**Report on the Architecture and Hyperparameters**

### ****1. Overview****

The provided code implements a Deep Q-Network (DQN) for training an agent to play Flappy Bird using reinforcement learning. The architecture leverages convolutional and fully connected layers to process frames and predict actions. Key components include a replay buffer, policy/target networks, and epsilon-greedy exploration.

### ****2. DQN Architecture****

#### ****2.1 Network Structure****

The neural network is defined in the DQN class, which consists of convolutional layers for feature extraction and fully connected layers for action value prediction.

* **Convolutional Layers:**
  + Layer 1: nn.Conv2d(1, 32, kernel\_size=8, stride=4), followed by ReLU
  + Layer 2: nn.Conv2d(32, 64, kernel\_size=4, stride=2), followed by ReLU
  + Flattening the output
* **Fully Connected Layers:**
  + Layer 1: nn.Linear(conv\_out\_size, 512), followed by ReLU
  + Layer 2: nn.Linear(512, 512), followed by ReLU
  + Layer 3: nn.Linear(512, n\_actions)

#### ****2.2 Input and Output****

* **Input:** Preprocessed frames of size (64, 64) passed as a single channel grayscale image.
* **Output:** Q-values for all possible actions (e.g., n\_actions = 2 for Flappy Bird).

### ****3. Replay Buffer****

The ReplayBuffer class implements experience replay to store and sample transitions:

* **Capacity:** 100,000 transitions
* **Stored Data:** (state, action, reward, next\_state, done)
* **Batch Sampling:** Mini-batches of size 64 are sampled for training.

### ****4. Hyperparameters****

#### ****4.1 Neural Network Hyperparameters****

* **Convolutional Layers:**
  + Layer 1: Filters = 32, Kernel Size = 8, Stride = 4
  + Layer 2: Filters = 64, Kernel Size = 4, Stride = 2
* **Fully Connected Layers:**
  + Layer 1: 512 neurons
  + Layer 2: 512 neurons

#### ****4.2 Training Hyperparameters****

* **Learning Rate:** 0.00025 (Adam optimizer)
* **Batch Size:** 64
* **Replay Buffer Size:** 100,000
* **Discount Factor (γ):** 0.99
* **Epsilon Decay:** 0.9997
* **Minimum Epsilon:** 0.0001
* **Target Network Update Interval:** 1,000 steps

#### ****4.3 Exploration (Epsilon-Greedy):****

* **Initial Epsilon:** 1.0
* **Decay Rate:** 0.9997 per step
* **Minimum Epsilon:** 0.0001

### ****5. Preprocessing and State Representation****

#### ****5.1 Frame Preprocessing****

The preprocess\_frame function processes raw game frames:

* **Cropping:** Removes irrelevant portions of the frame.
* **Color Conversion:** Converts RGB to HSV for masking.
* **Object Masking:** Creates a binary mask for relevant objects.
* **Resizing:** Scales the image to (64, 64).
* **Normalization:** Scales pixel values to [0, 1].

#### ****5.2 State Shape:**** (64, 64) (grayscale image with 1 channel)

### ****6. Training Logic****

#### ****6.1 Reward Shaping****

Rewards are shaped to encourage survival and penalize termination:

* -1: Episode termination
* 1.0: Positive score event
* 0.1: Survival

#### ****6.2 Training Schedule****

* Training occurs every step if sufficient transitions are in the buffer.
* Policy network parameters are updated via backpropagation.
* Target network parameters are updated every 1,000 steps.

#### ****6.3 Checkpointing and Logging****

* **Model Checkpoints:** Saved every 200 episodes or when the best reward improves.
* **Training Logs:** Episode number, epsilon value, and average rewards are logged.

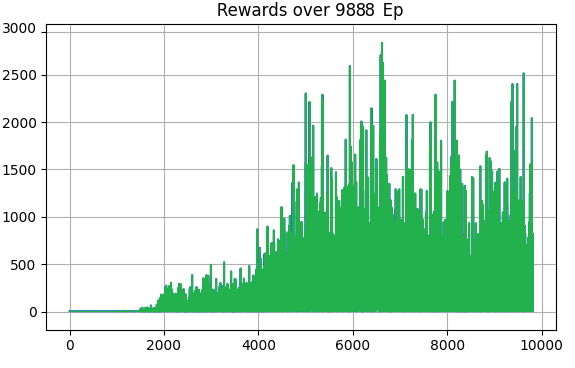
**7. Training Process:**

* **Experience Replay:** The agent stores experiences in a replay buffer and samples mini-batches to train the network. This helps break correlations between consecutive frames.
* **Q-learning Update:** The Q-values are updated using the Bellman equation, where the next state's maximum Q-value is used to estimate the expected Q-value.
* **Epsilon Decay:** The agent starts by exploring with a high epsilon value and gradually shifts towards exploiting learned behavior as epsilon decays.
* **Model Checkpoints:** Every 200 episodes, a checkpoint is saved. Additionally, when a higher reward is achieved, the model is saved as the best model. If the agent achieves a reward threshold of 5000, training stops, and the final model is saved.

**8. Model Saving and Plotting:**

* **Model Checkpoints:** At regular intervals, the agent's policy network is saved to a checkpoint file, which can be used to resume training.
* **Plotting:** Training statistics, including episode rewards and epsilon decay, are plotted after every 10 episodes and saved to a file for visualization.

**9. Overall Results  
  
Our best model has gotten the following results**



**Experimentation and Adjustments in Hyperparameters and Architectures**

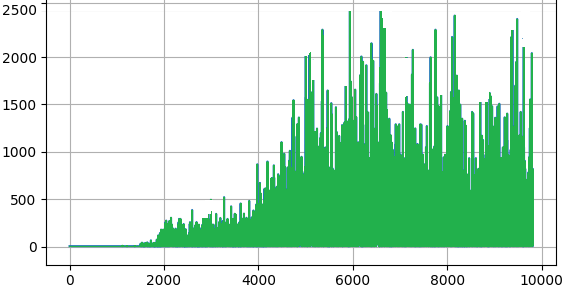
**Hyperparameter Exploration**

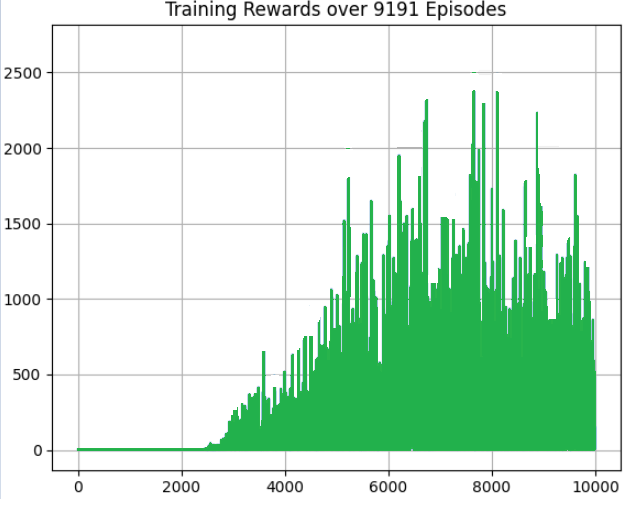
Throughout the training of our DQN-based agent, we experimented with various hyperparameter configurations to optimize performance. These included different settings for the epsilon-greedy strategy, batch size, discount factor (γ), learning rate, and target network update frequency. Below are some notable configurations tested:

* **Configuration 1:**
  + epsilon\_min = 0.00015
  + epsilon\_decay = 0.9996
  + target\_update = 950
  + steps = 5
  + batch\_size = 128
  + gamma = 0.985
  + epsilon = 0.95
* **Configuration 2:**
  + epsilon\_min = 0.00012
  + epsilon\_decay = 0.9998
  + target\_update = 1100
  + steps = 20
  + batch\_size = 256
  + gamma = 0.97
  + epsilon = 0.85
* **Configuration 3:**
  + epsilon\_min = 0.00025
  + epsilon\_decay = 0.9994
  + target\_update = 1050
  + steps = 15
  + batch\_size = 64
  + gamma = 0.995
  + epsilon = 1.1
* **Configuration 4:**
  + epsilon\_min = 0.0002
  + epsilon\_decay = 0.9999
  + target\_update = 1150
  + steps = 0
  + batch\_size = 48
  + gamma = 0.975
  + epsilon = 0.8
* **Configuration 5:**
  + epsilon\_min = 0.00018
  + epsilon\_decay = 0.9993
  + target\_update = 1025
  + steps = 8
  + batch\_size = 96
  + gamma = 0.99
  + epsilon = 0.92
* **Configuration 6:**
  + epsilon\_min = 0.0003
  + epsilon\_decay = 0.9995
  + target\_update = 1300
  + steps = 12
  + batch\_size = 128
  + gamma = 0.98
  + epsilon = 1.05

The results showed that while some configurations slightly improved performance, many produced either similar or lower results, emphasizing the sensitivity of the DQN algorithm to hyperparameter tuning.

With these tests we’ve got similar results to our best model in the best cases, such as:





#### Architecture Variants

We also explored different neural network architectures for our DQN agent, including variations in the number of layers, activation functions, and regularization techniques:

1. **Fully Connected Architectures:**
   * **4 layers:**
     + Nodes: [1024, 512, 256, 128]
     + Dropout: 0.2
     + Learning Rate: 0.001
     + Batch Sizes: 64 or 128
     + Activation: LeakyReLU
     + Optimizer: Adam
     + Additional Techniques: Learning rate scheduling, data augmentation (e.g., rotation)

 **Convolutional Architectures:**

* **Architecture 1:**

self.conv1 = nn.Conv2d(4, 32, kernel\_size=8, stride=4)

self.conv2 = nn.Conv2d(32, 64, kernel\_size=4, stride=2)

self.conv3 = nn.Conv2d(64, 64, kernel\_size=3, stride=1)

self.fc1 = nn.Linear(self.conv\_out\_size, 512)

self.fc2 = nn.Linear(512, num\_actions)

With ReLU activations:

x = torch.relu(self.conv1(x))

x = torch.relu(self.conv2(x))

x = torch.relu(self.conv3(x))

x = x.reshape(x.size(0), -1)

x = torch.relu(self.fc1(x))

x = self.fc2(x)

**Architecture 2:**

self.conv = nn.Sequential(

nn.Conv2d(input\_channels, 32, kernel\_size=8, stride=4, padding=0),

nn.LeakyReLU(),

nn.Conv2d(32, 64, kernel\_size=4, stride=2, padding=0),

nn.LeakyReLU(),

nn.Conv2d(64, 64, kernel\_size=3, stride=1, padding=0),

nn.LeakyReLU(),

)

self.fc = nn.Sequential(

nn.Linear(64 \* 7 \* 7, 512),

nn.LeakyReLU(),

nn.Linear(512, action\_dim),

)

#### Reward Adjustments

The reward function was another key focus of experimentation. Initially, the agent received:

* **Positive Reward:** +1 for passing a pipe.
* **Small Positive Reward:** +0.1 for staying alive.
* **Negative Reward:** -1 for crashing.

We modified these rewards to:

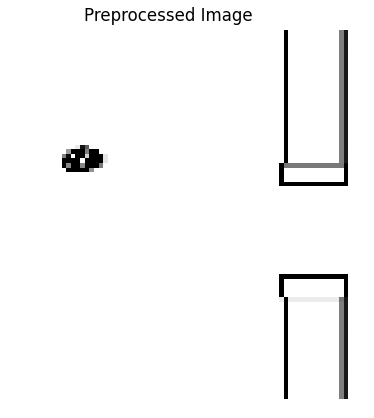
* Increase the reward for passing pipes to **+5**.
* Increase the reward for staying alive to **+0.2**.

### ****Initial Preprocessing Approach:****

The original preprocessing function aimed to perform several operations:

* **Crop and Grayscale Conversion:** The code first removed a portion of the image (observation[:-108, :, :]), likely to exclude unnecessary parts of the environment. The image was then converted to grayscale using OpenCV's cvtColor function.
* **Thresholding and Inversion:** After converting the image to grayscale, a binary threshold was applied to highlight the features of interest (using cv2.threshold). The image was then inverted with bitwise\_not to flip the black and white values.
* **Dilation:** A kernel was used to dilate the image, thickening certain elements to make them more noticeable. This can help emphasize objects of interest, such as pipes or the bird in the game.
* **Resizing:** Finally, the image was resized to 80x80 pixels, which is a common step to standardize the input size for the neural network.

This approach seems to have been an attempt to enhance object visibility (e.g., the bird, pipes) while reducing irrelevant background noise. However, it has been commented out, likely due to issues or experiments with alternative methods.



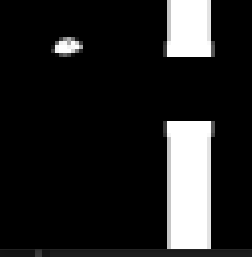
### 2. ****Simplified Grayscale and Resizing:****

The second approach, which is simpler, converted the frame into grayscale using Image.fromarray(frame).convert('L'), resized it to 84x84 pixels, and then normalized the pixel values by dividing by 255. This approach is very common in reinforcement learning tasks to reduce the complexity of the input and standardize it for processing by the neural network. By normalizing the pixel values, the model can better learn patterns without being affected by the original pixel intensity.

### 3. ****Current Preprocessing Approach:****

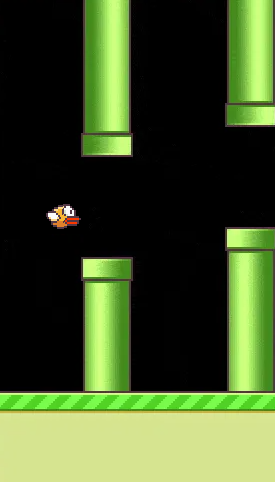
The most recent method involves a more sophisticated color-based background isolation technique, designed to separate the objects (bird and pipes) from the background:

* **Frame Cropping:** The frame is cropped to keep only the relevant top portion (frame[:400, :]), possibly to focus on the immediate area where the bird and pipes appear, leaving out irrelevant regions such as the ground.
* **Color Space Transformation (HSV):** The frame is then converted from RGB to HSV (Hue, Saturation, Value) color space using cv2.cvtColor. HSV is often preferred over RGB in image processing because it separates the chromatic content (color) from intensity (brightness), making it easier to filter specific colors.
* **Masking and Color Filtering:** Several masks are created using the cv2.inRange() function to isolate specific colors:
  + One mask captures the color of the pipes (green).
  + Others capture potential areas of interest, such as the background or the bird's color, based on predefined color ranges in the HSV space.
* **Background Isolation:** The background is isolated by combining the color masks with bitwise OR operations. The areas that correspond to the background are then subtracted, leaving only the objects (bird and pipes). This process effectively separates the agent's environment from the background distractions.
* **Result Construction and Normalization:** After creating the mask, the result is resized to 64x64 pixels and normalized by dividing by 255. This normalization ensures that the model receives input in a consistent range of values between 0 and 1, which helps with training stability.



### 4. ****De-Isolating the Background:****

As part of the background separation strategy, the function effectively isolates the objects (such as the bird and pipes) while removing the background. This allows the agent to focus on the important visual cues for its decision-making, which is crucial in environments like Flappy Bird, where the agent needs to focus on the position of the bird relative to obstacles.



These changes were aimed at incentivizing survival and strategic pipe navigation. However, these adjustments had limited impact on overall performance, as the agent still required careful tuning of other hyperparameters to achieve stable learning.