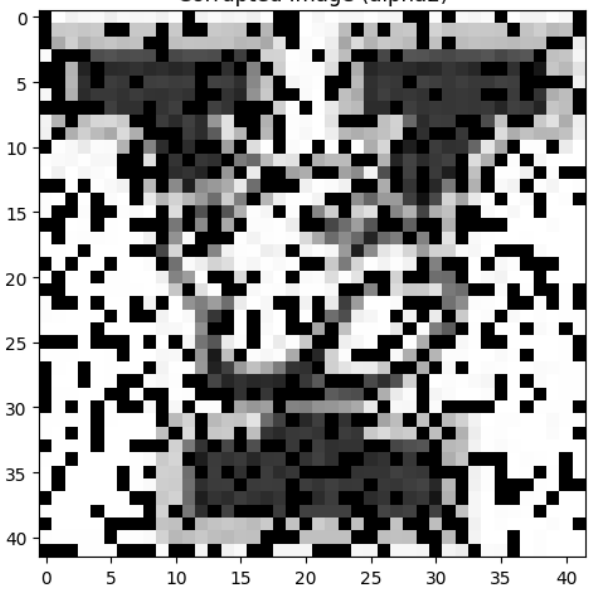
In [ ]:
```
# Try to experiment with different alpha values
alpha1 = 0.01
```

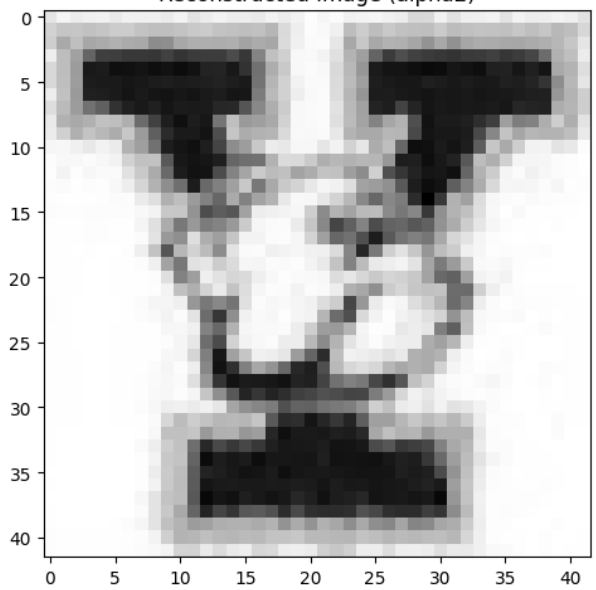
```
alpha2 = 0.1
alpha3 = 1

# closed form solution you derived above
x1 = np.linalg.inv(M.T @ M + alpha1 * P) @ M.T @ y
x2 = np.linalg.inv(M.T @ M + alpha2 * P) @ M.T @ y
x3 = np.linalg.inv(M.T @ M + alpha3 * P) @ M.T @ y
```

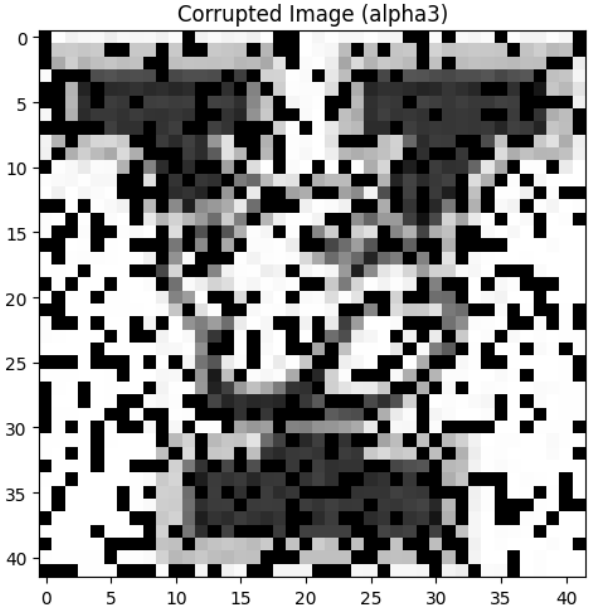
In [57]:
```
fig, axes = plt.subplots(3, 2, figsize=(10, 15))

# Reconstructed images
reconstructed_image1 = np.reshape(x1, (height_img, width_img))
reconstructed_image2 = np.reshape(x2, (height_img, width_img))
reconstructed_image3 = np.reshape(x3, (height_img, width_img))

# Plot corrupted and reconstructed images for alpha1
axes[0, 0].imshow(corrupted_image, cmap=plt.get_cmap('gray'))
axes[0, 0].set_title("Corrupted Image (alpha1)")
axes[0, 1].imshow(reconstructed_image1, cmap=plt.get_cmap('gray'))
axes[0, 1].set_title("Reconstructed Image (alpha1)")

# Plot corrupted and reconstructed images for alpha2
axes[1, 0].imshow(corrupted_image, cmap=plt.get_cmap('gray'))
axes[1, 0].set_title("Corrupted Image (alpha2)")
axes[1, 1].imshow(reconstructed_image2, cmap=plt.get_cmap('gray'))
axes[1, 1].set_title("Reconstructed Image (alpha2)")

# Plot corrupted and reconstructed images for alpha3
axes[2, 0].imshow(corrupted_image, cmap=plt.get_cmap('gray'))
axes[2, 0].set_title("Corrupted Image (alpha3)")
axes[2, 1].imshow(reconstructed_image3, cmap=plt.get_cmap('gray'))
axes[2, 1].set_title("Reconstructed Image (alpha3)")

plt.tight_layout()
plt.show()
```

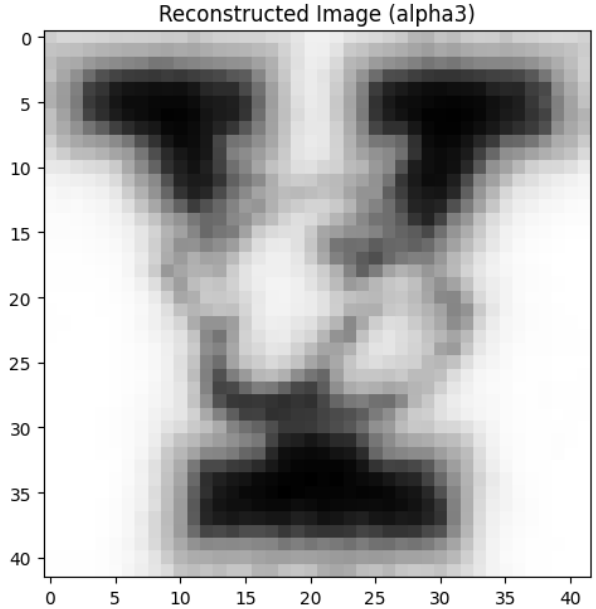

Corrupted Image (alpha1)


Reconstructed Image (alpha1)


Corrupted Image (alpha2)


Reconstructed Image (alpha2)


Corrupted Image (alpha3)


Reconstructed Image (alpha3)

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

Increasing the smoothing parameter alpha smooths the edges in the reconstructed image by penalizing sharp changes in neigboring pixel values. I can't see much difference between alpha=0.1 and alpha=1 reconstructions. The alpha=10 reconstruction is noticibly blurry. Increasing the degree of P offers additional tuning parameters, which may lead to a better fit, and allows one pixel's value to be informed by a wider net of neighboors' values because the degree d of the polynomial is the range d of nearest neigboors to a pixels whose values contribute to the P calculation for that pixel.

# 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 $\tau$ through a choice of actions using the policy. This can be rewritten as

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

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

## 2.1 Continuous action space with Gaussian policy

Suppose that the action space is continuous and $\pi_\theta(a \,|\, s)$ 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 $\theta$.

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 $\nabla_{\beta_1} \log \pi_\theta(a \,|\, s)$ and $\nabla_{\beta_2} \log \pi_\theta(a \,|\, s)$.

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

The policy estimation is done by maximizing J(theta), the expectation of the rewards,

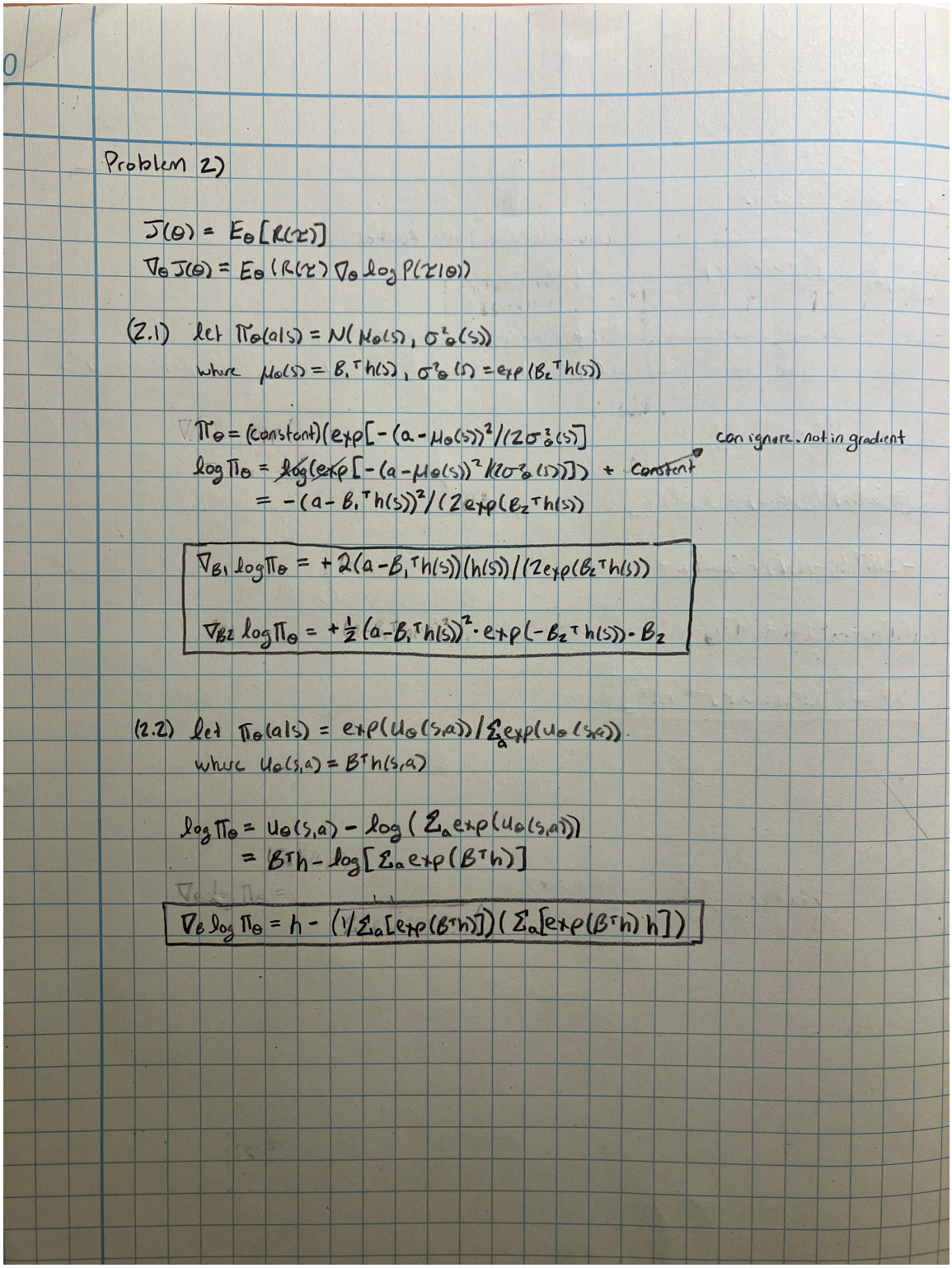
through gradient descent. The gradient of the log of the policy with respect to the paremeters identifies the direction in which each parameter can be adjusted to most increase the probability of taking actions that cause episodes with high rewards. The policy is then adjusted along this gradient to improve its performance. This gradient descent is done across all parameters in the DNN, not just B1 and B2.

## 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 2)

$J(\theta) = E_\theta[R(\tau)]$

$\nabla_\theta J(\theta) = E_\theta(R(\tau)\nabla_\theta \log P(\tau|\theta))$

(2.1) let $\pi_\theta(a|s) = N(\mu_\theta(s), \sigma^2_\theta(s))$

where $\mu_\theta(s) = B_1^T h(s)$, $\sigma^2_\theta(s) = \exp(B_2^T h(s))$

$\pi_\theta = (\text{constant})(\exp[-(a-\mu_\theta(s))^2/(2\sigma^2_\theta(s)]$ can ignore. not in gradient

$\log \pi_\theta = \cancel{\log}(\cancel{\exp}[-(a-\mu_\theta(s))^2/(2\sigma^2_\theta(s))]) + \text{constant}$

$= -(a-B_1^T h(s))^2/(2\exp(B_2^T h(s))$

$\nabla_{B_1} \log \pi_\theta = +2(a-B_1^T h(s))(h(s))/(2\exp(B_2^T h(s))$

$\nabla_{B_2} \log \pi_\theta = +\frac{1}{2}(a-B_1^T h(s))^2 \cdot \exp(-B_2^T h(s)) \cdot B_2$

(2.2) let $\pi_\theta(a|s) = \exp(u_\theta(s,a))/\sum_a \exp(u_\theta(s,a))$

where $u_\theta(s,a) = B^T h(s,a)$

$\log \pi_\theta = u_\theta(s,a) - \log(\sum_a \exp(u_\theta(s,a))$

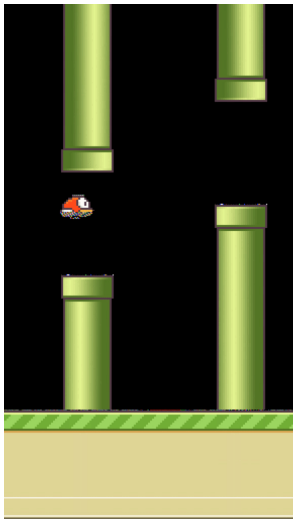
$= B^T h - \log[\sum_a \exp(B^T h)]$

$\nabla_\theta \log \pi_\theta = h - (1/\sum_a[\exp(B^T h)])(\sum_a[\exp(B^T h)\, h])$

# 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 [61]:
```
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 [62]:
```
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:

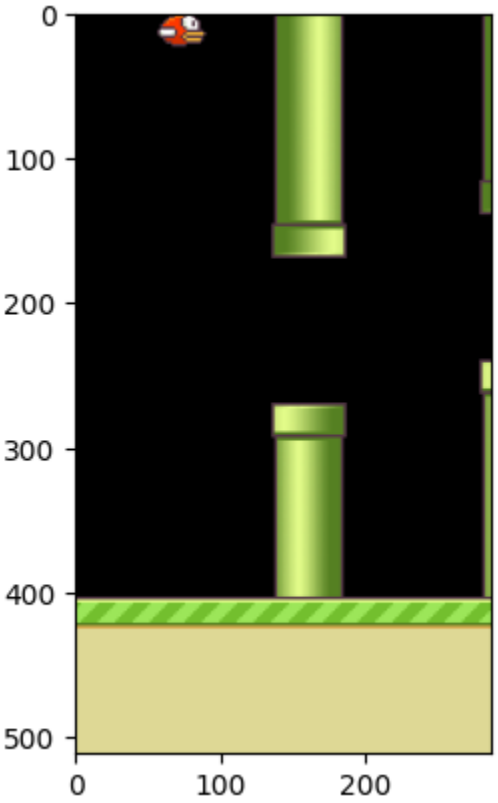
In [66]:
```
for i in range(587):

    # 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 [67]:
```
# show the image
import matplotlib.pyplot as plt
plt.imshow(image.transpose([1, 0, 2]))
plt.show()
plt.close()
```



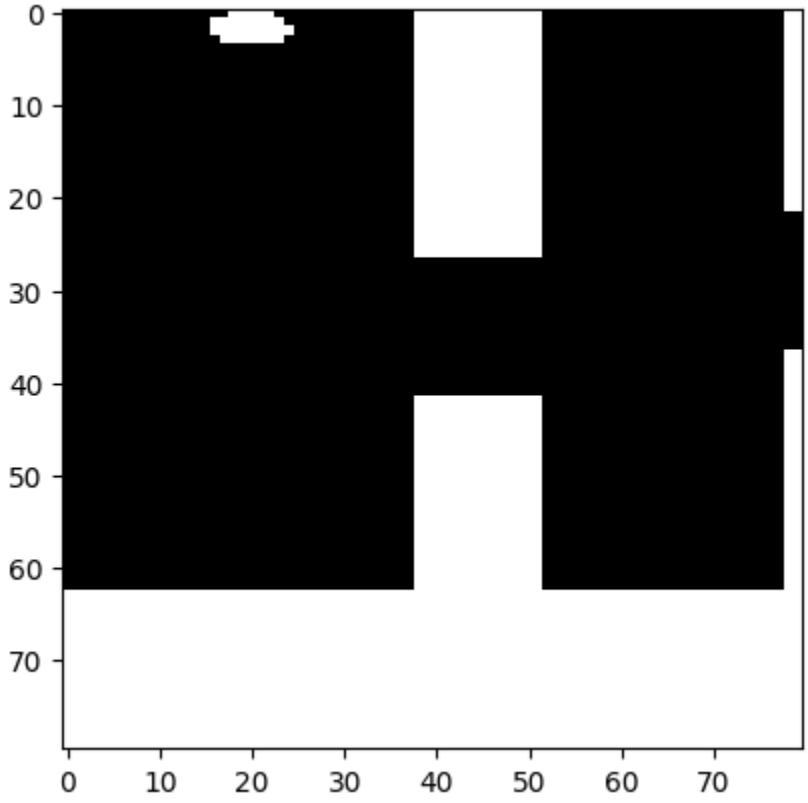

## Preprocessing the images

Alright, next we need to prepocess the images by converting them to grayscale and resizing them to $80 \times 80$ 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 [70]:
```
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 [71]:
```
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?

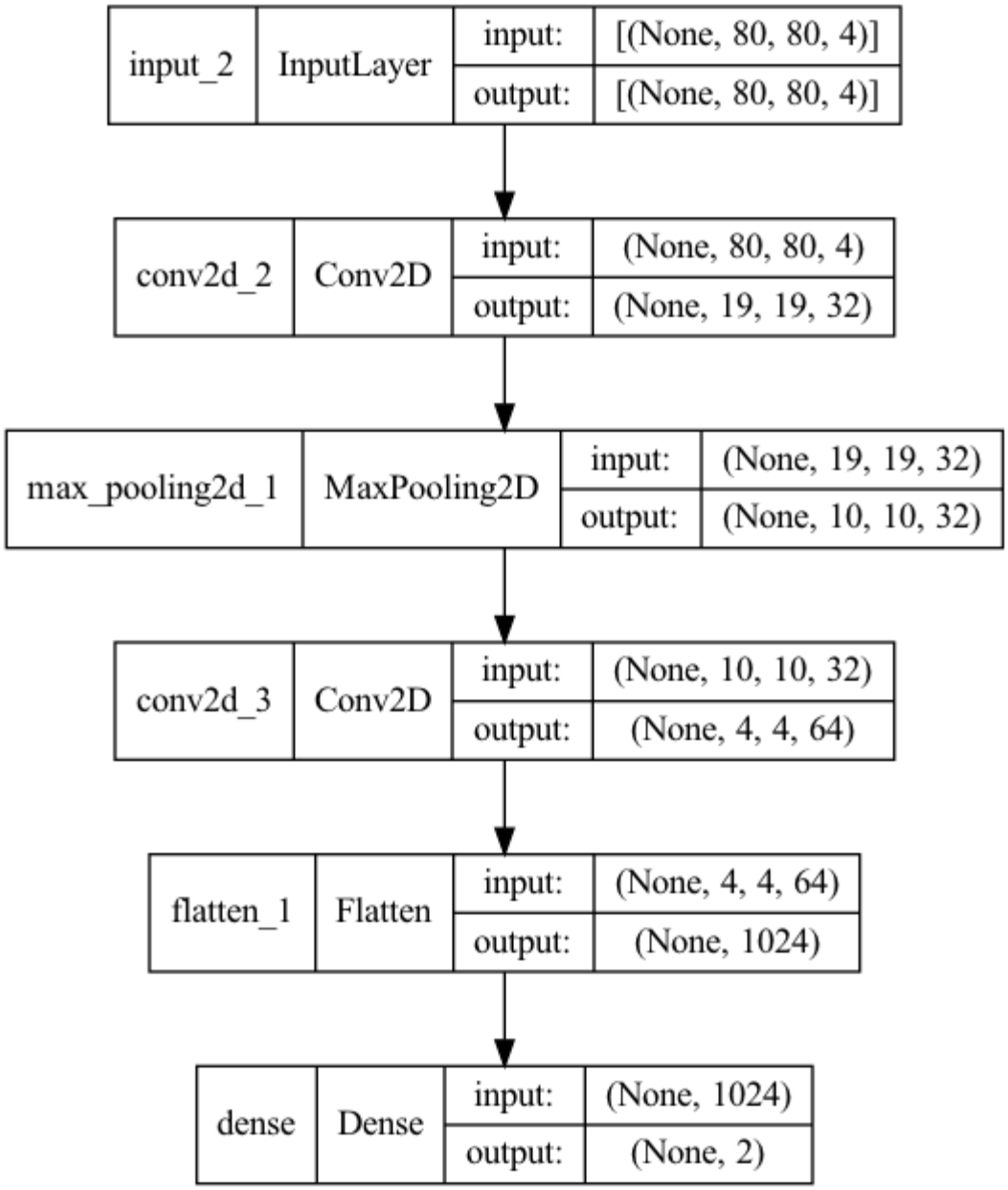
Give an intuitive explanation.

Four consecutive frames allow determination of the sprite's velocity. If the sprite is below the target, whether an action is correctly taken depends on whether the sprite is in that moment travelling up or down, which can be determing by the model considering the sprite's change in location across multiple frames.

## 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 [79]:
```
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.,
```

```
                            bias_initializer=initializers.Constant(0.01))(state)
    layer2 = layers.MaxPool2D(pool_size=(2, 2), strides=2, padding="SAME")(layer1)
    layer3 = layers.Conv2D(filters=64, kernel_size=3, strides=2, activation="relu",
                            kernel_initializer=initializers.TruncatedNormal(mean=0.,
                            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.,
                            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 [80]:
```
model = create_q_model()
print(model.summary())
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 80, 80, 4)]       0

 conv2d_3 (Conv2D)           (None, 19, 19, 32)        8224
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 80, 80, 4)]       0

 conv2d_3 (Conv2D)           (None, 19, 19, 32)        8224

 max_pooling2d_2 (MaxPooling  (None, 10, 10, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 4, 4, 64)          18496

 flatten_1 (Flatten)         (None, 1024)              0

 dense_1 (Dense)             (None, 2)                 2050

=================================================================
Total params: 28,770
Trainable params: 28,770
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 $\gamma$ is the discount factor and $\alpha$ can be viewed as the step size or learning rate for gradient ascent.

We'll set these as follows:

In [81]:
```
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 [117…
```
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?

Firstly, mantaining an arbitrarily long replay memory could strain the memory capacity of the computer itself (admitedly this probably isn't really an issue because of how relatively few bytes each frame is, but it would matter in general for training situations). The other big reason is that as the model gets better trained, the bird encounter new types of situations and to improve further, the model needs to learn how to handle those new situations, which it can only do if a new replay memory set is collected.

## 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 $\epsilon$, 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 $\theta$ refers to the parameters of our CNN.

In [83]:
```
# 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 [ ]:
```
# note: vscode gave suggestions on how to code this part.

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[0])

    # 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
        updated_q_value = np.zeros(batch_size)
        for i in range(batch_size):
            if terminal_sample[i]:
                updated_q_value[i] = reward_sample[i]
            else:
                next_state_tensor = tf.convert_to_tensor(state_next_sample[i])
                next_state_tensor = tf.expand_dims(next_state_tensor, 0)
                next_q_value = model(next_state_tensor, training=True)
                updated_q_value[i] = reward_sample[i] + gamma * np.max(next_q_v

        # train the model on the states and updated Q-values
        with tf.GradientTape() as tape:
            # compute the current Q-values for the samples
            state_sample_tensor = tf.convert_to_tensor(state_sample)
            current_q_values = model(state_sample_tensor, training=True)
            current_q_value = tf.gather_nd(current_q_values, tf.stack((tf.range

            # compute the loss
            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 [115…

```
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 [118…

```
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 0.2733778953552246
STEP 11000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.020653244107961655
STEP 11500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.03716724365949631
STEP 12000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.011714354157447815
STEP 12500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.027480322867631912
STEP 13000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.00568801024928689
STEP 13500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.014949658885598183
STEP 14000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.00806401763111353
STEP 14500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.008995527401566505
STEP 15000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.010906706564128399
STEP 15500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.21919341385364532
STEP 16000 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.013908205553889275
STEP 16500 | PHASE train | ACTION 1 | REWARD 0.1 | LOSS 0.009751500561833382
STEP 17000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.03614285588264465
STEP 17500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.04356595501303673
STEP 18000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.037368278950452805
STEP 18500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.3424623906612396
STEP 19000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.0355326347053051
STEP 19500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.24020010232925415
STEP 20000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.03712499514222145
STEP 20500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.03421327471733093
STEP 21000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.033225446939468384
STEP 21500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.03911549597978592
STEP 22000 | PHASE train | ACTION 1 | REWARD -1 | LOSS 0.039628930389881134
STEP 22500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.11595617234706879
STEP 23000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.08169691264629364
STEP 23500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.10373103618621826
STEP 24000 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.08186157792806625
STEP 24500 | PHASE train | ACTION 0 | REWARD 0.1 | LOSS 0.09620735049247742
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[118], line 1
----> 1 playgame()

Cell In[115], line 17, in playgame(start_from_ckpt, ckpt_path)
     14 loss_function = keras.losses.MeanSquaredError()
     16 # play the game
---> 17 dql_flappy_bird(model=model, optimizer=optimizer, loss_function=loss_functio
n)

Cell In[114], line 67, in dql_flappy_bird(model, optimizer, loss_function)
     65         next_state_tensor = tf.convert_to_tensor(state_next_sample[i])
     66         next_state_tensor = tf.expand_dims(next_state_tensor, 0)
---> 67         next_q_value = model(next_state_tensor, training=True)
     68         updated_q_value[i] = reward_sample[i] + gamma * np.max(next_q_value)
     70 # train the model on the states and updated Q-values

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/utils/traceback
_utils.py:64, in filter_traceback.<locals>.error_handler(*args, **kwargs)
     62 filtered_tb = None
     63 try:
---> 64   return fn(*args, **kwargs)
     65 except Exception as e:  # pylint: disable=broad-except
     66   filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/engine/trainin
g.py:490, in Model.__call__(self, *args, **kwargs)
    486     super().__call__(inputs, *copied_args, **copied_kwargs)
    488   layout_map_lib._map_subclass_model_variable(self, self._layout_map)
--> 490 return super().__call__(*args, **kwargs)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/utils/traceback
_utils.py:64, in filter_traceback.<locals>.error_handler(*args, **kwargs)
     62 filtered_tb = None
     63 try:
---> 64   return fn(*args, **kwargs)
     65 except Exception as e:  # pylint: disable=broad-except
     66   filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/engine/base_lay
er.py:1014, in Layer.__call__(self, *args, **kwargs)
   1010   inputs = self._maybe_cast_inputs(inputs, input_list)
   1012 with autocast_variable.enable_auto_cast_variables(
   1013     self._compute_dtype_object):
-> 1014   outputs = call_fn(inputs, *args, **kwargs)
   1016 if self._activity_regularizer:
   1017   self._handle_activity_regularization(inputs, outputs)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/utils/traceback
_utils.py:92, in inject_argument_info_in_traceback.<locals>.error_handler(*args, **k
wargs)
     90 bound_signature = None
     91 try:
---> 92   return fn(*args, **kwargs)
     93 except Exception as e:  # pylint: disable=broad-except
```

```
     94   if hasattr(e, '_keras_call_info_injected'):
     95     # Only inject info for the innermost failing call

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/engine/function
al.py:458, in Functional.call(self, inputs, training, mask)
    439 @doc_controls.do_not_doc_inheritable
    440 def call(self, inputs, training=None, mask=None):
    441   """Calls the model on new inputs.
    442
    443   In this case `call` just reapplies
   (...)
    456       a list of tensors if there are more than one outputs.
    457   """
--> 458   return self._run_internal_graph(
    459       inputs, training=training, mask=mask)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/engine/function
al.py:596, in Functional._run_internal_graph(self, inputs, training, mask)
    593   continue  # Node is not computable, try skipping.
    595 args, kwargs = node.map_arguments(tensor_dict)
--> 596 outputs = node.layer(*args, **kwargs)
    598 # Update tensor_dict.
    599 for x_id, y in zip(node.flat_output_ids, tf.nest.flatten(outputs)):

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/utils/traceback
_utils.py:64, in filter_traceback.<locals>.error_handler(*args, **kwargs)
     62 filtered_tb = None
     63 try:
---> 64   return fn(*args, **kwargs)
     65 except Exception as e:  # pylint: disable=broad-except
     66   filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/engine/base_lay
er.py:1014, in Layer.__call__(self, *args, **kwargs)
   1010   inputs = self._maybe_cast_inputs(inputs, input_list)
   1012 with autocast_variable.enable_auto_cast_variables(
   1013     self._compute_dtype_object):
-> 1014   outputs = call_fn(inputs, *args, **kwargs)
   1016 if self._activity_regularizer:
   1017   self._handle_activity_regularization(inputs, outputs)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/utils/traceback
_utils.py:92, in inject_argument_info_in_traceback.<locals>.error_handler(*args, **k
wargs)
     90 bound_signature = None
     91 try:
---> 92   return fn(*args, **kwargs)
     93 except Exception as e:  # pylint: disable=broad-except
     94   if hasattr(e, '_keras_call_info_injected'):
     95     # Only inject info for the innermost failing call

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/keras/layers/convolut
ional/base_conv.py:269, in Conv.call(self, inputs)
    266       outputs = conv_utils.squeeze_batch_dims(
    267           outputs, _apply_fn, inner_rank=self.rank + 1)
    268     else:
```

```
--> 269       outputs = tf.nn.bias_add(
    270           outputs, self.bias, data_format=self._tf_data_format)
    272 if not tf.executing_eagerly():
    273   # Infer the static output shape:
    274   out_shape = self.compute_output_shape(input_shape)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/tensorflow/python/util/traceback_utils.py:150, in filter_traceback.<locals>.error_handler(*args, **kwargs)
    148 filtered_tb = None
    149 try:
--> 150   return fn(*args, **kwargs)
    151 except Exception as e:
    152   filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/lpd/classes/SDS365/IMLvenv/lib/python3.10/site-packages/tensorflow/python/util/dispatch.py:1076, in add_dispatch_support.<locals>.decorator.<locals>.op_dispatch_handler(*args, **kwargs)
   1074 if iterable_params is not None:
   1075   args, kwargs = replace_iterable_params(args, kwargs, iterable_params)
-> 1076 result = api_dispatcher.Dispatch(args, kwargs)
   1077 if result is not NotImplemented:
   1078   return result

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.

In the early stage of training (within 2,000 steps in the explore phase), the flappy bird does not ever get past the first tube. The policy was randomly initialized and it this prelimary state seems to produce a flap action so often that the bird almost always colides with the top of the first tube and dies. After 5000 training steps, the bird is doing better. It many times passes the first tube rather than always dieing before it. As the bird explores, some actions it randomly takes like going through the tube opening, will lead to a larger reward. That becomes part of the replay memory, so that the parameters get trained through gradient descent to make the behaviour that led to that reward happen more often. As the sprite more often gets further, new situations are added to the buffer, but old situations aren't totally forgotten, as there is a buffer. This means the sprite is trained to do well throughout

the start and ongoing phases of game play.

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.