Deep Reinforcement Learning for Atari games aided with human guidance

kshitiz@stanford.edu

***Abstract*— Apply Deep Reinforcement Learning techniques to train an agent to play Atari games in a generic manner and develop approach to let the agent be taught/guided by a human teacher.**

I. INTRODUCTION

There is a growing trend of combining deep neural network with reinforcement learning to solve problem with high dimensional space state. The Atari 2600 games provides an excellent simulation environment to train and test such generic RL algorithms. Previously these problems were dealt by extracting handcrafted feature sets and the performance of such system relied on the quality of feature representation.

This projects focuses on studying and implementing one of the current techniques in Reinforcement Learning, which is Policy Gradient to let an AI agent automatically learn to play Atari games. The second step would be to come up with an approach to let the agent be taught/guided by a human teacher to help it discover strategies that involves multiple steps.

II. RELATED WORK

The ralated work in this area is primarily done by Google Deepmind. Minh et al. initially solved the problem using a off policy method by training a deep Q network to evaluate the Q-function for Q-learning(Mnih et al., 2013[1]). They later presented Asynchronous Advantage Actor- Critic(A3C) algorithm(Mnih et al., 2013,[2]) which is a type of Policy Gradient method to solve the same problem with significant reduction in training time. Although Policy gradient algorithms have been part of Reinforcement learning coursework for a long time, but their implementation with deep neural network is definitely state of the art.

III. BACKGROUND

A. Reinforcement Learning

We consider the standard reinforcement learning setting where an agent interacts with an environment $\mathcal{E}$ over a number of discrete time steps. At each time step *t*, the agent receives a state s_t and selects an action a_t from some set of possible actions $\mathcal{A}$ according to its policy π, where π is a mapping from states s_t to actions a_t. In return, the agent receives the next state s_{t+1} and receives a scalar reward r_t. The process continues until the agent reaches a terminal state after which the process restarts. The return

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

is the total accumulated reward from time step t with discount factor $\gamma \in (0;1]$ which takes care of the **credit assignment problem**. The goal of the agent is to maximize the expected return from each state s_t.

Action value $\mathcal{Q}^\pi(s;a) = \mathbb{E}[R_t|s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The optimal value function $\mathcal{Q}^*(s,a) = max_\pi \mathcal{Q}^\pi(s,a)$ gives the maximum action value for state s and action a achievable by any policy. Similarly, the value of state s under policy is defined as $\boldsymbol{v^\pi(s)} = \mathbb{E}\left[\boldsymbol{R_t}|\boldsymbol{s_t} = \boldsymbol{s}\right]$ and is simply the expected return for following policy from state s_t.

B. Policy Gradient

Policy gradient(PG) is a policy-based method that directly learns a parameterized policy $\pi(a|s;\theta)$ without consulting the action value function,$\mathcal{Q}^\pi$. Note that some policy gradient method like Action-critic still uses value function to estimate policy weights, θ, but value function is not required for action selection. For discrete action space, like in the case of games, the natural choice of parametrization is parametrized numerical preferences, $h(s,a,\theta) \in \mathcal{R}$ for each state action pair and policy function is expressed as exponential softmax distribution:

$$\pi(a|s;\theta) = \frac{exp(h(s,a,\theta))}{\sum_{b\in\mathcal{A}} exp(h(s,b,\theta))} \tag{2}$$

During learning phase, sampling over the above probability distribution elegantly solves the **exploration-exploitation dilemma**. Action value paramterization method like DQN(Mnih et al., 2013,[1]) generally uses a *ϵ-greedy exploration* approach - with probability ϵ choose a random action, otherwise go with the greedy action with the highest Q-value. This means that policy parametrization method allows the possibility of approaching determinism (action preference, $h(s,a,\theta)$, for optimal action becoming infinitely higher than other actions.) while in action value method there is always an ϵ probability of choosing random action. Other advantages of policy gradient includes policy function being simple to approximate compared to action value function and policy parametrization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system.

The action preference, $h(s,a,\theta)$, is computed by **deep neural network** where θ is the vector of all connection weights of the network. The θ vector is updated using **backpropogation** starting with policy gradients computed using algorithm mentioned below.

C. REINFORCE: Monte Carlo Policy Gradient

TABLE I: REINFORCE: Monte Carlo Policy Gradient Algorithm

```
Algorithm: REINFORCE
Input: a differentiable policy parameterization $\pi(a|s;\theta), \forall a \in \mathcal{A}, s \in \mathcal{S}$
Initialize policy weights $\boldsymbol{\theta}$
repeat until terminated:
    Using $\pi(a|s, \boldsymbol{\theta})$, generate episode $s_0, a_0, r_1, s_1, a_1, ...a_{T-1}, r_T$
    repeat for each step of the episode t = 0,1,...T-1:
        $R_t \leftarrow$ discounted reward from step t
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha R_t \bigtriangledown_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta})$
    end
end
```

Policy gradient maximizes the value function(expected return) of the starting state, $\boldsymbol{\eta(\theta) = v^{\pi_\theta}(s_0)}$ with respect to the policy weights, $\boldsymbol{\theta}$. Using the policy gradient theorem, the REINFORCE(Williams J,[3],Sutton et al. [4]) algorithm gives the **policy gradients, $\boldsymbol{\bigtriangledown_\theta \eta(\theta)}$**. The stochastic gradient ascent follows: :

$$\bigtriangledown_\theta \eta(\theta) = R_t \bigtriangledown_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta}) \tag{3}$$

$$\boldsymbol{\theta_{t+1}} \leftarrow \boldsymbol{\theta_t} + \alpha \bigtriangledown_\theta \eta(\theta) \tag{4}$$

The algorithm is summarized in Table I. Appendix A computes the derivative of a softmax function a simple action preference, $h(s, a, \theta)$, linear in feature.

D. Human Guidance

Advance video games involve strategies involving multiple steps. For example, in Mario, in order to get 1up, the agent has to first hit the brick hiding 1up and subsequently catch it. A simpler example is the tunneling strategy in breakout(Figure 1). Humans have high level abstract model enabling them to quickly figure out what is likely to give reward without ever actually experiencing it. Therefore human guidance can play an important role in such scenarios. There are two ways to achieve this:

- Inject a policy consisting of these complicated strategies. This policy can be computed using supervised learning based on sample of game episode collected from a human player.
- **Reward shaping**: Modifying the reward function so that it captures these strategies.

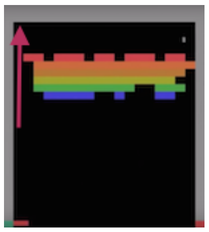


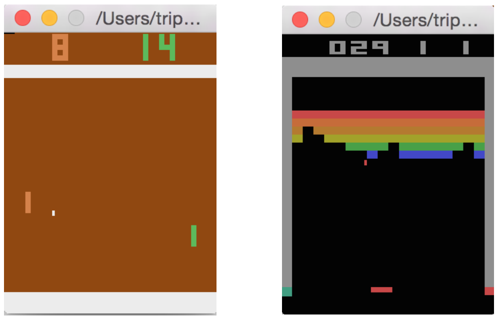
Fig. 1: Tunneling strategy in Breakout. The bricks of one of the sides are depleted quickly letting the ball reach the top and collect bricks rapidly.

As described in section IV later, we will test different reward function to train the agent to make it play the tunneling strategy in breakout.

E. Open AI Gym

We used OpenAI Gym (Brockman et al., 2016[5]) for simulating Atari environment. It is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games Gaming from Pixels like Pong or Breakout.

IV. Experiments



(a) Pong (b) Breakout

Fig. 2: Atari Games used in my training.

Training was done for 2 of the ATARI games: Pong and Breakout(Figure 2). The input was an image frame(210X160X3 byte array giving pixel values(0-255)). The action space consisted of 3 actions for both the games: move paddle up, move paddle down and no-op for Pong. Move paddle left, move paddle right and no-op/start-game for Breakout. For Pong, a reward of +1 was obtained when the ball went past the opponent, -1 if the agent misses the ball and 0 otherwise. For breakout, on every brick removal, +1 was received and 0 otherwise. So a Pong episode score ranges between [-21, +21] and for Breakout its 0 or more.

A. Model Architecture

The policy function approximator is a simple feed forward neural network as shown in Figure 3. The input to the network is the **difference of two successive image frame** received from game environment. Differnce is taken to capture the movement of objects which would have been missed otherwise. The hidden layer uses Relu as the activation function. And the output layer is followed by a softmax function.

B. Optimization

Stochastic gradient ascent with Standard non-centered RMSProp (Tieleman & Hinton, 2012) is used for annealing the learning rate. The update is given by:

$$g = \lambda g + (1 - \lambda) \bigtriangleup \theta^2 \tag{5}$$

$$\theta \leftarrow \theta + \alpha \frac{\bigtriangleup\theta}{\sqrt{g + \epsilon}} \tag{6}$$

where, λ is the decay rate.

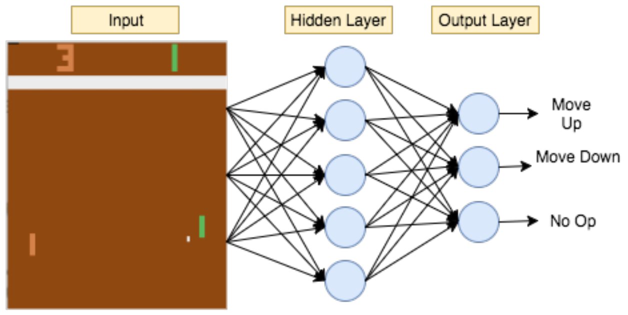


Fig. 3: Policy Network with 2 layers Fully Connected Net.

The input image frames were downsampled to 80 X 80 gray-scale for thinner network.(Image - Cropping was game specific)

C. Discounted Reward Function

For pong we had a single reward function where all the action leading to point(± 1) were only attributed with that point with a discount factor $\gamma = 0.99$.

For Breakout, we tested with following 3 reward strategies:

- **Reward 1**: Only the actions between scoring two consecutive non-zero points, r_{t_a} and r_{t_b} were attributed with point r_{t_b}(discounted).
- **Reward 2**: All the actions before scoring non-zero point r_t were attributed with r_t
- **Reward 3**: All the actions between scoring the non-zero points r_{t_a} and r_{t_b} (not necessarily consecutive) were attributed with all the points scored between $(t_a, t_b]$ such that there is only one paddle hit event between t_a and t_c and no paddle hit event between t_c and t_b, where r_{t_c} is the first non-zero point scored after scoring r_{t_a}.

The Reward 3 function is expected to favor the tunneling strategy as compared to Reward 1,2. Also, Reward 1,3 are expected to have better optimal score as Reward 2 is attributing the scores to action that are not involved in generating them.

D. Training

The training was done with policy parameter updated in a minibatch of 10 episodes. Both the games were trained with exactly same hyperparameters as shown in Table II. The hidden layer consisted of 1000 neurons.

TABLE II: Hyperparameters

parameters	Value
Hidden layer Neurons	1000
Learning Rate	0.5e-3
Discount Factor	0.99
Optimizer	RMSProp
RMSProp Decay Rate	0.95
Update Batch Size	10

V. Results

Agents were successfully trained to play Pong and Breakout. The learning curves are shown in Figure 4-5. The test results are summarized in table III.

Figure 6 shows the comparision between the three reward functions of breakout describe above. As expected Reward 1 and 3 performed better than Reward 2 but Reward 3 did not yet drive the agent to learn tunneling strategy.

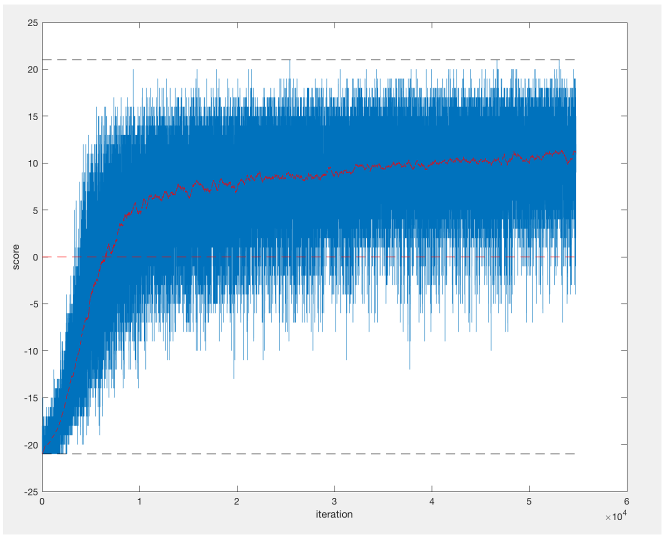


Fig. 4: Pong Learning Curve. The blue lines are episode scores. Red line is the running mean. Running mean after 60000 iteration = 11

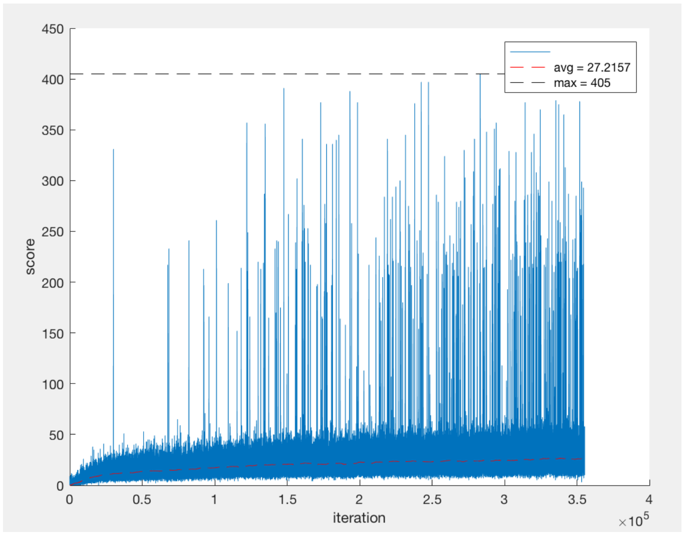


Fig. 5: Breakout Learning Curve. The blue lines are episode scores(Max = 405!). Red line is the running mean. Running mean after 350000 iteration = 27

VI. Conclusions

We were successfully able to train agents matching human level performance for pong and breakout. The reward shaping

TABLE III: Test Results

Game	Random Play	Human	Policy Gradient
Pong	-20.8	5.25	8.9±4.12
Breakout	1.3	33.5	25.5±6.4

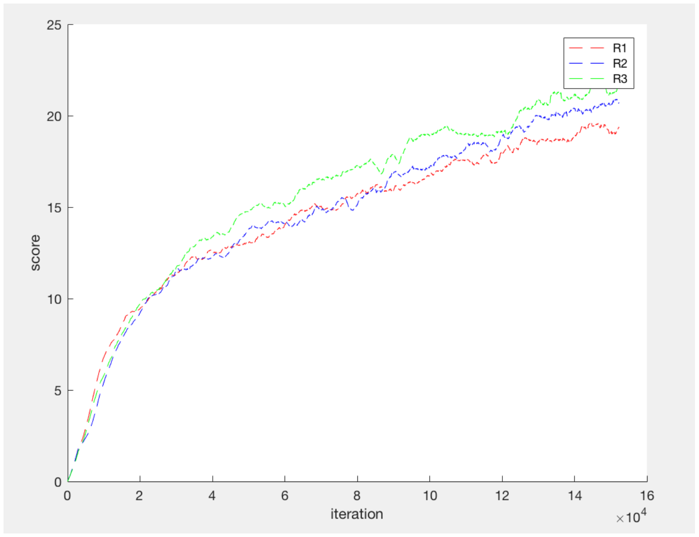


Fig. 6: Reward Function Comparison for Breakout.

experiment was unsuccessful in driving the agent to learn tunneling strategy in Breakout.

APPENDIX

A. Softmax cost function and its gradient

$$g_\theta(x) = \begin{bmatrix} p(y=1|x;\theta) \\ p(y=2|x;\theta) \\ \vdots \\ p(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} exp(\theta^{(j)T}x)} \begin{bmatrix} exp(\theta^{(1)T}x) \\ exp(\theta^{(2)T}x) \\ \vdots \\ exp(\theta^{(K)T}x) \end{bmatrix}$$

The cost function is given by,

$$J(\theta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)}=k\right\} \log \frac{exp(\theta^{(1)T}x^{(i)})}{\sum_{j=1}^{K} exp(\theta^{(j)T}x^{(i)})}$$

And the gradient is given by,

$$\nabla_{\theta^{(k)}} J(\theta) = -\sum_{i=1}^{m}[x^{(i)}(1\left\{y^{(i)}=k\right\} - \frac{exp(\theta^{(1)T}x^{(i)})}{\sum_{j=1}^{K} exp(\theta^{(j)T}x^{(i)})})]$$

ACKNOWLEDGMENT

REFERENCES

[1] Volodymyr Mnih and Koray Kavukcuoglu and David Silver and Alex Graves and Ioannis Antonoglou and Daan Wierstra and Martin A. Riedmiller, Playing Atari with Deep Reinforcement Learning, 2013
[2] Volodymyr Mnih and Adri Puigdomnech Badia and Mehdi Mirza and Alex Graves and Timothy P. Lillicrap and Tim Harley and David Silver and Koray Kavukcuoglu, Asynchronous Methods for Deep Reinforcement Learning, 2016
[3] Ronald J. Williams, Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning, 1992.
[4] Richard S Sutton and ANdrew G. Barto, Reinforcement Learning: An Introduction, 2016
[5] Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech https://openai.com/blog/, 2016
[6] Andrej Karpathy, http://karpathy.github.io/2016/05/31/rl/, 2016