Learning Slither.io

Amol Kapoor
Department of Computer Science
Oxford University

Adam Cobb
Department of Computer Science
Oxford University

Abstract

Reinforcement learning aims to create artificial intelligence models that can dynamically learn how to complete various tasks without preexisting information and minimal guidance. While these agents have shown great success in solving simple tasks, only recently has the scale and complexity of reinforcement learning models rivalled human intelligence and creativity. Thus, few models exist for tasks where dozens of players may be competing at one time. This paper presents a model for learning the popular internet multiplayer game Slither.io. Our agent was trained using a combination of inverse reinforcement learning methods and the reinforcement learning A3C algorithm. After significant training, our agent was unable to successfully learn the Slither.io environment.

1 Introduction

The goal of reinforcement learning is to create agents that can learn to operate in unknown environments. Because reinforcement learning agents can be applied in any system with a state and a reward system, reinforcement learning has applications in numerous fields. Reinforcement learning promises a generalized artificial intelligence. Such an artificial intelligence could help humans better understand and solve complex problems[1].

1.1 Early Research

Reinforcement learning is a rich field with a long history that was originally inspired by neuropsychological research into how humans learn. However, though reinforcement learning has been studied for many years, the principles of the field have remained fairly consistent over time. Reinforcement learning tasks consist of a set of environment states ***S***, a set of actions available to the agent ***A***, policies defining transitions between states and actions, and rules defining the immediate reward of a transition from one state to the other. A reinforcement learning agent interacts with the environment in discrete time steps. Starting from state ***s*** in ***S***, at each time ***t***, the agent receives an observation ***o*** and a reward ***r***. The agent then chooses an action ***a*** from the set ***A*** based on policy π , and moves the environment to a new state ***s'***. The goal of an agent is to maximize reward by changing the policies it follows[1].

A natural development from the basic definition of reinforcement learning tasks is the development of internal state representations by the agent. This internal representation, called a value function, maps a state to the amount of reward expected from that state based on a policy:

$$v_\pi(s) = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$$

where gamma is a parameter representing the discounted value of rewards from future states[1]. This definition for the value of a state naturally lends itself to an optimality equation, or the Bellman

Equation, defining the recurrence relationship between the value of a state and the value of its successors.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma(v_\pi(s')]$$

The definition for an optimal value function also naturally lends itself to a definition for the optimal policy - simply, take the action that leads to the state with the highest value, or greedy behavior.

$$\pi(s) = argmax \quad q_\pi(s,a)$$

where *q* represents the state action value, or the value of taking a certain action in a certain state[1][1]. Traditional reinforcement learning algorithms attempt to determine the value function through the iterative experience of the agent in the environment. Important algorithms include Monte-Carlo methods, TD(λ) methods, Q-Learning, and SARSA[1], though explaining each of them is out of the scope of this paper.

Because the agent must maximize reward, reinforcement learning algorithms need to determine a balance between exploration and exploitation. Without exploration, the agent may never find the optimal reward path; without exploitation, the agent may never actually maximize reward based on information that it has discovered. Mechanisms for balancing exploration and exploitation is an ongoing field of study within reinforcement learning[1].

1.2 Policy Gradient Methods and Actor Critic

Early reinforcement learning techniques focused on the value function approximation, with the default action policy being represented as greedy to the value function [2]. Value function approximation methods have some significant problems, including being oriented towards deterministic instead of stochastic policies and having discontinuous changes in action selection due to arbitrarily small value changes [2]. Further, many value function approximation methods do not have convergence guarantees. Policy gradient methods aim to solve these problems by estimating the policy directly. Actor critic methods are a subsection of policy gradient methods that combine value function approximation (the critic) to supply the policy (the actor) with low-variance knowledge of the performance of a given policy[3][4].

1.3 Deep Reinforcement Models

Traditional algorithms for reinforcement learning often store state values and policy actions in arrays representing each state. These storage mechanisms grow exponentially with new states, making traditional reinforcement learning algorithms untenable for complex problems that often have hundreds or thousands of states. Complex reinforcement learning tasks therefore require function approximation methods to determine optimal value functions and perform policy evaluation. Thanks to the explosion of computing power in the last decade, deep learning function approximators have seen significant success in solving complex reinforcement learning tasks[5][6].

Two common building blocks of deep learning models are particularly appealing for reinforcement learning algorithms. Convolutional networks have shown success in understanding position and identifying objects in images[7]. Long short-term memory modules (LSTMs) have shown success in understanding sequential data feeds over time[7]. Together, these elements of deep neural networks allow a reinforcement learning agent to use an unprocessed input stream (e.g. pixels on a computer screen) to understand how actions lead to outcomes.

[1]Finding the q function is trivial from the value function, and takes a similar form:

$$q_\pi(s,a) = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

1.4 A3C

While modern hardware allows for large neural networks, these systems are often limited to expensive and massive GPU server clusters. A recent learning algorithm proposed by Mnih et. al. optimizes deep reinforcement learning for easily available multicore CPUs[8]. The algorithm, known as Asynchronous Actor-Critic (A3C), asynchronously updates a global network from the experiences generated by numerous agents, dubbed workers, running simultaneously on different threads. These workers then synchronize with the global network after some amount of time or at the end of an episode. The A3C algorithm proposes the following update rule for parameters θ defining the value function and policy for a given state:

$$\nabla_{\theta'} log_{\pi}(a_t|s_t, \theta')A(s_t, a_t, \theta, \theta_v)$$

where A is an estimate of the advantage function, a function that estimates how much better an action was to take over any other action (or the advantage of taking that action). We use the advantage function proposed by Schulman et. al.[9], shown below:

$$\sum_{l=0}^{\infty}(\lambda\gamma)^l \delta_{t+1}^V$$

where δ^V is defined as $r_t + \gamma V(s_{t+k}, \theta_v) - V(s_t, \theta_v)$. This advantage function addresses the significant data requirements of previous policy gradient methods[9], and is therefore suitable for a large state environment like Slither.io.

The asynchronous method accounts for exploration through the asynchronous action of numerous simultaneous agents, each experiencing a different portion of the environment. In order to further encourage exploration, we follow Mnih et. al. in adding the entropy of the policy π to the loss function[8].

1.5 Inverse Reinforcement Learning

Inverse reinforcement learning, or apprentice learning, encapsulates methods and algorithms that teach an agent a reward function from an expert who already knows the environment[10]. For environments with numerous complex states, it can be difficult for an agent to 'make the first jump' to understanding how to play the game. We use inverse reinforcement learning methods interspersed with self-learning to better train the Slither.io agent. Current research in the field will compare the reward function proposed by the agent with the true reward function[11]. However, in our case the true reward function is not known. We instead treat the expert action as the 'true' action and compare action-by-action with a softmax cross entropy function. Gradients can then be backpropagated through the model based on the 'difference' between the actual action and the predicted one.

2 Model

2.1 Network Construction

Our model consists of a series of convolutional network layers with an LSTM network on top. The value function and the policy function are both determined by separate fully connected linear layers on top of the LSTM network. The final model can be viewed in Figure 1. State of the art research recommends using elu activation neurons[12], initialized using the He initialization function[13].

As per the A3C algorithm, at any given time there were multiple worker networks running in parallel. Gradients were collected based on the experiences and rewards of these threaded workers. The gradients were then used to update a global network located on the master thread. After an update, the global network would sync network variables with the worker that provided the update. Thus, at any given time, the global network contained the most recent updates from all of the parallel threads, while the individual threads would all eventually update to the most recent network[8]. This network structure can be viewed in Figure 2.

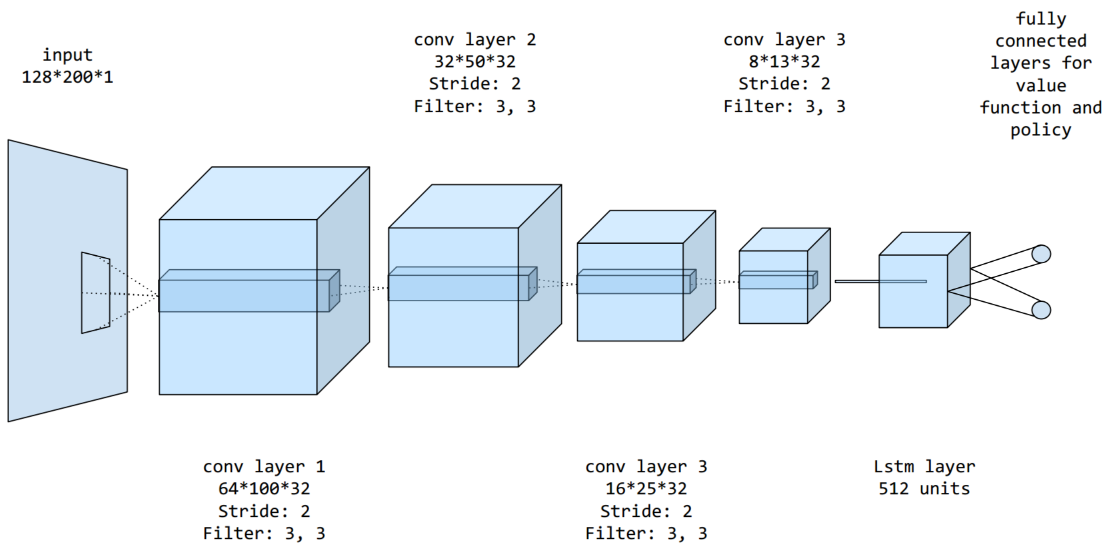


Figure 1: Visualization of the final model used for the Slither.io agent. The model consists of convolutional network layers each with a filter of 5x5 and a stride of 2, followed by an LSTM layer with 512 units, followed by two fully connected linear layers for the value function and the policy function respectively.

2.2 Training

The model was trained with a combination of inverse reinforcement learning and reinforcement learning techniques. In the inverse reinforcement learning stages, the model assumed the human action was the perfect action for the given state. The inverse reinforcement learning was used as a corrective measure. The model was allowed to self train for most of the training steps. If the model seemed to get caught in a rut - for example, only choosing a single action - inverse reinforcement learning would be used to correct the model. In this way, the model learned with a mix of self taught and guided learning.

The model used the softmax cross entropy function for its loss calculation. In the reinforcement learning stages, the model implemented the A3C algorithm stated above, and used the A3C loss calculation. To prevent exploding gradients and weights, the model also used clipped gradients, dropout, and L2 regularization. In both stages, the Adam Optimizer was used for propagating gradients.

3 Experiment

3.1 Setup

3.1.1 Model Construction

The model was built using Tensorflow 1.0 without CUDA. Convolutional layers were built using the implementation found in *tf.nn.conv2d*. The LSTM layers were built using the implementation found in *tf.contrib.rnn.LSTMCell*. Training and testing was done on an ASUS ROG GL551J.

While there are numerous hyperparameters that can affect the model, due to lack of time and computational ability we were unable to rigorously examine the optimal hyperparameter configuration. Similarly, we were unable to rigorously search for the optimal network structure. The full list of hyperparameters that we used for our final model can be found in Table 1.

3.1.2 Environment

Slither.io is a snake game where the goal is to create as long a snake as possible. The game takes in any one of six actions - straight, left, right, fast-straight, fast-left, and fast-right - and returns as

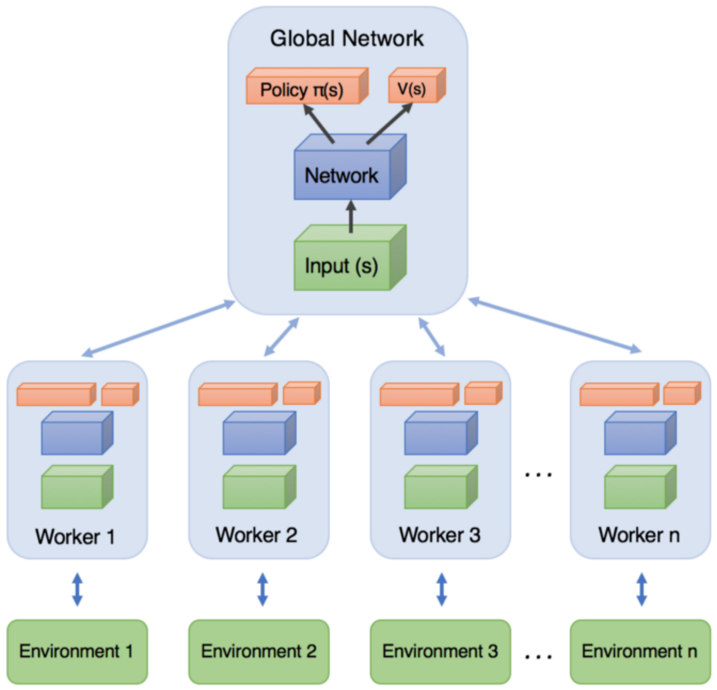


Figure 2: Visualization of A3C network[14]. The global network spins up n threads (generally n is less than or equal to the number of cores on the CPU), each of which runs a separate agent in its own version of the environment.

reward the delta in the length of the agent. The game is episodic, in the sense that after some amount of time the player agent will die (due to crashing into an opponent or the boundary of the stage) after which the game will reset from the beginning. Further, any fast action will cause the snake to lose length, therefore earning negative reward. If the action for speeding up is held until it is impossible to lose length, the agent will go back to its normal speed and will no longer accrue negative reward even if the fast action is held. The game has the capacity to host 500 players in a single server at a single time. After each death, the agent would begin again in a new server.

The Slither.io environment was set up using OpenAI Universe package[15]. At each time step, Universe would provide the model with a grey scale image representing the Slither.io environment. An example is shown in Figure 3. Thus, there were over eight million states. Universe would also provide the reward earned in the last step. At each frame, the model would input one of the six actions to the environment. Universe environment parameters can be found in Table 2. Due to limitations in computation, all training was done online - instead of storing and replaying memories, we had the network play in the Slither.io environment more often.

3.1.3 Data Collection

At the end of each episode or at the end of 1000 steps, the global network would collect data from the agent. For quantitative results, this data was compared to data collected on a random agent that, for each frame, selected a random action of the six actions available. For qualitative results, we observed and recorded the behavior of the agent in game.

Table 1: Hyperparameters for Slither.io agent network.

Parameter	Value
Network Construction	
Convolution Layers	4
Filter Shape	(3, 3)
Stride	(2, 2)
Output Channels	32
LSTM Units	512
Loss Function	
Value Function Loss Constant	0.5
Entropy Constant	0.01
Discount Constant	0.99
Learning Rate	0.1
Regularization	
Max Gradient Clipping	60.0
Dropout Probability	0.5
L2 Constant	1e-5

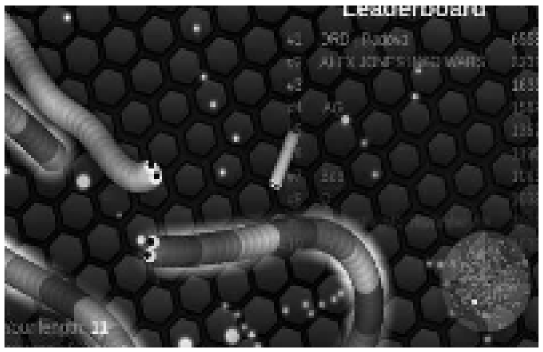


Figure 3: An example state observation given by Universe.

3.2 Results

3.2.1 Quantitative

After thousands of training steps, each representing an average of 300 different observation-action decisions played through by the Slither.io agent, our model was unable to perform better on average than the random model. The loss for the trained model over all time steps can be seen in Figure 4. The total reward over time of the trained model compared to that of the random model over all time steps can be seen in Figure 5. These results suggest that the model was not learning, or that the training time was not a sufficient.

It is important to note the significant jumps that can be observed both in the loss and the total reward. These jumps show the unpredictable nature of Slither.io, and likely make it difficult for an agent to determine how its actions impact the environment.

3.2.2 Qualitative

For qualitative tests we examined how our agent performed at various points in its training. If the agent met the following criterion, we considered the environment solved:

1. Minimize use of the speed up action to avoid negative reward.
2. Move to avoid the outer boundary.
3. Move to avoid running into enemy agents.

Table 2: Hyperparameters for Universe Slither.io environment.

Parameter	Value
Input Height	128
Input Width	200
FPS	10.0
No Reward Value	0
Game Over Reward Value	0
Number of Actions	6

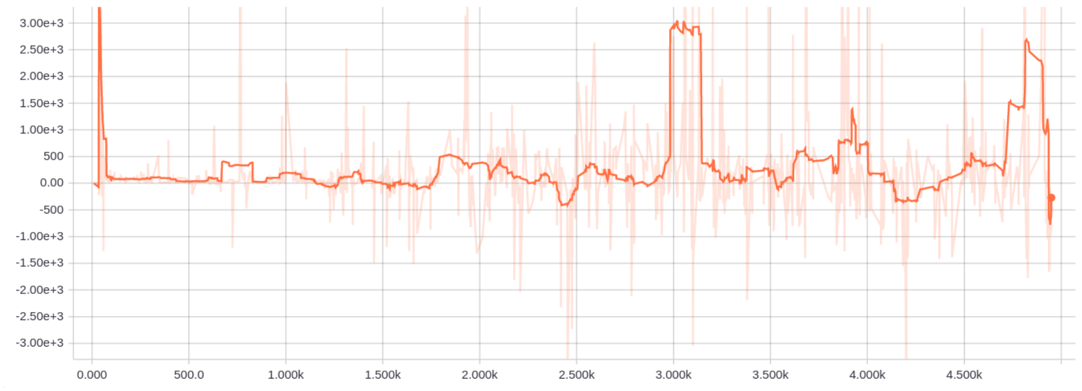


Figure 4: Loss at each time step for the trained model. The loss is smoothed with a rolling window average of about 500 steps. An agent that learned how to play the game from experience should have a decreasing average loss over time.

Our agent was unable to meet any of these goals.

Our final agent was tested using a reward scheme that did not punish game overs or not growing. However, it is worth noting that we did test using alternative reward schemes before settling on the final version. Surprisingly, whenever the reward scheme had any kind of negative reward on game over, the agent would generalize to choosing only a single action[2]. Single action selection occurred even after the implementation of alternating inverse reinforcement and regular reinforcement learning stages. Notably, the selected action distribution did change significantly for each inverse reinforcement learning stage. This suggests that, as expected, the human intervention successfully pulled the model back into a more exploratory state.

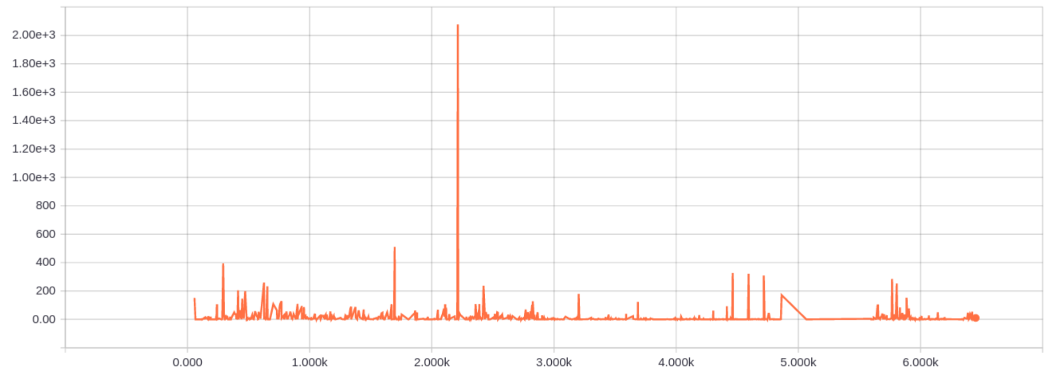
Single action selection differs significantly from the random agent, which (obviously) showed no changes in its action selection. We expected that the agent would learn to avoid enemy agents due to the game over negative reward. We believe that this was caused due to an error in the Universe environment package. The Slither.io environment would continue polling for actions for a few moments after the agent had died before a game over was registered and the negative reward given. Thus, the agent was unable to tie the actions that led to its death with the negative reward, as the negative reward was too far away from the most relevant actions.

When the reward scheme did not punish game overs, we found that the agent had an evenly spread distribution of action choices. This can be seen in Figure 6. Notably, we expected but did not observe a bias against selecting fast actions.

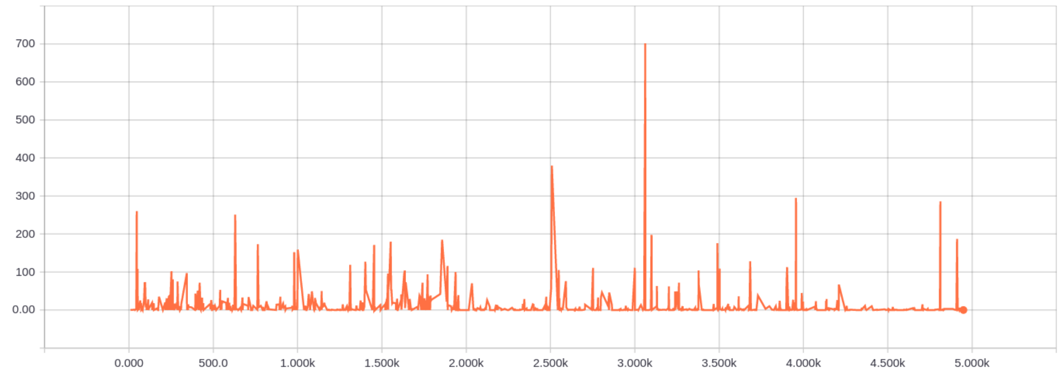
4 Conclusion and Future Work

In this paper we developed a reinforcement learning model to play Slither.io on the Universe platform, using state of the art reinforcement learning architectures and paradigms. Though our results left much to be desired, they indicated the potential for an active area of research.

[2] We do not show the data for this model. However, an example recording can be seen here: https://youtu.be/M9YVTZ3aQqA



(a) Trained agent.



(b) Random agent.

Figure 5: Comparison of total reward gathered by the trained agent and the total reward gathered by the random agent. Both agents have close to the same average score of about 20. An agent that learned how to play the game from experience should have an increasing total reward over time compared to the random control.

Slither.io is a complex and chaotic game with state transitions that are uniquely difficult to learn due to the nature of multiple different human players simultaneously playing at any moment and the sheer number of possible states. The lack of consistency between enemy agents made predicting enemy behavior an extremely difficult task. Though we trained for over a week, this was likely nowhere near the amount of time necessary for the agent to properly generalize how its actions impacted the environment. Further, it is very likely that our model architecture was severely limited by computation - for example, attempting to increase the number of convolution layers or decrease the stride would cause the system to hang. These issues were further compounded by an inability to test for hyperparameters due to time constraints. With more training time and computational resources, we believe that our model will successfully improve.

We found relative success in our alternating inverse reinforcement learning approach. Each inverse reinforcement learning stage led to a more exploratory agent. The ability to nudge an agent into a more exploratory state is useful for long term training, and can be viewed similar to a hands-off student-teacher relationship where the teacher intervenes to prevent the student when stuck. While we did not formalize our human intervention schedule, we suspect that a more rigorous implementation of alternating between reinforcement learning and inverse reinforcement learning would speed up agent training significantly. We further recommend memory playback[5], a popular mechanism to aid reinforcement learning tasks by using memory replay as a simulated supervised learning environment.

The A3C algorithm - and, to the best of our knowledge, reinforcement learning in general - does not have a great solution to highly inconsistent state transitions. When combined with a large state size, it can be difficult for an agent to understand how its actions impact the environment. This is a significant problem in environments with multiple other agents, and makes learning games like Slither.io particularly challenging. Most reinforcement learning paradigms attempt to address

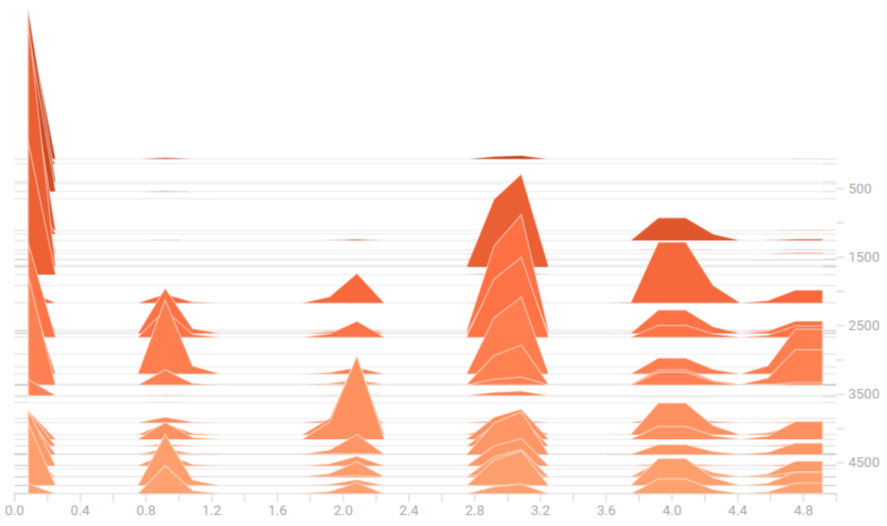


Figure 6: Frequency of action selected actions over time for a no-punishment reward scheme. The right hand axis is the step. The distribution peaks (six peaks for six actions) indicate the objective number of times a certain action was chosen on a given step. Note that as time goes on, the agent begins to select across all actions evenly. Even selection occurred even though the agent began by favoring a single action.

probabilistic state changes through the resulting expected value of the state in the agent value function. Though this may eventually converge, training time can be prohibitive. Attempting to tackle this problem without introducing external (i.e. not self learned) information to the model proved difficult.

While it is hard to predict what a deep learning model will learn, we suspect that our current architecture was insufficient at predicting the possible state transitions of a given state. Due to a lack of pooling layer, our architecture was unable to generalize the shape of an enemy agent across the environment. This significantly increased the number of states our agent had to process before our agent would learn to generalize. It also preventing enemy behavior in one area of the environment from influencing predicted outcomes of enemy behavior in another area of the environment. However, pooling layers remove the geo-spatial relationships between detected objects in an image. It is important to preserve these relationships through the convolutional layers for the network to understand the agent's location in relation to enemy agents and the environment. We propose a modification to our architecture that replaces the single stacked convolutional networks with multiple such stacks, some with pooling layers and some without. We believe that this would allow later layers of the network to identify both the presence of an enemy agent as well as its location. Further, we believe that a single LSTM layer is insufficient to accurately predict all of the potential future behaviors of an enemy agent. We propose a modification to our architecture that replaces our single unit LSTM layer with multiple LSTM units across the same layer. We suspect that each LSTM unit would be able to predict a different outcome given an input state, similar to how lower layers of a convolutional network can specialize to identify specific shapes.

Acknowledgments

Thank you Adam for an amazing class. I learned more than I could have hoped, and definitely feel better equipped for further studies into this fascinating field. I look forward to applying what I have learned both over the summer and well into the future!

References

[1] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 2012.

[2] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[3] 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):1291–1307, 2012.

[4] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *NIPS*, volume 13, pages 1008–1014, 1999.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] 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):529–533, 2015.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[8] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[9] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[10] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[11] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Deep inverse reinforcement learning. *CoRR*, 2015.

[12] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[14] Arthur Juliani. Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c), 2016.

[15] OpenAI. Universe, 2016.