Project

CS4246-AI Planning and Decision Making

Kritartha Ghosh	**Manasa Kashyap**	**Meenakshi Sundaram Viswanathan**
A0191466X	A0178462W	A0191324L

Abstract

Games have been an integral part of our lives for as long as we know and playing games have shown to have a lot of advantages such as development of imagination, improve cognitive, physical and emotional strengths together with helping people relax. Atari games have been around for more than a decade and the advancement in the field of artificial intelligence has encouraged the humans to try to apply them to various fields. In this project, we aim to build an agent which can play a game that surpasses human performance by exploring the use of various AI techniques such as reinforcement learning and neural networks.

1. Introduction

The exponentially increasing power of the computers has left the mankind hungry to explore fields where the power of computers can be put to good use. One such area is that of game playing. Recently, games have been the popular go-to choice to explore the applications of artificial intelligence. Most of these agents are known to make use of another powerful area, neural networks. With the advances in neural networks, the performance of some of these programs have been noted to surpass the performance of a human expert.

1.1 Learning Environment

In this project, we will be exploring a variety of reinforcement learning techniques on the Pixel Copter and Flappy Bird environment provided by OpenAI and PLE. PLE is a learning environment much like the Arcade Learning Environment Interface, which allows a fast implementation of reinforcement learning in Python. OpenAI Gym is a toolkit for developing and comparing reinforcement algorithms. In order to perform reinforcement learning, we would require observation state space, action space and reward which is luckily provided by the environment.

2. Related Work

In the year 1992, TD-Gammon[1] program was developed and it was a major milestone in the success of reinforcement learning. It was a program which learnt to play backgammon game by itself and managed to surpass human level performance. TD-gammon used a model-free learning algorithm and estimated the value function using a multi-layer perceptron with one hidden layer. Soon after that, many tried to use the same method to other games but were unsuccessful in achieving such results. Later, it was shown that combining model-free reinforcement algorithms with non-linear approximators or with any off-learning policy was causing the Q network to diverge. Lately, there has been a growing interest in using deep learning methods together with reinforcement learning. DeepMind technologies introduced the first deep learning model to successfully learn control policies directly from images using reinforcement learning. They used a convolutional neural network to train seven 2600 Atari games from Arcade Learning Environment with no adjustment of architecture or learning algorithm. They achieved state-of-the-art results in six out of seven games on which they tested the model[2].

Google DeepMind team later made a few improvements to DQN and presented it as DDQN(DDQN). DDQN reduced overestimation by decomposing the maximum operation in target into 2 steps: action selection and action evaluation. DDQN performed significantly better than DQN. The following results are noteworthy: the performance of Road Runner improved from 233% to 617%, performance of Asterix improved from 70% to 180% and Double Dunk's performance improved from 17% to 397%. Following these successes, there has been a lot of experimentation with trying different network architectures to obtain better performances.

3. Game Environment

PLE provides the game environment for the training and the testing. In order to use the interface, we have to first create the game instance, reset the environment to the initial state and then apply an action to generate the successor state. The input to the model is an action value which is an integer in the range of 0 and number of actions and it outputs a successor state in the form of an image. We also get a reward value from the transition together with a variable that indicates if the episode has terminated. The observation space is a matrix of (num_of_row_pixels x num_of_column_pixels x 3) which are the pixel values of a particular frame in the game.

Flappy Bird is a game wherein a user has to navigate a bird through the game environment trying to avoid obstacles in the form of vertical pipes using only one of the two options: Up or do nothing which by default will make the agent move down. Each pipe it passes through, a reward of +1 is gained whereas coming in contact with obstacles results in a penalty of 1 and the episode getting terminated.

Pixel copter is a side-scrolling game where in one has to navigate a pixel through a series of hurdles which are basically vertical blocks. The Up arrow causes the pixel to accelerate upwards and doing nothing will make it accelerate downwards respectively. Scoring is based on the number of vertical blocks that the player manages to pass through successfully. The game is over when the pixel comes in contact with any part of the hurdles. An image from a frame generated by the PLE is shown below in figure 1.

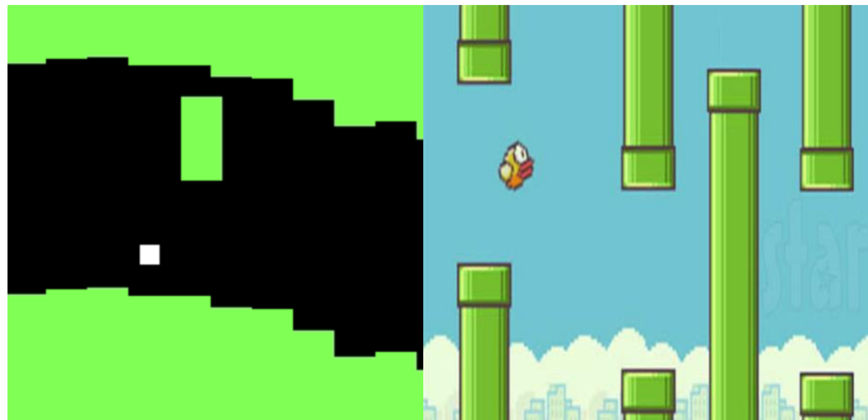

Fig 1. Frame of Pixel Copter and Flappy Bird

4. Pre-processing

The image from the environment is pre-processed in order to format before using it as an input to model. Firstly, the image is converted into greyscale as shown in Fig 2 and Fig3. This reduces the number of the channels of each image from 3 to 1. This is done because the colour does not play a major role in the action decision process and hence, converting it to greyscale reduces the size and allows the agent to learn faster. After which, a reduction process is applied and this reduces the image resolution. For Flappy Bird the resolution of original image was reduced by a fourth However, this was only applied for Flappy Bird as the resolution of the Pixel Copter is already low and reducing it further would have resulted in an extremely small image.

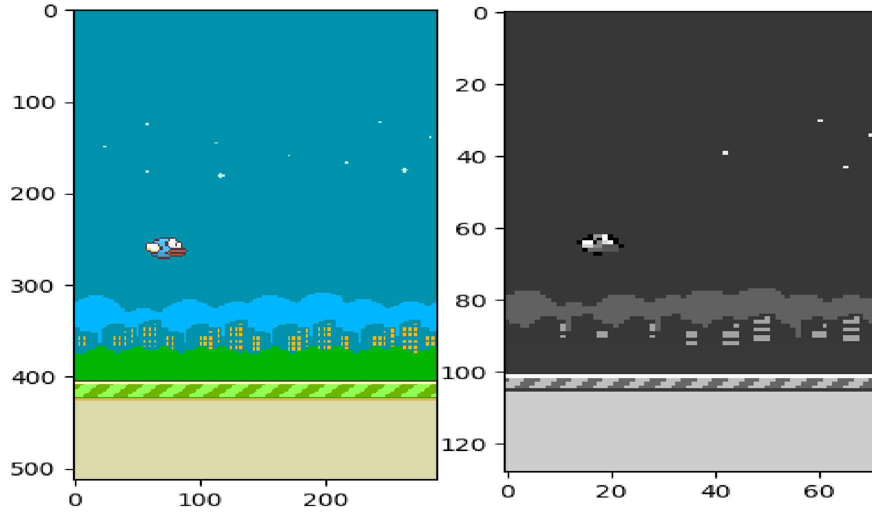

Fig 2. Images before and after image processing for Flappy Bird

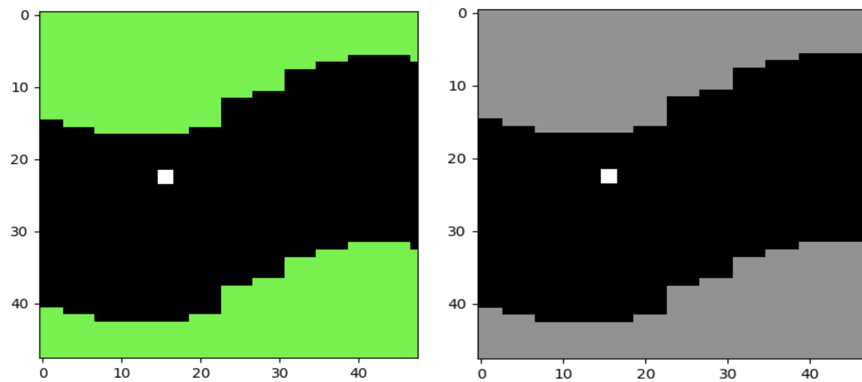

Fig 3. Images before and after image processing for Pixel Copter

4. Methods

4.1 Convolutional Neural Network Model

The implementation of DQN is done using the Keras library and we used a Google Cloud Engine with 8 core CPUs and a RAM of 52GB. The network architecture used in this project is the same as the one described in [2]. We used 3 convolution layers with ReLU activation function which is connected to 2 fully connected layers. The fully connected layers are separated by ReLU function just like in the convolution layers. The dimension of the final layer is equal to the number of valid actions allowed. In addition, we initialise replay memory to 50,000 observations. At the beginning of training, the replay memory is populated by choosing random actions for 10,000 steps. However, during this training step, the weights of the network is not updated. Training is only started once the replay memory buffer is almost filled. We have used the Adam optimization algorithm with a learning rate of 1e-5.

4.2 Deep Q Network

The Q function is approximated with a neural network that takes a state and action as input and outputs a corresponding Q value. Input to the network are 4 stacked frames of greyscale game screens. This is done to capture motion as a single frame will tell us nothing about the direction in which the bird or pixel moves. The Q value is updated as shown below:

$$Q[s,a] = Q[s, a] + \alpha(\text{frequency of } (s,a))(r + \gamma \max Q[s',a'] - Q[s, a])$$

The learning rate is given by α which controls the difference between newly computed Q-value and the previous one. The update equation exploits Bellman update equation.

In the Deep Q network, the Q values are updated slightly differently. Here, a forward pass is made on the current state s' to get the predicted Q-value for all the possible actions, then the same pass is made for the next state s1 and maxQ(s,a) is calculated. After this, the Q value for the action is calculated using γ maxQ[s',a'] while all other values are set to 0. The weights of the network are then updated using backpropagation.

4.3 Replay Memory

Replay memory can be thought of as supervised learning and it simplifies the testing. It ensures that the network converges and makes the system more stable as using experience replay, the behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. During the game play, all the experiences consisting of state, action, reward, next state are stored in a replay memory. The replay is written in a way that random mini batches are sampled from the replay memory and this helps the network converge to a local minimum.

4.4 Exploration vs Exploitation trade-off

There has always been a trade-off between exploration vs exploitation. Finding the perfect balance between them is what gives the optimal solution. Initially, the predictions of Q values are random due to the random initialisation of the network. During exploration, agent tries various actions and observe the return rewards whereas in exploitation, the agent carries out the action that maximises its reward. As the agent starts to prefer exploitation to exploration, the Q function starts to return more consistent values. In order to exploit the trade-off, a hyperparameter ε which is a probability that chooses between exploitation or exploration, initially set to 1. ε decreases over time at the rate of 9e-7. In the beginning, the agent chooses random actions to explore the state space as much as it can and then it settles down to a fixed exploration rate of 0.1.

Pseudocode DQN

```
Initialise Q with random weights
Reset the replay memory
Input initial state s
While(num_of_episodes) != 0
    While(!terminated)
        With probability ε select random action or compute a = argmaxQ(s,a)
        Get the reward r and state s'
        Store the experience history in replay memory
        Sample random transitions from replay memory
        Calculate output y for each minibatch transition
            If the next state is a terminal one then
                   y = reward
            Else
                   y = reward + gamma *maxQ(new state, new action)
            Calculate loss
Backpropagate the weights
```

5. Results

The agent was trained for a large number of episodes with a learning rate of 0.00025.

5.1 Pixel Copter

Pixel Copter was trained for a total of 36 hours. During training, we observed that the agent was able to play the game for a longer time and also that the scores increased with the number of episodes. During the course of evaluation, the agent was able to achieve an average score of -5 over 156 episodes if we use a random agent that is and it was able to run for about 100 steps, though some peaks were observed in between. It also obtained a maximum score of 30 and trained for about 200 steps. Training it for a longer time, might yield better results. Fig. 4 and 5 show the results for training and evaluation respectively.

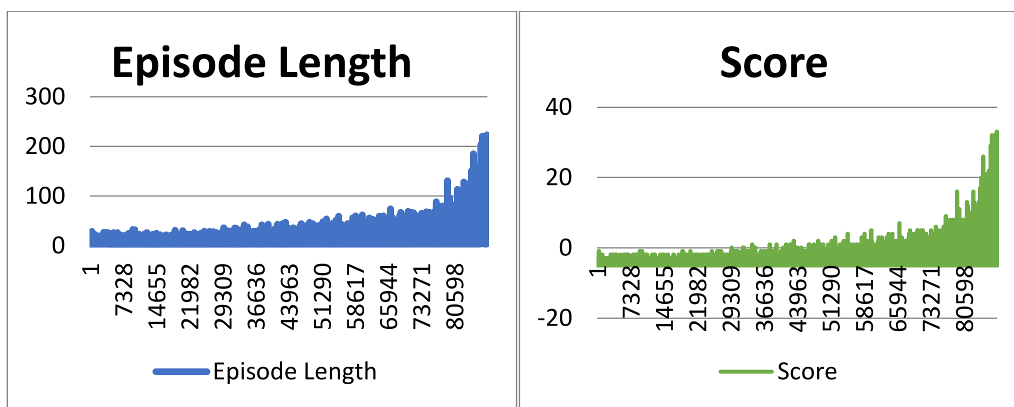

Fig 4. Graph of episode number vs the length of episode(left) and score obtained for each episode(right) during training for Pixel Copter

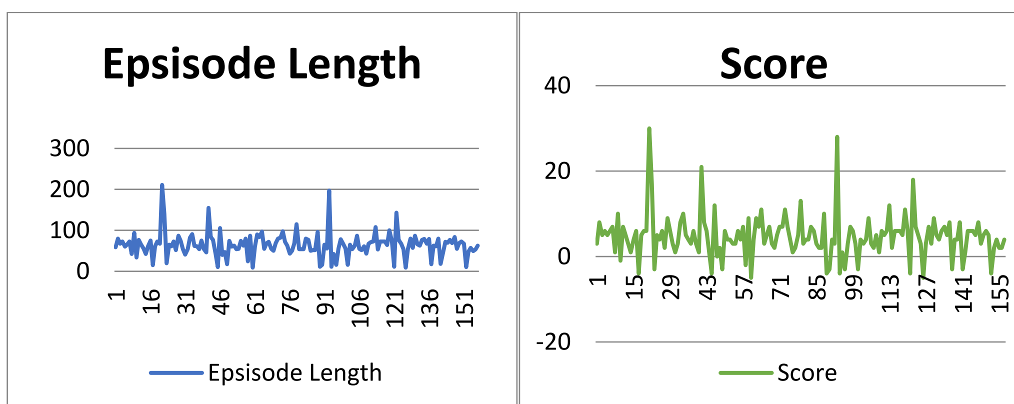

Fig 5. Graph of episode number vs the length of episode(left) and score obtained for each episode(right) during evaluation for Pixel Copter

5.2 Flappy Bird

Flappy Bird was trained for over 24 hours on the Cloud Virtual Machine and we were able to train a model using DQN to learn Flappy Bird with a non-zero performance. We trained the model over 9011 episodes. During training, we observed that with the score obtained and the amount of time for which each episode lasted was almost consistent without any improvement. This goes on to show that the it hasn't converged and tuning the hyperparameters will provide better results. During the evaluation phase, we observed that the agent for Flappy Bird learns to cross the pipes only when the opening is at a certain height. From this we deduced that it needs more exploration and tuning the hyperparameter ε will result in better performance. Figure 6 and 7 shows the performance of the model.

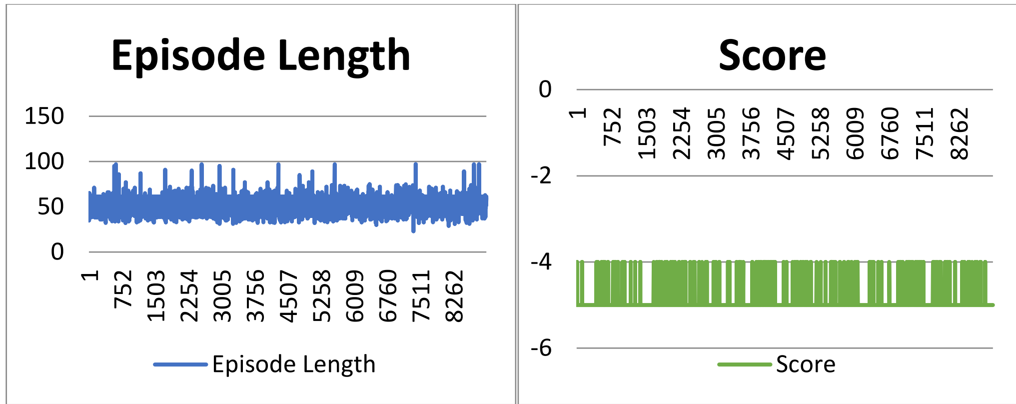

Fig 6. Graph of episode number vs the length of episode(left) and score obtained for each episode(right) during training for Flappy Bird

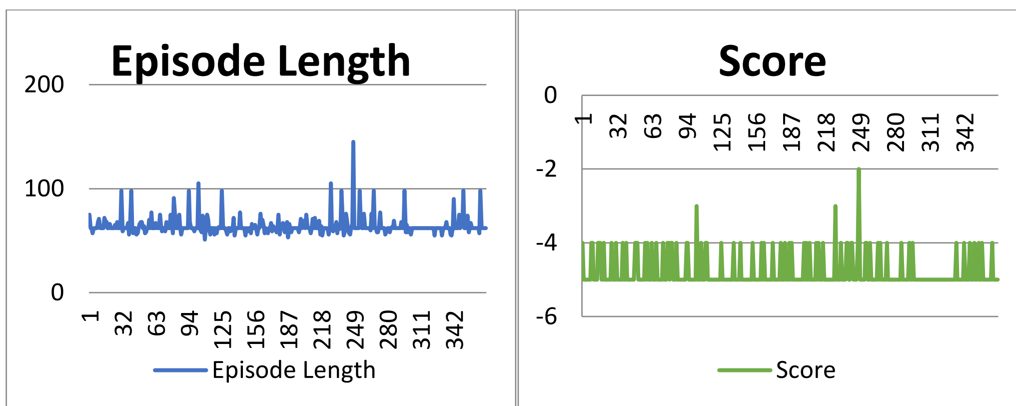

Fig 7. Graph of episode number vs the length of episode(left) and score obtained for each episode(right) during evaluation for Flappy Bird

6. Conclusion

The results can be significantly improved if the model was trained for a longer time and for a larger number of episodes. Performance of DDQN is much better on Pixel Copter as compared to Flappy Bird. In order to improve its performance, we can try to tune the hyperparameters such as learning rate and the exploration-exploitation trade-off variable as well as letting it train for more episodes. In addition, we can also try algorithms like Asynchronous Actor-Critic to extract better performance.

References

[1] G. Tesauro, "Temporal Difference Learning and TD-Gammon", in Commun. ACM, vol. 38, no. 3, pp. 55-68, March 1995.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", arXiv:1312.5602v1, December 2013.