Game Playing with Deep Q-Learning using OpenAI Gym

Robert Chuchro
chuchro3@stanford.edu

Deepak Gupta
dgupta9@stanford.edu

Abstract

Historically, designing game players requires domain-specific knowledge of the particular game to be integrated into the model for the game playing program. This leads to a program that can only learn to play a single particular game successfully. In this project, we explore the use of general AI techniques, namely reinforcement learning and neural networks, in order to architect a model which can be trained on more than one game. Our goal is to progress from a simple convolutional neural network with a couple fully connected layers, to experimenting with more complex additions, such as deeper layers or recurrent neural networks.

1. Introduction

Game playing has recently emerged as a popular playground for exploring the application of artificial intelligence theory. With the addition of convolutional neural networks, performance of game playing programs has seen improvement that can even go beyond the ability of even expert level human players. This demonstrates the powerful learning capabilities of computers in a variety of challenging environments. Designing a model which is agnostic to its environment allows us to investigate a core problem of artificial intelligence, which is the concept of general intelligence. The benefit to interfacing with OpenAI Gym is that it is an actively developed interface which is adding more environments and features useful for training. One of the core challenges with computer vision is obtaining enough data to properly train a neural network, and OpenAI Gym provides a clean interface with dozens of different environments.

1.1. Learning Environment

In this project, we will be exploring reinforcement learning on a variety of OpenAI Gym environments (G. Brockman et al., 2016). OpenAI Gym is an interface which provides various environments which simulate reinforcement learning problems. Specifically, each environment has an observation state space, an action space to interact with the environment to transition between states, and a reward associated with performing a particular action in a given state. This information is fundamental to any reinforcement learning problem.

The input to our model will be a sequence of pixel images as arrays (Width x Height x 3) generated by a particular OpenAI Gym environment. We then use a Deep Q-Network to output a action from the action space of the game. The output that the model will learn is an action from the environments action space in order to maximize future reward from a given state. In this paper, we explore using a neural network with multiple convolutional layers as our model.

2. Related Work

The best known success story of classical reinforcement learning is TD-gammon, a backgammon playing program which learned entirely by reinforcement learning [6]. TD-gammon used a model-free reinforcement learning algorithm similar to Q-learning. Attempt to explore on success of TD-gammon were less successful like applying the same techniques on Go and checkers as it was later concluded that stochasticity in the dice rolls helps exploring state space and makes value function particularly smooth [7].

The core of the problem is to apply non linear function approximation such as a neural network to reinforcement learning problem. Traditionally reinforcement learning problems tend to be unstable or diverge when a non linear function is applied to represent the action value aka Q value[1].Learning to control agents directly from high dimensional sensory inputs like vision is one of the long standing challenges of reinforcement learning[2]. Most of the recent algorithms and works that have used deep learning models to approximate Q function for RL exercise has been made possible due to recent breakthroughs in computer vision [3,4,5].First deep learning model to tackle reinforcement learning problem was introduced by DeepMind Technologies [2] to successfully learn control policies directly from image input using reinforcement learning. They used a convolutional neural network to train seven Atari 2600 games from Arcade Learning Environment with no adjustment of architecture or learning algorithm. They were able to achieve state-of-the-art results in six of the seven games it was tested on, with no adjustment of the architecture or hyperparameters.

Following the success of DQN, the Google DeepMind team experimented with applying deep convolution network to Double Q-learning algorithm [8,9], the motivation being standard Deep Q-learning algorithms were known to overestimate values under certain conditions. If overestimations are not uniform they can negatively impact the quality of resulting policy [10].Double DQN builds on the idea of Double Q-learning which aims at reducing overestimations by decomposing the max operation in the target into action selection and action evaluation [8]. Google DeepMind team concluded that DDQN clearly improves over DQN. Noteworthy examples include Road Runner (from 233% to 617%), Asterix (from 70% to 180%), Zaxxon (from 54% to 111%), and Double Dunk (from 17% to 397%). DDQN also estimated to show low variance in results than the corresponding DQN model.

There has been experimentation with different network architectures within convolution network domain such as Dueling Networks which contain two separate estimators: one for the state value function and one for the state-dependent action advantage function.

This helps in generalizing learning across actions without imposing any change to the underlying reinforcement learning algorithm[11].Google DeepMind team experimented with applying dueling networks on DDQN with prioritized experience replay [12] and uniform networks without any prioritized experience replay buffer.They evaluated dueling network architecture on 57 Atari games and found prioritized dueling agent performs significantly better than both the prioritized baseline agent and the dueling agent alone Recent works by Google DeepMind involves asynchronous variant of action critic model which surpasses current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU [13].Asynchronous variant of actor critic model finds inspiration from General Reinforcement Learning Architecture (Gorila) and some earlier work which studied convergence properties of Q learning in the asynchronous optimization setting [15].

3. Data Format

3.1. Interacting with Environment

All of the data used for training our model comes from interacting with the OpenAI Gym Interface (CITE). Interacting with the Gym interface has three main steps: registering the desired game with Gym, resetting the environment to get the initial state, then applying a step on the environment to generate a successor state.

The input which is required to step in the environment is an action value. More specifically, it is an integer ranging from $[0, num_actions)$. The information that we receive back is the successor state, represented as an image array of form (Width x Height x 3), where the third dimension is the RGB values at each pixel. In addition, we receive a reward value from the transition as well as an indicator to notify if the particular episode has terminated.

In the case of Flappy Bird, the observation space is a matrix of (288x512x3) representing the pixel values of the image of a particular frame in the game. Upon passing in between a set of pipes, a reward of 1 is received. If Flappy Bird collides with a pipe, the reward is -5 and the episode ends. A reward of 0 is given in all other states. The action space that we will learn our policy on is just two actions, 0 or 1. A sample image from a frame generated by the OpenAi Gym environment is shown in figure [1].

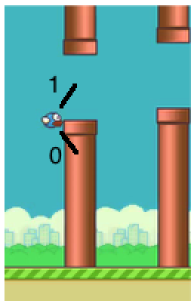


Figure 1. Sample frame of Flappy Bird. Action of 1 causes the bird to go up, while 0 allows gravity to drag it down.

3.2. Input Preprocessing

Once we receive the image from the Gym environment, we apply a couple steps to format the image to finally use an input to the model. First, we convert the image to gray scale, which should reduce discrepancies between episodes with different background settings (night/day, pipe/bird color). It also reduces the size of the third dimension from 3 for RGB to 1. Next, we De-noise the image using adaptive thresholding. In Flappy Bird, the background contains some unnecessary pixels that can create some noise for the neural network, such as stars in the night background. Applying adaptive Gaussian thresholding (CITE) incorporates each pixel's neighboring values to determine whether its value is noise or not. This same algorithm generalizes well format other games, as shown in figure [2].

Afterwards, we normalize values to be between 0 and 1 and reduce the image to 80x80 pixels. This allows for consistent input sizes across games as well as decreasing dimensionality of each layer of our network, allowing for faster passes and fewer parameters per layer. Finally, we stack last 4 frames as input to network (CITE) to form the 80x80x4 input to the network. For the reward values, we clip all incoming rewards such that positive rewards are all 1, negative rewards are -1, and 0 rewards remain 0. This

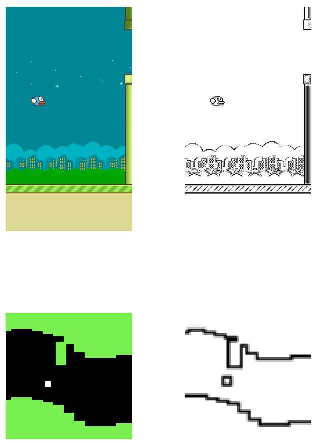
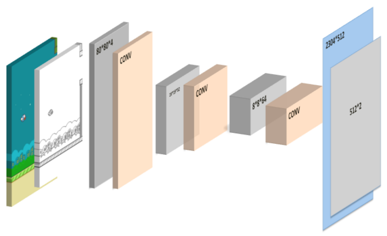


Figure 2. Image preprocessing for Flappy Bird (top) and Pixel Copter (bottom)



Network Architecture				
Type	Classes / Filters	Filter Size	Stride	Activation
Conv-1	32	8x8	4	ReLU
Conv-2	64	4x4	2	ReLU
Conv-3	64	3x3	1	ReLU
Fully Connected-1	512			ReLU
Fully Connected-2	# of actions			Linear

Figure 3. Details of layers of our current network, based off the DeepMind DQN model (V. Mnih et al., 2013), but with a larger number of classes / filters in each layer

should remove the need for tuning learning rates across different games.

4. Methods

4.1. Neural Network Model

Our implementation of DQN is using the Tensor Flow library (H. M. Abadi et al., 2016), and our models are running on Google Cloud Compute Engine with 8 cores and an Nvidia Tesla K80 GPU. The Network architecture is outlined in figure [4]. We are using 3 convolution layers with ReLU non-linearity activations, followed by 2 fully connected layers. Fully connected layers are separated by ReLU function similar to convolution layers. The final output layer dimension is equal to the number of valid actions allowed in the game. The values at this output layer represent the Q function given the input state for each valid action. At startup, we initialize all the weight matrices using normal distribution with a standard deviation of .01. We also initialize replay memory to 50,000 observations. At beginning of training, we first populate the replay memory by choosing random actions for 10,000 steps and we are not updating network weights during this preliminary training step. Once replay memory buffer is partially filled, we start training. We use Tensor Flows Adam optimization algorithm with an initial learning rate of 1e-5.

4.2. Deep Q Network

Neural networks are exceptionally good for learning features for highly structured data. Our Q function can be represented with a neural network that takes the state (last four game screens stacked) and action as input and outputs the corresponding Q-value. This approach has the advantage where if we want to perform Q value update or pick an action with highest Q-value we just need to do one forward pass through the network and have all Q values for all the actions immediately. Input to the network are 4 stacked frames of 80x80 gray scale game screens. Output of the network are Q-values for each possible action (2 for Flappy Bird and Pixel Copter). Q-values are continuous values which makes it a regression task that can either be optimized by using L2 loss, equation [] or Huber loss, equation [].

$$Loss = (reward + maxQ(s', a') - Q(s, a))^2 \qquad (1)$$

$$Loss = (reward + maxQ(s', a') - Q(s, a))^2 \qquad (2)$$

Pseudo code for Q learning algorithm is as follows

```
initialize Q[num_states, num_actions]
observe initial state s
while condition
  select and carry out an action a
  observe reward r and new state s'
  $Q[s, a] = Q[s, a] + \alpha(r + \gamma maxQ[s', a'] - Q[s, a])$
  s=s'
```

α is the learning rate that controls how much is difference between previous Q-value and newly proposed Q-value. When α is equal to 1 then update equation is similar to Bellman equation.

For Deep Q network given a transition $< s, a, r, s' >$, the Q-table update rule can be re written as:

- Do a feedforward pass for the current state s to get predicted Q-values for all actions.
- Do a feedforward pass for the next state s and calculate maximum overall network outputs $maxQ(s, a)$.
- Set Q-value target for action to $r + \gamma * maxQ(s, a)$. For all other actions make output equal to 0.
- Update the weights using back propagation (using gradient descent to optimize on this).

4.3. Replay Memory

This can be coined as the most important trick to get network to converge. Due to the stochastic nature of game play, approximating Q values using non linear functions is not very stable. Replay memory is one of the tricks that helps in converging the network and making learning stable. During game play, all the experiences $< s, a, r, s >$ are stored in a replay memory of fixed size D (a hyperparameter). During training, random mini batches from replay memory are sampled and this helps in breaking the similarity of continuous training samples and helps drive the network towards a local minimum. Replay memory makes training process similar to supervised learning and simplifies debugging and testing of algorithm

4.4. Exploration vs Exploitation

At the beginning of the training cycle predictions of Q network are random due to random initialization of the network.Agent tends to perform exploration in which it tries various actions and observe rewards for these actions.As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. Q-learning incorporates the exploration as part of the algorithm. But this exploration is greedy, it settles with the first effective strategy it finds. A simple and effective fix to this problem is to introduce a hyperparameter epsilon which determines the probability to choose between exploration or exploitation. [1] actually decreases over time from 1 to 0.1 in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate [1].

Now we can write Deep Q-learning algorithm as :

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
for episode in 1:M
  reset open gym environment
  while episode is not done :
    select an action a
    with probability epsilon select a random action
    otherwise select a= argmax Q(s,a')
    carry out action a
    observe reward r and new state s'
    store experience <s,a,r,s'> in replay memory D

    sample random transitions <ss,aa,rr,ss'>
    from replay memory D
    calculate output y for each minibatch transition
      if next state ss' is end of episode then y=rr
      else y=rr + gamma * max Q(ss',aa')
    train Q network using (y -Q(ss,aa))^2
    s=s'
until we finish iterating M episodes
```

4.5. Target Network

Before we explain DDQN, it is better to understand the intuition behind target networks. A target network is used to generate the target-Q values that will be used to compute the loss for every action during training. Using one network for both estimations is not very stable as Q-networks values shift, and if we are using a constantly shifting set of values to adjust our network values,then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target networks weights are fixed, and only periodically updated to the primary Q-networks values. In this way training can proceed in a more stable manner [1].

4.6. Double DQN

Double DQN is based on Double Q learning algorithm and motivation behind this algorithm is that regular DQN often overestimates Q-values of potential actions to take in a given state. This over-estimation would not have been a problem if all the actions are overestimated but that is not always the case.In order to correct this authors of DDQN[9]proposed a simple trick : instead of taking the max over Q-values when computing the target-Q value for training step, use your primary network to chose an action, and target network to generate the target Q-value for that action

So equation of Q target can be written as :

$$y = reward + \gamma Q_t arget(s', argmax(Q(s', a'))) \quad (3)$$

Now we can write Double Deep Q-learning algorithm as:

```
initialize replay memory D
initialize action-value function Q with random weights
initialize action-value function for Q_target
observe initial state s
numIterations =0
for episode in 1:M
  reset open gym environment
  while episode is not done :
    select an action a
      with probability epsilon select a random action
      otherwise select a= argmax Q(s,a')
    carry out action a
    observe reward r and new state s'
    store experience <s,a,r,s'> in replay memory D

    sample random transitions <ss,aa,rr,ss'>
    in replay memory D
    calculate output y for each minibatch transition
      if next state ss' is end of episode then y=rr
      else
        $y = rr + gamma * maxQ_t arget(ss', argmax(Q(ss', aa')))$
    train Q network using $(y - Q(ss, aa))^2$
    s=s'
    increment numIterations
    if numIterations % targetNetworkUpdateThreshold :
      update target network weights equal to primary
      network until we finish iterating M episodes
```

5. Experiments

We measure the performance of a particular model using metrics provided by the OpenAI Gym interface, namely the average score over 100 episodes of a particular environment. We will explore different reinforcement learning techniques as well as experiment with the tuning of hyperparameters.

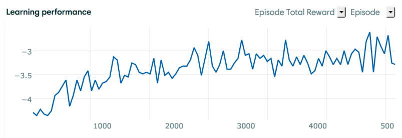


Figure 4. Flappy Bird scores when training a first epoch of 5000 episodes with standard exponentially decaying ϵ function

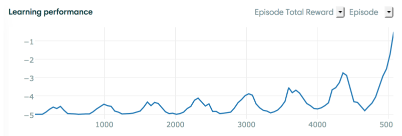


Figure 5. Flappy Bird scores when training a first epoch of 5000 episodes with our sinusoidal decaying function. Peaks continue to rise as training progresses, instead of getting stuck.

5.1. Sinusoidal Exploration Decay

Typically, reinforcement learning algorithms begin with a high exploration rate, and decay it as training progresses. The decay method is traditionally a linear or exponential decay. In our early trials, we found that after a certain number of iterations, training would progress would flatten, but then continue to improve when a new epoch was started. The most significant difference in our implementation when a new epoch begins is the reset of the exploration hyperparameter, ϵ. As shown in figure [], with an exponentially decaying ϵ, the episode score for our model levels out about half way through, essentially wasting half an epoch of training time. In contrast, figure [] shows our sinusoidal decay allowing the model to continue to improve across the entire epoch, finishing with a better top score.

We have introduced a new ϵ decaying function: one which exponentially decays over episodes in a sinusoidal fashion.

$$\epsilon = \epsilon_0 \cdot \epsilon_d^x \cdot \frac{1}{2}(1 + cos(\frac{2\pi xn}{X})) \tag{4}$$

- ϵ_0 is initial epsilon
- ϵ_d is decay rate
- n is number of mini epochs
- X is number of training episodes
- x is current training episode number

The main motivation behind this function was for our model to be able to escape local optima while training with an already reduced ϵ. Since it is difficult to model an ideal exploration rate for a particular environment, a decaying sinusoidal function makes less assumptions about the expected decay of ϵ, hopefully finding the correct decay rate periodically during the entire epoch. An analogy can be made about the suggested strategy. It focuses on the idea how one should use car brakes on a slippery road (without ABS): it is difficult for a driver to apply the brakes an optimal amount while maintaining rolling friction, but if the driver pumps the brakes repeatedly, they will achieve the correct amount of pressure periodically.

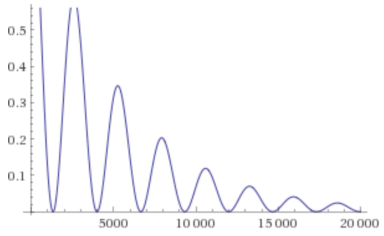


Figure 6. Plot of equation []. Save the model at every minimum, creating a mini-epoch. Parameters used: X=20000, n=7, ϵ_0=1, ϵ_d=.9998

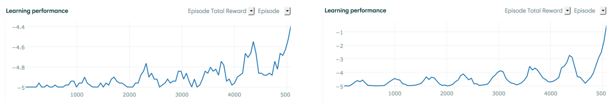


Figure 7. Preliminary OpenAi Gym score results for Flappy Bird using DQN (left) and DDQN (right)

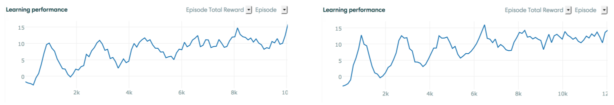


Figure 8. Preliminary OpenAi Gym score results for Pixel Copter using DQN (left) and DDQN (right)

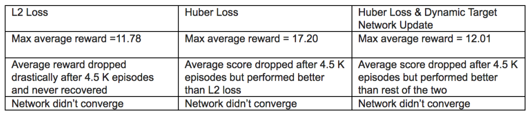
5.2. DQN vs DDQN

We ran 5000 training iterations of both Flappy Bird and Pixel Copter using DQN and then Double DQN. The results show an improvement in best average 100 episode score for Flappy Bird as shown in figure []. Interestingly, for Pixel Copter, there was very little difference in performance, as shown in figure []. These results offer evidence that DDQN is not a guaranteed improvement over DQN.

5.3. DDQN Variations

The following section describes details about three experiments for DDQN.

L2 Loss: In this experiment we tried to optimize L2 loss using Adam Optimizer Huber Loss: In this experiment we tried to optimize Huber Loss using Adam Optimizer Huber Loss and Dynamic Target Network Update: In this experiment we tried to optimize Huber Loss using Adam Optimizer. We also did dynamic target network updates. Instead of updating target network to primary network after every N constant iterations we instead updated target network dynamically using an experimental formula where intuition was to freeze target network for longer if the current



L2 Loss	Huber Loss	Huber Loss & Dynamic Target Network Update
Max average reward =11.78	Max average reward = 17.20	Max average reward = 12.01
Average reward dropped drastically after 4.5 K episodes and never recovered	Average score dropped after 4.5 K episodes but performed better than L2 loss	Average score dropped after 4.5 K episodes but performed better than rest of the two
Network didn't converge	Network didn't converge	Network didn't converge

Figure 9. Preliminary OpenAi Gym score results for Flappy Bird using DQN (left) and DDQN (right)

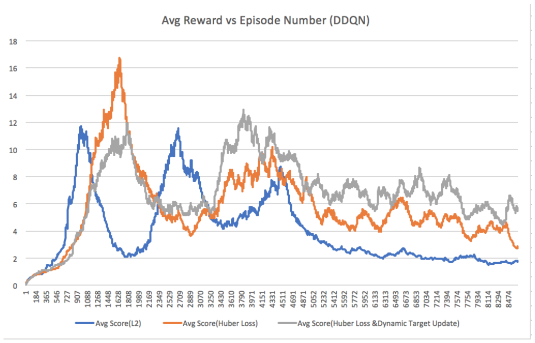


Figure 10. Preliminary OpenAi Gym score results for Flappy Bird using DQN (left) and DDQN (right)

network is achieving a higher score. The formula used is listed in equation [5].

$$updateTargetThresh = 750(avgReward + 1) + 9000 \tag{5}$$

6. Results

6.1. Flappy Bird

Ultimately, we were able to train a model using DQN to learn Flappy Bird with superhuman performance. This was accomplished with two training epochs totaling 18000 episodes and about 7 million iterations. Figure [] shows the OpenAI Gym submission of the second epoch, which is currently the third highest submission on OpenAI Gym with a best 100 episode average score of 62.26. This likely perform better with more training since each exploitation spike was continuing to increase. Figure [] shows the predicted Q-value of our model throughout the same training epoch, which follows a similar shape to the score graph. It is worth noting that even though the predicted Q-value was beginning to converge to about 8, the score of the actual game was continuing to increase.

6.2. Pixel Copter

Preliminary results for Pixel Copter using DQN have achieved a promising score of 16.89 with less than 2 million iterations, and is expected to surpass human performance.

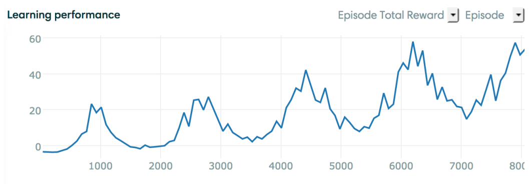


Figure 11. OpenAi Gym submission of score over about 7 million iterations

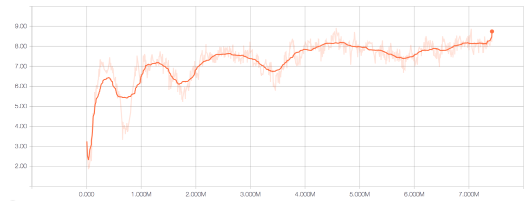


Figure 12. DQN predicted Q-value over the course of training

7. Conclusion

We were able to receive superhuman results on Flappy Bird using Deep Reinforcement Learning. We reduced training time by utilizing our new sinusoidal epsilon function. We can further reduce training time and improve performance by using Double Deep Q-Learning The first epoch of Double Deep Q-Learning for Flappy Bird shows a much smoother score curve over episodes, confirming the increased stability hypothesis, as well as improved performance.

Our model was able to transfer over to another game, Pixel Copter. Preliminary results are promising, can significantly improve if given the same amount of training time as our successful Flappy Bird model. Double Deep Q-Learning does not seem to offer as significant of an improvement in Pixel Copter compared to Flappy Bird. This may be due to the simplicity of the graphics in Pixel Copter.