5-2018

Improving Asynchronous Advantage Actor Critic with a More Intelligent Exploration Strategy

James B. Holliday
University of Arkansas, Fayetteville

Improving Asynchronous Advantage Actor Critic with a More Intelligent Exploration Strategy

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

James Bradley Holliday
University of Arkansas
Bachelor of Science in Computer Science, 1998

May 2018
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____

Michael Gashler, Ph.D.
Thesis Director

_____

M. Gordon Beavers, Ph.D.
Committee Member

_____

Xintao Wu, Ph.D.
Committee Member

Abstract

We propose a simple and efficient modification to the Asynchronous Advantage Actor Critic (A3C) algorithm that improves training. In 2016 Google's DeepMind set a new standard for state-of-the-art reinforcement learning performance with the introduction of the A3C algorithm. The goal of this research is to show that A3C can be improved by the use of a new novel exploration strategy we call "Follow then Forage Exploration" (FFE). FFE forces the agents to follow the best known path at the beginning of a training episode and then later in the episode the agent is forced to "forage" and explores randomly. In tests against A3C implemented using OpenAI's Universe-Starter-Agent, FFE was able to show on average that it reached the maximum score faster.

Acknowledgments

Thank you to Dr. Michael Gashler. Without your help this thesis may have never taken shape.

Contents

1 Introduction **1**

2 Related Work **5**

3 Asynchronous Advantage Actor Critic (A3C) **7**

4 Follow Then Forage Exploration **9**

5 Experiments **13**

6 Conclusion and Discussion **18**

References **19**

Appendix **21**

Chapter 1

Introduction

The three main branches of Machine Learning (ML) are supervised, unsupervised, and reinforcement learning and all have been aided by the introduction of Deep Neural Networks (DNNs). A DNN is a neural network with multiple hidden layers between the input and output layers, which enables the neural network to approximate more complex functions.

Stated simply, in reinforcement learning an agent explores an environment seeking to learn an optimal policy. Each "location" an agent experiences is called a state. A policy is defined as a mapping from states to corresponding actions. When a reward is discovered, the reinforcing component of reinforcement learning takes place whereby the state and action that lead to the reward is given a corresponding utility or value, and that value is propagated (with some discount factor) to states and actions leading up to the reward. This results in a path where an agent can follow the higher utilities to maximize its long-term rewards. In order for the agent to discover any reward the agent must choose actions. Actions move the agent from state to state. Before rewards are known all states are equally valued and generally random actions are made until the agent discovers a reward. Still the found reward might not be the only reward nor the best reward possible so there needs to be a mixture of the agent following the increasing utilities (exploiting what it knows) and choosing actions that lead to unknown states (exploring what it does not know). This trade-off is a classic problem in reinforcement learning and is generally known as "Exploitation versus Exploration."

Q-Learning (Watkins, 1992) is an example of this type of reinforcement learning. Q-Learning works by learning a utility function we denote as Q(i,a) where the inputs of the function are states and actions. States here are the same as defined by Markov Decision Processes (MDP) (Bellman 1957). 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 (a) for a given state (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 (Bellman 2003):

$$Q(i,a) \leftarrow (1-\alpha^k)Q(i,a) + \alpha^k \left[r(i,a,j) + \gamma \max_{b \in \mathcal{A}(j)} Q(j,b) \right] \tag{1.1}$$

In this formula, α is a learning rate, which controls how much the Q value is changed for each occurrence of a and i. The closer 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(i,a,j) is the reward function, that for a given state (i), action (a) and subsequent state (j) returns a reward value. The two states of the reward function are i and j where i is the current state where action a is performed to arrive in state j. The reward is added to the maximum Q value when we check each possible action (b) for state (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 a in state i always leads to state j. In stochastic environments the learning rate is decreased to zero over time, where stochastic is defined as there is a chance that action a in state i leads to state 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.

There are many known methods for balancing between exploitation and exploration. When the state and action space is discrete, optimal solutions are possible. Bayesian Reinforcement Learning (Ghavamzadeh, 2015) is an example of reinforcement learning that can generate an op-

timal 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 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 is very small.

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 (Williams, 1991). 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.

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 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 reinforcement learning 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. This leads to agents exploring more where exploration is most needed, which leads to faster learning times.

Chapter 2

Related Work

In the area of reinforcement learning the last few years have been filled with landmark achievements and ground breaking research (Schmidhuber, 2015). In 2013, Mnih introduced Deep Q Learning in the form of the Deep Q Network (DQN) (Mnih 2013, 2015). DQN uses Q learning as described in the introduction except in the case of DQN the Q table is replaced by a DNN. DQN was able to reach human and beyond human level ability playing several specific Atari games. In 2016, Asynchronous Reinforcement learning (Mnih, 2016) 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. Our research is an effort to improve A3C.

Exploration research in reinforcement learning is not a new topic (Kaelbling, 1996), but since the emergence of deep learning many new efforts have been made to improve this important aspect of reinforcement learning. Because reinforcement learning 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. (Stadie, 2015) (Houthooft, 2016) and (Pathak, 2017) 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. (Silver, 2016) and (Tang, 2017) 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 (Silver, 2016), which was able to defeat some of the worlds 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.

(Dorigo, 2006) 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 reinforcement learning (Grondman, 2012) 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. (Baird, 1993) showed how to generate a signal called an advantage (A) for state (s) and action (a). Where Q is the same Q value as in Q learning and V is the value associated with given state (s).

$$Advantage : A(s,a) = Q(s,a) - V(s) \tag{2.1}$$

(Sutton, 2000) 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.

Chapter 3

Asynchronous Advantage Actor Critic (A3C)

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 (Sutton, 2000). The actor calculates the policy, (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 (Mnih, 2013) relied on a single agent while for their research, A3C, (Mnih, 2016) 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 (). The entropy (H) of the policy is also calculated.

$$Value\ Loss:\ L = \Sigma(R - V(s))^2 \tag{3.1}$$

$$Policy\ Loss:\ L = -log(\pi(s)) * A(s) - \beta * H(\pi) \tag{3.2}$$

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 regards 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.

Chapter 4

Follow Then Forage Exploration

When any state in an environment is equally likely and there isnt much progression from start to end, exploring at any time makes sense, but if an actual path from start to end has been found then exploring at the beginning of a run through an environment is less valuable than following the best known path and exploring later. We define this idea of a run through an environment as a single episode, and for our experiments this would be a single Atari game played from start to game over. In simple terms, the agent already knows what to do in the beginning, and it also knows what to do when the reward is near, but it is not sure of what is best to do in the middle. This is the foundational idea supporting FFE. With FFE we can ensure that the agent exploits closer to the beginning and end of the episode and is more likely to explore in the middle.

To evaluate A3C (Mnih, 2016) used multiple experiments, but the majority of those tests were playing various Atari games. That was accomplished through a program called the Atari Learning Environment (ALE). 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 (USA). USA is a Python program that we used as our benchmark. We then used a modified version of USA with FFE to compare against the benchmark results.

Algorithm 1 Asynchronous advantage actor-critic pseudo-code for each actor-learner thread. (Simplified)

//Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
//Assume thread-specific parameters vectors θ' *and* θ'_v
//Assume thread-specific variables m, n, o *and* p
Initialize m, n, o and $p \leftarrow 0$
Initialize thread step counter t $\leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state s_t
    repeat
        Perform FFE
        Receive reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t **or** $t - t_{start} == t_{max}$
    if terminal s_t **then**
        Perform FFE Update
    end if
    $R = \begin{cases} 0 \text{ for terminal } s_t \\ V(s_t, \theta'_v) \text{ for non-terminal } s_t \end{cases}$
    for $i \in \{t-1, ..., t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients θ' and θ'_v
    end for
    Perform asynchronous update of θ' and θ'_v
until T >Tmax

Algorithm 2 Follow then forage pseudo-code for an actor-learner thread. (FFE)

if $m > 0$ **then**
    Choose a_t according to policy $\pi(a_t|s_t;\theta')$
    $m \leftarrow m - 1$
else
    if $n \geq 0$ **then**
        Choose a_t from a uniform distribution in a
        $n \leftarrow n - 1$
    else
        Choose a_t according to policy $\pi(a_t|s_t;\theta')$
    end if
end if
Perform a_t

Algorithm 3 Follow then forage parameter update pseudo-code for an actor-learner thread. (FFE Update)

//Assume meta parameter Φ *and* Ψ
$U \leftarrow Unif$ ▷ Unif denotes the random number generator
$o \leftarrow o + 1$
$p \leftarrow (o-1)/o * p + (1/o) * t$
$m \leftarrow U * p$
if $U < \Phi$ **then**
    $n \leftarrow 0$
else
    $n = U * (p - m) * \Psi$
end if

Algorithm 1 describes a simplified version of A3C, derived from work by (Mnih, 2016). It is not a primary contribution of this work. Algorithms 2 and 3 are the primary contributions of this thesis. Algorithm 2 determines how the agent balances exploration with exploration. Algorithm 3 describes how parameters are updated.

Algorithm 3 keeps track of a running value of the agents completed episode count (o) and average episode length (p). For example, if FFE is used while learning to play an Atari game this would be the number of actions required on average for an Atari game to reach the game over state. At the end of each episode (ie. game) p is calculated from previous experience and is multiplied by a random number percentage to set the Follow/Forage threshold. After each action the agent will decrement its Follow (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 (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 exploiting. The use of the p ensures that the values for m and n are dynamic and diverse in each episode.

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 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. The intuition for only foraging half of the time 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. 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.

Chapter 5

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 USA, and for our modified version we added FFE to the default USA implementation. The environments tested were various Atari games as implemented by OpenAIs GYM.

All of 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 of 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 USA. 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 5.1 shows USA modified with FFE (indicated in blue) outperforms the default USA (orange) by reaching a higher score faster. For this test entropy was left at the default value.

We also tested a range of different entropy scalers to see how entropy affected the performance of the algorithm. Figure 5.2 shows that when entropy is not used FFE alone is not sufficient to generate a good result (NoE). Figure 5.2 also shows when we tried scaling entropy by .001 (SmallE) instead of the default .01 (DefaultE) we obtained an interesting result: SmallE actually starts to improve in a fewer number of episodes, but both algorithms reach the max reward about the same time. We found that both SmallE and DefaultE 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 be seen in Figure 5.3. 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.

Our next experiments involved more challenging Atari games. We tested Boxing, Amidar, and Beamrider. Figure 5.4 shows the results of experiment of the default USA and USA 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 default USA by reaching a higher score faster. Figure 5.5 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 (Mnih, 2016) which was 176 after four days of training. Lastly, Figure 5.6 shows the results of the experiment with the same algorithms playing Atari Beamrider. In this experiment FFE failed to outperform default USA. This result is perhaps due to instability in the models DNN. Figures 5.4, 5.5 and 5.6 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 USA states that the algorithm is tuned for good Pong performance. Meaning USA might perform poorly on other Atari games. Here is a list of some differences between (Mnih, 2016) and USAs implementation of A3C. The original A3C used a shared optimizer for all agents, and USA uses distinct optimizers for each agent. Also to note USA 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.

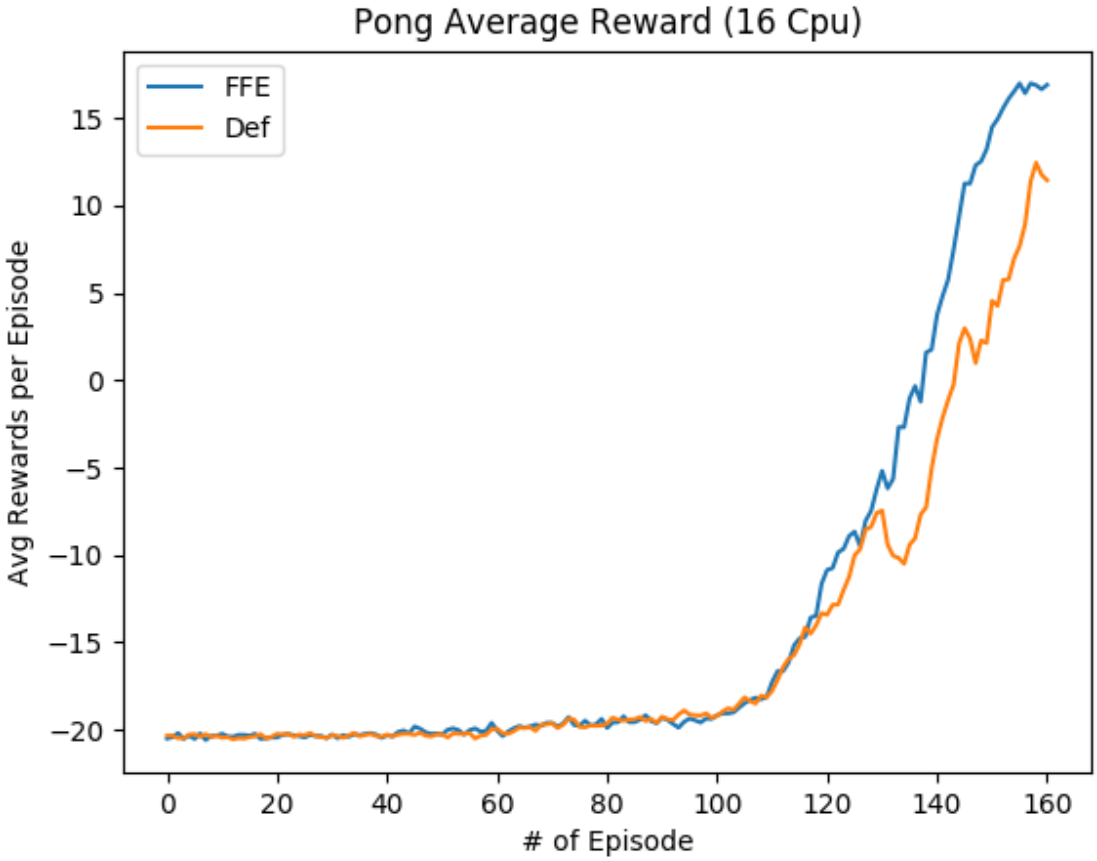

Figure 5.1: shows the average reward all the agents achieved per episode playing Atari Pong. Def represents the Universe-Starter-Agent with default configuration. FFE represents the Universe-Starter-Agent modified with Follow Then Forage exploration. This chart limits the results to each algorithm performing 4 million global steps.

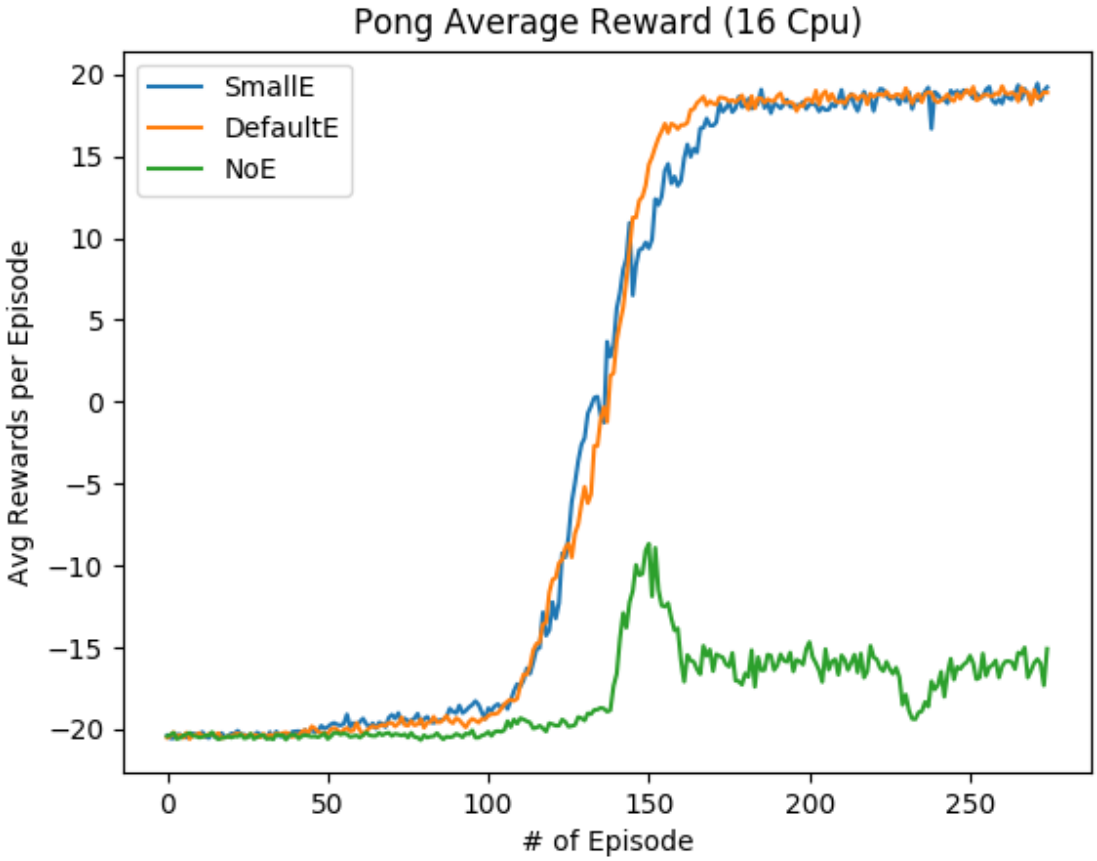

Figure 5.2: shows the average reward all the agents achieved per episode. Each line represents the Universe-Starter-Agent modified with Follow Then Forage exploration. SmallE represents the algorithm using an entropy scaled by 0.001. DefaultE represents the algorithm using an entropy scaled by 0.01. NoE represents the algorithm with entropy scaled to zero. This chart limits the results to each algorithm performing 10 million global steps.

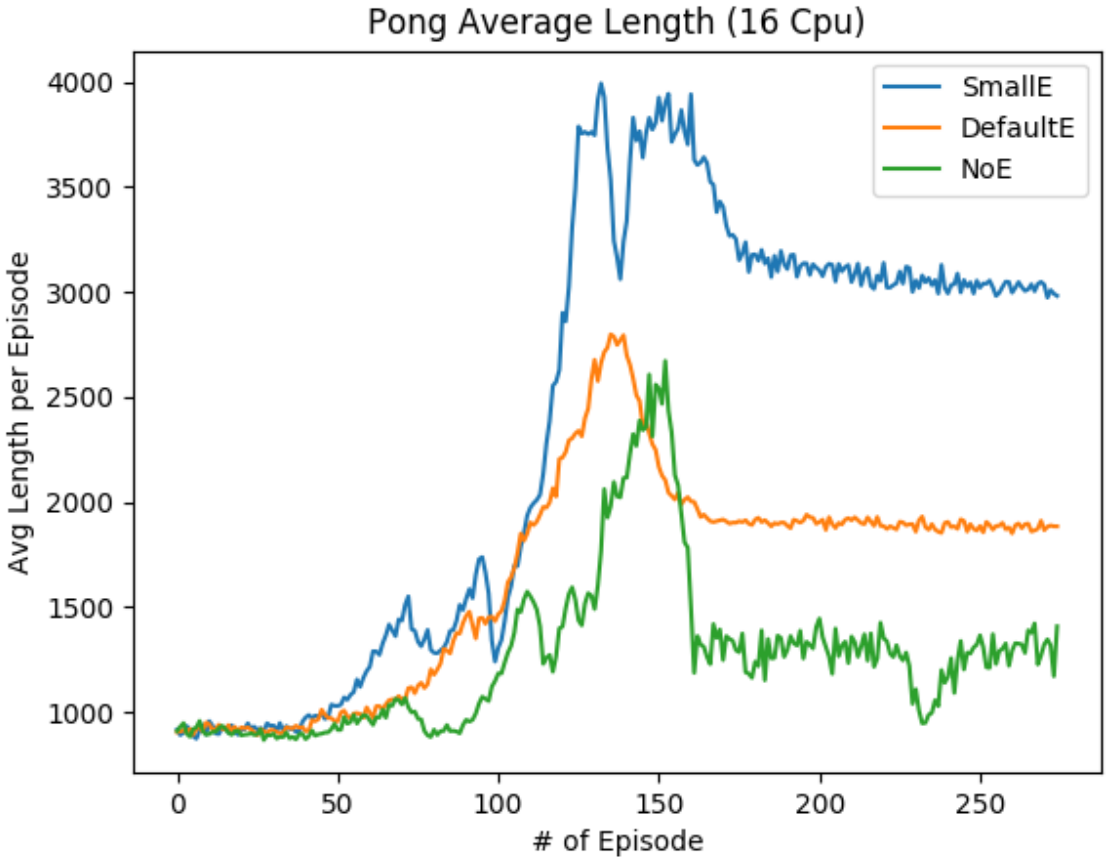

Figure 5.3: shows the average length (or number of moves) to complete an episode for all the agents per episode. Each line represents the Universe-Starter-Agent modified with Follow Then Forage exploration. SmallE represents the algorithm using an entropy scaled by 0.001. DefaultE represents the algorithm using an entropy scaled by 0.01. NoE represents the algorithm with entropy scaled to zero. This chart limits the results to each algorithm performing 10 million global steps.

Chapter 6

Conclusion and Discussion

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. (Mnih, 2016) listed a Pong training time at taking two hours. (Mnih, 2016) 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 evaluate an implementation of A3C that uses a shared optimizer instead of distinct optimizers.

Entropy plays an interesting role in all of 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 also scaled the computed Forage value smaller. Because of these parameters more testing is needed to determine their optimal values.

References

Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, page 679, 1957.

Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28, Nov 2006.

Mohammad Ghavamzadeh, Shie Mannor, Joelle Pineau, Aviv Tamar, et al. Bayesian reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 8(5-6):359, 2015.

Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert 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)*, 42(6):1291, 2012.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, page 1109, 2016.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237, 1996.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 02 2015.

Volodymyr Mnih, Adri Puigdomnech, Mehdi Mirza Badia Badia, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *33rd International Conference on Machine Learning (ICML)*, page 1928, 2016.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, volume 2017, 2017.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85, 2015.

David Silver, Aja Huang, Arthur Guez Chris J. Maddison, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 01 2016.

Bradly C Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, page 1057, 2000.

Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, page 2750, 2017.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279, 1992.

Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241, 1991.

Appendix

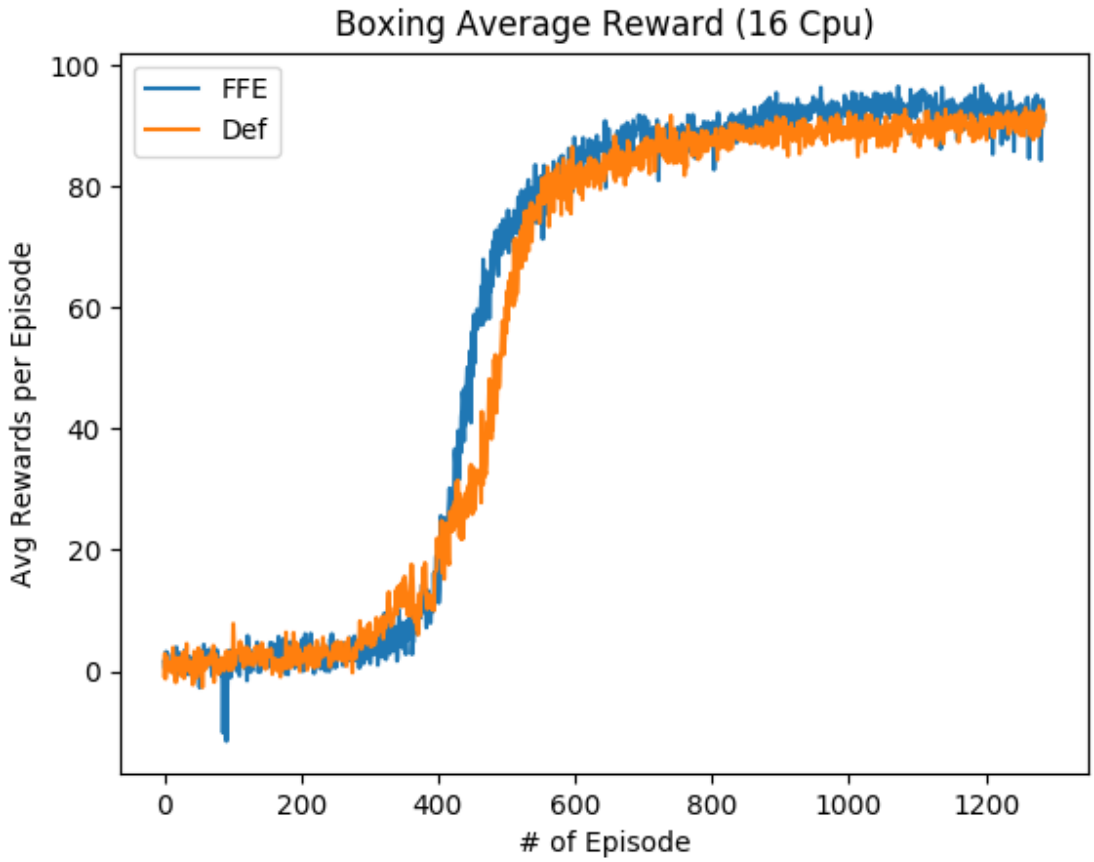

Figure 5.4: shows the average reward all the agents achieved per episode playing Atari Boxing. Def represents the Universe-Starter-Agent with default configuration. FFE represents the Universe-Starter-Agent modified with Follow Then Forage exploration. This chart is limited to reporting the results of each algorithm performing 40 million global steps.

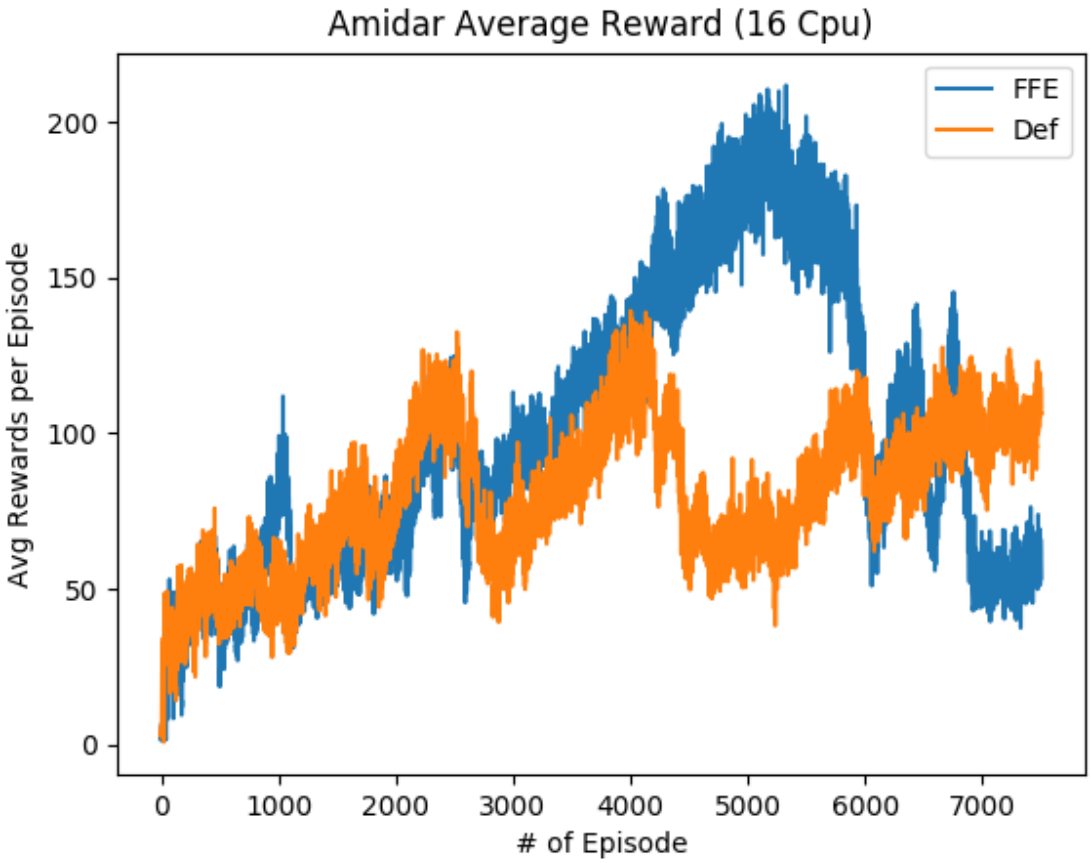

Figure 5.5: shows the average reward all the agents achieved per episode playing Atari Amidar. Def represents the Universe-Starter-Agent with default configuration. FFE represents the Universe-Starter-Agent modified with Follow Then Forage exploration. This chart is limited to reporting the results of each algorithm performing 60 million global steps.

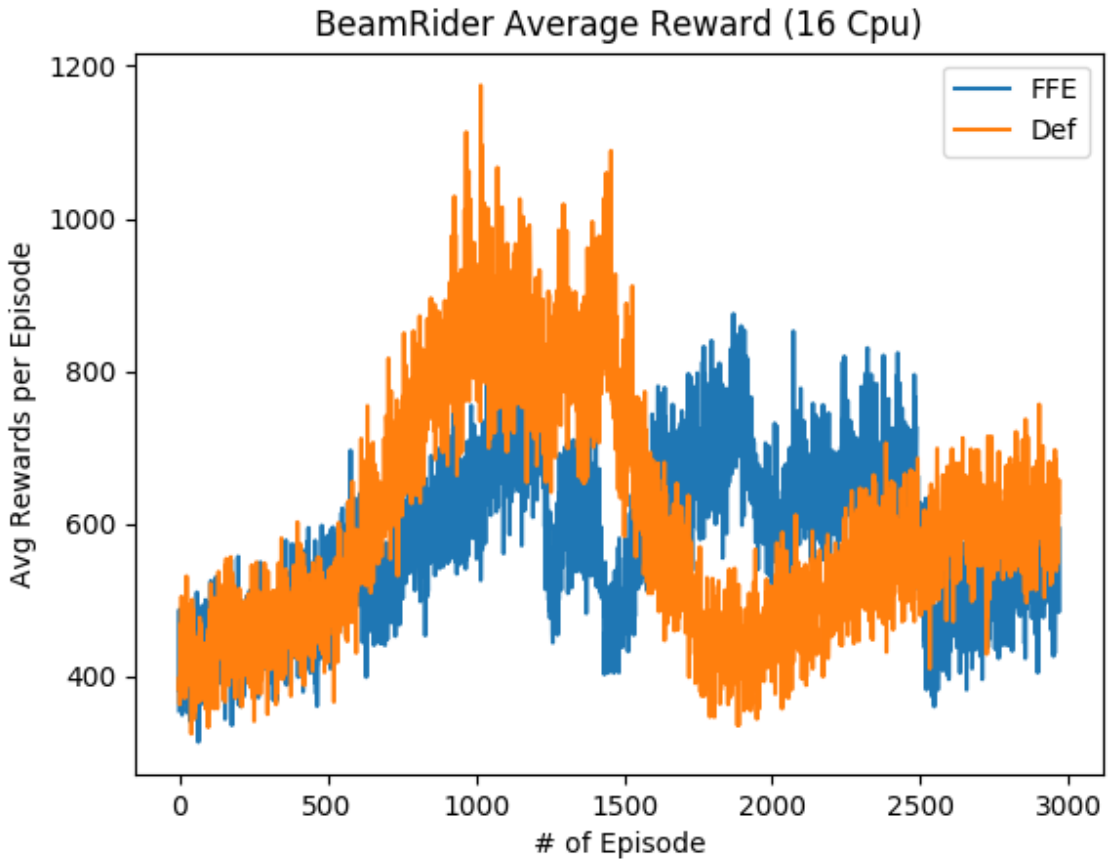

Figure 5.6: shows the average reward all the agents achieved per episode playing Atari Beamrider. Def represents the Universe-Starter-Agent with default configuration. FFE represents the Universe-Starter-Agent modified with Follow Then Forage exploration. This chart is limited to reporting the results of each algorithm performing 100 million global steps.