Machine Learning Engineer Nanodegree

Capstone Project: Implementation of Deep Q-learning on OpenAI’s Gym Environment

David Peabody

August 28th, 2017

Abstract

In this project I aim to implement a deep reinforcement learning approach on OpenAI’s Gym environments. In a minor modification to the project proposal I intend to use CartPole-v0 as the primary learning environment rather than Pong-v0 and if possible scale the same approach through to other environments, specifically the Atari environments. Initially I will implement a form of the Q-learning algorithm along with a neural net coded in Keras. The overarching goal of this project will be to provide a clear, step-by-step explanation of how the algorithm works to solidify the knowledge for myself and for others.

I. Definition

Project Overview

Reinforcement learning (RL) is an exciting subject area because it represents the ability to train machines to find solutions (policies) to complex dynamic problems without hard coding instructions which could be prohibitively expensive effort wise or even impossible.

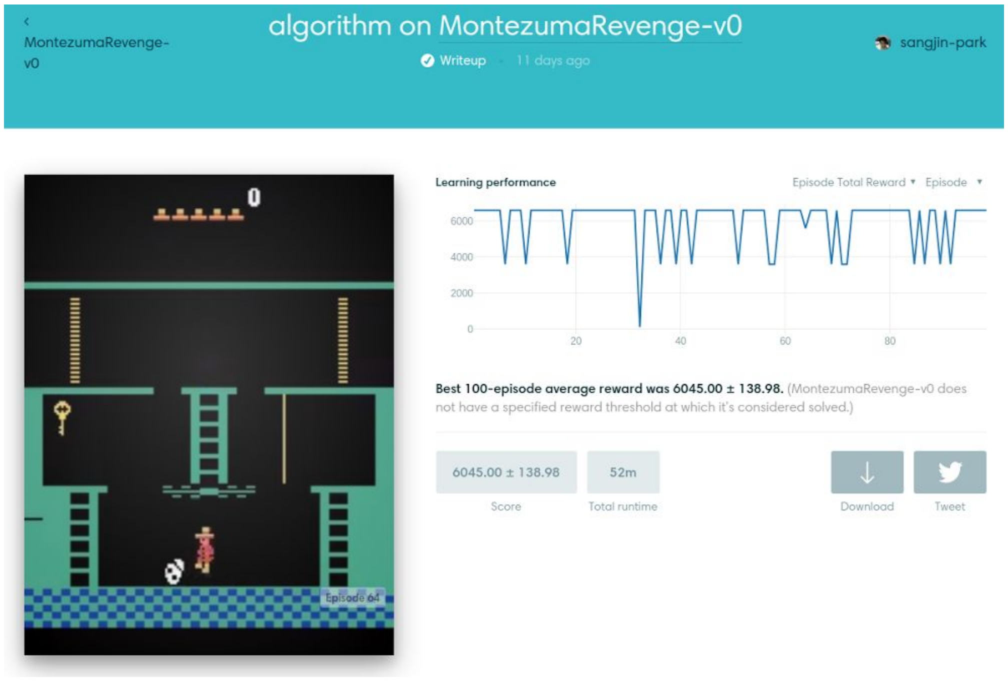
Advances in RL could lead to progress in manufacturing [1], robotic control [2], medical research, finance and engineering. Further, research into how agents can learn most efficiently can provide insight into how humans learn or more often how there are other ways to learn beyond how humans traditionally have.

Deep reinforcement learning is an active area of research in the machine learning community with weekly releases of new papers covering innovations including, algorithm design and implementation [3], new reward mechanisms & policies [4],varying neural network structures, and parameter innovations such as injecting noise into the algorithms parameters [5].

In this project I will implement a deep reinforcement learning algorithm to solve OpenAI's CartPole-v0 environment and then as a stretch goal, try to implement a modified version of the algorithm on the Breakout-v0 Atari game.

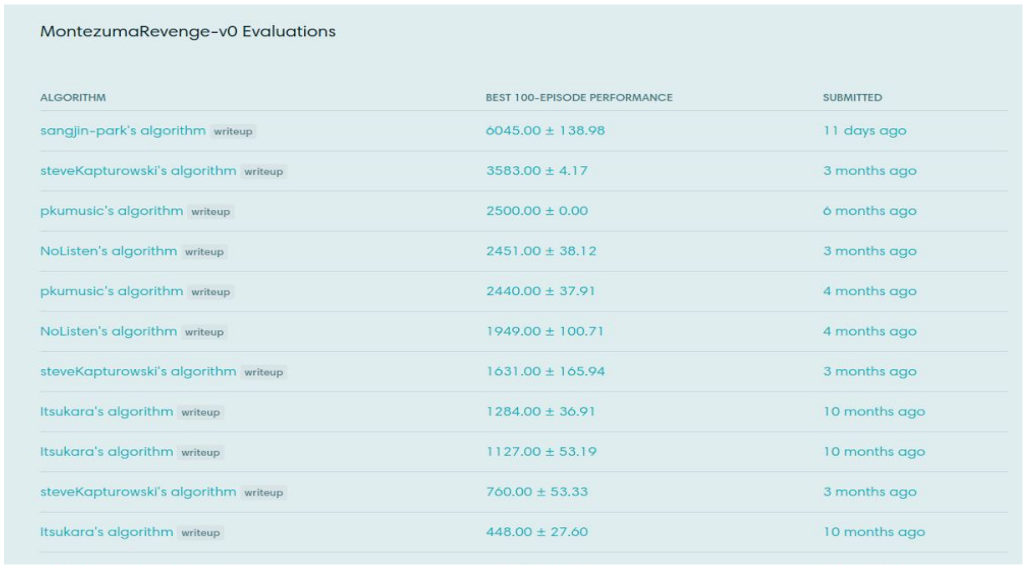
Games and control problems have become a popular testing ground for reinforcement learning. These differing environments enable algorithms to be benchmarked against each other, identifying their various strengths. Openai, has provided public access to their gym environment, which combines the whole Atari Catalog along with various control problems into a series of environments with a single interface which is the same for all [6]. This single interface enables a single algorithm to be easily generalized to a wide variety of different environments, this tests the general learning capabilities of reinforcement learning algorithms. The OpenAI gym also provides scoring and benchmarking metrics so that algorithms can be compared against each other [7].

Fig 1: Example scoring metric



Best 100-episode average reward was 6045.00 ± 138.98. (MontezumaRevenge-v0 does not have a specified reward threshold at which it's considered solved.)

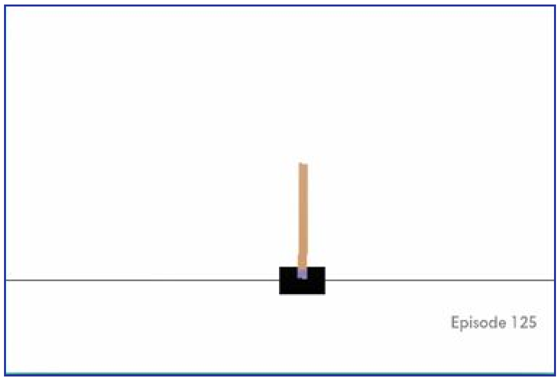
Fig 2: Example leaderboard of recent tests against an environment



MontezumaRevenge-v0 Evaluations

ALGORITHM	BEST 100-EPISODE PERFORMANCE	SUBMITTED
sangjin-park's algorithm writeup	6045.00 ± 138.98	11 days ago
steveKapturowski's algorithm writeup	3583.00 ± 4.17	3 months ago
pkumusic's algorithm writeup	2500.00 ± 0.00	6 months ago
NoListen's algorithm writeup	2451.00 ± 38.12	3 months ago
pkumusic's algorithm writeup	2440.00 ± 37.91	4 months ago
NoListen's algorithm writeup	1949.00 ± 100.71	4 months ago
steveKapturowski's algorithm writeup	1631.00 ± 165.94	3 months ago
Itsukara's algorithm writeup	1284.00 ± 36.91	10 months ago
Itsukara's algorithm writeup	1127.00 ± 53.19	10 months ago
steveKapturowski's algorithm writeup	760.00 ± 53.33	3 months ago
Itsukara's algorithm writeup	448.00 ± 27.60	10 months ago

Problem Statement



The CartPole-v0 environment which will be our base test environment in which we will implement our deep q-learning algorithm, is a simple control problem. The goal of the problem is to balance the pole in an upright vertical position by moving the cart left or right. This represents a single joint pivot problem.

In the CartPole environment you are provided feedback on the angle of the pole from vertical and a reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

The CartPole-v0 environment defines "solving" as getting average reward of 195.0 (max score is 200) over 100 consecutive trials, this will be the goal for the project.

The solution implemented in this project will be a basic deep Q-learning algorithm. The deep Q-learning algorithm combines a deep neural network with an implementation of the Q-learning algorithm (shown below).

Equation 1) $$Q(s,a) = r(s,a) + \gamma(\max(Q(s',a'))$$

Where the aim at each state (s) is to choose the action (a) which maximizes the utility function, Q(s,a). In the equation above, the utility function, Q(s,a), is the estimate of how beneficial a given action is from a specific state. This is calculated by taking into account the immediate reward, r(s,a), plus the discounted (γ) maximum reward for the future state (s').

A deep neural network is used because of the neural nets ability to make predictions based on a large high dimensional input space. An example for this will be the Atari environments where the input space is an RGB image of the video stream, which is an array of shape (210, 160, 3).

My focus for this project will be to not only successfully implement a deep Q-learning solution but also to provide a didactic implementation to solidify my own knowledge and hopefully to provide a clear guide for other students just getting to grips with deep reinforcement learning.

Breakout-v0

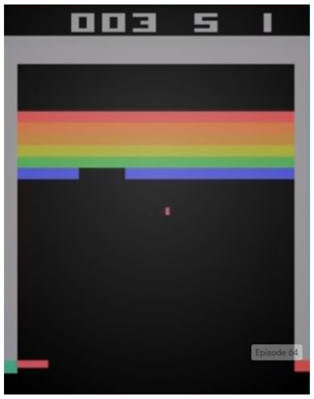
As mentioned above my stretch goal for this project will be to implement a modified version of the deep q-learning algorithm on Breakout-v0, a game in the Atari environment. As a brief aside you may wonder why I have considered this a stretch goal. The difficulties with the Atari environments are mainly due to the huge increase in state space which comes with image input. To break this down, the input space is an array every timestep of size (210, 160, 3), flattened this is an input of 100,800. This input space can be shrunk with preprocessing, however it still represents a sizable challenge in terms of:

- Time it takes the algorithm to learn
- Hardware required to run the algorithm in an acceptable time
- Stability of the algorithm

As a go by, in the DDQN paper by Hasselt et al. [8], who implemented a Double Deep Q-Learning Algorithm (described later) on the Atari environments stated: “On each game, the network is trained on a single GPU for 200M frames, or approximately 1 week.” As the reader can imagine a week is not an insignificant time period to be running an experiment.

In addition, as mentioned above, the instability of the algorithm can be an impediment to its ability to learn, and the larger the state space and more stochastic the more the algorithm has to be tuned to avoid chronic instability.

So in short, I will implement the algorithm on the Breakout-v0 environment. But, no promises on the results.



As a short overview of Breakout-v0: Breakout is a traditional arcade game which involves moving a paddle left or right to hit a ball and break the bricks above, each brick broken scores a point. In every episode (new game) you are given 5 balls (lives) in which the aim is to score as many points as possible, every time you fail to hit the ball with the paddle, the ball is lost and you lose a life. This environment does not have a score at which it is considered solved, the aim is strictly to score as many points as possible.

Metrics

As mentioned in the above section, the model will be evaluated against OpenAI's platform. The gym platform takes the average reward received for a the best 100 - episode period. The gym platform is designed specifically for reinforcement learning hence why algorithms are compared on the average rewards received. In addition, as a further method we will track the number of episodes required to produce an effective model.

The CartPole environment provides a reward of +1 for every timestep the pole remains upright, the environment resets when you reach 200 timesteps (the max score).

The Breakout environment provides a + 1 reward for every brick broken. There is no max stated score but for comparison the best 100 episode average score currently achieved is 760 [9]

Internal metrics

As the metrics proposed for this project are fairly straightforward I thought I would briefly discuss the metrics involved internally in the DQN. In this implementation, we use the Mean Squared Error (MSE) as the loss function.

Equation 2) MSE = $\frac{1}{n} \sum_{i=1}^{n} (\hat{Y}i - Yi)^2$

Where $\hat{Y}i$ is the target and Yi is the predicted value of the i-th sample.To train the neural network on a sample we call the following code:

```
self.model.fit(state, target_f)
```

This code provides the state and the target, if the current episode is finished the target is equal to the actual reward received and if it is not finished, is equal to the predicted max utility for the given state and action. The MSE of the target and the current predicted reward is then taken and backpropogated through the network adjusting the weights proportionally to update the model.

II. Analysis

Data Exploration

OpenAI Gym

Unlike traditional supervised learning, reinforcement learning problems do not come packaged with a labeled dataset in hand. In RL the dataset is built up based on the experiences of the agent.

For this project the input and space in which the agent acts is OpenAI's Gym environment, specifically the CartPole-v0 environment. OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms [10].

The two key aspects of the Gym environments are:

1) Providing an easy to use, consistent interface so that a single algorithm design can be tested against a wide variety of environments with few to no alterations. Thus, illuminating the strengths and weaknesses of an algorithm in various situations.

2) Implementing strict versioning requirements to the environments so that the relative capabilities of algorithms can be compared against each other, apples to apples for the foreseeable future.

Environments

CartPole-v0

As described by OpenAI:

"A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center."

The Cartpole environment represents a simple classic control problem, which can actually be solved fairly easily by traditional process control solutions. However, traditional process control solutions are not generalizable and have to be customized to the specific problem.

State Space

The state space of our environment can be found through the following code:

```
print(env.observation_space)
#> Box(4,)
```

This represents that for every observation of the environment we recieve 4 pieces of information.

```
print('Observation = {}'.format(observation))

#> Observation = [ 0.12057927  1.41783334 -0.15178413 -2.14444717]
```

Above is a sample observation from a test run, the 4 pieces provided represent the position of cart, velocity of cart, angle of pole, rotation rate of pole respectively.

Actions

In the CartPole environment we are limited to two actions as shown by the code below:

```
print(env.action_space)
#> Discrete(2)
```

These Two actions are represented by 0 & 1 where 0 = left and 1 = right. Thus advantageous actions will act upon the cart and aim to stabilize the pole by countering the rotation and the angle of the pole with an opposite force.

Rewards

As mentioned before, the reward structure is that for each timestep the pole remains in an upright position (no more than 15 degrees from vertical or 2.4 units from the center) the agent is rewarded +1. In this environment there is no negative rewards (-1) or neutral rewards (0).

This summarizes the CartPole environment which takes actions as inputs and outputs observations and rewards.

Breakout-v0

For a brief description of Breakout-v0 please see the problem statement section above when an intuitive description was given.

State Space

Each observation is an image of the arcade screen this RGB image is provided to us as a numpy array, in the shape of 210, 160, 3 as shown in the following code:

```
print(observation.shape)
#> (210, 160, 3)
```

As mentioned above this large state space represents a fairly big challenge as when flattened, this observation has a size of 100,800. Added to this the ball and paddle can be in many different locations or trajectories at any given moment, making the number of possible states very large and an added challenge to learn an optimum solution for.

Actions

In Breakout the agent only controls the paddle, which can move left or right, in addition in this environment there are several other actions as shown in the code below:

```
print(env.unwrapped.get_action_meanings())
```

```
#> ['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

Where:

- NOOP = No action (paddle remains in place)
- FIRE = Start new game after episode ends
- RIGHT = Move paddle right
- LEFT = Move paddle left

Rewards

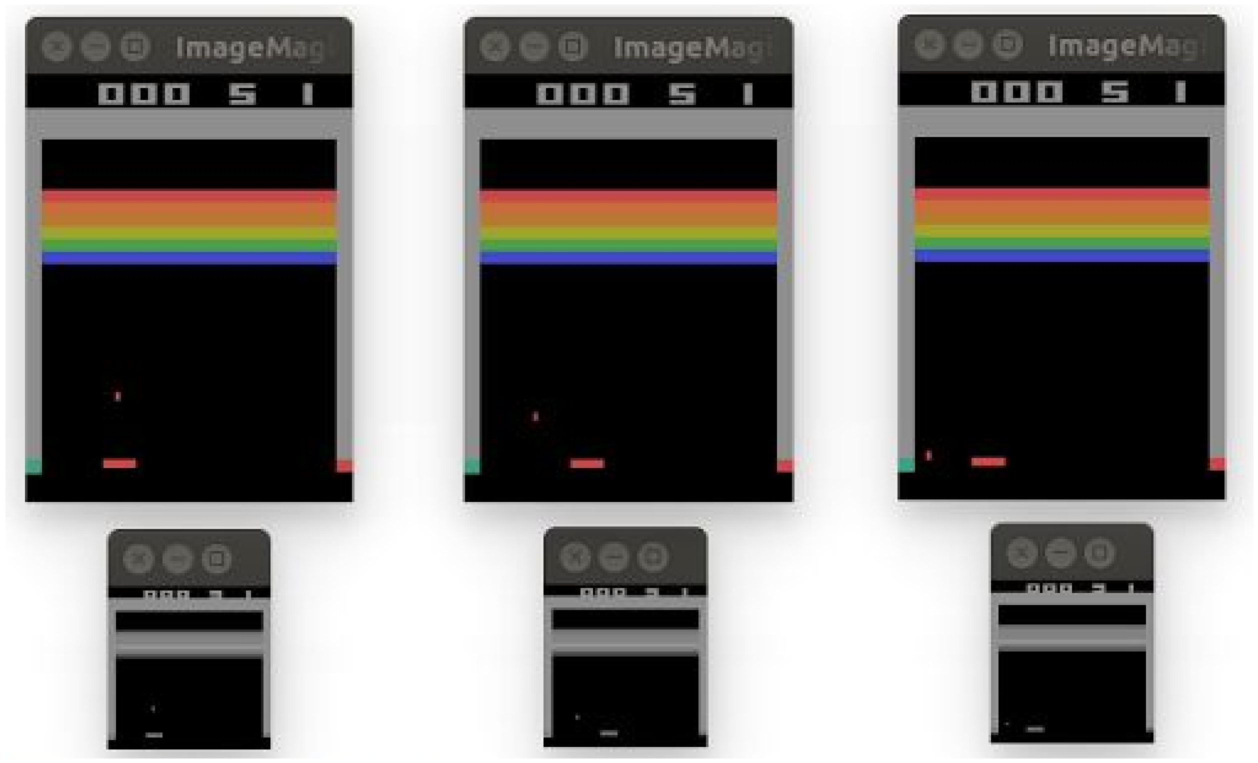
In Breakout the agent receives a positive reward (+1) for every brick that is broken, for all other timesteps which do not result in a brick being broken the agent receives a neutral reward (0)

As you can see from the breakdown of the environments above both the CartPole and Breakout environments are fairly similar as far as taking actions, providing a observation and a reward. The key difference between the environments is that the CartPole environment provides a fairly small state space whereas the Breakout environment provides a full RGB image at every time step which will add some complications in terms of processing.

Exploratory Visualization

To provide a better understanding of the state information provided at each step of the Atari environment in OpenAI gym, I have provided a few frames which will be described below. Although CartPole is the main environment that will be investigated the state information as described above is fairly simple and less interesting visually, hence using an Atari example below.

Fig 3: Sample output of unprocessed and processed observations



The first 3 colored frames are the RGB (210, 160, 3) shaped array which comes directly from the observations of the environment. I have chosen a rough sequence of outputs but not a frame by frame reproduction (there are around 60 frames per second and as such a sequence of 3 consecutive frames would not have enough difference to be readily discernable by the human eye).

The second set of three frames are our processed versions. An important step when passing images to a neural network is to reduce the irrelevant information and also the size of the image being transferred. Reducing irrelevant information, in this case the color, improves the learning rate as the algorithm is not getting distracted by making causal assumptions about coincidental changes in color. In addition reducing the image size reduces the processing power / time required on each frame.

This initial processing to get from the first set of images to the second follows the following format:

- The initial RGB image with shape 210, 160, 3 is converted to grayscale with shape 210, 160, 1
- The grayscale image is then resized from an array with shape 210,160,1 to a shape of 84, 84, 1

As we can see we started with an image that when flattened had a size of 100,800 and have reduced this through processing to an image that when flattened would have a size of 7,056, while not losing any important information.

A brief note on how these images were produced:

A full implementation of the image processing will be produced in section III, however that will not cover the printing of the individual frames as that is a memory intensive process, which is not required when running the algorithm. These images were produced using the Python PIL module calling the following code before and after processing to produce the 2 sets of images.

Unprocessed Images:

```
new_state, reward, done, info = env.step(action)

xxx = new_state

img = Image.fromarray(xxx, 'RGB')
```

Processed Images:

```
new_state = RGBprocess(new_state)

xxx = new_state.reshape(new_state.shape[1], new_state.shape[2])

img = Image.fromarray(xxx, 'L')

img.show()
```

Algorithms and Techniques

Q-Learning

Q-Learning is a reinforcement learning technique which can be used to find an optimum policy for any finite Markov decision process. In its simplest implementation the Q-Learning algorithm is a value iteration program whose key goal is at each step to chose the action which maximises the utility function Q(s,a). A formal definition was shown in Equation 1 above and repeated here:

Equation 1) $Q(s,a) = r(s,a) + \gamma(\max(Q(s',a')))$

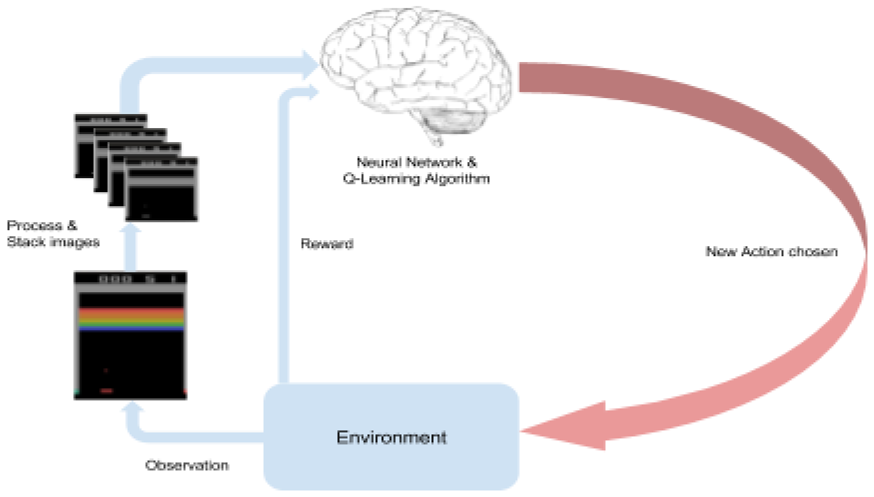
Where the aim at each state (s) is to choose the action (a) which maximizes the utility function, Q(s,a). In the equation above, the utility function, Q(s,a), is the estimate of how beneficial a given action is from a specific state. This is calculated by taking into account the immediate reward, r(s,a), plus the discounted (γ) maximum reward for the future state (s’).

This equation works well for a small state space with a limited number of action such as in the self driving cab project, completed earlier in the course, where the Q values for any given state and action combination are built up in a table which can then be reference to find the optimum future solution.

This solution breaks down when you encounter a problem with a large state space and many possible actions. In this case we must use a technique to predict the future utility.

Deep Q-Learning

Fig 4: Simple Deep Q-Learning flow diagram



The above simplified diagram of a Deep Q-Network shows the basic process the algorithm goes through to learn and optimum policy. However what this diagram oversimplifies is what is going on in the brain image, there is really two parts to the brain image, the Q-Learning algorithm as shown previously and the neural network.

The Q-Learning algorithm can be assumed to work exactly as stated above with one small modification. To deal with the large state space and the impracticality of building a Q-Table for every possible state action pair, we instead use the neural network to predict the future reward max(Q(s',a')). Thus equation 1 above becomes:

Equation 3: $Q(s,a) = r(s,a) + \gamma(\max(\text{model.predict}(Q(s', a'))$

Thus we are using the neural network to predict a possible value for Q(s', a'). This prediction starts off as being arbitrary based on the original neural network initialization. As the algorithm explores the state space and updates the neural network, training it on state, action and real reward pairs the predicted Q- value becomes more accurate until it converges on the real value.

Neural Network Architecture

CartPole-v0

The CartPole environment as described earlier is a fairly simple environment and so the neural network required is also a pretty basic network. This network is defined by 3 fully connected layers, layer 1 being the input, layer 2 the hidden layer and layer 3 the output. We have used a relu activation function and the Adam optimizer. We experimented with larger networks with more hidden layers and more nodes on each layer but this did not improve the DQN and may have contributed to over fitting.

Code to build the model:

```
model = Sequential()

model.add(Dense(20, input_dim=self.state_size, activation='relu'))

model.add(Dense(20, activation='relu'))

model.add(Dense(self.action_size, activation='linear'))

model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
```

Summary of the model:

```
Layer (type)            Output Shape          Param #
=================================================================
dense_1 (Dense)           (None, 20)            100
_________________________________________________________________
dense_2 (Dense)           (None, 20)            420
_________________________________________________________________
dense_3 (Dense)           (None, 2)            42
=================================================================
Total params: 562

Trainable params: 562

Non-trainable params: 0
_________________________________________________________________
```

Breakout-v0

As described previously even after processing the state images, the Atari environments are much more complicated and have a larger state space. In addition, information about what is happening in the environment has to be pulled out of the images. For this we return to our old friend, convolutional neural networks. In our network below we have used a similar architecture as described in the original DeepMind paper "Playing Atari with Deep Reinforcement Learning" [11].

Code to build the model:

```
model = Sequential()

model.add(Conv2D(32, kernel_size = 8, subsample = (4, 4), activation='relu', padding = 'same', input_shape = sizes))

model.add(Conv2D(64, kernel_size=4, subsample=(2, 2), activation='relu', padding='same'))

model.add(Conv2D(64, kernel_size=3, subsample=(1, 1), activation='relu', padding='same'))

model.add(Flatten())

model.add(Dense(256, activation='relu'))

model.add(Dense(self.action_size))

model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
```

Summary of the model:

```
Layer (type)            Output Shape           Param #
=================================================================
conv2d_4 (Conv2D)         (None, 21, 21, 32)       8224
_________________________________________________________
conv2d_5 (Conv2D)         (None, 11, 11, 64)       32832
_________________________________________________________
conv2d_6 (Conv2D)         (None, 11, 11, 64)       36928
_________________________________________________________
flatten_2 (Flatten)       (None, 7744)             0
_________________________________________________________
dense_3 (Dense)           (None, 256)              1982720
_________________________________________________________
dense_4 (Dense)           (None, 4)              1028
=================================================================
Total params: 2,061,732
Trainable params: 2,061,732
Non-trainable params: 0
_________________________________________________________________
```

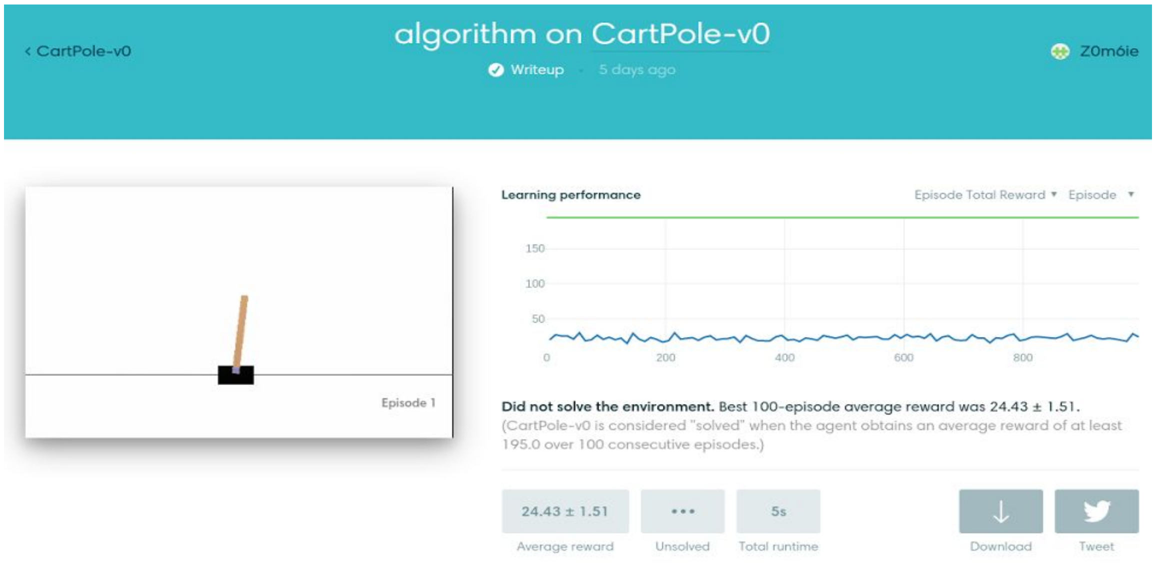
Note 1: A detailed explanation of a convolutional neural network architecture and how it works is beyond the scope of this project for more information please see the following link for an in depth explanation [12].

Note 2: See section III for a discussion of the parameters used.

Benchmark

For both the CartPole & Breakout environments I have created a random action benchmark and uploaded it to OpenAI gym along with a write-up. I believe this sort of initiative is important for the community to have basic benchmarks to compare their algorithms against. The random action benchmark records the score received by only taking random actions with no learning.

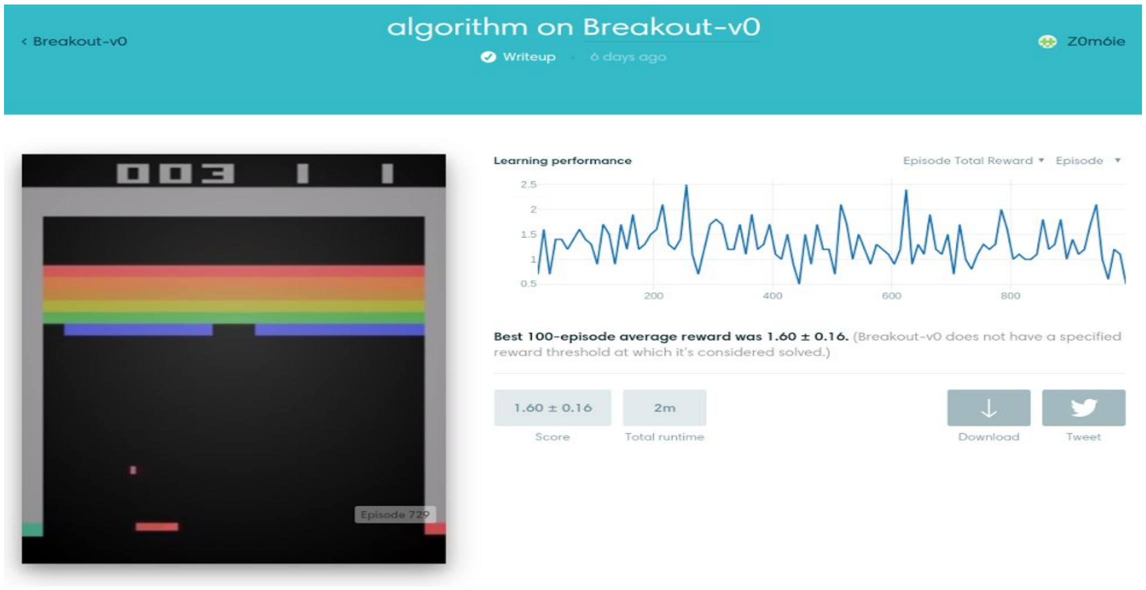
Fig 5: CartPole-v0 Random Action Benchmark



Did not solve the environment. Best 100-episode average reward was 24.43 ± 1.51. (CartPole-v0 is considered "solved" when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.)

As shown in the upload the best 100 episode score over a 1000 episode test was **24.43 +/- 1.51**. Please follow the link to see a GIF of the test and for the code I used to run the test [13].

Fig 6: Breakout-v0 Random Action Benchmark



Best 100-episode average reward was 1.60 ± 0.16. (Breakout-v0 does not have a specified reward threshold at which it's considered solved.)

As shown in the upload the best 100 episode score over a 1000 episode test was **1.60 +/- 0.16**. Please follow the link to see a GIF of the test and for the code I used to run the test [14].

III. Methodology

For the Methodology section I am providing a walk through of the code and so will deviate slightly from the prescribed write up, however I will still cover all the major sections just in a different order.

The code for the Deep Q-Learning algorithm is in two files:

- main.py file which sets up the environment, calls the agent and logs the results.
- agent.py file which has the preprocessing, neural network building, learning and action selection.

I will begin by describing and explaining the main.py file and then cover the agent.py file.

main.py

Import Statements

```
import sys

import gym

from gym import wrappers

from agent import Agent

import numpy as np
```

Fairly standard list of imports, sys to enable some arguments to be added to running the program, gym is the OpenAI gym environment and wrappers enables monitoring and uploading the file to the gyms scoreboard. From agent import Agent is our call to be able to run the functions in our agent.py file. Finally we import numpy to enable some array resizing.

Program setup

```
batch_size = 32

episodes = sys.argv[1] if len(sys.argv) > 1 else 1500

episodes = int(episodes)

env_name = "CartPole-v0"

env = gym.make(env_name)
env = wrappers.Monitor(env, env_name, force=True)

agent = Agent(env.observation_space.shape[0], env.action_space.n)

agent.f
```

In this section we select which environment we want the gym to use and call “gym.make” to create the environment. I added a “sys.argv” statement to allow the user to specify the number of episodes they want the environment to run for. If no number is provided the program defaults to running for 1500 episodes. At the end of this section we pass the observation shape and number of possible actions through to our agen.py file. This allows us to call functions through the “agent.function” statement. Finally, “agent.f” is an opening of an CSV file to log our loss and Q-value predictions.

The “Meat” of the Program

```
for i_episodes in range(episodes):

  state = env.reset()

  state = np.reshape(state, [1, env.observation_space.shape[0]])

  index = 0

  done = False

  while not done:

    #env.render()

    action = agent.act(state)

    new_state, reward, done, info = env.step(action)

    new_state = np.reshape(new_state, [1, env.observation_space.shape[0]])

    agent.remember(state, action, reward, new_state, done)

    state = new_state

    index += 1

  agent.memory_replay(batch_size)

  if done:

    print("{} episode, score = {} ".format(i_episodes + 1, index + 1))

    agent.save_model()

agent.f.close()
```

```
env.close()
```

The above section should be fairly easy to follow without much explanation, so I will just provide a quick logical overview of what is going on.

For each episode (an episode can be though to a single game which runs until you win or lose) the agent calls an observation from the environment. This is the state in which the agent is currently in. From that observation the agent calls "agent.act(state)" to decide what action to take and then enacts the action on the environment. When the agent takes an action in the environment it receives back a new state, the reward if any received and a statement of whether the episode has finished or not.

The agent then consigns that state, action, reward, new state and done statement to memory. This action is part of creating a memory history to sample from in the memory replay described below.

Finally I have a few book keeping statements to print the score per episode, save the model and close the CSV log and close the environment.

Memory Replay

Memory replay is one of the many tricks which makes deep reinforcement learning a viable solution. A key issue with deep reinforcement learning is that if you only trained the model live on the state, action, reward, new state group which you saw at that exact moment it would take an extremely long time for the model to converge on anything. In addition because all the training samples would be in sequence, you could easily drive the model to converge on a local minimum.

Memory replay is a technique to counter this, instead of training live on just the most recent example the memory replay samples a random batch from the stored memory and trains the model on each of those examples. In our code this means on every episode we train our model on 32 random examples, thus hopefully providing enough experiences for our model to learn and avoid local minimums.

agent.py

Class

```
class Agent:

    def __init__(self, state_size, action_size):

        self.state_size = state_size

        self.action_size = action_size

        self.memory = deque(maxlen=2000)
        self.gamma = 0.95   # discount rate

        self.epsilon = 1.0  # exploration rate

        self.epsilon_min = 0.00  # exploration will not decay further

        self.epsilon_decay = 0.000995

        self.learning_rate = 0.001

        self.model = self._build_model()

        self.weight_backup = 'model_weights.h5'

        self.f = open('csvfile.csv', 'w')
```

The class Agent contains all the major functions called in the program and as listed above all the main parameters. For example, self.gamma is the discount rate shown in Equation 1, in most situations the reward you receive at that moment is fairly insignificant so it makes sense not to highly discount future rewards, hence why we have chosen a gamma close to 1.

We also assign the epsilon start, decay and minimum rate, epsilon is the exploration factor. When epsilon is 1 every action taken is a random action and when epsilon is 0, no random actions are taken. At the end of every episode the epsilon is reduced by the decay rate until zero.

Our final parameter is the learning rate, which determines how much of an impact each new training example has on what we have learnt so far, too small and you will never learn, too large and every result could change your whole model, creating a very unstable result.

Build Model

```
  def _build_model(self):

    model = Sequential()

    model.add(Dense(20, input_dim=self.state_size, activation='relu'))

    model.add(Dense(20, activation='relu'))

    model.add(Dense(self.action_size, activation='linear'))

    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))

    print(model.summary())

    return model
```

See section II, Neural Network Architecture for an explanation.

Act

```
def act(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])
```

In the act function we call a random number between 0 & 1 if that number is less than our epsilon (exploration factor) we take a random action. If however that number is more than our epsilon we use our model to predict the utility value of each action and then select the action that has the highest utility value.

Remember

```
def remember(self, state, action, reward, new_state, done):
    self.memory.append((state, action, reward, new_state, done))
```

This simple function consigns our state, action, reward, new state & done statement to our memory. The memory is a deque with a max length of 2000. This stops the memory taking up too much of our real working memory and crashing the computer as well as ensures we are sampling from more recent memories.

Memory Replay

```
def memory_replay(self, batch_size):
    if len(self.memory) < batch_size:
        return
    Sample = random.sample(self.memory, batch_size)
    for state, action, reward, new_state, done in Sample:
        target = reward
        if not done:
            target = reward + self.gamma * np.amax(self.model.predict(new_state)[0])
        target_f = self.model.predict(state)
        target_f[0][action] = target
        history = self.model.fit(state, target_f, epochs=1, verbose=0)
```

Please see the memory replay section above for an explanation as to why we do experience replay. The memory replay function loops through the random sample taken from our memory. If the done statement is true i.e. the episode is over, the target is made to equal the reward received. If the done statement is false then the target equals the Q(s, a) equation. We use this target to update the predicted utility values for the state and action pair. This new target value is then used to update our model.

Processing

The code shown thus far in the method has all been from the agent built for CartPole and simple control environments. In these environments no processing is required. In the Atari environments processing is required to reduce the size of the images and for stacking, these processing steps are described below.

Image processing

```
  def RGBprocess(self, raw_img):

      processed_observation = Image.fromarray(raw_img, 'RGB')

      processed_observation = processed_observation.convert('L')

      processed_observation = processed_observation.resize((84, 84))

      processed_observation = np.array(processed_observation)

      processed_observation = processed_observation.reshape( 1, processed_observation.shape[0],
processed_observation.shape[1])

      return processed_observation
```

As described above, the initial image processing step converts the RGB image to grayscale and then reduces its size to 84, 84. The image is then slightly reshaped to work with keras and provide a dimension for the images to be stacked on.

Stacking

```
  def stack(self, processed_observation):

      I_4 = self.old_I_3 if self.old_I_3 is not None else np.zeros((1, 84, 84))

      I_3 = self.old_I_2 if self.old_I_2 is not None else np.zeros((1, 84, 84))

      I_2 = self.old_I_1 if self.old_I_1 is not None else np.zeros( (1, 84, 84))

      I_1 = processed_observation

      processed_stack = np.stack((I_4, I_3, I_2, I_1), axis=3)

      self.old_I_4 = I_4
```

```
    self.old_I_3 = I_3

    self.old_I_2 = I_2

    self.old_I_1 = I_1

    return processed_stack
```

If our algorithm was fed only a single unstacked image in isolation it would not be able to pick up any sense of movement or cause and effect. Stacking images is an important step to overcome the limitations of a single image taken from a video stream. By stacking together 4 images in a sequence we are able to get an idea of motion of the objects in the video stream. In the code above every new frame is added to the end of the stack for the 3 previous frames.

Refinement

Ahhh refinement *cough* tinkering, The final code that is being presented went through many revisions, not so much for functionality, which came quite quickly, but instead for readability.

Some examples of this are in the pre-processing section, where I initially started off using OpenCV. I moved away from OpenCV because i had issues trying to get it to install properly on one of my computers and so could not test implementations. I then moved to a pure numpy implementation, while this was an impressive implementation in terms of understanding what makes up an image in an array, it was not very didactic and was fairly hard to understand. I ultimately ended up with the current PIL implementation. The PIL implementation is quite clear and is a pretty standard package that most people have ready access to.

Another section which saw a lot of change is the Memory Replay function. This section required the most trial and error to get it working and required me to review as much other code as i could find on the subject to get a complete understanding of the why and how. For many of my early runs, I implemented a train_on_batch which instead of training the model on each of the 32 samples individually, trains on all samples at the end of the memory replay process. While this may be a more efficient implementation (I am unsure) it added additional reading complexity to the code. I ultimately felt like someone who was new to reinforcement learning would struggle to understand that implementation.

Double Deep Q-Network (DDQN)

The largest refinement made was ultimately my implementation of DDQN. DDQN helps combat one of the fundamental issues of a simple DQN implementation which is that a simple DQN tends to overestimate the

Q-value which can lead to the algorithm taking longer to converge and being less stable in finding the optimum Q-value.

DDQN combats this by decoupling the action selection from the Q-value estimation, so that you have 2 separate neural networks making the predictions, with the new network being updated with the weights of the original at every X number of steps. See code below for the original DQN and the refined DDQN:

DQN

```
        if not done:

            target = reward + self.gamma * np.amax(self.model.predict(new_state)[0])
```

DDQN

```
            future_action = self.model.predict(new_state)[0]

            future_q = self.target_model.predict(new_state)[0]

            target = reward + self.gamma * future_q[np.argmax(future_action)]
```

For both the CartPole & the Atari environment I have provided a DQN & a DDQN implementation.

Results and a comparison will be shown in the Results section.

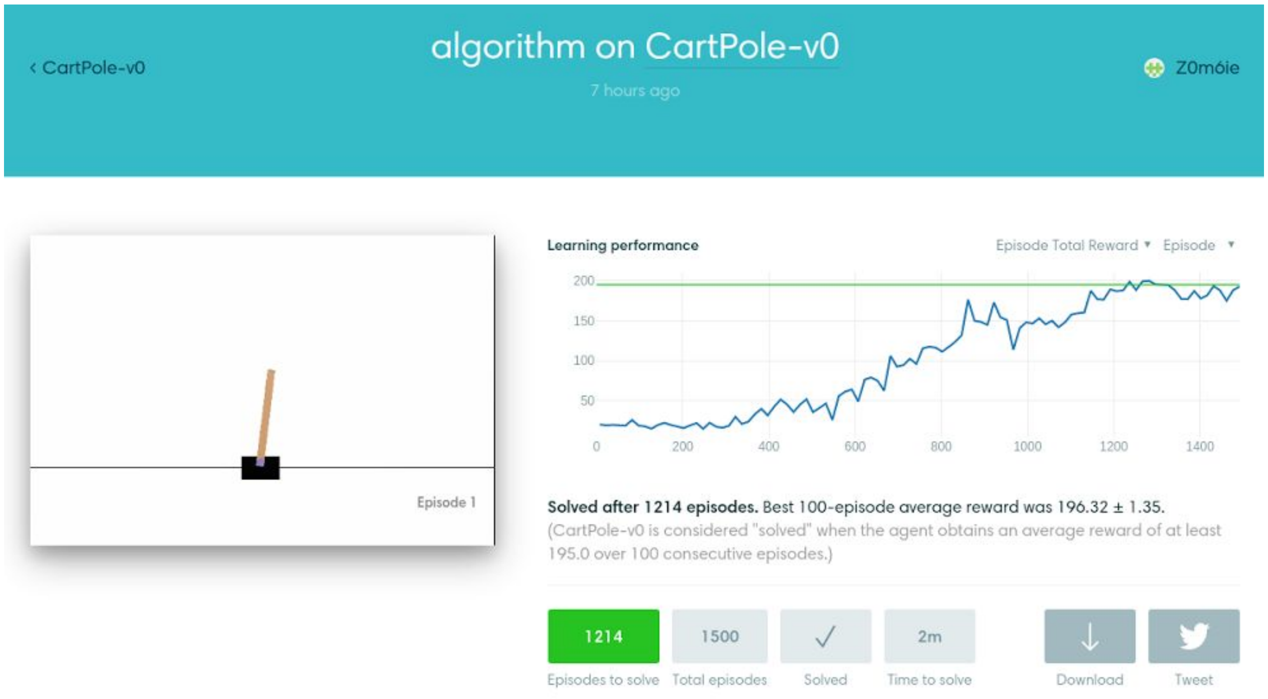
IV. Results

Model Evaluation and Validation

A Tale of Two Environments

CartPole-v0

Fig 7: DQN Implementation Results



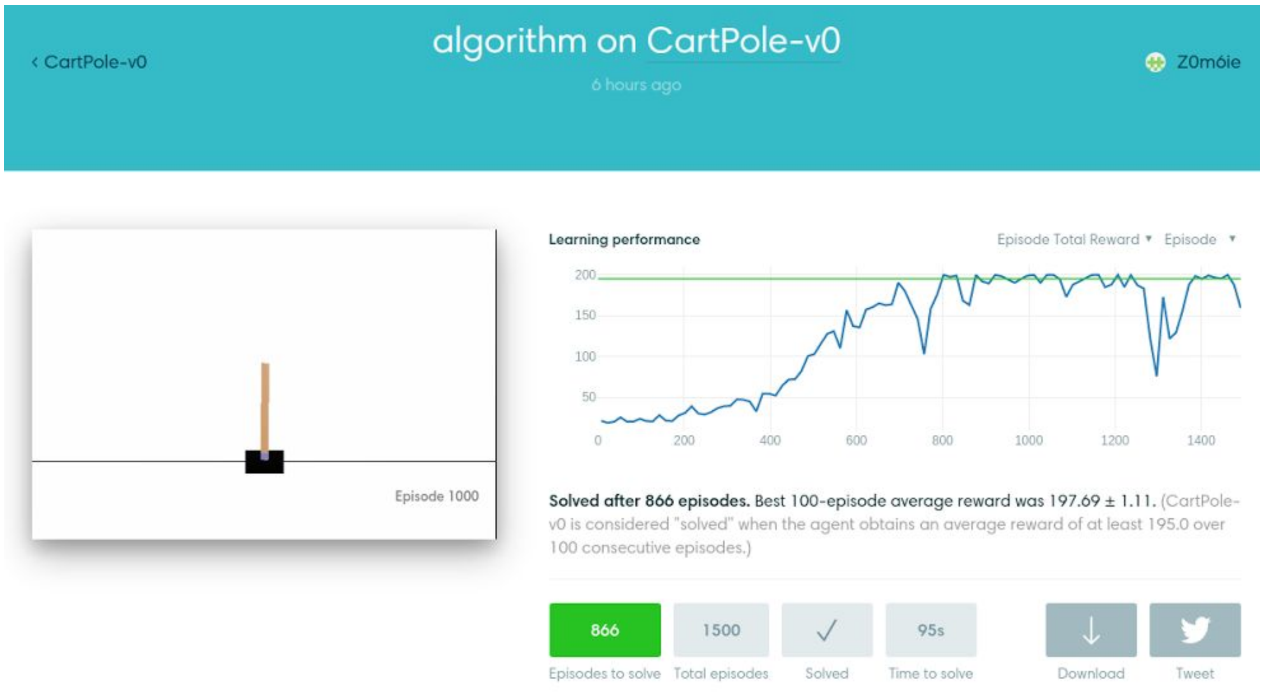
Solved after 1214 episodes. Best 100-episode average reward was 196.32 ± 1.35. (CartPole-v0 is considered "solved" when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.)

As we can see above my implementation of a Deep Q-Learning model solves the CartPole environment in **1214** episodes and has an average best score over 100 episodes of **196.32 +/- 1.35.** For a link to my results click here.

This is a great result in that the algorithm always learns and most of the time will solve the environment in between 800 - 3000 episodes.

The final parameters used include a gamma of 0.98 and a learning rate of 0.0005. These choices seemed to improve the stability of the training, however individual runs are not necessarily consistent, as I have had runs which solve in 700 episodes and others that have not solved over 3000 (although still performing far better than random, just not meeting OpenAI's definition of solving).

Fig 8: DDQN Implementation Results



Solved after 866 episodes. Best 100-episode average reward was 197.69 ± 1.11. (CartPole-v0 is considered "solved" when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.)

As we can see above my implementation of a Double Deep Q-Learning model solves the CartPole environment in **866** episodes and has an average best score over 100 episodes of **197.69 +/- 1.11.** For a link to my results click here.

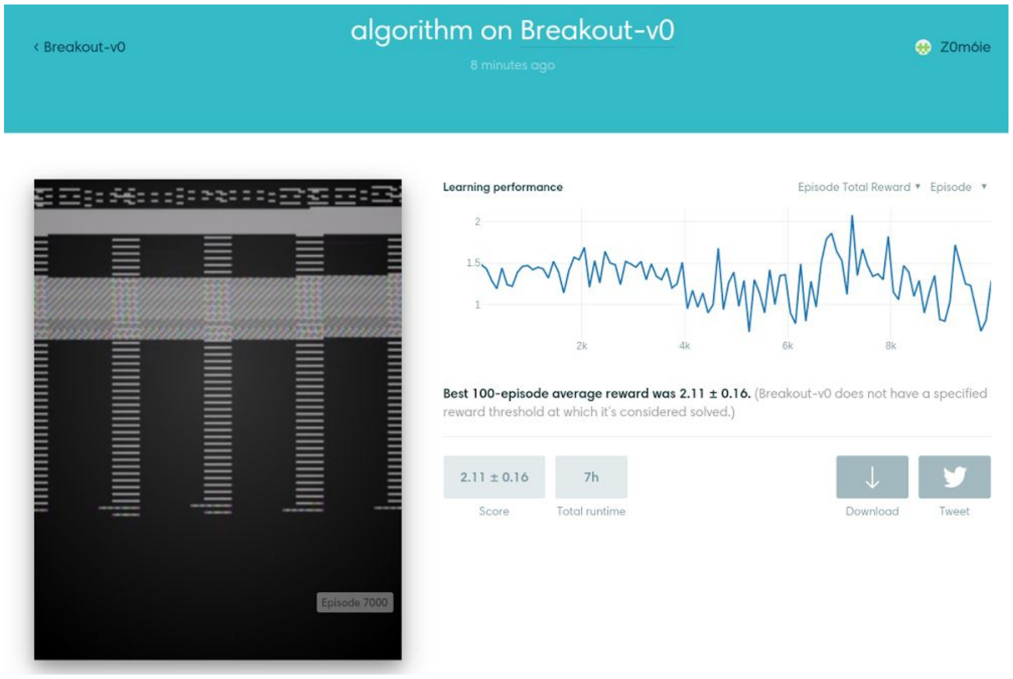
Again this is a fantastic result in that the algorithm always learns and most of the time will solve the environment in between 800 - 3000 episodes.

In general the DDQN solves the environment slightly faster than the DQN model and if we delve into the Q-value estimates there are some other interesting difference which will be described in detail in the Free Form Visualization section of the conclusion.

What is noticeable by anyone who runs the tests multiple times is that the learning of both models is not very stable. It will quite often perform at max score for a while and then collapse back to a very low score and continue to jump around. To a certain extent I believe this is due to the nature of a very simplified Q-learning algorithm. It could also be due to an unoptimised neural network, throughout the building process I tried a few different architectures however I was never able to find something that consistently improved on my base implementation.

Breakout-v0

Fig 9: DQN Implementation Results



Best 100-episode average reward was 2.11 ± 0.16. (Breakout-v0 does not have a specified reward threshold at which it's considered solved.)

... A picture paints a thousand words...

As we can see in the above figure, the implementation did not do well, the best 100-episode average was **2.11 +/- 0.16**. Results can be seen here.

In the end I had many issues with my Atari implementation, I was unable to run it on my home computer without exceeding my memory and the program coming to a halt. I was able to run it on a remote GPU service "Floyd Hub" this worked adequately but due to the remote server I was unable to visualize the agent as it trained, in addition as you can see from the image above this remote operation causes some issues with the video feed (I have checked that the fracturing of the video input is nothing to do with my pre-processing and even happens when you run a simple random action test where no image processing is involved.

A Ray of Hope?

While it would be easy to think the implementation was a complete fail and not working at all, it is not actually as clear cut. In some of my longer tests for single episode periods the agent scored 12 - 16 points on occasion. This is far above the max score I saw in any of my random action tests (more than double). While I guess this could be random, it does not seem likely. In addition when reviewing the Q-value estimates, the agent is making Q-value predictions however it seems to suffer from what I term exploding Q-values. That is Q-value predictions that increase far beyond possible maximum scores for the environment. In addition in our run above it does start performing at a level just above the random benchmark between the 6 -8 thousand episode mark.

If I was to guess at issues the implementation may still have I would target:

- Initialization values for the convolutional neural network
- Implement error clipping on the loss function (number 1 concern)
- Check the image processing (pretty confident it is not the cause, but you never know)

For the remainder of the report I will focus on the CartPole results and only briefly reference the Atari implementation as I don't believe at this point there is too much more to be gained looking at the Atari implementation. On a final note I ultimately stopped tinkering with the Atari implementation as having to run it on a remote GPU service was becoming quite expensive, when you consider the implementations in the original deepmind paper were run for a week each, which costs about $67 and when you do multiple test runs that price goes up quite quickly.

Justification

As seen in Fig: 7 and Fig: 8 we soundly beat the random action benchmark for the CartPole Problem. See below for a table comparison of the results:

Fig 10: Benchmark Comparison.

	CartPole-v0	**Breakout-v0**
Random Action Benchmark	24.43 +/- 1.51	1.60 +/- 0.16
DQN	196.32 +/- 1.35	1.92 ± 0.17
DDQN	197.69 +/- 1.11	2.11 +/- 0.16

As we can see in the above table all results with the exception of the DQN Atari implementation beat the random action benchmark with statistical significance. However I would still consider the Breakout results debatable. What is not debatable however, is our implementation on the CartPole environment which exceeded my expectations.

V. Conclusion

Free-Form Visualization

For the free-form visualization I decided to take a look at the differences on the CartPole environment between my DQN and DDQN implementation. Specifically how the Q-value estimation differs between them. To gather the data for this I implemented a write to CSV section which records the last loss and Q-value estimate received during the memory replay for each episode. I did not log the Q-value estimate for every single step of the memory replay as that would have resulted in millions of datapoints which would be slow to analyse and not provide any more useful information.

The results below show a very interesting comparison and gets to the heart of the difference between DQN & DDQN. In figure 11, the DQN implementation you can see the telltale hump between the 127 & 190 episode mark, this is inevitably found in almost all simple DQN implementations and represents the over estimation of the Q-value and then correction. If you refer to figure 12 the DDQN implementation you can see that does not exist. This is because the decoupling of the Q-value estimation from the action selection helps reduce the Q-value over estimation. In addition, it adds some stability to the estimated Q-value which does not vary as much in the DDQN implementation.

Fig 11: DQN Q-value Estimate

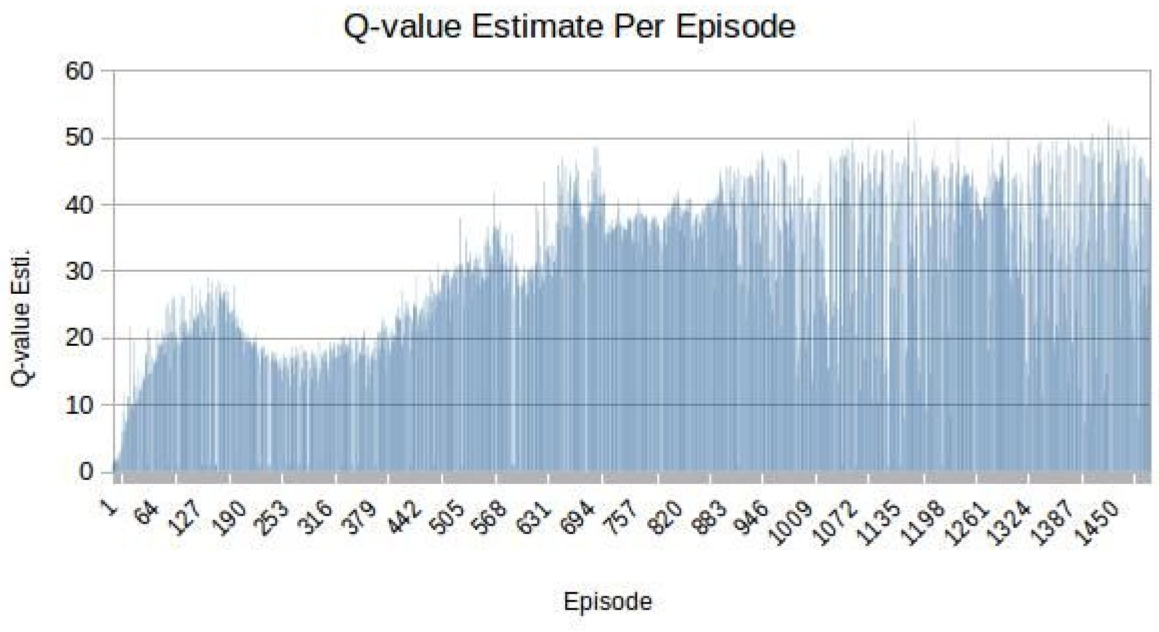
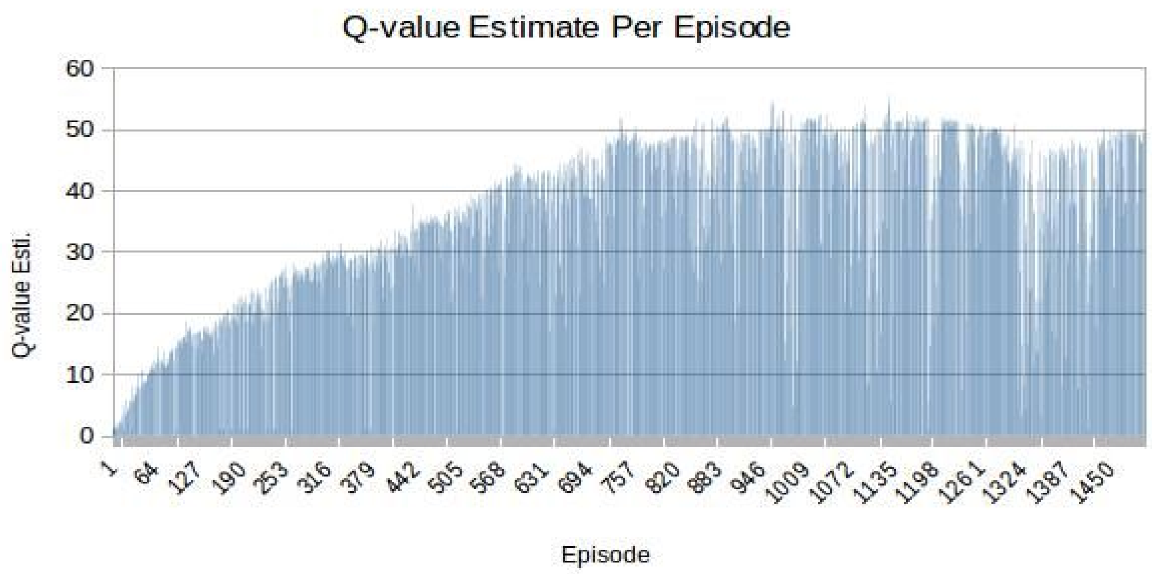


Fig 12: DDQN Q-value Estimate



Reflection

In this project I implemented a Deep Q-learning Algorithm successfully on the CartPole-v0 environment. The steps taken can roughly be summarised as follows:

- Install the OpenAI gym environment & dependencies (Not trivial)
- Develop a random action test implementation to ensure system is working
- Create a random action benchmark for CartPole & Breakout
- Build upon the random benchmark to implement Q-learning in CartPole
- Tinker till happy with results
- Transition to Breakout, implement image processing & convolutional NN
- Implement DDQN in both environments
- Realise the need for more detailed results, collect loss & estimated Q-value info
- Run long term Breakout tests, realise how much it costs, stop tests.
- Analyse final results against the benchmarks.

I have really enjoyed this project and learnt a lot. My understanding of Q-learning and image processing has improved greatly. The most difficult parts of the project were implementing the Memory Replay function and the image processing required, as that was completely new to me. The most frustrating part was ultimately the 100+ runs executed in the Atari environment without being able to get a good result, and yet feeling very close to success the whole time (I still have the urge to go back into the code and keep tinkering, expecting it all to fall into place).

Improvement

The most obvious improvement I would make would be to implement a loss clipping function to clip the training errors to between +1 & -1. I think this would vastly improve the stability for both environments and possibly solve the issues on the Atari environment. I believe this would solves these problems because when i have analysed the losses at each training step every once in awhile the loss is huge and greatly changes the model. Limiting the loss to always only making small changes should fix this.

Why haven't I implemented it yet?

Currently this goes slightly beyond my understanding of how to do it correctly. I have read various analyses of changing loss functions and implementing loss clipping and I have found that it is easy to do wrong. It is easy to get into a situation where the loss beyond a specific point ends up always being zero or affecting your activation function of the nodes in the neural network, to the point where the node can effectively be killed off or given what researchers term “brain death”

References

I have chosen not to to relist all the references listed throughout the report, Instead I will cite resources which helped me with my implementation.

Cited Works

Deep Reinforcement Learning: Pong from Pixels. Web. 02 Sept. 2017.

"Ben Lau." *Using Keras and Deep Q-Network to Play FlappyBird | Ben Lau*. Web. 02 Sept. 2017.

"Deep Q-Learning with Keras and Gym." *Deep Q-Learning with Keras and Gym · Keon's Blog*. Web. 02 Sept. 2017.

"Guest Post (Part I): Demystifying Deep Reinforcement Learning." *Intel Nervana*. 22 June 2017. Web. 02 Sept. 2017.

Juliani, Arthur. "Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond." *Medium*. Medium, 02 Sept. 2016. Web. 02 Sept. 2017.

Parthasarathy, Dhruv. "Write an AI to Win at Pong from Scratch with Reinforcement Learning." *Medium*. Medium, 25 Sept. 2016. Web. 02 Sept. 2017.

"Q-Learning .*;." *A Painless Q-Learning Tutorial*. Web. 02 Sept. 2017.