FOLLOW THEN FORAGE EXPLORATION: IMPROVING ASYNCHRONOUS ADVANTAGE ACTOR CRITIC

James B. Holliday and T.H. Ngan Le

Department of Computer Science & Computer Engineering,
University of Arkansas, Fayetteville, Arkansas, USA

ABSTRACT

Combining both value-iteration and policy-gradient, Asynchronous Advantage Actor Critic (A3C) by Google's DeepMind has successfully optimized deep neural network controllers on multi agents. In this work we propose a novel exploration strategy we call "Follow then Forage Exploration" (FFE) which aims to more effectively train A3C. Different from the original A3C where agents only use entropy as a means of improving exploration, our proposed FFE allows agents to break away from A3C's normal action selection which we call "following" and "forage" which means to explore randomly. The central idea supporting FFE is that forcing random exploration at the right time during a training episode can lead to improved training performance. To compare the performance of our proposed FFE, we used A3C implemented by OpenAI's Universe-Starter-Agent as baseline. The experimental results have shown that FFE is able to converge faster.

KEYWORDS

Reinforcement Learning, Multi Agents, Exploration, Asynchronous Advantage Actor Critic, Follow Then Forage

1. INTRODUCTION

In general, Machine Learning (ML) can be categorized into either supervised, unsupervised or reinforcement learning. Our work in this paper focuses on the last category. Stated simply, in reinforcement learning (RL) an agent gradually learns the best (or near-best) strategies based on trial and error which are performed through random interactions with the environment. The incorporation of the responses of these interactions help to improve the overall performance. Meaning the agents actions aim at both learning (explore) and optimizing (exploit). Exploitation is to make the best decision given current information whereas exploring is to gather more information. Many researches have been conducted to find the best strategies for the trade-off between exploitation and exploration. The trade-off between learning and optimizing is a classic problem in RL and is generally known as Exploitation versus Exploration.

There are many known methods for balancing between exploitation and exploration. When the state and action space is discrete, optimal solutions are possible. Bayesian RL [1] is an example of RL that can generate an optimal solution. However, when the state/action spaces are not discrete or the number of states grows very large, those previously optimal solutions become impractical. In these cases, we turn to heuristic approaches that are not perfect but are workable. The simplest approaches are random and greedy methods. With random choices the agent always chooses its action randomly during training. With greedy choices the agent always chooses its action based on the best-known utility both during training as well as execution. The most commonly implemented non-ideal approach is the ε-greedy method. ε-Greedy exploration is a combination of random and greedy where the variable, ε, determines a rate at which the agent will choose randomly or choose greedily. Generally, as the agent learns the algorithm will decay ε towards zero, so that over time more exploiting and less exploring takes place. This ensures the agent can satisfactorily explore, while still acting nearly optimally when ε is very small.

In the area of RL the last few years have been filled with landmark achievements and ground breaking research [2]. In 2013, Mnih introduced Deep Q−Learning in the form of the Deep Q−Network (DQN) [3] [4]. Q−Learning [5] is an example of this type of RL. Q−Learning works by learning a utility function which we denote as $Q(s_i,\alpha)$ where the inputs of the function are states s_i and actions α. States here are the same as defined by Markov Decision Processes (MDP) [6]. This Q function defines the policy that will control the agent's actions. An optimal policy is a mapping from states to the corresponding actions the agent should take to maximum rewards in the long run. That is, it computes the action α for a given state s_i that will yield the highest discounted horizon utility. After learning has fully converged, the optimal policy will be known. The utility function in Q−Learning is updated with the following formula derived from the Bellman equation [7]:

$$Q(s_i,\alpha) \leftarrow (1-\eta^k)Q(s_i,\alpha)+\eta^k\left[r(s_i,\alpha,s_j)+\gamma \max_{\beta\in\mathcal{A}(j)} Q(s_j,\beta)\right] \qquad (1)$$

In this formula, η is a learning rate, which controls how much the *Q* value is changed for each occurrence of *a* and *i*. The closer η is to 1, the faster the old *Q* value will be forgotten and replaced by the value computed by the rest of the formula. The term $r(s_i,\alpha,s_j)$ is the reward function, that for a given state s_i, action α and subsequent state s_j returns a reward value. The two states of the reward function are s_i and s_j where s_i is the current state where action α is performed to arrive in state s_j. The reward is added to the maximum *Q* value when we check each possible action β for state s_j and discount that *Q* value by the discount factor (γ). For γ, values near 0 prioritize immediate rewards, whereas values closer to 1 prioritize long-term utility. For most cases γ must be tuned to the problem at hand, but will always fall between 1 and 0. Over the course of training the factor *k* can be used to adjust the learning rate, which controls how much of the newer information derived from the sum of the reward and discounted future *Q* value replaces the old *Q* value. In fully deterministic environments a learning rate of 1 is optimal, where deterministic is defined as action α in state s_i always leads to state s_j. In stochastic environments the learning rate is decreased to zero over time, where stochastic is defined as there is a chance that action α in state s_i leads to state s_j. Because of this stochastic behavior it is not ideal to always replace the *Q* value with new information, so overtime as trust for the *Q* value grows it is changed less and less as the *Q* values are updated. *Q* values can be stored in a lookup table (*Q*−table). The *Q*−table can then be queried by the agent to make decisions based on the returned *Q* values. DQN built on top of *Q*−Learning by replacing the *Q*−table with a deep neural network (DNN).

DQN was able to reach human and beyond human level ability playing several specific Atari games. In 2016, Asynchronous RL [8] and specifically the development of A3C significantly improved previous efforts playing specific Atari games. A3C was able to produce better results than had been previously recorded by DQN and was able to learn much faster. A3C models its policy probabilistically where the policy output provides a probability for each possible action. These probabilities sum to 1 according to the current distribution of the model. In the case of A3C multiple agents choose actions nearly greedily (argmax of the policy output), but also seek to maximize an entropy term [9]. Entropy is used to skew the values used by the neural network optimizer in a manner that encourages heterogeneity in the way it assigns probability to the possible actions. The purpose for entropy is to “encourage diversity” in the action selection, so that the algorithm doesn’t settle on a small select group of actions or action sequences. Entropy influences the error signal which influences how the model is trained. It causes the model to lean away from giving 100% of the probability to a specific action. This indirectly causes the agent to explore because it chooses actions probabilistically according to the distribution in its model.

Our research is an effort to improve A3C. Our contribution can be summarized as follows: While A3C demonstrates good training performance, we show that A3C’s training performance can be improved by adding FFE as an exploration strategy. FFE changes the way A3C chooses some of the actions to perform. In A3C the argmax of the policy output layer of its DNN is always selected, but as mentioned above that output is influenced by entropy so exploration is encouraged. FFE builds on top of that by controlling when to stop choosing the argmax of the output and choose an action randomly.

Analogous behavior can be observed in the physical world in ants. As ants search for food they begin by following a pheromone trail from ants that have gone before them, but as the pheromone grows weak, they start to explore on their own. Similarly, at the start of an episode of RL with FFE, an agent will exploit its knowledge to take actions that lead to higher rewards, but after following the reward trail for a variable length of time the agent stops exploiting and starts exploring. After some exploring following begins again. This leads to agents exploring more where exploration is most needed, which leads to faster learning times.

2. RELATED WORK

Exploration research in RL is not a new topic [10], but since the emergence of deep learning many new efforts have been made to improve this important aspect of RL. Because RL builds on the discovery of rewards, important to a models improvement. Many times rewards are very sparse or only take place at the end of the episode. In those cases generating intrinsic rewards can create stepping stones towards actual rewards. [11] [12] and [13] attempt to do this by adding additional DNNs to their systems structure. Those additional DNNs learn what part of the state space is well known and what part is unknown and generate an intrinsic reward to explore places that are less familiar to the model. Some of these researchers call this intrinsic reward, curiosity. Our method is much simpler and does not require the expensive cost of additional networks to improve learning.

Another classical system for improved exploration is keeping track of (or counting) every unique state the agent visits. In this way it can encourage the model to explore states that have a lower count or no count at all. The challenge here is that as states grow in size and complexity processing all this information is costly and the chances of the agent seeing all the possible states is unlikely. [14] and [15] try to overcome this challenge by using Monte Carlo tree search or hashing algorithms to estimate states. A historic achievement came as a result of related research that created AlphaGo [14], which was able to defeat some of the world’s best GO players. In the case of AlphaGo the number of possible states in a 19×19 GO board is: $\sim 2.082 \times 10^{170}$, but Monte Carlo searching combined with DQN proved capable of navigating this massive state space. Again, these efforts are considerably more costly and complex to implement than our simple method.

[16] described in detail the idea of Ant Colony Optimization (ACO) as a form of swarm intelligence. They explained how it could be applied to computer intelligence. Their applications for ACO are similar to FFE as they used it in a different domain.

Actor Critic RL [17] is similar to Q−Learning except with Actor Critic the policy and utility or value are separated into their own functions meaning the policy is independent of the value. In this case the policy is known as the actor and the value is known as the critic. The actor chooses actions and the critic critiques the actions. The critiquing is done by critic estimating a value at the start of an action or sequence of actions and comparing that with the actual value that was generated by the end of the action or sequence. The difference in the estimated value and the actual value is used as a signal that can be used to train both the actor and the critic. [18] showed how to generate a signal called an advantage A for state s and action α. Where Q is the same Q value as in Q−Learning and V is the value associated with given state s.

$$Advantage : A(s,\alpha) = Q(s,\alpha) - V(s) \qquad (2)$$

[19] showed how to estimate the advantage instead of calculating Q values. The discounted reward that is used in calculating Q values is used as the replacement for the actual Q value. The formula for estimated advantage is the same as above except the Q function is replaced with the discounted reward.

3. Asynchronous Advantage Actor Critic (A3C) - Revisited

The three As of A3C stand for Asynchronous, Advantage, and Actor. The C of A3C stands for Critic. The algorithm is asynchronous because it relies on more than one agent playing the environment at the same time. For A3C, advantage, A(s) is the estimated advantage [19]. The actor calculates the policy, $\pi(s)$, in the form of probabilities for each possible action for a given state in the form of a softmax output. The critic estimates the value of a given state $V(s)$ in the form of a linear output.

One of the benefits of A3C is that it uses many agents to explore different regions of isolated but equal environments, which is one of the reasons A3C learns much faster than its predecessor DQN. DQN [3] relied on a single agent while for their research, A3C, [8] utilized sixteen agents each running their own environment. A3C creates agents or workers that each have their own DNN and environment, such as an Atari game. Each agent operates in a separate thread, but they share the same actor and critic which are represented by a global DNN. However, the agents only operate on their own local copy of the global DNN. This network functions as both the actor and critic by using shared input and hidden layers but distinct output layers. One output layer is for the policy and the other output layer is for the value. At the start of a cycle each agent copies the global DNN over its local DNN, and collects experience as it plays the game. The experience is in the form of states, actions, and values. When the agents experience is large enough it is used to determine the discounted reward, R, and advantage, A. Once the discounted reward and advantage are known losses can be calculated for the value V and the policy π. The entropy H of the policy is also calculated.

$$Value\ Loss : L = \Sigma(R - V(s))2 \qquad (3)$$
$$Policy\ Loss : L = -log(\pi(s)) * A(s) - \beta * H(\pi) \qquad (4)$$

The entropy correlates with the spread of action probabilities output from the policy. When the probabilities are relatively equal entropy will be small, but when the probabilities are spread out the entropy is large. Entropy acts as a neutralizer that encourages the model to be conservative in regard to how strongly it thinks it knows the correct action. The agent takes the losses and uses them to calculate gradients that are used to optimize its local DNN parameters. The updated local DNN is then copied over the global DNN. This causes the global DNN to be constantly updated by the agents. This training process is repeated until convergence is detected. A3C relies solely on entropy to control exploration and exploitation. [8] has shown A3C is very capable, but our modifications to A3C with FFE can improve its training performance. Adding FFE to A3C will force A3C to explore more by making more random choices and not only entropy-based choices.

4. OUR PROPOSED FOLLOW THEN FORAGE EXPLORATION

In our observation of RL learning, there are different times when more exploring or more exploiting offer more value to the learning process. For instance, when there isn't much progression made during a run, exploring at any time makes sense. We define the idea of a run as a training episode, and for our experiment a run is a single Atari game played from start to game over. If there isn't much progression the agent doesn't yet "understand" how different actions lead to different states and rewards. At this time maximum exploration will lead to the quickest increase in learning. If the agent has learned a meaningful path from start to end, then exploring at the beginning is less valuable than following the path and exploring later. When that path is sub-optimal, then additional exploration can introduce the agent to unfamiliar states. These experiences with unfamiliar states accelerate the agents learning. The central idea supporting FFE is that forcing exploration at the right time during a training episode can lead to improved training performance. Our implementation of FFE puts limits on exploiting and exploring. This is done to ensure the agent exploits closer to the beginning and end of a run, and is forced to explore in the middle.

Algorithm 1 describes our proposed algorithm which is an improvement version of A3C [8]. Our main contribution is shown at two steps, namely, Perform FFE and Update FFE. The first step of Perform FFE is detailed in the Algorithm 2 whereas Update FFE is described in Algorithm 3 as follows:

- **Algorithm 2 - Perform FFE**. This algorithm aims at selecting the best action α during training. FFE, when following, uses A3C's normal method of action selection. At a point in time when foraging should begin, FFE starts to select actions randomly. That point in time is determined by the Follow Action Limit m, and it is computed in Update FFE Algorithm 3. This forces FFE to perform more exploration than A3C normally would. However, too many random choices performed together in A3C will generally lead to poorer training performance, so the exploration must be limited. To enforce this limit FFE stops foraging and returns to following. This takes place after a controlled amount of exploration actions. This second Forage Action Limit n is also calculated in Update FFE (Algorithm 3). The injection of random exploration forces the agents into potentially less experienced states allowing the model to train faster. The other important aspect of this injection is that it happens approximately in the middle of the episode. It is our intuition that this will be the most valuable time in the episode for additional exploration to take place.

- **Algorithm 3 - Update FFE**. This step aims to keep track of a running value of the agents completed episode count o and average episode length p. For example, if FFE is used in a run this would be the number of actions required on average for the game to reach the game over state. At the end of each episode (i.e. game) p is calculated from previous experience and is multiplied by a random number percentage to set the Follow/Forage Action Limits, m and n.

After each action the agent will decrement one of it's two Action Limits. If FFE is following it decrements its Follow Action Limit m value until that variable is zero. While m is greater than zero the agent always exploits. When m reaches zero the agent switches to Forage mode, where after each action the agent will decrement its Forage Action Limit n value until that variable is zero. While n is greater than zero the agent always explores randomly. Once n reaches zero the agent returns to always following/exploiting. The use of the average episode length *p* ensures that the Action Limits *m* and *n* are dynamic and diverse in each episode.

[8] used multiple experiments to evaluate A3C, but the majority of those tests were playing various Atari games. That was accomplished through a program called the Atari Learning Environment (ALE) [20]. ALE is a simulator that can receive inputs that mimic Atari controller inputs and produce appropriate visual output that show an Atari game being played. ALE has many Atari games implemented. ALE itself has multiple implementations and for our research we used OpenAIs Gym (GYM) implementation of ALE. For our research, we used OpenAIs standard A3C implementation called Universe-Starter-Agent [21] as baseline during our benchmark and comparison against our proposed FFE.

In our experimentation we determined that best results were obtained with a large proportion of following and only a relatively small amount of foraging. The last if statement of the Algorithm 3 was added to ensure that foraging was controlled. We use the meta-parameters Φ and Ψ to limit the foraging. Parameter Φ controls the frequency of foraging actions. Then we use the difference of *p* and the value for *m* and scale that down by Ψ so that *n* is again limited. For our testing Φ was set to 0.5 and Ψ was set to 0.15. As stated, before this was to limit introducing too many poor action choices into the model. Also the 0.15 value for the scalar Ψ was used because *m* could be set to a potentially small value and then *n* could end up very large. This scalar ensured foraging never dominated following. The intuition for these limiting parameters was to allow the model to flex while foraging. By flex we mean that the model would not be presented with so many potentially poor action choices that the model was pulled away from improvement.

5. ALGORITHMS

Algorithm 1 Pseudo code of our improvement FFE to A3C. Our contribution is mainly shown at line 12, 18

```
1:  //Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
2:  //Assume thread-specific parameters vectors $\theta'$ and $\theta'_v$
3:  //Assume thread-specific variables m (Follow Action Limit), n (Forage Action Limit), o (episode count)
    p (average episode length)
4:  Initialize $m, n, o$ and $p \leftarrow 0$
5:  Initialize thread step counter t $\leftarrow 1$
6:  repeat
7:      Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
8:      Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
9:      $t_{start} = t$
10:     Get state $s_t$
11:     repeat
12:         Perform FFE                                                    ▷ See Algorithm 2
13:         Receive reward $r_t$ and new state $s_{t+1}$
14:         $t \leftarrow t+1$
15:         $T \leftarrow T+1$
16:     until terminal $s_t$ or $t - t_{start} == t_{max}$
17:     if terminal $s_t$ then
18:         Perform FFE Update                                             ▷ See Algorithm 3
19:     end if
20:     $R = \begin{cases} 0 \text{ for terminal } s_t \\ V(s_t, \theta'_v) \text{ for non-terminal } s_t \end{cases}$
21:     for $i \in \{t-1, \ldots, t_{start}\}$ do
22:         $R \leftarrow r_i + \gamma R$
23:         Accumulate gradients $\theta'$ and $\theta'_v$
24:     end for
25:     Perform asynchronous update of $\theta'$ and $\theta'_v$
26: until T >Tmax
```

Algorithm 2 Pseudo code of our proposed FFE to select best action α

```
1:  //Assume thread-specific variables m (Follow Action Limit) and n (Forage Action Limit)
2:  //m and n were initialized to 0 in Algorithm 1
3:  //Update FFE (Algorthim 3) will reset m and n to new values > 0
4:  if Follow: $m > 0$ then
5:      //Follow Action Limit > 0: choose action normally
6:      Choose $\alpha_t$ according to policy $\pi(\alpha_t|s_t;\theta')$
7:      $m \leftarrow m-1$
8:  else
9:      if Forage: $n > 0$ then
10:         //Forage Action Limit > 0: choose action randomly
11:         Choose $\alpha_t$ from a uniform distribution in $\alpha$
12:         $n \leftarrow n-1$
13:     else
14:         //Both Action Limits = 0: return to choosing action normally
15:         Choose $\alpha_t$ according to policy $\pi(\alpha_t|s_t;\theta')$
16:     end if
17: end if
18: Perform $\alpha_t$
```

Algorithm 3 Pseudo code of our proposed Update FFE

```
1: //Assume meta parameter $\Phi$ (limits forage frequency) and $\Psi$ (forage scaler)
2: //m and n = 0 when Update FFE is performed
3: $U \leftarrow Unif$                    ▷ Unif denotes the random number generator
4: Increment episode count: $o \leftarrow o + 1$
5: Calculate average episode length: $p \leftarrow (o-1)/o * p + (1/o) * t$
6: Set Follow Action Limit: $m \leftarrow U * p$
7: if $U < \Phi$ then //Decide if Forage will happen in next training episode
8:     Choose not to Forage: $n \leftarrow 0$
9: else
10:     Choose to Forage and set Forage Action Limit: $n \leftarrow U * (p - m) * \Psi$
11: end if
```

6. Experiments

To validate our research we ran numerous tests comparing a default implementation of A3C with a version of A3C modified with FFE. For our benchmark we used OpenAIs Universe-Starter-Agent [21], and for our modified version we added FFE to the default [21] implementation. The environments tested were various Atari games as implemented by OpenAIs GYM.

All our experiments were conducted on a Microsoft Azure Data Science Virtual Machine for Linux. Some initial tests were conducted using 8 cpu virtual machines, but for all the recorded experiments in this paper a Standard DS5 v2 Promo (16 vcpus, 56 GB memory) virtual machine was used. A3C was configured with all the default settings from [21]. For each experiment three test runs were done, and the results were averaged together for the results presented. For the Atari game, Pong, the score is calculated based on the total score of the agent minus the total score of the computer opponent. The game/episode is over when either the agent or computer opponent achieves a score of 21. Figure 1 shows A3C modified with FFE (indicated in blue) outperforms the baseline A3c (orange) by reaching a higher score faster. For this test entropy was left at the default value. This result shows that FFE was able to speed up the learning process for A3C. FFE created a much steeper and stable rate of learning. This shows that the intuition for FFE has merit and more extensive testing is warranted.

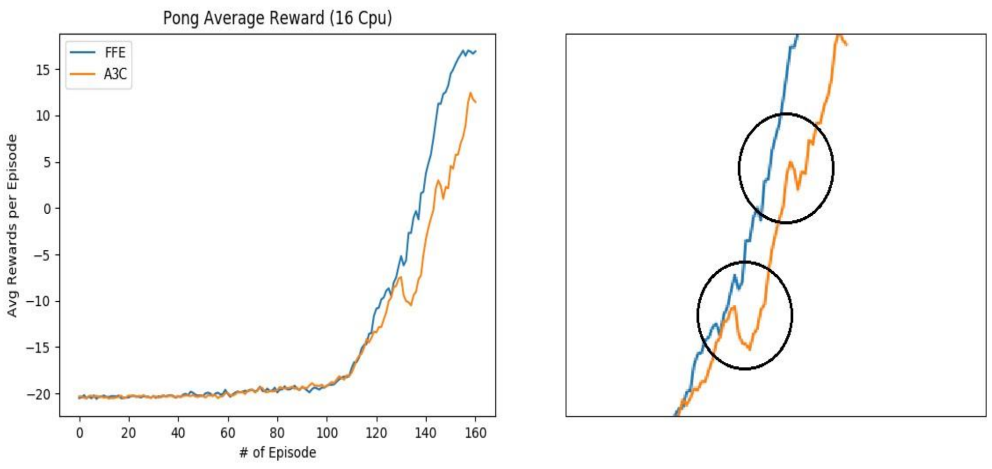

Figure 1: shows the average reward all the agents achieved per episode playing Atari Pong. The right side shows a section of the same chart highlighting the improved stability of FFE. This chart limits the results to each algorithm performing 4 million global steps.

We also tested a range of different entropy scalars to see how entropy affected the performance of the algorithm. Figure 2 shows that when entropy is not used FFE alone is not sufficient to generate a good result (labled as 0.0). Figure 2 also shows when we tried scaling entropy by .001 (labeled as 0.001) instead of the default .01 (labeled as 0.01). We found that both .001 and .01 learned in about the same number of episodes total, but by scaling entropy by .001 this caused the agents to play less optimally and thus the agents took more actions to complete the episodes, which can also be seen in Figure 2. The curves of the average game length rise and then fall because the game takes longer when both players scores get close to 21, but then as the agent starts to play much better than the computer opponent the average length starts to decrease until it plateaus when the agent is winning episodes 21 to 0. This experiment shows that FFE alone is not able to replace entropy as an exploration strategy.

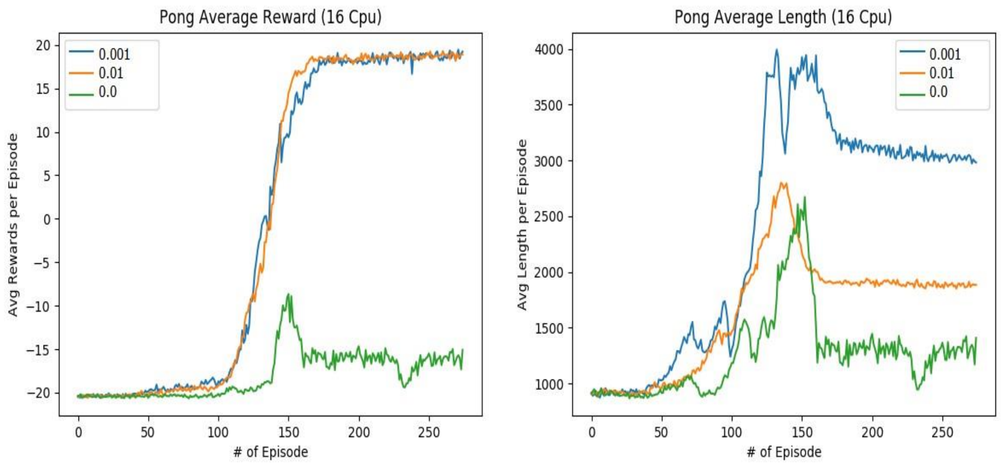

Figure 2: shows how entropy scaling effects training performance for AC3 modified with FFE. The entropy value is scaled according to the key values. [8] used a value 0.01 as the scalar value. This chart limits the results to each algorithm performing 10 million global steps.

Our next experiments involved more challenging Atari games. We tested Boxing, Amidar, and Beamrider. Figure 4 shows the results of experiment of the baseline A3C and A3C modified with FFE playing Atari Boxing. Boxing has a maximum score of 100. When either player punches the other player successfully they are rewarded with a point, and the game is over when either opponent reaches a score of 100 or time runs out. The final score is the agent's score minus the computer opponents score. While the results of both algorithms are close, FFE narrowly outperforms baseline A3C by reaching a higher score faster. Figure 5 shows the results of the experiment with the same algorithms playing Atari Amidar. Amidar has no maximum score. That being the case our agents did not score very high. Still our results were better than the high score achieved by A3C LSTM [8] which was 176 after four days of training. Lastly, Figure 6 shows the results of the experiment with the same algorithms playing Atari Beamrider. In this experiment FFE failed to outperform baseline A3C. This result is perhaps due to instability in the models DNN. Figures 3, 4 and 5 are found in the appendix. In our tests training 100 million global steps of A3C took approximately 24 hours. The cost associated with that much Microsoft Azure virtual server usage limited how much we could train.

The documentation concerning our baseline A3C states that the algorithm is tuned for good Pong performance. Meaning baseline A3C might perform poorly on other Atari games. Here is a list of some differences between [8] and [21] implementation of A3C. The original A3C used a shared optimizer for all agents, and baseline A3C uses distinct optimizers for each agent. Also to note baseline A3C is designed to be able to play games in real time, so it stores experience in a separate process while the optimizer ran, and the original A3C would force the agents to wait while the optimizer ran.

7. CONCLUSIONS

FFE demonstrates that relying on entropy alone is not the most efficient method to train A3C. Exploration utilizing FFE can allow the learning process (training) to be improved. We compared a version of A3C equipped with FFE against the default A3C with several different Atari games, and found that FFE improved results in the majority of cases. These results are promising, and provide evidence that FFE improves the default exploration strategy utilized by A3C.

When training Pong we were able to reach the maximum score in less than forty minutes for a best case. [8] listed a Pong training time at taking two hours. [8] showed the result of a good score for Breakout in less than four hours, in our research we did not have meaningful results after twenty four hours of training on Breakout. For future work we plan to utilize a more cost-efficient computing environment to allow training on more diverse and difficult environments.

For future work we will research moving FFE's forage to other times during the episodes. Perhaps foraging at the beginning or end of the episode could lead to better performance. We will evaluate an implementation of A3C that uses a shared optimizer instead of distinct optimizers.

Entropy plays an interesting role in all the experiments. For future work we plan to analyze entropy more closely. FFE also utilizes meta-parameters. We set Forage to only happen in half of the episodes and scaled the computed Forage value smaller. Because of these meta-parameters more testing is needed to determine their optimal values.

REFERENCES

[1] M. Ghavamzadeh, S. Mannor, J. Pineau, A. Tamar and others, "Bayesian reinforcement learning: A survey," *Foundations and Trends® in Machine Learning,* vol. 8, p. 359, 2015.

[2] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks,* vol. 61, p. 85, 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop,* 2013.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski and others, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, p. 529, 2 2015.

[5] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine learning,* vol. 8, p. 279, 1992.

[6] R. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics,* p. 679, 1957.

[7] R. Bellman, Dynamic programming, Courier Corporation, 2013.

[8] V. Mnih, A. Puigdomènech, M. M. B. Badia, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *33rd International Conference on Machine Learning (ICML),* 2016.

[9] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connection Science,* vol. 3, p. 241, 1991.

[10] L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research,* vol. 4, p. 237, 1996.

[11] B. C. Stadie, S. Levine and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *arXiv preprint arXiv:1507.00814,* 2015.

[12] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck and P. Abbeel, "Vime: Variational information maximizing exploration," in *Advances in Neural Information Processing Systems,* 2016.

[13] D. Pathak, P. Agrawal, A. A. Efros and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning (ICML)*, 2017.

[14] D. Silver, A. Huang, A. G. Chris J. Maddison, L. Sifre, G. V. D. Driessche, J. Schrittwieser and others, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, p. 484, 1 2016.

[15] H. Tang, R. Houthooft, D. Foote, A. Stooke, O. X. Chen, Y. Duan, J. Schulman, F. DeTurck and P. Abbeel, "# Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning," in *Advances in Neural Information Processing Systems*, 2017.

[16] M. Dorigo, M. Birattari and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, p. 28, 11 2006.

[17] I. Grondman, L. Busoniu, G. A. D. Lopes and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, p. 1291, 2012.

[18] L. C. Baird III, "Advantage updating," 1993.

[19] R. S. Sutton, D. A. McAllester, S. P. Singh and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000.

[20] M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *CoRR*, vol. abs/1207.4708, 2012.

[21] G. Brockman, C. Olsson and A. Ray, *Universe Starter Agent*, 2016.

APPENDIX

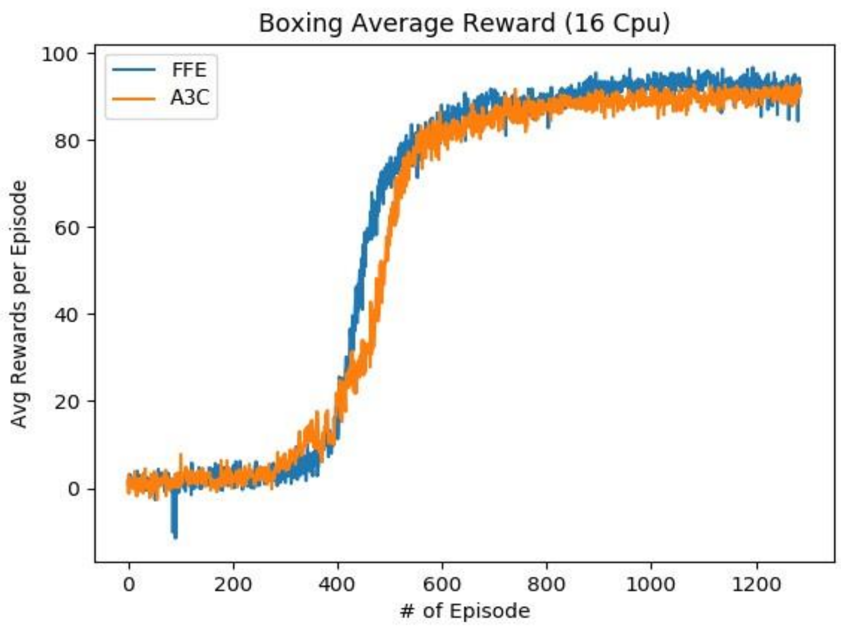

Figure 3: shows the average reward all the agents achieved per episode playing Atari Boxing. This chart is limited to reporting the results of each algorithm performing 40 million global steps.

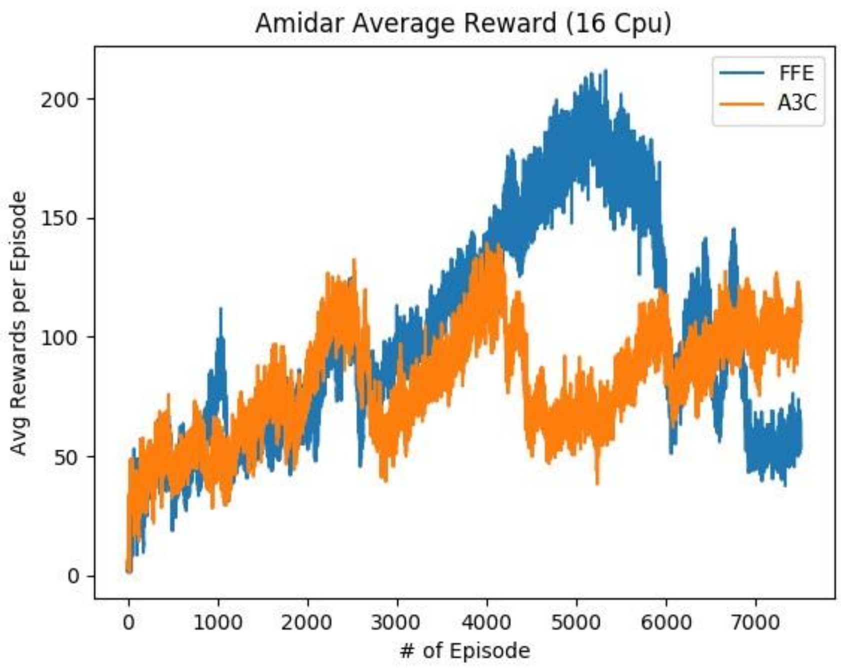

Figure 4: shows the average reward all the agents achieved per episode playing Atari Amidar. This chart is limited to reporting the results of each algorithm performing 60 million global steps.

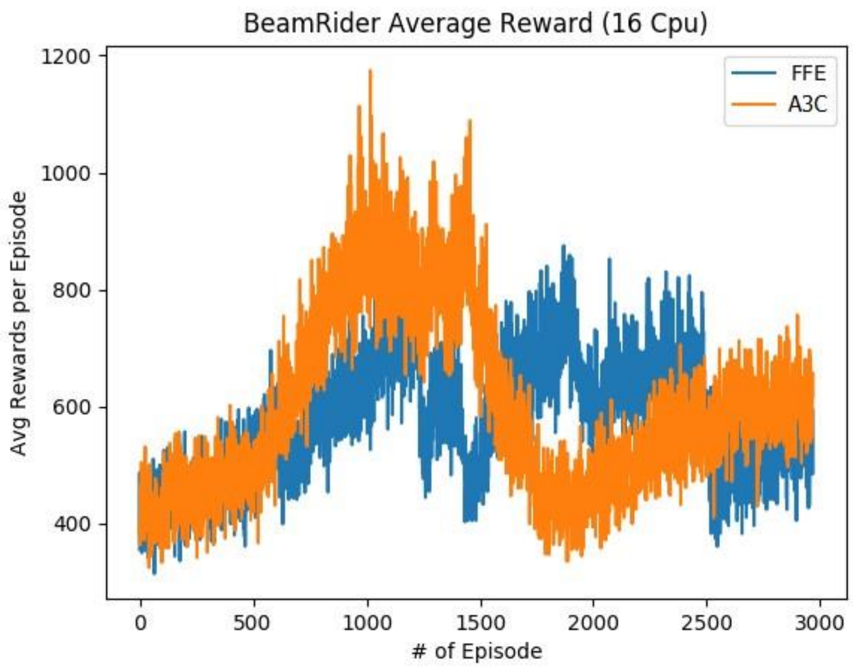

Figure 5: shows the average reward all the agents achieved per episode playing Atari Beamrider. This chart is limited to reporting the results of each algorithm performing 100 million global steps.