Copter Bot Based on Deep Q-Learning

Harivignesh Rajaram
CU Boulder
hara3180@colorado.edu

Pranav Kumar Sivakumar
CU Boulder
prsi6835@colorado.edu

Saikrishna Jaliparthy
CU Boulder
saja8821@colorado.edu

Abstract

Game Playing can easily be automated using many general Reinforcement Learning based AI techniques, which caused us to explore many deep learning related models that works with similar analogies but with better performance than the former ones. In this project we implement an all-encompassing Deep-learning model which has its roots in the Q-learning that can be used to well automate any game (if trained on them), working on a Markov Decision process. While reinforcement learning agents have achieved some successes in a variety of domains ,their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. Our goal here is to develop a generic Deep Q-learning model on the PixelCopter game- a simple and pixelated version of the classic Copter game.

1 Introduction

Game bots are a popular means of exploring the application of Reinforcement learning. The paradigm of learning by trial-and-error, solely from rewards or punishments, is known as reinforcement learning (RL). With the inclusion of Deep learning concepts, the performance of automated game playing programs has seen much advancement that can go beyond the ability of even expert level human players.

Copter is an interesting classic game to work on with RL since it doesn't have a final goal state to achieve rather just the high score from staying alive for a longer time. It also provides us with the opportunity to upskill our Machine Learning knowledge and experience.

Game bots for this game can either be built using supervised learning or by reinforcement learning. But, If we want to build a bot using supervised learning, we need to have hundreds of hours of gameplay videos of world's best players playing this game which is tagged with their corresponding move at a particular frame which can then be given as input for the model. Given that we have such a dataset and high powerful GPU machines this problem can be tackled as a supervised classification problem. But instead, we can easily train a program by directly interacting with the game environment using a trial and error approach. Therefore Reinforcement learning would be the best choice for this kind of problems. The main difference in labels among the above techniques is that in supervised learning We have a target label for each of the training example and in reinforcement learning, We would have sparse and time-delayed labels – the rewards. Based only on those rewards the agent (the Copter) has to learn to behave in the environment. The PixelCopter game window looks like Figure 3.

2 Related Work

Deep reinforcement learning has received considerable attention in recent years due to its potential to automate the design of representations in RL. Deep reinforcement learning and related methods have

.

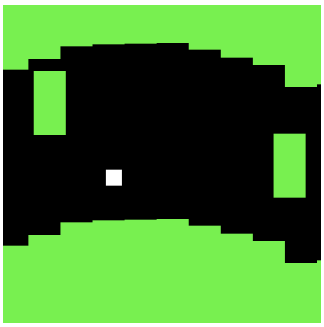

Figure 1: A game window.

been applied to learn policies to play Atari games [1][2] and perform a wide variety of simulated and real-world robotic control tasks[3][4] [5].

Perhaps one of the best-known success stories of reinforcement learning is TD-gammon, a backgammon playing program which learnt entirely by reinforcement learning and self-play, and achieved a superhuman level of play [6]. It was a model-free reinforcement learning which is similar to Q-learning, where the value function was approximated using a multi-layer perceptron with one hidden layer. However, other early attempts on TD-gammon, including applications of the same method to chess, Go and checkers were less successful. This led to a widespread belief that the TD-gammon approach was a special case that only worked in backgammon, perhaps because the stochasticity in the dice rolls helps explore the state space and also makes the value function particularly smooth.

Deep Q Network(DQN) combines Q-learning with a flexible deep neural network and was tested on a varied and large set of deterministic Atari 2600 games, reaching human-level performance on many games. In many ways, this setting is a best-case scenario for Q-learning, because the deep neural network provides flexible function approximation with the potential for a low asymptotic approximation error, and the determinism of the environments prevents the harmful effects of noise. We have implemented a similar kind of model in our project.

3 Data Format

3.1 Preprocessing

Game Environment is generated with the help of a python module named pygame and also utilizing the Pygame Learning Environment library. After getting the images from the environment in-order to make training faster, it is important to do some image processing. Firstly, images are converted from color image to gray scale followed by reshaping the image down to 80×80 pixels. Here the shape of a single frame is $1 \times 1 \times 80 \times 80$ Then consecutive 4 frames are stacked together and fed into the gonna-be proposed neural network and also the shape of a stacked frame is $1 \times 4 \times 80 \times 80$ which is essential for the TensorFlow-based development.

4 Model

4.1 Neural Network Model

In the model architecture shown in figure 2, the image stack of size $1 \times 4 \times 80 \times 80$ is given as input to the first convolution layer which contains 32 8×8 filters with subsample of size 4 are applied on it, the output from first convolution layer is of size $20 \times 20 \times 32$ is given as input to the second convolution layer which applies 64 4×4 filters with subsample of size 2 on it, it is tailed by the another convolution layer which applies 64 3×3 with subsample of size 1 the input to the third convolution layer is of size $10 \times 10 \times 64$ in which it output is of size $10 \times 10 \times 64$. All the three convolution layers have ReLU activations. The third convolution layer is followed by a dense layer with 512 rectifier units and the final output layer is a dense layer whose dimensions are equal to the number of validation actions in the game.In our case its two valid actions(Pressing a key or pressing nothing). Kindly refer to Figure 2 for elucidation.

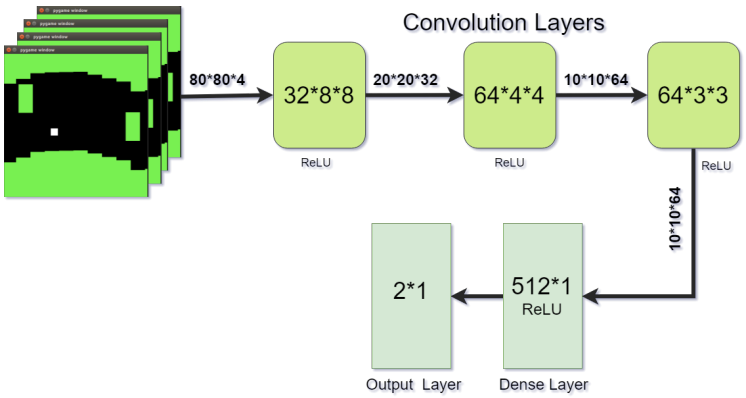

Figure 2: Model Architecture.

Algorithm 1 Deep Q-learning with the Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for $episode = 1, M$ **do**
    Initialize sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With Probability ϵ select a random action a_t
        otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $S_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi t + 1 = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = r_j$ for terminal ϕ_{j+1}
        Set $y_j = r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta)$ for non terminal ϕ_{j+1}
        Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \theta))^2$ according to the equation 3
    end for
end for

4.2 Deep Q-Learning

A reinforcement learning algorithm called Q-Learning is utilized as it has been proven that for any finite MDP, at a given state Q-learning can find an optimal policy such that the reward is maximum by taking into account the successive states. For our problem, the state is the current position of the copter and the obstacles.

The action is either pressing the key or not. We reward the agent for each successful move and punish it when it collides the border or an obstacle. With every game played, the bot observes the previous states and its actions. Over the outcomes, it punishes or rewards the state-action pairs. Q-learning is primarily represented by the Q-function, Q(s,a) which denotes the maximum discounted future reward when we perform action a in state s. It also indicates how good that action is. Suppose you are in state s and not sure which action to choose (a or b), then take the one which has the maximum Q- value obtained from the Q-function.

The Q-function is defined as: Q(s,a) = $r + \gamma \times max_{a'} \times Q(s', a')$

In plain English, it means maximum future reward for this state and action (s,a) is the immediate reward r plus maximum discounted future reward for the next state s', action a' (with gamma as the discount factor). The idea behind the Deep Q-learning is that we compress the Q-table(with all the Q-values) using the neural network and represent the Q- function using the weights. The output from our Convolution Neural network are the Q-values of each possible actions. In a way that the loss function of the network is modified as, $L = [r + \gamma max_{a'} Q(s', a') - Q(s, a)]^2$

A Deep Q-network is a multi-layered neural network that for a given state s outputs a vector of action values $Q(s, .; \theta)$, where θ are the parameters of the network. For an n-dimensional state space and an action space containing m actions, the neural network is a function from R^n to R^m. Two important ingredients of the DQN algorithm are the use of a target network, and the use of experience replay.

The full algorithm, which we call the Deep Q-learning is presented in Algorithm 1. This approach has several advantages over standard online Q-learning [7].First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency.Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically.

4.3 Experience Replay

For the Experience Replay [8] observed transitions are stored for some time and sampled uniformly from this memory bank to update the network. Both the target network and the experience replay dramatically improve the performance of the algorithm[2].

Based on the used neural network model architecture the Q-values obtained are not stable. So, in-order to overcome this problem we use the concept called the experience replay. During the game-play all the episodes (s, a, r, $s^{'}$) are stored in replay memory D. When training of the network, random mini-batches from the replay memory(D) are used instead of the most recent transition, which will greatly improve the stability.

4.4 Exploration Vs Exploitation

This is one more important concept to look into for reinforcement learning algorithm models. The questions are about the amount of agent's time that should be spent on exploiting the existing known-good policy and also the time that it should be focused on exploring new possibly better actions.

In order to maximize future reward, they need to balance the amount of time that they follow their current policy (a greedy approach), and the time they spend exploring new possibilities that might be better.A popular approach is called ϵ greedy approach. Based on this approach, the policy tells the agent to try a random action some percentage of the time, as defined by the variable ϵ (epsilon), which is a number between 0 and 1. The strategy will help the RL agent to occasionally try something new and see if we can achieve ultimate strategy.

5 Experiments and Results

Conventional Training(laptops) is not enough for developing this type of model. So we opted for using on- demand cloud computing services, AWS. We trained our model in an instance(p3.2xlarge) that has a NVIDIA Tesla V100 GPU (16GB), for 16 hours on about 2 million frames.

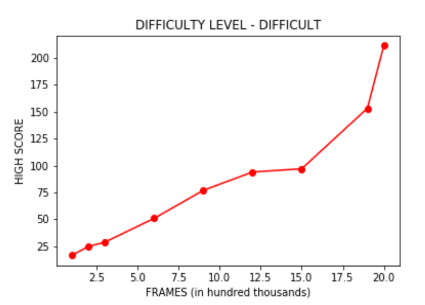

Figure 3: Scores Vs Frames

Based on the obstacle size , agent velocity, etc, the version of our game is considered hard. The below graph illustrates the increasing high-scores of the bot with the increase in the number of iterations during training in the difficult version. The weights stored are updated for every 1000 frames. The initial 3200 frames were observed with random actions only. Then in next 1 million frames, the model predicts the actions with the exploration co-efficient (ϵ) gradually decreasing from 0.01 to 0.0001. For further 1 million frames the ϵ is kept constant. Every time a mini-batch of tuples from the Replay Memory is extracted for the model to train. The learning rate, batch size and the Replay memory was 0.00001,32 and 50000 respectively.

From the graph we can observe that the until the 1.25 million frames high score increases at stead state. After that from 1.25 million the high score increases rapidly so that it reaches to high score of 211 at 2 million frames. So, from that we can say that as the number of training frames increases highscore also increases. As the accuracy of the model increases with the increase in the number of frames, it is less plausible to compute any accuracy metrics for validation. Also the game environment varies depending upon the settings set by the developer. Moreover, a different model trained on a similar training set on a similar environment is highly unlikely to be discovered, so we haven't made any benchmarks. But our algorithm is generic such that it can be implemented over any similar kind of game environment.

6 Conclusion

We observed that as the training frame count increases the model learns to achieve higher average scores.We have developed an efficient Copter bot using a Deep Q-Learning based CNN model. The developed model is generic in the sense that it can also be trained on many other games too. We could also try developing a similar model based on Double Deep Q-learning and verify whether it improves the score or not.

Acknowledgments

The major work involved in this project are finding and setting up the game environment, preprocessing, deciding and building the Deep Neural network model using Tensorflow, Q-learning based algorithmic development, finding the best parameters and finally, deploying the model in the (AWS) for faster training. Since we are a small team of only three members we had contributed to the project in equal proportions.

References

[1] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[2] 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, 2015.

[3] Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control. *Machine learning*, 84(1-2):137–169, 2011.

[4] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.

[5] Tim de Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. The importance of experience replay database composition in deep reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*, 2015.

[6] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[7] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[8] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.