Asynchronous Deep Q-Learning for Breakout with RAM inputs

Edgard Bonilla, Jiaming Zeng, Jennie Zheng

***Abstract*— We implemented Asynchronous Deep Q-learning to learn the Atari 2600 game Breakout with RAM inputs. We tested the performance of the our agent by varying network structure, training policy, and environment settings. We saw the he most notable improvement through changing the environment settings. Furthermore, we observed interesting training effects when we used a Boltzmann-Q Policy that encouraged exploration by putting an upper bound on the greediness of the algorithm.**

I. INTRODUCTION

Reinforcement learning allows machines to iteratively determine the best behavior in a specific environment based on feedback and performance. Inspired by behavioral psychology, reinforcement learning imitates how humans learn to make decisions under uncertainty and trains an agent to gain control of an environment. For situations with complex sensory inputs, deep neural networks are frequently employed as the agent, termed a deep Q-network [6], to provide a rich representation for learning.

In this project, we train a deep Q-network agent to learn a policy for playing the Atari 2600 game Breakout. In Breakout, the player is presented with a screen lined with eight rows of bricks on the top and a ball that bounces off the top and sides of the walls. A brick is destroyed when hit by the ball and points rewarded. At the start of the game, the ball is released from the top of the screen in a random direction. By controlling a paddle at the bottom of the screen, the player must catch the ball and bounce it back to hit the bricks lined at the top. If the ball falls off the screen, the player dies and a life is lost. The goal is to achieve the highest possible score by destroying bricks with a given number of lives. By only giving the agent the RAM states of the game, we explore how well the agent generalizes the environment and develops an optimal policy for playing by varying different settings for both the environment and the algorithm. Figure 1 shows a performance example of our trained agent.

In Section III, we explain the formulation of our model and outlines the algorithm used to estimate the optimal policy for game play. In Section IV, we explore the effect on agent performance by varying the network structure of our deep Q-network, the policy used for action selection, and the settings of the Breakout environment.

II. RELATED WORKS

The framework of reinforcement learning has made significant advances in recent decades, especially in the optimization of game strategies using high-dimensional sensory input. In Mnih et al. (2015) [6], they described the first deep neural network based agent, a deep Q-network (DQN), that successfully masters difficult control policies for Atari 2600 computer games using only raw pixel inputs. To stabilize the deep neural network, they employed experience replay and a target network. In Mnih et al. (2016) [5], they introduced an alternative way to stabilize the deep neural network. Instead of employing experience replay, they use a series of parallel agents to asynchronously train on the environment, making DQNs employable for a wider spectrum of algorithms and applications. Our work is heavily inspired by their research.

Fig. 1: *The game performance of the agent on one of the best played games at three different stages. Visit here for more agent performance:* `https://youtu.be/B0xk_HQtOSU`*. Additional videos can be seen in the video description.*

III. METHODOLOGY

A. MDP Formulation

We model Breakout as a Markov Decision Process (MDP)[8]. Specifically, at each discrete time t, an agent observes a state s_t from an environment ε and selects an action a_t from the action space using policy $\pi(s_t)$. It then receives a reward r_t and the next state s_{t+1}, which only depends on the previous one. The process is restarted when the agent reaches a terminal state. The future reward at each time step is discounted by a factor $0 \geq \gamma < 1$, i.e. $R_t = \sum_{i=t}^{T_{max}} \gamma^{i-t} r_i$. The goal of the agent is to take actions that maximize the expected future reward, $E[R_t|s_t, a_t]$. We estimate the expected future reward at state s by the quality function $Q(s, a)$, which is discussed in Section III-B in more detail.

For Breakout, we employed OpenAI Gym's Atari game environment [2]. The environment outputs the state s as either internal memory (RAM) or screen images; we mainly focus on RAM inputs for this paper. OpenAI Gym's default action space contains six actions that correspond to the Atari's controls. To speed up training, we reduced the action space to left, right, and do nothing. When given action a_t at time t, the environment outputs $\{s_{t+1}, r_{t+1}, terminal\}$,

TABLE I: DQN Architecture

Layer	Input	Activation	Output
flatten1	1x128	None	None x 128
Dense2	None x 128	ReLU	None x 128
Dense3	None x 128	ReLU	None x 128
Dense4	None x 128	ReLU	None x 128
Dense5	None x 128	Linear	None x 6

where r_{t+1} is the score the agent has at time $t+1$ and $terminal$ tells us whether the agent has lost its last life.

The MDP formulation can be summarized as follows:

1) *Environment states:* 1x128 RAM state of Breakout
2) *Action Space:* Left, Right, Do nothing
3) *Transition Rules:* We used a policy π to select the next action to the game. Our choice of π is explained in Section III-C.
4) *Reward Rules:* The score of the game
5) *Terminal Rules:* Given a number of lives l, the game terminates when the ball falls off the screen l times.

B. Asynchronous Deep Q-Learning

In Q-learning, we define the maximum expected future reward for an action a in state s as the quality function $Q(s,a)$; it can be proven that the optimal quality function follows the Bellman optimality equation (Equation 1) [8],

$$Q(s,a) = \mathbf{E}_{s' \in \varepsilon}[r + \gamma \max_{a'} Q(s',a') \big| s,a] \quad (1)$$

where s' represent the next possible states and r the reward.

For an MDP, if we had a $Q(s,a)$ that satisfies Equation 1, we could construct the optimal policy π by selecting the action that maximizes $Q(s,a)$ for each state s. Q-learning represents algorithms that try to approximate $Q(s,a)$ for a given problem and then use their approximations to establish the optimal policy.

Similar to prior reinforcement learning algorithms, we use the Bellman equation to iteratively approximate the optimal $Q(s,a)$. while sampling the space of possible states $s' \in \varepsilon$.

However, RAM inputs contain 2^{128} possible states, making it far too large for a direct search. Therefore, we approximate $Q(s,a) \approx Q(s,a;\theta)$ by using a deep Q-network. The network is updated through stochastic gradient descent with loss function

$$L_i(\theta_i) = \mathbf{E}\Big(r + \gamma \max_{a'} Q_i(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\Big)^2 \quad (2)$$

Our DQN structure is presented in Table I, where the inputs to the network are the 1x128 RAM states and the outputs are the $Q(s,a)$ values for each of the valid actions a on state s. This particular network structure is selected after experiments seen in Section IV-A. We used the optimizer Adam [4] to perform stochastic gradient descent on our network.

One challenges in stabilizing the estimation of DQNs with stochastic gradient descent is the problem of highly correlated observation sequences, which is especially important in Breakout. To combat this, we use asynchronous updates and employ the use of a target network[6].

In asynchronous Q-learning, we have multiple agents, or threads, training the DQN in parallel. Each thread operates on its own environment and at each time step computes the Q-learning loss gradient on a shared network. To reduce the correlation of the gradient's target values, we use a slowly changing target network for the gradient update, and each thread uses a different exploration policy. Additionally, to increase the robustness of the algorithm, the gradients are built up over a fixed number of timesteps before applied to the network, similar to a minibatch update in Mnih et. al. (2015) [6].

In addition to breaking the observation correlation and stabilizing learning, asynchronous Q-learning is comparable or even better than a single agent using a replay memory in both performance, speed, and memory use [2].

The full algorithm can be seen in Algorithm 1. $T_{\max}$ is the number of steps we trained the model. T_{target} is the number of steps we refresh the target network. T_{main} is the number of steps we take to build up the Q-learning loss gradient. The parameter values used for training can be seen in the Appendix.

Algorithm 1 Asynchronous Q-learning

1: initialize global counter T = 0, thread counter t = 0 and thread memory m for τ threads
2: initialize $Q(s,a)$ main network with uniform weights θ
3: set target network $\hat{Q}(s,a)$ weights $\hat{\theta}$ = θ
4: observe state s_0
5: **while** $T \le T_{\max}$ **do**
6: perform action a based on $Q(s,a)$ and ϵ
7: observe state s and reward r from action a

$$y = \begin{cases} r & \text{for terminal } s_{t+1} \\ r + \gamma \max_{a'} \hat{Q}(s_{t+1},a'), & \text{for non-terminal } s_{t+1} \end{cases}$$

8: store experience (s_t, a, y) in m
9: $T = T + 1$
10: $t = t + 1$
11: $s_t = s_{t+1}$
12: **if** t mod $T_{\text{main}} == 0$ **or** s_{t+1} is terminal **then**
13: asynchronously update θ using the loss $L = \frac{1}{|m|}\sum_m (Q(s,a) - y)^2$
14: **end if**
15: **if** T mod $T_{\text{target}} == 0$ **then**
16: update target network $\hat{\theta}$ = θ
17: **end if**
18: **end while**

To implement the DQN, we employed Keras [3], a high-level neural networks library for deep learning, and Tensorflow [1].

C. Policy Selection

To study the exploration-exploitation dilemma, we implemented two different policies π to select the action value in our DQN network. Finding the balance between exploring for new knowledge and exploiting the current knowledge

is one of the key trade-offs to improving performance. We used the linearly annealed ϵ-greedy policy as a baseline and compared it to a variation of the Boltzmann-Q policy that encourages exploration in Section IV-B.

1) Linearly annealed ϵ-greedy policy: Given a state s, the algorithm chooses the action $a_{\max} = \operatorname{argmax}_a Q(s, a)$ with probability $1 - \epsilon$ and a random action with probability ϵ.

During the training cycle, we anneal ϵ linearly from 1 to some thread-dependent ϵ_f over the C iterations of the algorithm. The rate of annealing depends on the number of training episodes.

2) Boltzmann-Q policy with constraint on greediness: For a state s, we set the probability of taking action a to be

$$\pi(s, a) \propto \exp\left(\frac{\beta Q(s, a)}{\max_a Q(s, a) - \min_a Q(s, a)}\right).$$

This way, the actions are ranked in order of quality, favoring actions with a higher Q value.

By imposing the additional constraint that

$$\left(\frac{\pi(s, a_{\max})}{\pi(s, a_{\min})}\right)_{\text{Boltzmann-Q}} = \left(\frac{\pi(s, a_{\max})}{\pi(s, a_{\min})}\right)_{\epsilon\text{-greedy}}$$

we can find a map $\beta(\epsilon)$ that imposes an upper bound on the greediness of the algorithm, as compared to the ϵ-greedy described above. The mapping imposed by this condition is:

$$\beta(\epsilon) = \log\left(|\mathcal{A}|\,(1/\epsilon - 1) + 1\right) \tag{3}$$

where $|\mathcal{A}|$ is the size of the action space. Using this map gives us a point of comparison for the performance of both exploration policies.

IV. RESULTS

For all the following experiments, we ran asynchronous Q-learning with 8 threads. For computing, we used Stanford Farmshare's Barley cluster. The hyperparameters used for the following experiments can be found in the Appendix.

A. Network Architecture Experimentation

We tested various DQN architectures to select the most promising for use in training by varying the number of densely connected layers k and the number of nodes in each layer n. We experimented with $k = 2, 3, 4$ and $n = 128, 256$.

Figure 2 shows the results after 42 hours of training on the Stanford Farmshare Barley cluster. In general, larger network complexity showed increased computation time, but the performance of all the networks are comparable in the first 20 million steps, with a slight divergence afterwards. To compensate both for speed and performance, we selected the DQN with $k = 3$ and $n = 128$ for subsequent model training. Compared to the other models, this specific architecture took the least amount of time to achieve the highest rewards and did not show a significant drop-off after the linear annealing phase of training.

We also note that in Figure 2, the average max Q values seem to drop off after initially hitting a high value. One possible reason is that the model failed to continue exploring non-optimal actions when it had the opportunity to do so. In Section IV-B, we test whether increased exploration will improve performance.

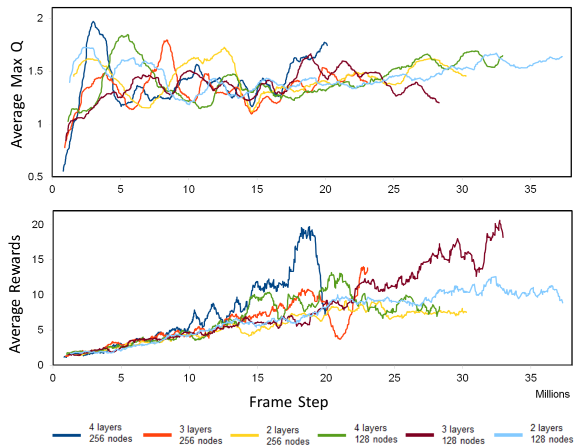

Fig. 2: *The moving averages for max Q and rewards for the first 40 million training frames. The different network structures are compared in speed and performance after training for 42 hours.*

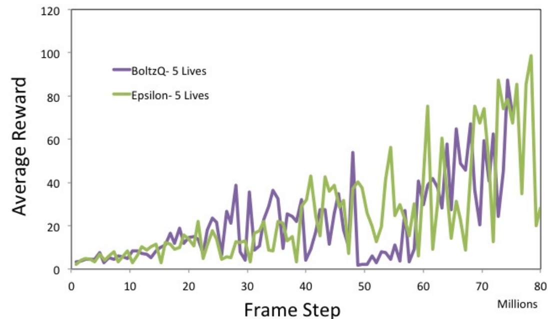

Fig. 3: *Average reward over 100 test episodes, trained using 5 lives for the environment setting. Each test is performed approximately every 800,000 frames. Testing parameters are summarized in the Appendix.*

B. Using Boltzmann-Q with constraint on greediness

Figure 3 shows that average test result at each time step for DQN using both ϵ-greedy and Boltmann-Q policies. For approximately every 800,000 frames, we test the agent for 100 gaming episodes with $\epsilon_{\text{testing}}$. The agent has 5-lives (i.e. 5 times to lose the ball) until the game reaches a terminal state.

Figure 4 and Figure 5 show the episode training rewards and average episode max Q values. We make the following observations from training and testing:

- As seen in Figure 3, the average testing scores of both policies are very similar. In the 5-lives setting, ϵ-greedy seems to achieve higher scores than Boltzmann-Q.
- The learning of ϵ-greedy appears to be steadier than Boltzmann-Q.
- At about 50 million steps in Figure 3, Boltzmann-Q encounters a steep drop in performance, coinciding with a drop an the average max Q per episode shown in Figure 5. The algorithm was able to explore and return to previous levels.

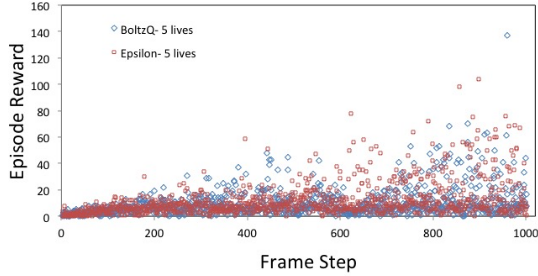

Fig. 4: *The rewards achieved by agent for each DQN training episode using 5-lives for the environment. We note that most of the training rewards barely break 40.*

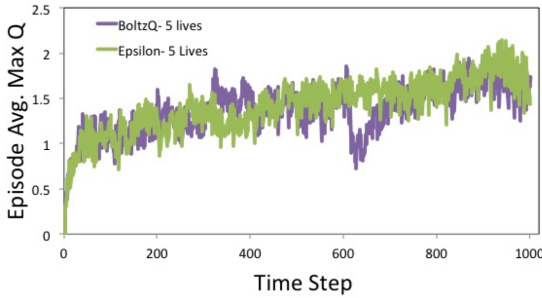

Fig. 5: *The average Max Q for each training episode using 5-lives for the environment.*

- For both policies, the performance is very unstable and the rewards vary from single to triple digits throughout the testing process. We suspect the agents are overfitting the DQN and getting stuck in local minimums that do not generalize well from time to time.
- In Figure 4, we noticed that Boltzmann-Q performs at a comparatively lower level than ϵ-greedy, which we hypothesize to be due to the exploration bound we implemented during training. However, the average max Q values and performance on testing are comparable, as seen in Figures 3 and 5.
- The agents seldom increase their scores above 50 points during training, as shown in Figure 4. However, during testing they oftentimes score well above this number, which shows some degree of generalization for the DQN fitting.
- During Boltzmann-Q training, we noticed an interesting "procrastination" phenomenon. The algorithm seems to learn that it is not vital to play "well" until the life counter hits one. In fact, many times, approximately $\frac{1}{2}$ of the score is scored in the last life. We hypothesize that because of the encouraged exploration, the agent now spends the first 4 lives exploring instead.
- In 80 million steps, the highest testing score achieved is 306 by ϵ-greedy and 283 by Boltzmann-Q.

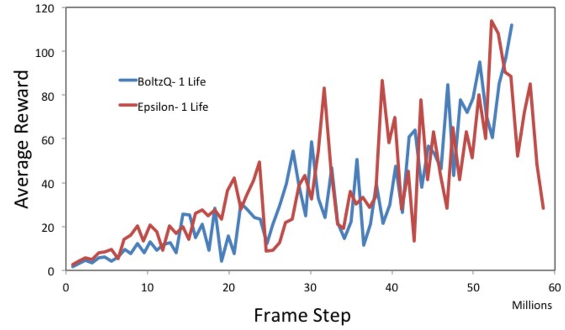

Fig. 6: *Testing average reward over 100 episodes for approximately every 800,000 frames, trained using 1-life for the environment setting.*

To further explore the "procrastination" phenomenon seen with Boltzmann-Q, we experiment with how changing the number of lives will affect performance in Section IV-C.

C. Reducing number of lives to terminal

Instead of 5-lives, the agent now has 1-life until the game reaches a terminal state. The exact same experiment as Section IV-B is ran and the average test rewards are graphed in Figure 6. The average episode training rewards and max Q for 1-life is very similar to the ones for 5-lives.

We make the following observations:

- Comparing Figure 3 to 6, the agent using a 1-live environment reaches high performance much faster than 5-lives. In 60 million steps, 1-life performance is already comparable to 80 million with 5-lives. This result was expected for the Q-Boltzmann policy due to the "procrastinator" behavior we observed during training.
- Without the propagation of rewards from previous trials, the constraint on game lives effectively punishes suboptimal play, leading to better results for both policies.
- The average performance of both policies with number of lives held constant are very similar. This is clearly shown in both Figures 3 and 6.
- Upon changing the number of training lives, the average score of Boltzmann-Q saw a larger improvement than ϵ-greedy (see Figures 3 and 6). This observation is again consistent with the "procrastinator" behavior mentioned above.
- In 60 million steps, the highest testing score achieved is 396 by ϵ-greedy and 338 by Boltzmann-Q.

Changing the environment settings gave us the most noticeable improvement in both training speed and agent performance.

V. CONCLUSION

In this paper, we explore the effect on speed and performance of asynchronous deep Q-learning on the Atari game Breakout while varying the network structure, the policy π, and environment settings.

Network structure seems to have a larger impact on the speed of training than the reward performance. We then chose the network structure with best speed-performance trade-off for further training. By varying the policy, we saw noticeable effects of encouraged exploration during the training process, such as the fascinating "procrastinator" effect. However, Boltzmann-Q with constraint on greediness does not show a performance improvement over ϵ-greedy in performance. The most noticeable training speed and agent performance improvement was seen when we varied the environment settings. By constraining games lives, we are essentially punishing sub-optimal, leading to performance improvements.

For future work, we would like to extend our code to accommodate image inputs. Exploring the reasons for why our model appears to be overfitting the DQN would also be interesting. Moreover, we would like to further study the exploration vs. exploitation trade-off and better understand the effect of "procastination" seen in the Boltzmann-Q policy by experimenting with different bounds on exploration and exploitation. We suspect that experimenting with different Atari games with larger action sets will help us differentiate the behavior of the two policies.

APPENDIX

Table II contains the list of hyperparameters used for our experiments.

Hyperparameters	**Values**	**Description**
τ	8	Number of threads used in training
γ	0.99	The discount factor
$T_{\max}$	80,000,000	Total number of training frames
T_{target}	40,000	Frame frequency with which we update the target network
T_{main}	32	Frames frequency for gradient update
ϵ_0	1	Initial epsilon for policy
ϵ_f	0.5 (3), 0.1 (3), 0.01 (2)	Final epsilon (threads count) we annealed to
C	1,000,000	The number of frames we annealed epsilon for
learning rate	0.0001	Learning rate used for Adam optimizer
$\epsilon_{\text{testing}}$	0.005	Epsilon used for testing
l_{testing}	5	Number of lives used for testing

TABLE II: Hyperparameters used for results shown in Section IV

REFERENCES

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. arXiv preprint arXiv:1606.01540.

[3] Chollet, F. keras. GitHub (2015). https://github.com/fchollet/keras.

[4] Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[5] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. arXiv preprint arXiv:1602.01783.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

[7] Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: An introduction (Vol. 1, No. 1). Cambridge: MIT press.

[8] Watkins, C. J. C. H. (1989). Learning from delayed rewards (Doctoral dissertation, University of Cambridge).