Artificial Intelligence Final Project
On

Using Deep Reinforcement Learning on How To Play Flappy Bird.

Submitted By
Siva Rama Krishna Kambuluri
Pranay Reddy Muthyala

Under the Guidance of
Vahid Behzadan (Assistant Professor)

UNIVERSITY OF NEW HAVEN
1920
CONNECTICUT

Project Objective:-

The project goal is to use Deep Q Learning Algorithm, which is applied to Atari 2600 games, this can be generalized to Flappy Bird game.

In this Project, we show that deep reinforcement learning is very effective at learning how to play game flappy bird. Here the agent is not given information about what the bird or pipes look like, it must learn these representations and directly use the input and score to develop an optimal strategy. Our agent uses a convolutional neural network to evaluate the Q-function for a variant of Q- learning.

Flappy Bird is a game in which the player tries to keep the bird alive as long as possible. The bird automatically falls towards the ground by due to gravity, and if it hits the ground, it dies and the game ends. The bird must also navigate through pipes. The pipes restrict the height of the bird to be within a range as the bird to pass through them. If the bird is too high or too low, it will crash into the pipe and die. Therefore, the player must time jump's properly to keep the bird alive as it passes through these obstacles. The game score is measured by how many obstacles the bird successfully passes through. Therefore, to get a high score, the player must keep the bird alive for as long as possible as it encounters the pipes.

Training an agent to successfully play the game is especially challenging because our goal is to provide the agent with only pixel information and the score. The agent is not provided with information regarding what the bird looks like, what the pipes look like, or where the bird and pipes are. Instead, it must learn these representations and interactions and be able to generalize due to the very large state space.

Installation Dependencies/ Tools and technologies used:

- Python
- TensorFlow
- Pygame
- OpenCV-python

Background:

Reinforcement learning develops control patterns by providing feedback on a model's selected actions, which encourages the model to select better actions in the future. At each time step, given some state's, the model will select an action a, and then observe the new state's and a reward r based on some optimality criterion.

We specifically used a method known as Q learning, which approximates the maximum expected return for performing an action at a given state using an action-value (Q) function. Specifically, return gives the sum of the rewards until the game terminates, where the reward is discounted by a factor of γ at each time step. We formally define this as:

$$R_t = \sum_{t'=1}^{T} \gamma^{t'-t} r_t. \quad (1)$$

We then define the action-value function:

$$Q^*(s, a) = E[R_t \mid s_t = s, a_t = a] \quad (2)$$

Note that if the optimal Q function is known for State's, we can write the optimal Q function at preceding state's as the maximum expected value of $r + \gamma Q^*(s, a`)$. This identity is known as Bellman equation.

$$Q^*(s,a) = E[r + \gamma * \max_{a'} Q^*(s', a') \mid s, a] \tag{3}$$

The intuition behind reinforcement learning is to continually update the action-value function based on observations using the Bellman equation. It has been shown by Sutton et al 1998 [2] that such update algorithms will converge on the optimal action-value function as time approaches infinity. Based on this, we can define Q as the output of a neural network, which has weights θ, and train this network by minimizing the following loss function at each iteration i:

$$L_i(\theta_i) = E[(y_i - Q(s, a; \theta_i))^2] \tag{4}$$

Where y_i represents the target function we want to approach during each iteration. It is defined as:

$$y_i = E[r + \gamma * \max_{a'} Q(s', a';\ \theta_{i-1}) \mid s, a] \tag{5}$$

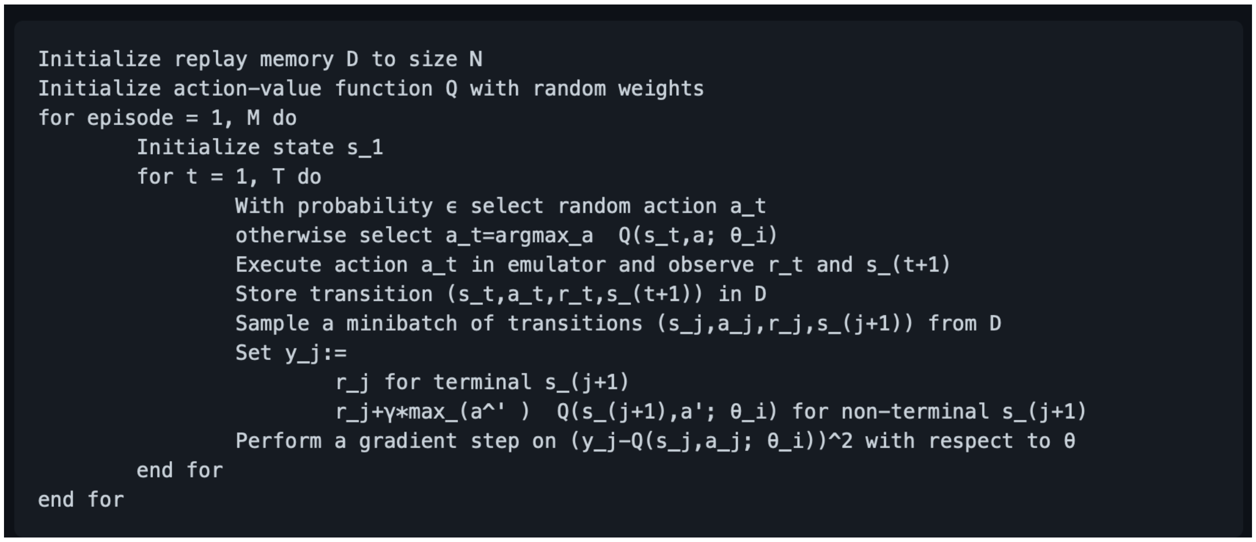
Note that when i is equal to the final iteration of an episode (colloquially the end of a game), the Q function should be 0 since it is impossible to attain additional reward after the game has ended. Therefore, when i equals the terminal frame of an episode, we can simply write:

$$y_i = E[r \mid s, a] \tag{6}$$

What is Deep Q-Network?

It is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards.

Deep Q Learning Algorithm:



```
Initialize replay memory D to size N
Initialize action-value function Q with random weights
for episode = 1, M do
        Initialize state s_1
        for t = 1, T do
                With probability ϵ select random action a_t
                otherwise select a_t=argmax_a  Q(s_t,a; θ_i)
                Execute action a_t in emulator and observe r_t and s_(t+1)
                Store transition (s_t,a_t,r_t,s_(t+1)) in D
                Sample a minibatch of transitions (s_j,a_j,r_j,s_(j+1)) from D
                Set y_j:=
                        r_j for terminal s_(j+1)
                        r_j+γ*max_(a^' )  Q(s_(j+1),a'; θ_i) for non-terminal s_(j+1)
                Perform a gradient step on (y_j-Q(s_j,a_j; θ_i))^2 with respect to θ
        end for
end for
```

Experimental Methods:

Network Architecture

We first preprocessed the game screens with following steps:

1. Convert image to grayscale
2. Resize image to 80x80
3. Stack last 4 frames to produce an 80x80x4 input array for network

The architecture of the network is shown in the figure below. The first layer convolves the input image with an 8x8x4x32 kernel at a stride size of 4. The output is then put through a 2x2 max pooling layer. The second layer convolves with a 4x4x32x64 kernel at a stride of 2. We then max pool again. The third layer convolves with a 3x3x64x64 kernel at a stride of 1. We then max pool one more time. The last hidden layer consists of 256 fully connected ReLU nodes.

The final output layer has the same dimensionality as the number of valid actions which can be performed in the game, where the 0th index always corresponds to doing nothing. The values at this output layer represent the Q function given the input state for each valid action. At each time step, the network performs whichever action corresponds to the highest Q value using a ϵ greedy policy.

Training:

At first, we initialize all weight matrices randomly using a normal distribution with a standard deviation of 0.01, the set the replay memory with a max size of 500,000 experiences. we started training by choosing actions uniformly at random for the first 10,000 times steps, without updating the network weights. This allows the system to populate the replay memory before the training begins.

Results:

See the link for videos of the DQN in action:
https://www.youtube.com/watch?v=D2tyW0NcvQE

we found that for flappy bird the good results can be achieved with more training for sure. We took like 16 hours of game time, Qualitatively the network started to play at a decent level. This indicates that improvement as it means that the network is expecting to receive a greater reward per game as it trains for longer. I think as we continue to train the network, the value of the maximum Q value should plateau as the network reaches as optimal state.

References:

1. Flappy Bird game: https://github.com/sourabhv/FlapPyBird
2. Deep Learning on videogames: https://github.com/asrivat1/DeepLearningVideoGames