AI Agent for Ice Hockey Atari 2600

Emman Kabaghe (emmank@stanford.edu)
Rajarshi Roy (rroy@stanford.edu)

1 Introduction

In the reinforcement learning (RL) problem an agent autonomously learns a behavior policy from experience in order to maximize a provided reward signal. Games have always been an important testbed for AI, frequently being used to demonstrate major contributions to the field. The Atari Learning Environment **[1]**, which allows the testing of AI agents for ATARI 2600 games has served as a standardized testbench and metrics platform for recent advances in reinforcement learning algorithms such as deep Q learning **[2]** and asynchronous multi-actor learning **[3]**. In this project we developed AI agents using various reinforcement learning techniques for the Atari 2600 game "Ice Hockey (1981)" from Activision software **[4]**. We chose this game in particular because we found it challenging to beat the computer as human players. From a reinforcement learning perspective, developing an AI for the game is non-trivial since there is no feedback for majority of state transitions. The only reward feedback to the agent is delayed till the opponent or the agent scores a goal. Furthermore, the game lets the agent control two players so it is interesting to observe the learning of optimal strategies for each of the players.

2 Related Work

In the standard reinforcement learning setting, an agent interacts with an environment over a number of discrete steps. At each time step t, the agent observes state s_t and selects an action at from a set of possible actions according to some policy π that maps states to actions. Upon taking the action, the agent receives the next state s_{t+1} and some reward r_t . The process continues until an episode ends. The return is the sum of rewards accumulated throughout the episode. The aim of reinforcement learning algorithms is to learn an optimal policy that maximizes the expected return **[5]**.

The underlying model of the state space in reinforcement learning is a Markov Decision process (MDP). The model describes states, transition probabilities from and to pairs of states, and rewards associated with state transitions. Given a model whose transitions and rewards are known, policy iteration and value iteration algorithms can find the best policy to maximize the expected return **[5]**. However, if the model transitions probabilities and rewards are no known, Monte Carlo approaches can be taken to populate transitions probabilities and rewards by counting experience. However, model-free approaches can circumvent the model altogether and directly learn an optimal policy from experience **[6]**. The only aspect of the MDP that is retained however are the Q and V values. The value, V(s) of a state corresponds to the expected return if optimal actions taken from the state. The Q(s,a) value of a state-action pair is the expected return if the action a is taken from state s. Thus V(s) = max(Q(s,a) for all possible actions a). Value based model free approaches such as SARSA **[7]** and Q-learning **[8]** attempt to directly learn the Q values of state-action tuples from experience (s,a,r,s'). While policy based model free approaches such as policy gradients attempt to directly learn optimal policies **[9]**.

There have been several developments to model free approaches that make the Atari game AI learning problem feasible. Tracking values (V/Q) for every (state/state-action pair) is not feasible due to the large pixel state space of the Atari screen frame (256 pixel levels)^(210(height)x160(width)x3(channels)) = 2^268800. Function approximators tackle this problem by creating a mapping function between the (state/state-action pairs) to the (V/Q) values directly **[26]**. Good results have been obtained with both

“shallow” and “deep” function approximators. Shallow function approximators are composed of hand engineered features such as blobs positions or difference of blob positions that are processed from the screen and fed into a linear function **[10]**. Deep function approximators such as those used in DQN (deep Q-learning) are multi-layered neural networks with initial convolution layers that learn to extract features followed by one or more fully connected layers **[2]**. In our work, we used a hybrid function approximator neural net. We first extract from the raw frame, a feature set of the position of elements in the game of ice hockey (puck and player positions). Then a two layer neural network on this feature set is used to predict values. The neural net in our approach is used for faster training than a deep neural net since it has a much smaller state space instead of raw pixels.

Since DQN, there have been other improvements to the reinforcement learning loss minimization framework that propagate feedback to the function approximator to learn its parameters. DQN follows the standard 1-step Q learning which models the loss of a state transition (s,a,r,s’) with the [r+gamma*V(s’)-Q(s,a)]^2 that is based on the underlying MDP model assumption **[11]**. Other methods such as Double-DQN **[12]**, Dueling DQN **[13]**, n-step DQN **[14]**, bootstrapped DQN **[15]** and prioritized replay DQN **[16]** have been shown to perform better for many Atari games AIs. Recently, methods that exploit multiple independent actors exploring on separate instances of the game environment to update a common model such as Gorila **[17]** and asynchronous methods **[3]** have shown good performance improvements and also learning improvements as model updates from several independent actors are no longer strongly correlated and generalizes the function approximator better. Advantage actor-critic, which is a loss minimization approach that is an hybrid of value based and policy based model-free learning has shown to consistently outperform other approaches in the asynchronous setting (A3C: asynchronous advantage actor-critic) **[3]**.

The very recent Tensorpack open-source framework **[18]** allows for GPU based implementation of asynchronous methods using actors on multiple CPU threads and tensorflow based GPU neural net function approximators. Recent research on GPU based A3C or GA3C has consistently performed well like the CPU based implementation **[19]**. In our work, we evaluated the 1-step Q learning and the advantage actor-critic model-free learning frameworks on the GPU based asynchronous Tensorpack framework.

3 Task Definition

The setup of the game [Figure 1] has two players of the yellow team (opponent) versus two players of the blue team (AI agent). At any given time, only one player (the one closer to the puck) from each team can be controlled. An input to our system is the RGB values of the pixels in the screen (210x160x3 array) of every frame. We process the pixel input into a state space consisting of positions of players and puck that will be discussed later. The output from our system are 18 actions:
[don’t move, move up, move left, move down, move right, move diagonal up-left, move diagonal down-left, move diagonal down-right, move diagonal up-right]X[shoot, don’t shoot].

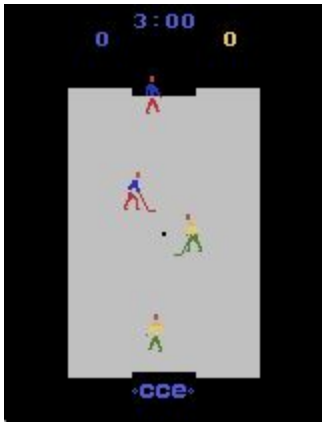


Figure 1: Screenshot of Atari Ice Hockey

Upon generating an action, the game proceeds a timestep and returns the pixels for the new frame. The second input to our system is a reward of [-1,0,1]. A reward of -1 is returned for the frame where the opponent scores a goal. A reward of 1 is returned for the frame where the agent scores a goal. For all other frames a reward of 0 is returned.

Our goal is to maximize the overall score, which is the difference (agent's score - opponent's score) over the duration of a game. The overall score is equal to the sum of all rewards returned during the duration of a game. The duration of a game, also termed an episode, is 24 timesteps/second x 60 seconds/minute x 3 minute = 4320 timesteps.

During our observation of the game we discovered an exploit that allows the agent to immediately shoot the puck from the reset position off the left wall into the goal if executed perfectly. The path of the puck for the exploit is shown in [Figure 2]. Accounting for the puck's decceleration, velocity and path, we measured an approximate rate of 30 goals per episode if the exploit is perfectly executed every time. It is to be noted that after any goal, the puck and player positions reset. This allows for the execution of the exploit after every goal. We refer to the overall score of 30 as the exploit score.

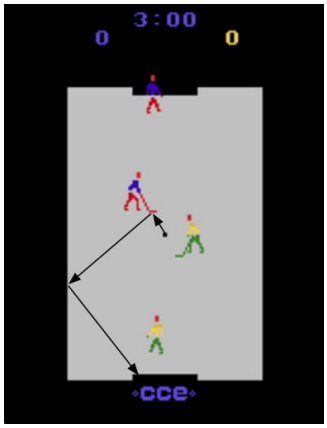


Figure 2: Pathway of puck in optimal exploit

According to two reinforcement learning papers **[15,16]**, the random player score is -11.2 and -9.7. We take the average of these scores -10.45 as the random agent score. This is consistent with our observation of a random agent. Similarly, the human player score is cited to be 0.5 and 0.9. We take the average of these scores 0.7 as the human player score. Again, this is consistent with our experience. Finally, the baseline DQN score **[16]** is -3.8. We choose the lowest benchmark (random agent score) of -10.45 as our baseline and the highest benchmark (exploit score) of 30 as our oracle [Table 1].

Benchmark	**Score**
Random agent **(baseline)**	-10.45
DQN	-3.8
Human player	0.7
Exploit **(oracle)**	30

Table 1: Baseline, oracle and other benchmarks summary

4 Infrastructure

We used the OpenAI Gym toolkit which integrates the Arcade Learning environment (a simple object-oriented framework that allows researchers and hobbyists to develop AI agents for Atari 2600 games) and the Stella Atari Emulator. OpenAI Gym provides the game simulator and allowed us to focus on writing the reinforcement learning algorithms. The OpenAI Gym platform provides us with the following classic reinforcement learning “agent-environment loop” [Figure 3].

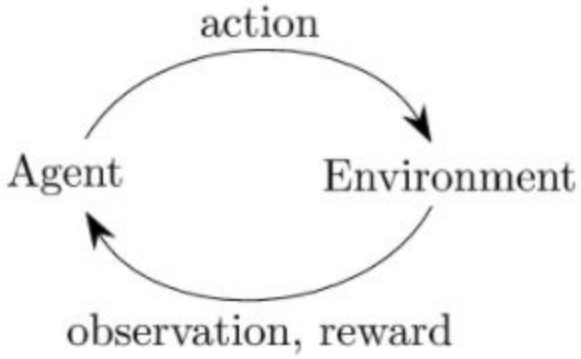


Figure 3: Agent-environment loop on OpenAI gym

For Atari Ice Hockey, we define these terms as:

- **Action**: This is the set of all possible actions of a game. In our case, this is all actions that can be performed at any given time step (move up, move down, move right, move left and trigger (hit the puck)).
- **Observation**: This represents the the current state of the environment at a particular frame. In the case of Ice Hockey, this will represent the current state of the game i.e (the positions of both sets of players and the position of the puck).
- **Reward**: This represents a reward (if any) of the previous action. In the case of Ice Hockey, this will convey the change in score caused by the previous action. If the previous action was a shot that ended up in the opponent's goal, the reward would be a score of 1. If the action did not lead to a goal for either player, the reward would be 0. However, if the opponent scored, the reward would be -1.

We used the Tensorpack open-source framework **[18]** that allows for GPU based implementation of asynchronous methods using actors on multiple CPU threads and tensorflow **[20]** based GPU neural net function approximators. Tensorpack's API allows us to specify the number of actors in any of the asynchronous methods as well as some other information such as batch size , number of history frames (how many frames to add to the game state) and the number of iterations per epoch.

Tensorpack also offer the ability to define the function approximator model and the model-free learning loss minimization framework with numpy **[21]** and tensorflow **[20]**. A deep function approximator with 4 convolutional layers and one fully connected layer similar to that in A3C **[3]** was provided. As described further in the next section, we modified the function approximator to a hybrid of our own feature detector and two hidden layers. Tensorpack also provided an example advantage-actor-critic loss minimization head to the neural net. We changed this to a 1-step Q-learning loss minimization head to the neural net for our Q-learning evaluations.

Unfortunately, tensorpack is focused on asynchronous learning model experimentation on the OpenAI gym platform and is not very flexible to the data format. It expects the data to be in the 2D pixel array x 3 color channel format. We spent a significant amount of effort in debugging and modifying tensorpack core functions to handle our feature detector that outputs a vector of various player and puck positions.

The compute resource we used was a personal desktop computer with an Intel i7 Processor and a single Nvidia GTX 1080 GPU. The neural net framework was Tensorflow **[20]**, CuDNN v5 **[22]** and CUDA Toolkit 8.0 **[23]**.

5 Approach

5.1: Function Approximator:

Due to the time scale of this project and the effort in tuning deep neural nets with convolutional layers, we chose a function approximator approach similar to shallow reinforcement learning **[10]**. However, since our project is focused on the AI agent for the specific game of Ice Hockey, we designed a feature detector specific to the game instead of generic features like B-PROS, B-PROST and Blob-PROST that were used in shallow reinforcement learning. Our feature detector detects the horizontal (x) and vertical (y) positions of the four players and the puck in the game.

Hand coded matrix operations in numpy processes the raw frame pixels and outputs the following array: **[agentplayer1_xpos, agentplayer1_ypos, agentplayer2_xpos, agentplayer2_ypos, opponentplayer1_xpos, opponentplayer1_ypos, opponentplayer2_xpos, opponentplayer2_ypos, puck_xpos, puck_ypos]**

The feature detectors for the puck and players are custom coded based on the color values of the pixels of the sprites.

For example, the detector for the puck uses the following operations on the input pixel frame:
1) Extract just the hockey court from the frame's green channel:

```
puckdet = np.copy(observe[42:187,32:128,1])
```

2) Top goalpost is black so white it out:

```
puckdet[0:4,32:64] = 255
```

3) Bottom goalpost is black so white it out:

```
puckdet[142:145,32:64] = 255
```

4) Clip all non-zero (non black pixels) to 1. That way, only pixels belonging to the puck (which is black) will be 0. The do a 1-x operation to make the puck pixels 1 and other pixels 0.

```
puckdet = (1-np.clip(puckdet, 0, 1))
```

5) Ge the x-y position of the puck pixels using the nonzero operation

```
rawpuckidx = np.transpose(np.nonzero(puckdet))
```

6) Average the puck pixel positions to obtain the final puck x-y position

```
puck_y = int(np.mean(rawpuckidx, axis=0)[0])
puck_x = int(np.mean(rawpuckidx, axis=0)[1])
```

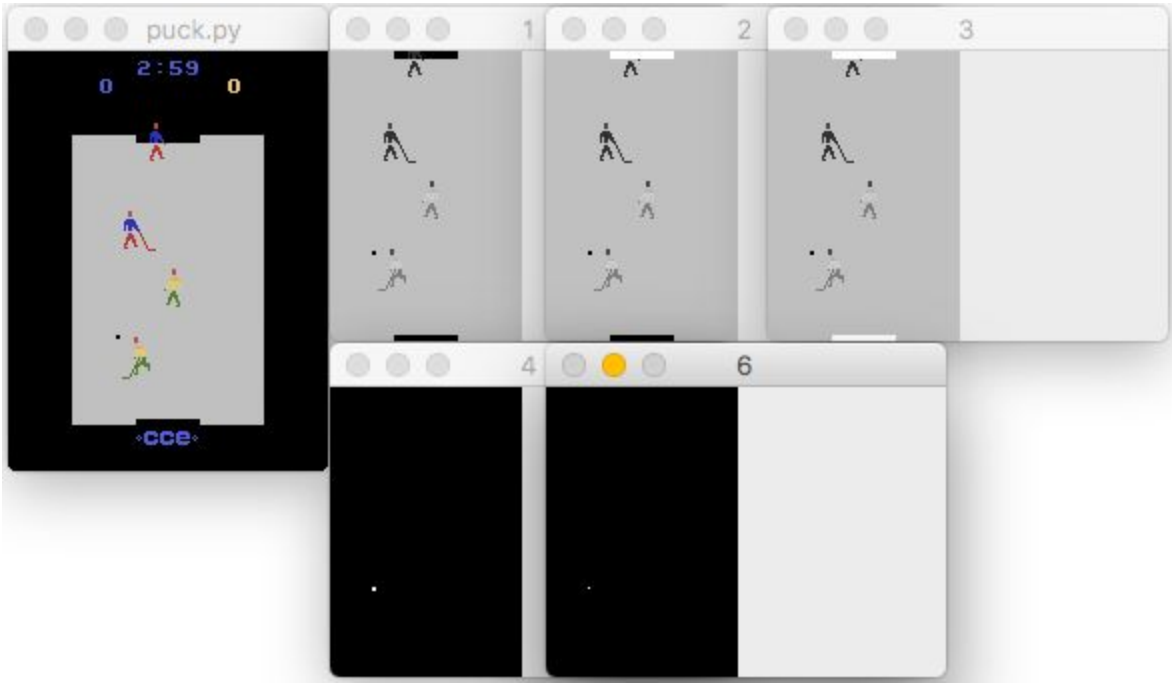


Figure 4: Stages of operations on the frame to obtain the puck position

Similar operations are used to detect the players using the jersey color and head color. The player locations correspond to the neck of the players.

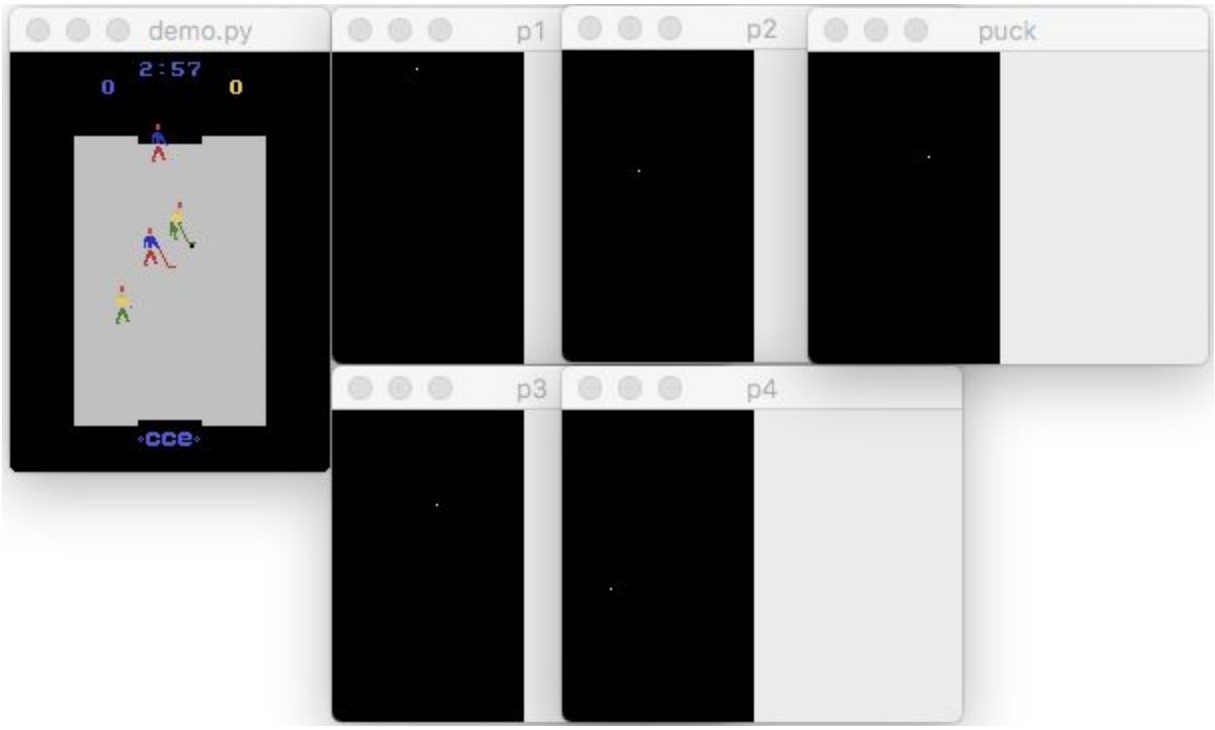


Figure 5: Visualized output positions from feature detector

The game does not respond to input actions for the first second of the game and after a position reset due to a goal. To encode this information, two timers are added to the feature vector: timesteps since beginning of game, timesteps since last goal. Thus final intermediate state representation is:
`[agentplayer1_xpos, agentplayer1_ypos, agentplayer2_xpos, agentplayer2_ypos, opponentplayer1_xpos, opponentplayer1_ypos, opponentplayer2_xpos, opponentplayer2_ypos, puck_xpos, puck_ypos, timer_startgame, timer_lastgoal]`
This 12 element vector representation of the 4 most recent frames are then used as a 48 element vector intermediate state representation. This vectors thus encodes direction, velocity and acceleration of elements.

The 48 element state vector is then used as input to a neural network with two hidden layers that output a 512 element vector. For advantage actor-critic, this vector goes through a fully connected step to produce the policy vector (18 elements corresponding to actions) and a value scalar [Table 2]. For Q-learning, the 512 element vector goes through a full connected step to produce the Q vector (18 elements corresponding to actions) [Table 3].

Layer:	Output Dimension:
Input frame observation	(210x160x3)
Feature detector with history	48
Fully Connected	512
PReLU	512
Fully Connected	512
PReLU	512
Fully connected, Fully connected	1 (value), 18 (policy)

Table 2: Advantage actor critic function approximator

Layer:	Output Dimension:
Input frame observation	(210x160x3)
Feature detector with history	48
Fully Connected	512
PReLU	512
Fully Connected	512
PReLU	512
Fully connected	18 (Q)

Table 3: Q learning function approximator

Due to the time-limitations in this project we could not experiment expansively with various activation functions and hidden layer numbers.

5.2: Advantage Actor Critic and GPU based asynchronous toolkit implementation:

Actor-critic reinforcement learning is a temporal difference learning method that attempts to explicitly represent the policy independent of the value function. The vast majority of reinforcement learning methods learn either the value function only or policy $\pi(a_t|s_t,\theta)$ only. Not that a_t is an action taken at time step t, s_t is the current state and θ is the set of parameters of the policy function . Actor-critic aims to combine both the value function approximation and the policy-based learning. Actor-critic achieves this by separating the actor from the critic. The actor is the policy structure and it is used to select actions to take in a given state. The critic is is the estimated value function and it “criticizes” the actions made by the actor. The actor follows a particular policy and is therefore on-policy. The critic learns a value function which is then used to update the

actor's policy parameters in a manner which leads to performance improvement. The output of the critic is in essence how happy, or how unhappy the critic is with the action taken by the actor. The critic has the form of the standard temporal difference shown below:

$$V^{\pi}(s_t;\theta_v) = r_{t+1} + \gamma V^{\pi}(s_{t+1};\theta_v)$$

where :
r_{t+1} is the reward after taking an action from state s and observing state s_{t+1}
γ is the discount factor
$V^{\pi}(s_{t+1})$ is the estimated expected utility for following policy π from state s_{t+1}
$V^{\pi}(s_t)$ is the current estimate of the expected utility for following policy π in state s
θ_v is the learning parameter of the value function

For the actor, there are many methods for updating parameters θ after seeing the rewards of the environment. One example of such a method is the standard REINFORCE update **[24]**. This method involves performing gradient ascent on the expected return for selecting an action in the current state and following a policy π. The REINFORCE method updates the the policy parameters in the direction $\nabla_{\theta} log\, \pi(a_t|s_t;\theta)R_t$ where R_t is the total accumulated return of an episode at the time step t. The variance of the estimate can be reduced by introducing a learned function of the state $b_t(s_t)$ which is conveniently called the baseline **[24]**. The resulting direction update (gradient) after we subtract the baseline is $\nabla_{\theta} log\, \pi(a_t|s_t;\theta)(R_t - b_t(s_t))$. In advantage actor critic, it turns out that a commonly used estimate for the baseline is $V^{\pi}(s_t)$ **[25]**. The value $R_t - b_t(s_t)$ can be seen as an estimate of the "advantage" of action a_t in state s_t. This follows from the fact that R_t is an estimate of the expected return for selecting action a_t in state s_t and following policy π (commonly denoted as $Q^{\pi}(a_t, s_t)$ in most reinforcement learning literature). This is advantage actor critic.

In CPU based Asynchronous Advantage Actor Critic (A3C), multiple agents play concurrently and asynchronously update the policy and value parameters (θ_v, θ) using gradient descent. **[25]**. Each agent calculates gradients based on an exploration policy and then sends updates to a central parameter server after a certain maximum number of actions, or when a terminal state is reached. Because different actors can use different exploration policies and thus experience vastly different episodes, the parameter updates to the central server are less likely to be correlated reducing the need for experience replay **[25]**.

Our implementation of Asynchronous Advantage Actor Critic (A3C) is GPU based. This is handled in the Tensorpack toolkit we use. As with the CPU based implementation, the actors act asynchronously. However, unlike in CPU based A3C, the GPU based implementation does not replicate the model from multiple actors. We only have one GPU instance of the model. Furthermore, the actors do not perform any parameter updates. Instead, the actors queue policy requests in a "prediction tower" before taking an action. Once an action is available, the actors then interact with the game simulation environment performing the policy and observing (reward, new state) experiences. After a specific number of iterations (6000 in our case), these (reward, new state) experiences are then submitted into what is known as a "training tower". Behind the prediction and training towers are asynchronous predictor and trainer threads respectively. These run on the GPU. The predictor threads remove requests from the prediction tower and send a single inference query to our neural network model on the GPU. Once predictions are available, the actors receive their requested policies from the predictors. The trainer threads on the other hand, remove (reward, new state) experiences from the trainer tower and submit them to the GPU for model parameter updates.

5.3: Q learning:

Due to the flexibility of tensorpack, Q learning did not require any further modification to the overall tensorpack asynchronous framework after A3C was set up with the feature detector based function approximator. The function approximator for the 18 element Q values vector is exactly the same as that of the policy vector in A3C. However the loss minimization framework for Q learning is purely value based.

For a state transition (s_t , a , r_{t+1} , s_{t+1}) the Q value recurrence is defined as **[8]**:

$$Q(s_t, a; \theta_q) = r_{t+1} + \gamma V(s_{t+1}; \theta_q)$$

where :

r_{t+1} is the reward after taking an action from state s and observing state s_{t+1}

γ is the discount factor

$V(s_{t+1}; \theta_q)$ is the estimated expected utility for following the optimal action from state s_{t+1}

Note that: $V(s_{t+1})$ = $max_a Q(s_{t+1}, a; \theta_q)$

$Q(s_t, a; \theta_q)$ is the current estimate of the expected utility for taking action a in state s

θ_q is the learning parameter of the q function

Thus the target = $r_{t+1} + \gamma V(s_{t+1}; \theta_q)$ is first forward propagated using the function approximator.

Then the gradient for HuberLoss($Q(s_t, a; \theta_q)$, target) is back propagated to update θ_q. Note that no back propagation happens through target.

6 Experiments and Discussion

We visually verified that our feature detector implementation works. A video of the working feature detector visualized [**https://youtu.be/HnfYlNktuRY**] indicates the detected positions of the players and the puck.

After fine tuning our parameters, we were able to train our both our models for 96 hours (4 days) and we managed to obtain good results. We first present graphs showing the mean (average) score over 50 episodes and well as the max score of these episodes [Figure 6].

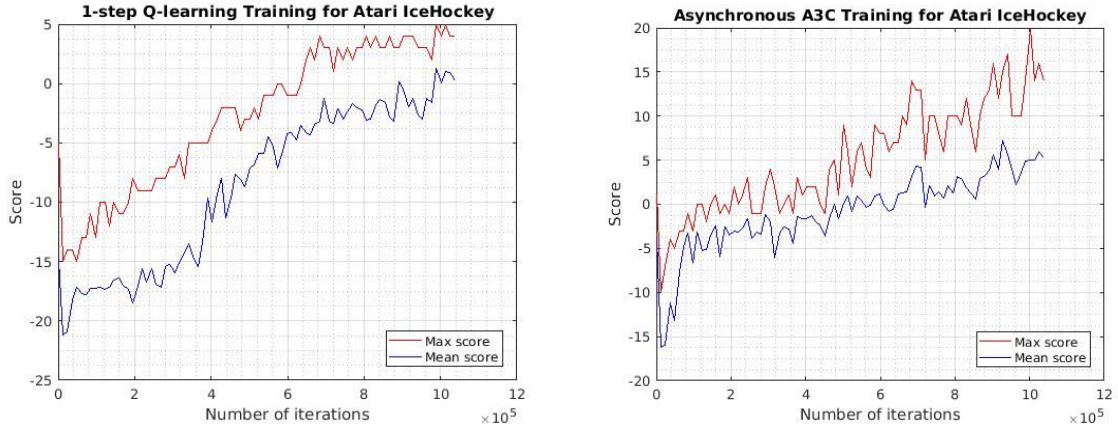


Figure 6: 50 episode mean and max scores over training iterations

Asynchronous Advantage Actor Critic outperforms 1-step Q-learning in both the mean and max scores. It can be seen that A3C reaches a winning score much quicker than 1-step Q-learning. In fact in the data that we collected, 1-step Q-learning only managed to win after about 600000 iterations while AC3 was winning at less than 200000 iterations. A3C was able to learn how to score much quicker than 1-step Q-learning.

Another interesting data point is the how the iterations/second are affected by batch size for both the raw pixel implementation version of the algorithms and our detector implementation. It can be seen below that our detector runs more iterations/second than the raw pixel implementation for both A3C and 1-step Q-learning. This is because of the absence of the convolutional neural network layers in our detector implementation.

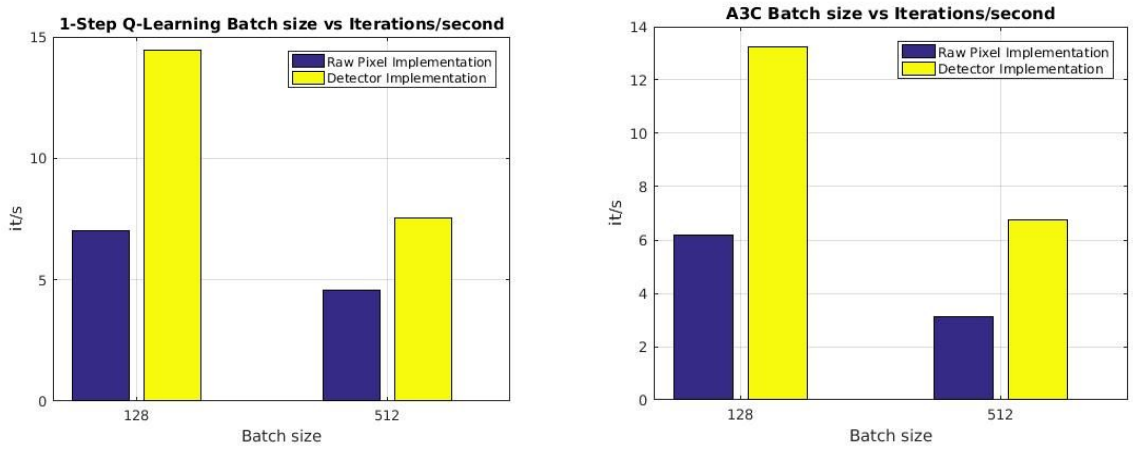


Figure 7: Training speed (iterations/second) for batch sizes 128 and 512.

For both the algorithms, the function approximator without detector iterates twice as fast as the deep function approximators that process raw pixels [Figure 7].

Looking at the raw scores helps us see the statistical performance but this gives us no intuition about the nature of the gameplay of the AI agent. Only videos of the gameplay show what the agent has really learnt. We analysed some of the game play produced by the A3C model. At 180000 iterations, as can be seen in the video [**https://youtu.be/T1Af_r-MnLg**], our agent was just beginning to learn. Both the players can perform various actions but for the most part these actions appeared to be random. There is no fixed strategy in play yet. The agent is mainly exploring the action and state space. The agent tries to follow the puck in some instances. The screenshot for the final score at this stage is shown below [Figure 8].

Figure 8: End episode screenshot at 180000 training iterations

At about 360000 iterations [**https://youtu.be/oM0YxFJOJro**], we observe that the “goalie” becomes very good at preventing the opponent from scoring. The “goalie” also realizes that it is best to stay in goal as

opposed to rushing out when an opponent is approaching the goal. It is fascinating that the first strategy picked up by the agent is defensive strategy. While the “goalie” is very good, the forward player is still sub standard. He shoots towards goal but most of his shots are off target and he has not yet quite figured out the optimum angles on which to bounce the puck off the wall. This explains the low number of goals scored by the agent.

At 528000 iterations [**https://youtu.be/XNrcmZk_2kc**], we notice the foward player become adept at scoring goals. In this game, our agent ties with the opponent, which is remarkable. The forward player has now figured out how to take angled shots off the left side of the wall as soon as he gets the ball. Notice that the players don't try to move around and dribble with the ball once in possession. The primary objective is to shoot as soon as possible so as to score. The screenshot shows the final score of this game.

Figure 9: End episode screenshot at 528000 training iterations

Our best score is 19-3 [**https://youtu.be/CejlbspRzik**]. At 1038000 iterations, the agent is both adept at defending and at scoring. The AI agent has developed a clear strategy for winning games. The goalie stays in goal and defends, while the forward player always uses the same angled shot off the left wall as soon as the game restarts. It is remarkable progress. The screenshot below shows the result:

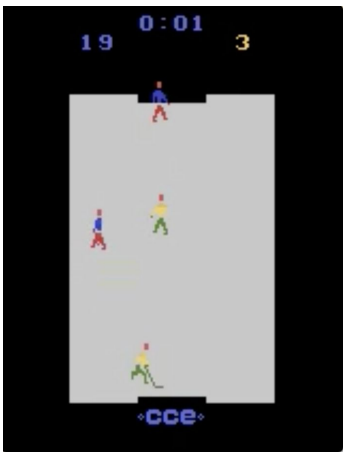
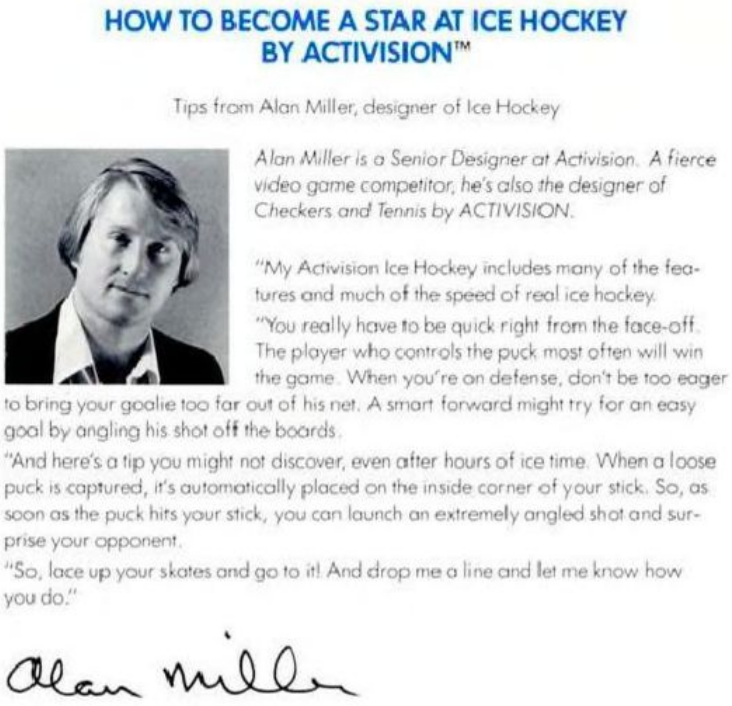


Figure 10: End episode screenshot at 1038000 training iterations

7 Conclusion

We explored advantage actor-critic and q-learning with a custom feature detector based function approximator to create an AI agent for the Atari 2600 game Ice Hockey. Our best AI agent which was based on advantage actor-critic trained in an asynchronous GPU accelerated setting using a desktop computer scored 17 more goals than the opponent. Our baseline was -10.45 using a random agent and our oracle was 30 using an exploit in the game strategy. Human level performance for Ice Hockey is 0.7. A more beautiful qualitative result is that our AI agent arrived to the exact same optimal strategy as hinted by the game's designer Alan Miller in the game's manual [Figure 11]:

"The player who controls the puck most often will win the game. When you're on defence, don't be too eager to bring your goalie too far out of his net. A smart forward might try for an easy goal by angling his shot off the boards."



HOW TO BECOME A STAR AT ICE HOCKEY BY ACTIVISION™

Tips from Alan Miller, designer of Ice Hockey

Alan Miller is a Senior Designer at Activision. A fierce video game competitor, he's also the designer of Checkers and Tennis by ACTIVISION.

"My Activision Ice Hockey includes many of the features and much of the speed of real ice hockey.

"You really have to be quick right from the face-off. The player who controls the puck most often will win the game. When you're on defense, don't be too eager to bring your goalie too far out of his net. A smart forward might try for an easy goal by angling his shot off the boards.

"And here's a tip you might not discover, even after hours of ice time. When a loose puck is captured, it's automatically placed on the inside corner of your stick. So, as soon as the puck hits your stick, you can launch an extremely angled shot and surprise your opponent.

"So, lace up your skates and go to it! And drop me a line and let me know how you do."

Alan Miller

Figure 11: Ice Hockey game manual **[4]**

Over the course of this project, we understood cutting edge reinforcement learning techniques and a bulk of the Tensorpack asynchronous learning framework in order to implement our agent. We would like express our gratitude to our mentor teaching assistant Tianlin Shi for his guidance on this project and the fantastic course staff of Stanford CS221 for the foundation that enabled us to execute this project.

8 References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. J. Artif. Intell. Res. (JAIR), 47:253–279, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human- level control through deep reinforcement learning. Nature, 518 (7540):529–533, 2015.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Int'l Conf. on Machine Learning (ICML), 2016.

[4] AtariAge. (n.d.). Retrieved December 16, 2016, from http://atariage.com/manual_html_page.php?SoftwareLabelID=241

[5] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. Vol. 1. No. 1. Cambridge: MIT press, 1998

[6] Fonteneau, Raphael, et al. "Model-Free Monte Carlo-like Policy Evaluation." *AISTATS*. 2010.

[7] Shteingart, H; Neiman, T; Loewenstein, Y (May 2013). "The Role of First Impression in Operant Learning". J Exp Psychol Gen. 142 (2): 476–88.

[8] C. J. C. H. Watkins and P. Dayan. Technical Note: Q-Learning. Machine Learning, 8(3-4), May 1992.

[9] Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS* (Vol. 99, pp. 1057-1063).

[10] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael Bowling. 2016. State of the Art Control of Atari Games Using Shallow Reinforcement Learning. In Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems (AAMAS '16). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 485-493.

[11] Howard, Ronald A. "Dynamic Programming And Markov Processes" (1960).

[12] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-learning." *CoRR, abs/1509.06461* (2015).

[13] de Freitas, Nando. Dueling Network Architectures for Deep Reinforcement Learning. No. arXiv: 1511.06581. 2015.

[14] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. Vol. 1. No. 1. Cambridge: MIT press, 1998

[15] Osband, I., Blundell, C., Pritzel, A., & Van Roy, B. (2016). Deep Exploration via Bootstrapped DQN. *arXiv preprint arXiv:1602.04621*.

[16] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[17] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Mas- sively parallel methods for deep reinforcement learning. In Deep Learning Workshop, ICML, 2015.

[18] Zhou, Shuchang, et al. "Tensorpack". Retrieved December 16, 2016, from https://github.com/ppwwyyxx/tensorpack.

[19] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons: "GA3C: GPU-based A3C for Deep Reinforcement Learning", 2016; [http://arxiv.org/abs/1611.06256 arXiv:1611.06256].

[20] Abadi, Martın, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467* (2016).

[21] Oliphant, Travis E. *A guide to NumPy*. Vol. 1. USA: Trelgol Publishing, 2006.

[22] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." *arXiv preprint arXiv:1410.0759* (2014).

[23] Nvidia, C.U.D.A. "Compute unified device architecture programming guide." (2007).

[24] Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8(3):229–256, 1992.

[25] V. Mnih, A. Puigdomenech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. ArXiv preprint arXiv:1602.01783, 2016.

[26] Grounds, Matthew and Kudenko, Daniel. Parallel rein- forcement learning with linear function approximation. In *Proceedings of the 5th, 6th and 7th European Confer- ence on Adaptive and Learning Agents and Multi-agent Systems: Adaptation and Multi-agent Learning*, pp. 60– 74. Springer-Verlag, 2008.