Motivation

Many recent publications have explored the use of reinforcement learning for Atari games, achieving comparable performance for many games, and superhuman performance for a select few including PongMnih et al. [2015], He et al. [2016], Guo et al. [2014], Silver et al. [2016], Mnih et al. [2015]. All of these algorithms utilized a convolutional neural network approach based on the pixel input of the game image. In this work we would like to focus on Pong, and see how the learning performance will be affected if the state space is instead composed of the positions and velocities of the game objects. Our hypothesis is that the richer state space of the game image will lead to higher performance. However, our revised state space is of interest because of its ability to learn policies based on the game objects, as humans do.

Problem Definition

The state, action, and reward definitions are below. The state space is composed of the y positions and y velocities of paddles one and two, and the x and y position and velocity of the ball. The action space is a discrete space of dimension three. The learning agent trains the control for paddle two, and plays against the default AI of paddle one, which is based on the position of the ball. The reward is one when paddle two hits the ball.

$$S = \left\{ \begin{array}{cccccccc} P_{1,y_{pos}} & P_{1,vel} & P_{2,y_{pos}} & P_{2,vel} & B_{x_{pos}} & B_{y_{pos}} & B_{x_{vel}} & B_{y_{vel}} \end{array} \right\} \quad \text{R} = \left\{ \begin{array}{cc} 1 & B_{x_{pos}} = P_{x_{pos}}, B_{y_{pos}} \in P_{1,y_{range}} \\ 0 & \text{else} \end{array} \right\}$$

As stated above, the action space is discrete of dimension three, and controls the speed of paddle two. The paddle either stays still, goes down, or goes up, if A is zero, one, or two respectively. Paddle one either stays still if it is at the same position as the ball, goes down if the ball is below it, or goes up if the ball is above it.

$$A \in \{0,1,2\} \quad P_{1,y_{vel}} = \left\{ \begin{array}{cc} 0 & A=0 \\ -P_{1,vel_{max}} & A=1 \\ P_{1,y_{vel_{max}}} & A=2 \end{array} \right\} \quad P_{2,y_{vel}} = \left\{ \begin{array}{cc} 0 & B_{x_{pos}} = P_{2,y_{pos}} \\ -P_{2,y_{vel_{max}}} & B_{x_{pos}} < P_{2,y_{pos}} \\ P_{2,y_{vel_{max}}} & B_{x_{pos}} > P_{2,y_{pos}} \end{array} \right\}$$

Proposed Experiments

In this work, we will test two hypothesis. The first is that the reduced dimension of the modified state space decreases the performance. The second is that the speed of the second paddle affects the performance, as faster velocities in the opponent paddle will lead to reduced reward for suboptimal policies. These are tested in the following experiments

1. Experiment 1: We will benchmark the performance of our custom environment against Pong-V0 in OpenAI gym, using the DQN algorithm from Keras RL.
2. Experiment 2: We will record the performance of our algorithm for multiple speeds of the second paddle, and test our algorithms robustness to this parameter variation.

1 Justification for RL Approach

1.1 Classic Pong Formulation

In the above Pong Formulation, where the positions and velocities of the game object are used, the cardinality of the state space is given by the following equation.

$$|S_{\text{classic}}| = \left[\text{Max}\left(P_{1,pos}\right)\right] \cdot \left[\text{Max}\left(P_{2,pos}\right)\right] \cdot \left[Max\left(B_{x,pos}\right)\right] \cdot \left[\text{Max}\left(B_{y,pos}\right)\right]$$
$$\cdot \left[2 \cdot \text{Max}\left(P_{1,vel}\right)\right] \cdot \left[2 \cdot \text{Max}\left(P_{2,vel}\right)\right] \cdot \left[2 \cdot \text{Max}\left(B_{x,vel}\right)\right] \cdot \left[2 \cdot \text{Max}\left(B_{y,vel}\right)\right]$$

We now substitute the parameters for our experiment, in which the screen is 400×600 pixels and the maximum velocity is 20 pixels per second for all objects.

$$|S_{\text{classic}}| = [400Px.] \cdot [400Px.] \cdot [600Px.] \cdot [400Px.] \cdot \left[2 \cdot 20\frac{Px.}{\text{sec}}\right] \cdot \left[2 \cdot 20\frac{Px.}{\text{sec}}\right] \cdot \left[2 \cdot 20\frac{Px.}{\text{sec}}\right] \cdot \left[2 \cdot 20\frac{Px.}{\text{sec}}\right]$$

The cardinality of the transition matrix is the square of the cardinality of the state matrix.

$$|P_{classic}| = |S_{classic}|^2$$

The enormous cardinality of the transition matrix necessitates the use of Reinforcement Learning for this problem.

1.2 Convolutional Pong Formulation

In the Convolutional Pong Formulation used in previous work, where the pixel input is used, we can see the cardinality of the state space is much larger. The problem is framed in an identical fashion, except for the state and reward definitions which are given below.

$$S' = \left\{ \ R_{x\in X, y\in Y} \quad G_{x\in X, y\in Y} \quad B_{x\in X, y\in Y} \ \right\} \quad \text{R}' = \left\{P_{2,score} - P_{1,score}\right\}$$

The cardinality of the state space becomes the following

$$|S_{\text{convolutional}}| = \left[\text{Max}\left(X_{\text{pos}}\right)\right] \cdot \left[\text{Max}\left(Y_{\text{pos}}\right)\right] \cdot \left[\text{Max}\left(R_{val}\right)\right] \cdot \left[\text{Max}\left(G_{val}\right)\right] \cdot \left[Max\left(B_{val}\right)\right]$$

We now substitute the parameters for this experiment, in which the screen is 400×600 pixels and the RGB space is composed of 255 voxels.

$$|S_{\text{convolutional}}| = [400Px.] \cdot [600Px.] \cdot [255Voxels] \cdot [255Voxels] \cdot [255Voxels]$$

The cardinality of the transition matrix is the square of the cardinality of the state matrix.

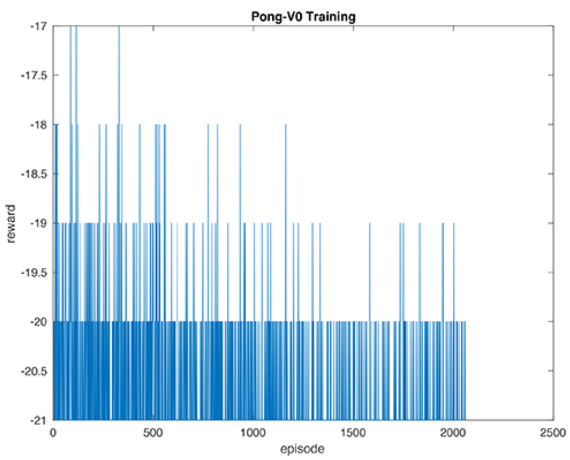
$$|P_{\text{convolutional}}| = |S_{\text{convolutional}}|^2$$

The enormous cardinality of the transition matrix necessitates the use of Reinforcement Learning for this problem as well.

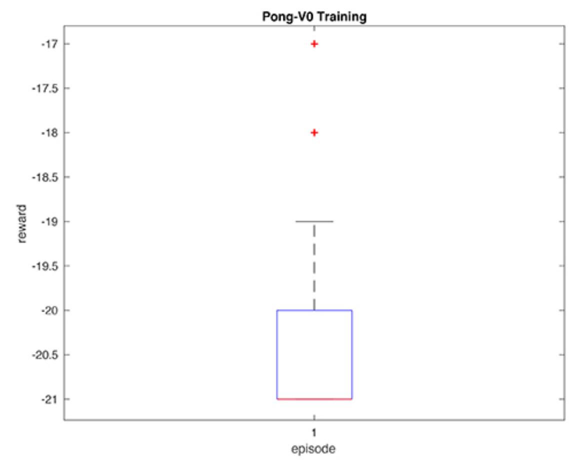
Results

Experiment 1

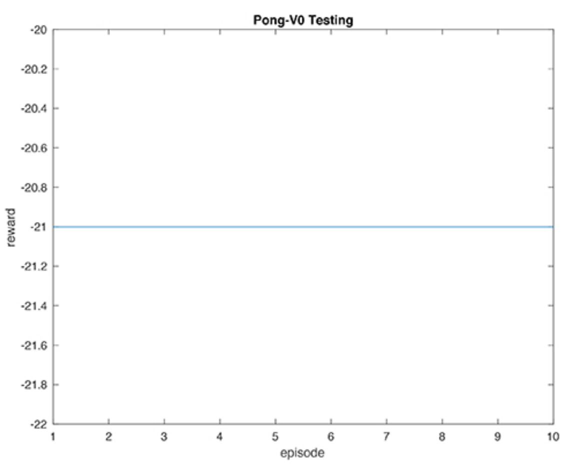
The baseline, the Pong-V0 environment, from Open AI Gym, is trained with the DQG Atari algorithm from Keras RL (Appendix A). Figure 1 shows plots and box plots for both the training and testing phases. The simulation was run for 170000 steps (pixel updates) for twenty-four hours using the CPU version of Tensor Flow on a 2016 MacBook Pro with a 2.9GHz Intel i7 processor, before being tested for ten episodes. It can be observed from Figure 1 that this simulation, which was trained with a reward of the difference in score between the two paddles and a state input of the game image, fails completely even after the twenty-four hour training period.



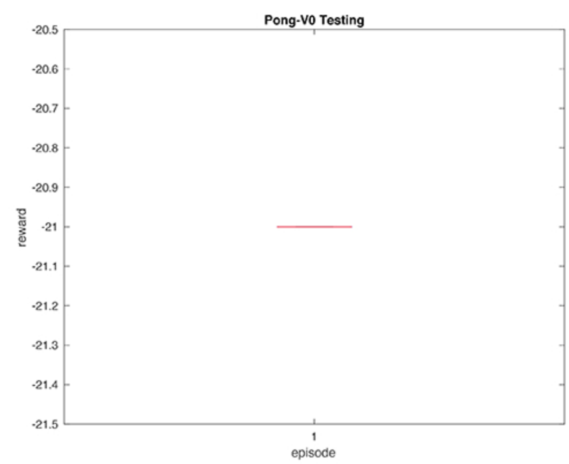
(a) Training Plot



(b) Training Box Plot



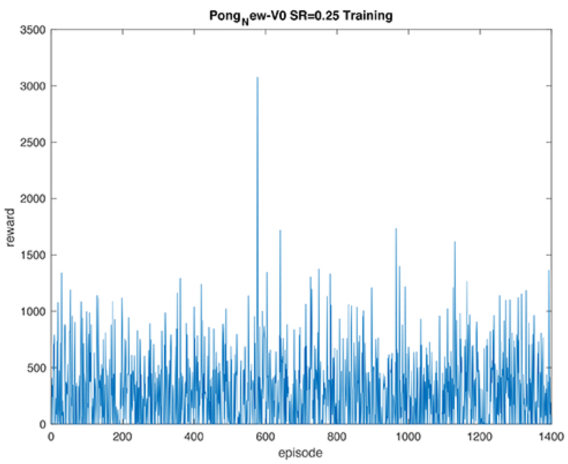
(c) Test Plot



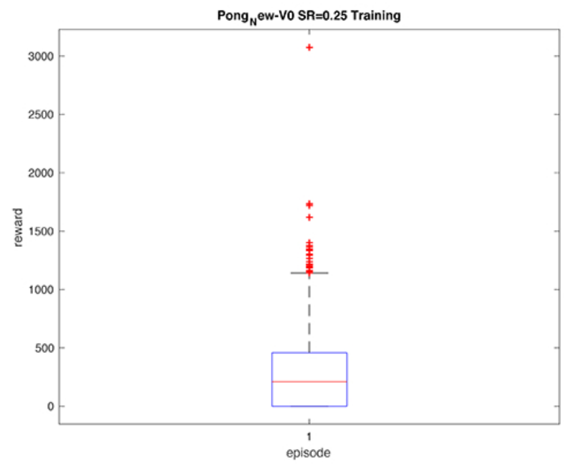
(d) Test Box Plot

Figure 1: Pong-V0

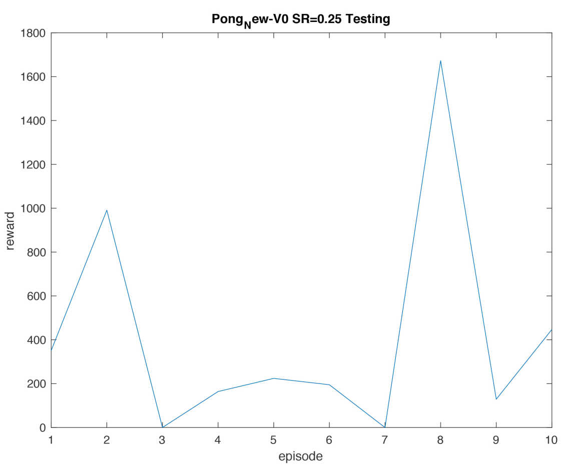
Our environment, Pong New - V0 (Appendix B), which uses the classic control framework from Open AI Gym, was trained with the Classic Control DQN Algorithm from Keras RL (Appendix C). We decreased the simulation time by a factor of ten, running the simulation for 175000 steps over a period of three hours on the same hardware system before testing for ten episodes. This revised learning algorithm, which learns a defensive strategy where the reward is given for hitting the ball, succeeds eighty percent of the time after only training for three hours, with the positions and velocities of the game objects as state input. During this experiment, the ratio of the maximum speed of the paddle we train to the one we compete against is four to one.



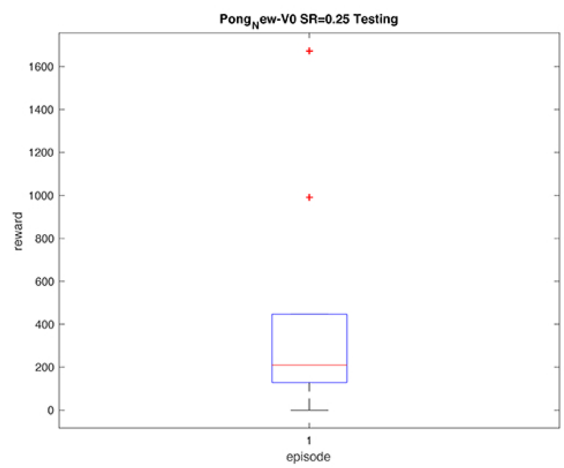
(a) Training Plot



(b) Training Box Plot



(c) Test Plot

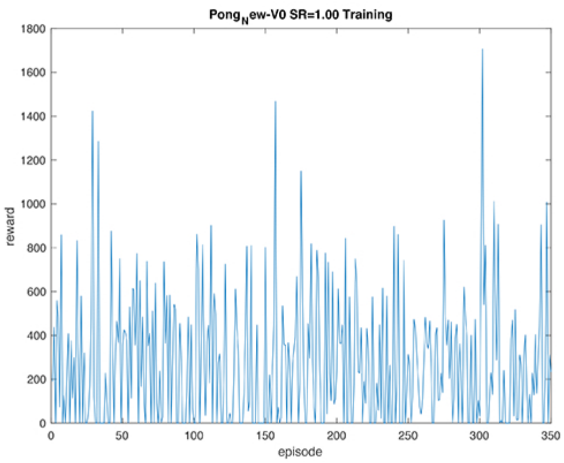


(d) Test Box Plot

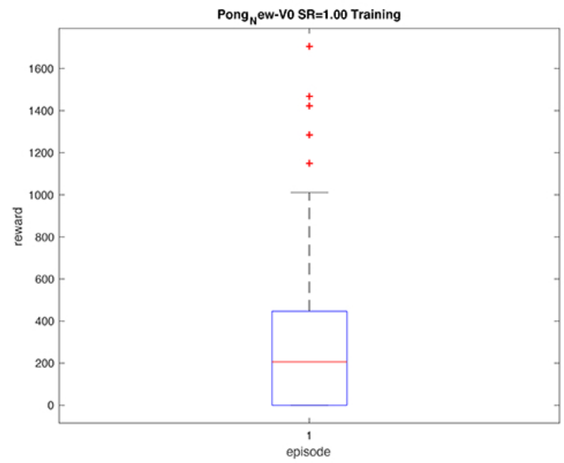
Figure 2: Pong New $- V0$ SR $= 0.25$

Experiment 2

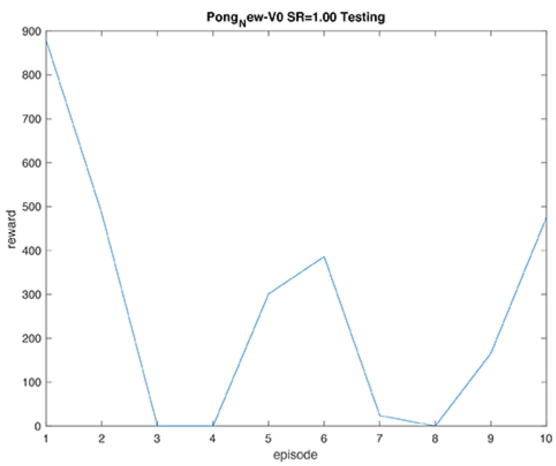
We now repeat the previous experiment on our Pong New - V0 environment, decreasing the ratio of the maximum speed of the paddle we train to the one we compete against to one to one. We observe seventy percent success rate for this experiment, showing the robustness of our environment to this parameter variation.



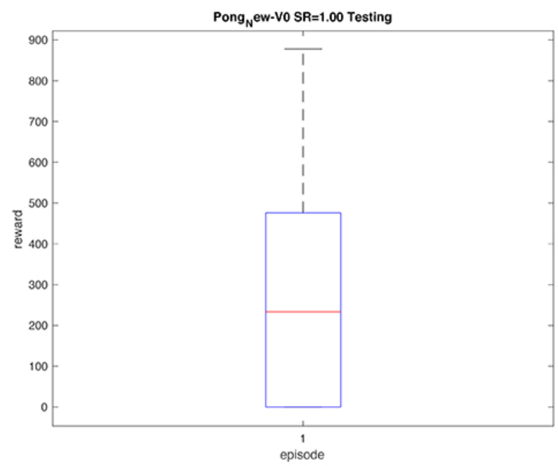
(a) Training Plot



(b) Training Box Plot



(c) Test Plot



(d) Test Box Plot

Figure 3: Pong New $- V0 \quad \mathrm{SR} = 1.00$

2 Utilized Environments

For the training environment, we utilized Open AI Gym Brockman et al. [2016]. For the DQN network, we utilized **Keras RL**, which implements the DQN algorithm utilized in Mnih et al. [2015] as part of its package of modern Deep RL algorithms built on the **Tensor Flow** training environment. Finally, for the Pong New - V0 environment, we created our new environment based on an existing python implementation of the classic **Pong Game** using **Pygames**, and our own previous implementation of **Pong in Java**.

3 Discussion

It is sensible that our approach trains faster, as the dimension of the state space and the one step return on our reward expedite training. The baseline Pong - V0, with its high dimensional image input, and a score based reward which will not give high return until a winning policy is found, cannot be expected to train as efficiently. Since all winning policies lie in the convex hull of all defensive policies, it is sensible to perform two-shot learning for the Pong environment, where the policy trained with our environment is used as a baseline for imitation learning in the original Pong - V0 environment. The state spaces of the two environments can be related to each other by either storing the game image data during training of our environment, or by modifying the Pong - V0 environment to first perform object classification via supervised learning as in Kulkarni et al. [2016], and then learn a policy based on the classified position and velocities of the game objects.

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

Frank S He, Yang Liu, Alexander G Schwing, and Jian Peng. Learning to play in a day: Faster deep reinforcement learning by optimality tightening. *arXiv preprint arXiv:1611.01606*, 2016.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 3675–3683, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Appendix

A: DQN Algorithm for Pong - V0: Atari DQN From Keras-RL

```python
from __future__ import division
import argparse
from PIL import Image
import numpy as np
import gym
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Convolution2D, Permute
from keras.optimizers import Adam
import keras.backend as K
from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, BoltzmannQPolicy, EpsGreedyQPolicy
from rl.memory import SequentialMemory
from rl.core import Processor
from rl.callbacks import FileLogger, ModelIntervalCheckpoint
import scipy
INPUT_SHAPE = (84, 84)
WINDOW_LENGTH = 4
class AtariProcessor(Processor):
    def process_observation(self, observation):
        assert observation.ndim == 3  # (height, width, channel)
        img = Image.fromarray(observation)
        img = img.resize(INPUT_SHAPE).convert('L')  # resize and convert to grayscale
        processed_observation = np.array(img)
        assert processed_observation.shape == INPUT_SHAPE
        return processed_observation.astype('uint8')  # saves storage in experience memory
    def process_state_batch(self, batch):
        # We could perform this processing step in `process_observation`. In this case, however,
        # we would need to store a `float32` array instead, which is 4x more memory intensive than
        # an `uint8` array. This matters if we store 1M observations.
        processed_batch = batch.astype('float32') / 255.
        return processed_batch
    def process_reward(self, reward):
        return np.clip(reward, -1., 1.)
parser = argparse.ArgumentParser()
parser.add_argument('--mode', choices=['train', 'test'], default='train')
parser.add_argument('--env-name', type=str, default='Pong-v0')
```

```python
parser.add_argument('--weights', type=str, default=None)
args = parser.parse_args()
# Get the environment and extract the number of actions.
env = gym.make(args.env_name)
env = env.unwrapped
np.random.seed(123)
env.seed(123)
nb_actions = env.action_space.n
def _step(a):
    reward = 0.0
    action = env._action_set[a]
    lives_before = env.ale.lives()
    for _ in range(4):
        reward += env.ale.act(action)
    ob = env._get_obs()
    done = env.ale.game_over() or (args.mode == 'train' and lives_before != env.ale.lives())
    return ob, reward, done, {}
env._step = _step
input_shape = (WINDOW_LENGTH,) + INPUT_SHAPE
model = Sequential()
if K.image_dim_ordering() == 'tf':
    model.add(Permute((2, 3, 1), input_shape=input_shape))
elif K.image_dim_ordering() == 'th':
    model.add(Permute((1, 2, 3), input_shape=input_shape))
else:
    raise RuntimeError('Unknown image_dim_ordering.')
model.add(Convolution2D(32, 8, 8, subsample=(4, 4)))
model.add(Activation('relu'))
model.add(Convolution2D(64, 4, 4, subsample=(2, 2)))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3, subsample=(1, 1)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
memory = SequentialMemory(limit=1000000, window_length=WINDOW_LENGTH)
processor = AtariProcessor()
policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, value_test=.05,
                              nb_steps=1000000)
dqn = DQNAgent(model=model, nb_actions=nb_actions, policy=policy, memory=memory,
```

```
                 processor=processor, nb_steps_warmup=50000, gamma=.99, target_model_update=10000,
                 train_interval=4, delta_clip=1.)
dqn.compile(Adam(lr=.00025), metrics=['mae'])
if args.mode == 'train':
    weights_filename = 'dqn_{}_weights.h5f'.format(args.env_name)
    checkpoint_weights_filename = 'dqn_' + args.env_name + '_weights_{step}.h5f'
    log_filename = 'dqn_{}_log.json'.format(args.env_name)
    callbacks = [ModelIntervalCheckpoint(checkpoint_weights_filename, interval=250000)]
    callbacks += [FileLogger(log_filename, interval=100)]
    history_0 = dqn.fit(env, callbacks=callbacks, nb_steps=1750000, log_interval=10000)
    dqn.save_weights(weights_filename, overwrite=True)
    history_1 = dqn.test(env, nb_episodes=10, visualize=False)
elif args.mode == 'test':
    weights_filename = 'dqn_{}_weights.h5f'.format(args.env_name)
    if args.weights:
        weights_filename = args.weights
    dqn.load_weights(weights_filename)
    history_1 = dqn.test(env, nb_episodes=10, visualize=False)
scipy.io.savemat('history_0.mat', history_0.history, appendmat=True, format='5', long_field_names=False, do_compression=False, oned_as='row')
scipy.io.savemat('history_1.mat', history_1.history, appendmat=True, format='5', long_field_names=False, do_compression=False, oned_as='row')
```

B: Pong New - V0 Environment

```
import logging
import math
import gym
from gym import spaces
from gym.utils import seeding
import numpy as np
from os import path
import random
import pygame, sys
from pygame.locals import *
WHITE             = (255,255,255)
RED               = (255,0,0)
GREEN             = (0,255,0)
BLACK             = (0,0,0)
MAX_BALL_VEL      = 20
WIDTH             = 600
HEIGHT            = 400
BALL_RADIUS       = 20
PAD_WIDTH         = 8
PAD_HEIGHT        = 80
HALF_PAD_WIDTH    = PAD_WIDTH // 2
HALF_PAD_HEIGHT   = PAD_HEIGHT // 2
paddle1_pos       = [HALF_PAD_WIDTH - 1,HEIGHT//2]
paddle2_pos       = [WIDTH +1 - HALF_PAD_WIDTH,HEIGHT//2]
paddle1_vel       = 0
paddle2_vel       = 0
ball_pos          = [WIDTH//2, HEIGHT//2]
ball_vel          = [0, 0]
r_score_threshold = 100
l_score_threshold = 100
max_paddle1_vel   = 20
max_paddle2_vel   = 20
min_paddle1_vel   = -20
min_paddle2_vel   = -20
max_paddle1_pos  = HEIGHT - HALF_PAD_HEIGHT
max_paddle2_pos  = HEIGHT - HALF_PAD_HEIGHT
max_ball_pos     = [WIDTH - BALL_RADIUS, HEIGHT - BALL_RADIUS]
max_ball_vel     = [MAX_BALL_VEL, MAX_BALL_VEL]
min_paddle1_pos  = HALF_PAD_HEIGHT
min_paddle2_pos  = HALF_PAD_HEIGHT
min_ball_pos     = [BALL_RADIUS, BALL_RADIUS]
min_ball_vel     = [-MAX_BALL_VEL, -MAX_BALL_VEL]
logger = logging.getLogger(__name__)
global l_score, r_score, reward, reward_curr
global paddle1_pos, paddle2_pos, ball_pos, ball_vel
```

```python
class PongEnv(gym.Env):
    metadata = {
        'render.modes' : ['human', 'rgb_array'],
        'video.frames_per_second' : 30
    }
    def __init__(self):
        global l_score, r_score, reward, reward_curr
        global high, low
        global paddle1_pos, paddle2_pos, ball_pos, ball_vel, paddle1_vel, paddle2_vel
        global ball_pos, ball_vel
        def ball_init():
            global ball_vel
            horz     = 0
            vert     = 0
            while (horz == 0) or (vert == 0):
                horz     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
                vert     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
            if random.randrange(0,2) is not 0:
                horz = - horz
            ball_vel = [horz, -vert]
        r_score = 0
        l_score = 0
        reward  = 0
        high = np.array([max_paddle1_pos, max_paddle1_vel, max_paddle2_pos, max_paddle2_vel, max_ball_pos[0], max_ball_pos[1], max_ball_vel[0], max_ball_vel[1]])
        low = np.array([min_paddle1_pos, min_paddle1_vel, min_paddle2_pos, min_paddle2_vel, min_ball_pos[0], min_ball_pos[1], min_ball_vel[0], min_ball_vel[1]])
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Box(low, high)
        self._seed()
        self.viewer = None
        self.state = None
        self.steps_beyond_done = None
        self.steps_to_done = 0
        self.state = self.np_random.uniform(low, high, size=(8,))
        state = self.state
        (paddle1_pos[1], paddle1_vel, paddle2_pos[1], paddle2_vel, ball_pos[0], ball_pos[1], ball_vel[0], ball_vel[1]) = state
        ball_pos = [WIDTH//2, HEIGHT//2]
        ball_init()
        self.state = (paddle1_pos[1], paddle1_vel, paddle2_pos[1], paddle2_vel, ball_pos[0], ball_pos[1], ball_vel[0],
```

```
ball_vel[1])
    def _seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]
    def _step(self, action):
        self.steps_to_done += 1;
        global paddle1_pos, paddle2_pos, paddle1_vel, paddle2_vel, l_score, r_score
        global l_score, r_score, reward, reward_curr
        global ball_pos, ball_vel
        def ball_init():
            global ball_vel
            horz     = 0
            vert     = 0
            while (horz == 0) or (vert == 0):
                horz     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
                vert     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
            if random.randrange(0,2) is not 0:
                horz = - horz
            ball_vel = [horz, -vert]
        assert self.action_space.contains(action), "%r (%s) invalid" %(action, type(action))
        state = self.state
        (paddle1_pos[1],paddle1_vel,paddle2_pos[1],paddle2_vel, ball_pos[0],ball_pos[1],ball_vel[0],ball_vel[1]) = state
        #update paddle velocity
        if ball_pos[1] < paddle1_pos[1]:
            paddle1_vel = -max_paddle2_vel
        elif ball_pos[1] > paddle1_pos[1]:
            paddle1_vel = max_paddle2_vel
        else:
            paddle1_vel = 0
        if action == 0:
            paddle2_vel = -max_paddle1_vel
        elif action == 1:
            paddle2_vel = max_paddle1_vel
        elif action == 2:
            paddle2_vel = 0
        if paddle1_pos[1] > HALF_PAD_HEIGHT and paddle1_pos[1] < HEIGHT - HALF_PAD_HEIGHT:
            paddle1_pos[1] += paddle1_vel
        elif paddle1_pos[1] < HALF_PAD_HEIGHT and paddle1_vel > 0:
            paddle1_pos[1] += paddle1_vel
        elif paddle1_pos[1] > HEIGHT - HALF_PAD_HEIGHT and paddle1_vel < 0:
```

```
            paddle1_pos[1] += paddle1_vel
        elif paddle1_pos[1] < HALF_PAD_HEIGHT and paddle1_vel < 0:
            paddle1_pos[1] -= paddle1_vel
        elif paddle1_pos[1] > HEIGHT - HALF_PAD_HEIGHT and paddle1_vel > 0:
            paddle1_pos[1] -= paddle1_vel
        if paddle2_pos[1] > HALF_PAD_HEIGHT and paddle2_pos[1] < HEIGHT - HALF_PAD_HEIGHT:
            paddle2_pos[1] += paddle2_vel
        elif paddle2_pos[1] < HALF_PAD_HEIGHT and paddle2_vel > 0:
            paddle2_pos[1] += paddle2_vel
        elif paddle2_pos[1] > HEIGHT - HALF_PAD_HEIGHT and paddle2_vel < 0:
            paddle2_pos[1] += paddle2_vel
        elif paddle2_pos[1] < HALF_PAD_HEIGHT and paddle2_vel < 0:
            paddle2_pos[1] -= paddle2_vel
        elif paddle2_pos[1] > HEIGHT - HALF_PAD_HEIGHT and paddle2_vel > 0:
            paddle2_pos[1] -= paddle2_vel
        ball_pos[0] += int(ball_vel[0])
        ball_pos[1] += int(ball_vel[1])
        if int(ball_pos[1]) <= BALL_RADIUS:
            ball_vel[1] = - ball_vel[1]
        if int(ball_pos[1]) >= HEIGHT + 1 - BALL_RADIUS:
            ball_vel[1] = - ball_vel[1]
        if int(ball_pos[0])<=BALL_RADIUS+PAD_WIDTH and int(ball_pos[1]) in range(int(paddle1_pos[1]-HALF_PAD_HEIGHT),int(paddle1_pos[1]+HALF_PAD_HEIGHT),1):
            ball_vel[0]  = -ball_vel[0]
            ball_vel[0]  *= 1.1
            ball_vel[1]  *= 1.1
        elif int(ball_pos[0]) <= BALL_RADIUS + PAD_WIDTH:
            #reward      += 2
            r_score      += 1
            ball_pos     = [WIDTH//2, HEIGHT//2]
            ball_init()

        if int(ball_pos[0])>=WIDTH+1-BALL_RADIUS-PAD_WIDTH and int(ball_pos[1]) in range(int(paddle2_pos[1]-HALF_PAD_HEIGHT),int(paddle2_pos[1]+HALF_PAD_HEIGHT),1):
            reward += 1
            ball_vel[0]  = -ball_vel[0]
            ball_vel[0] *= 1.1
            ball_vel[1] *= 1.1
        elif int(ball_pos[0]) >= WIDTH + 1 - BALL_RADIUS - PAD_WIDTH:
            l_score     += 1
            ball_pos = [WIDTH//2, HEIGHT//2]
```

```
            ball_init()
        self.state = (paddle1_pos[1], paddle1_vel, paddle2_pos[1], paddle2_vel, ball_pos[0], ball_pos[1], ball_vel[0], ball_vel[1])
        done =  (l_score > l_score_threshold) or (r_score > r_score_threshold)
        done = bool(done)
        done = bool(done)
        if not done:
            self.steps_beyond_done = self.steps_beyond_done
            reward = reward
        elif self.steps_beyond_done is None:
            self.steps_beyond_done = 0
            reward = reward
        else:
            if self.steps_beyond_done == 0:
                logger.warning("You are calling 'step()' even though this environment has already returned done = True. You should always call 'reset()' once you receive 'done = True' -- any further steps are undefined behavior.")
            self.steps_beyond_done += 1
            self.steps_to_done      = 0
            reward = 0
        if r_score > r_score_threshold:
            r_score = 0
            l_score = 0
            reward_curr = 0
        if l_score > l_score_threshold:
            r_score = 0
            l_score = 0
            reward_curr = 0
        return np.array(self.state), reward, done, {}
    def _reset(self):
        self.steps_to_done = 0
        self.steps_beyond_done = None
        global l_score, r_score, reward, reward_curr
        global paddle1_pos, paddle2_pos, ball_pos, ball_vel
        def ball_init():
            global ball_vel
            horz     = 0
            vert     = 0
            while (horz == 0) or (vert == 0):
                horz     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
                vert     = random.randrange(-MAX_BALL_VEL,MAX_BALL_VEL)
```

```python
            if random.randrange(0,2) is not 0:
                horz = - horz
            ball_vel = [horz, -vert]
        r_score = 0
        l_score = 0
        reward  = 0
        self.state = self.np_random.uniform(low, high, size=(8,))
        state = self.state
        (paddle1_pos[1], paddle1_vel, paddle2_pos[1], paddle2_vel, ball_pos[0], ball_pos[1], ball_vel[0], ball_vel[1]) = state
        ball_pos = [WIDTH//2, HEIGHT//2]
        ball_init()
        self.state = (paddle1_pos[1], paddle1_vel, paddle2_pos[1], paddle2_vel, ball_pos[0], ball_pos[1], ball_vel[0], ball_vel[1])
        return np.array(self.state)
    def _render(self, mode='human', close=False):
        if close:
            if self.viewer is not None:
                self.viewer.close()
                self.viewer = None
                pygame.quit()
                sys.exit()
            return
        if self.viewer is None:
            pygame.init()
            fps = pygame.time.Clock()
            window = pygame.display.set_mode((WIDTH, HEIGHT), 0, 32)
            pygame.display.set_caption('Hello World')
            window.fill(BLACK)
            pygame.draw.line(window, WHITE, [WIDTH // 2, 0],[WIDTH // 2, HEIGHT], 1)
            pygame.draw.line(window, WHITE, [PAD_WIDTH, 0],[PAD_WIDTH, HEIGHT], 1)
            pygame.draw.line(window, WHITE, [WIDTH - PAD_WIDTH, 0],[WIDTH - PAD_WIDTH, HEIGHT], 1)
            pygame.draw.circle(window, WHITE, [WIDTH//2, HEIGHT//2], 70, 1)
            ball_pos_int = [int(ball_pos[0]), int(ball_pos[1])]
            pygame.draw.circle(window, RED, ball_pos_int, 20, 0)
            pygame.draw.polygon(window, GREEN, [[paddle1_pos[0] - HALF_PAD_WIDTH, paddle1_pos[1] - HALF_PAD_HEIGHT], [paddle1_pos[0] - HALF_PAD_WIDTH, paddle1_pos[1] + HALF_PAD_HEIGHT], [paddle1_pos[0] + HALF_PAD_WIDTH, paddle1_pos[1] + HALF_PAD_HEIGHT], [paddle1_pos[0] + HALF_PAD_WIDTH, paddle1_pos[1] - HALF_PAD_HEIGHT]], 0)
```

```
            pygame.draw.polygon(window, GREEN, [[paddle2_pos[0] - HALF_PAD_WIDTH, paddle2_pos[1] - HALF_PAD_HEIGHT], [paddle2_pos[0] - HALF_PAD_WIDTH, paddle2_pos[1] + HALF_PAD_HEIGHT], [paddle2_pos[0] + HALF_PAD_WIDTH, paddle2_pos[1] + HALF_PAD_HEIGHT], [paddle2_pos[0] + HALF_PAD_WIDTH, paddle2_pos[1] - HALF_PAD_HEIGHT]], 0)
            myfont1 = pygame.font.SysFont("Comic Sans MS", 20)
            label1 = myfont1.render("Score: "+str(l_score), 1, (255,255,0))
            window.blit(label1, (50,20))
            myfont2 = pygame.font.SysFont("Comic Sans MS", 20)
            label2 = myfont2.render("Reward: "+str(reward), 1, (255,255,0))
            window.blit(label2, (210, 20))
            myfont2 = pygame.font.SysFont("Comic Sans MS", 20)
            label2 = myfont2.render("Score: "+str(r_score), 1, (255,255,0))
            window.blit(label2, (470, 20))
            pygame.display.update()
            fps.tick(200)
        if self.state is None: return None
        else: return None
```

C: DQN Algorithm for Pong New - V0: Classic Control DQN From Keras-RL

```python
import numpy as np
import gym
import scipy
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten
from keras.optimizers import Adam
from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy
from rl.memory import SequentialMemory
ENV_NAME = 'pong_new-v0'
env = gym.make(ENV_NAME)
np.random.seed(123)
env.seed(123)
nb_actions = env.action_space.n
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
memory = SequentialMemory(limit=50000, window_length=1)
policy = EpsGreedyQPolicy()
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory, nb_steps_warmup=50000, target_model_update=1e-2, policy=policy)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])
hist_0 = dqn.fit(env,nb_steps=175000, visualize=False,verbose=2, nb_max_episode_steps=10000)
dqn.save_weights('dqn_{}_weights.h5f'.format(ENV_NAME),overwrite=True)
hist_1 = dqn.test(env, nb_episodes=10, visualize=False)
scipy.io.savemat('hist_0.mat',history_0.history,appendmat=True,format='5',long_field_names=False,do_compression=False,oned_as='row')
scipy.io.savemat('hist_1.mat',history_1.history,appendmat=True,format='5',long_field_names=False,do_compression=False,oned_as='row')
```