Aditya Hans Prasad
Professor Richard Granger
Computational Neuroscience, COSC16
November 2019

Reinforcement Learning: A Mouse Learns To Distinguish Cheese from Traps

In this project, I will demonstrate how I designed a program and environment in Unity to teach a mouse with no prior knowledge of cheese vs trap to find cheese and avoid traps. I did so using a technique known as 'Reinforcement Learning'. My work here is implemented in Unity and C#, using Unity's ML Agents open-source plugin. The mechanisms here are largely inspired by ML Agents's collector neural network models. I was very interested in actually visualizing the implications of the reinforcement learning algorithms and architecture we learnt in class, and this is my attempt at recreating the mouse creature we learnt about in class.

A BRIEF INTRODUCTION TO REINFORCEMENT LEARNING

Reinforcement learning works similarly to how a pet dog learns how to do tricks — roll over, give its paw, or sit. What this means is that when our player performs the task we want it to perform properly, we give it a reward. Rewards are scalar values which are dependant on the extent to which the player completes its task. Additionally, we can also give a negative reward as a punishment for the player doing something we do not want it to do. We iterate on respawning the player and letting it makes its decisions, and provide it rewards based on the outcomes of its decisions. At the end of each iteration i.e. each round, we provide the player its reward, and in the next iteration, the player seemingly "learns" from the previous rounds and bases its future decisions off of the past.

Functionally, our players decisions are based off states. The current state of the player is derived from its previous states, and the output of the player depends on its current state. The output of the player then determines its future states, with rewards being the parameters that drive the change of states. Unlike supervised learning, we do not provide the player tags for single states, but provide it with rewards that allow it to associate sequences of states with wins or losses.

POLICY FUNCTIONS

Unity's ML Agents uses Policy Learning, which implements a Policy function. A policy function is a function that maps the agent's current state (or sequence of states) and maps it to an action to take. It therefore takes a state as a combination input of observation vectors and past carried over vectors and produces a prediction for what is the optimal set of actions to take for a given set of inputs. What is an optimal action? In this context, an optimal action is one that will lead to the agent to the maximum predicted rewards in the future. However, we can parameterize this to prioritize sooner rewards to later rewards or vice-versa. One prominent implementation of reinforcement learning algorithms is use policy learning techniques, which essentially apply a collection of policy functions to a multi-layered neural network in order to predict the best sets of action given input state vectors.

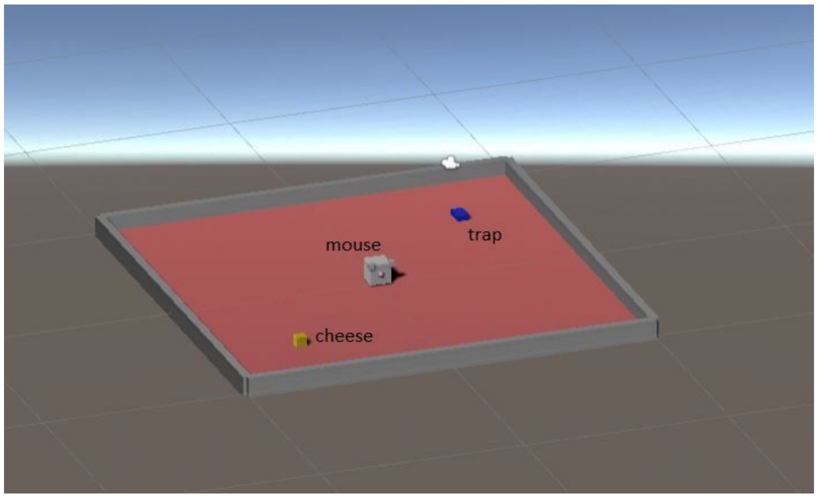
MACHINE LEARNING IN UNITY

I decided to implement my visualisation in Unity, using its open-source machine learning and artificial intelligence plugin, ML-Agents. ML Agents supports Reinforcement learning with a very comprehensive structure and framework. I will attempt to explain it below. Essentially, Unity makes it such that our environment is inhabited by two main components -- an agent and an academy. The agent is the character that makes decisions, that interacts with the environment, and the entity that iteratively learns and changes behaviour. The academy provides the agents the structure and rules by which it should learn -- it throws at the agent its rewards, its punishments and respawns objects as needed. ML-Agents is a plugin that is built on Tensorflow, and therefore uses and creates Tensorflow neural network models for Unity to use. As such, I did not need to implement the Reinforcement learning algorithm from scratch, but merely needed to apply the algorithm to a specific application by setting up the environment, the agent, and the academy. In designing and implementing this environment, I had to pay careful attention to what my inputs and outputs would be for the neural network model, and how to train the model with hyperparameter tuning in order for it to perform it task in the best way possible.

THE ENVIRONMENT

In my experiment, the first step was to create an environment for the mouse to move around in, since reinforcement learning is highly environment dependent. Below is an image of the environment, with an explanation for each component.

The **mouse** is our agent. He must learn to move around, earning rewards when he eats the **cheese**, and earning negative rewards (i.e. punishments) when he eats the **trap** poison**.** Cheese and traps are randomly generated around the platform (this is done by the MouseArea script). Associated to the mouse is an agent script -- this C# script provides methods and heuristics for how the mouse moves, behaves, and learns. More on that below.



PROGRAM DESIGN
Inputs: For the agent to see, I used Unity ML Agents' RayPerception3D heuristic, which provides us the **input** data. Our mouse essentially sees out the vector angles *(20, 90, 160, 45, 135, 70, 110)* from its forward axis to a distance of 50f, searching for objects of the type "food" and "trap". These 7 vectors provide us our observational input to reinforcement the neural network. These observation inputs are taken by the 'Mouse' C# script.
Outputs: The outputs to our approach are the mouse's actions. In this case, the mouse can either rotate to the left or right and either go forward or back. The mouses decisions in these processes are built off its previous learnings in previous runs, and soon the mouse learns to rotate and move in a manner that it goes towards cheese and avoids traps. These are also provided by the Mouse C# script.
Environment: Everytime the mouse eats cheese, it receives a reward of 3. Another cheese is randomly generated on the board. Likewise, everytime the mouse eats a trap, its reward goes down by 1. Another trap is generated on the board. These are controlled by the MouseArea and Item C# scripts.
Rewards: Additionally, when our mouse collides with a cheese, trap, or wall, it is given a reward of 3, -3 and -0.5 respectively. This reward mechanism is an important input to the neural network training process, and is the only contextualizing input that tells the mouse if it is doing something wrong or right.

TRAINING
Once I set up the environment, I wanted my mouse to finally learn. Unity ML Agents implements Reinforcement Learning through a technique known as Proximal Policy Optimization (PPO), which takes an input and provides mappings to subsequent actions in accordance with reward policies in order to maximize an agents rewards. PPO predicts the best subsequent set of actions to take from the current state. Training therefore involves many, many iterations and generations of the mouse running around the area and eating cheese, running into

traps. From each generation of action, the PPO algorithm applies its previous learnings to the next generation of mice, so that when we stop our training process, the mouse runs fully in accordance with the rules and environment we have set for it.

I ran the algorithm with a set of default hyperparameters first. I found that my mouse was very tentative and hesitant in his movements, and was jittering front and back without actually viewing the locations of cheeses and traps. I realized that I had to tune the default parameters in accordance with the input space, the rewarding objects, and the output space i.e. the set of actions that the agent should take.

PARAMETER TUNING

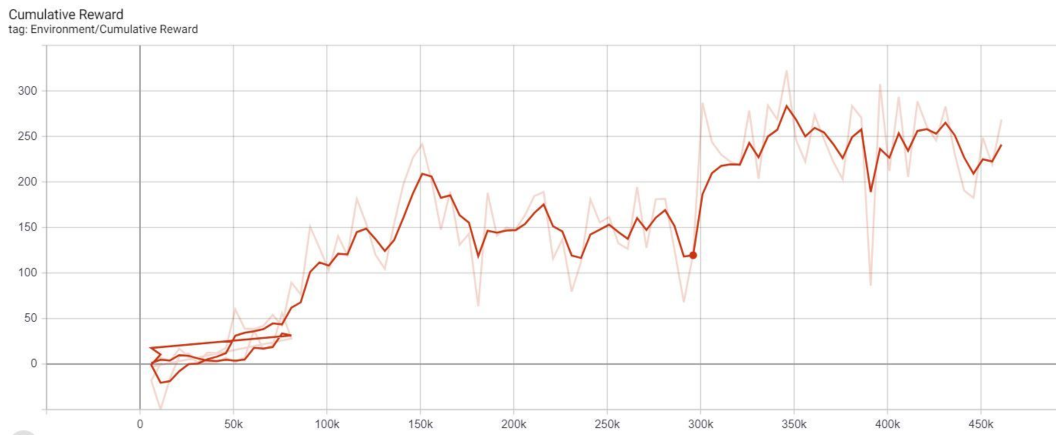
After much fine tuning and a few hours of training, I finally arrived at a decent generation of mouse. Below, I will list the important parameters I tuned and fit in order to effectively teach the mouse. Additionally, I monitored the learning process and the changes in the neural network's parameters via Tensorboard, a plugin that visualizes the evolution of Tensorflow neural networks. The definitions of these parameters are from Unity's ML Agents parameter tuning documentation.

1. **Trainer Type: PPO -** *We want to use the Proximal Policy Approximation implementation of Reinforcement Learning as provided by ML-Agents using Tensorflow frameworks.*
2. **Lambda: 0.99 -** *Lambda refers to how much the learning model's learning is influenced by its predicted rewards in the future i.e. its value estimate in the current state.*
3. **Buffer Size: 12000 -** *Buffer size refers to how long or how many observations the model should take before it updates its learning.*
4. **Beta: 0.001 -** *Beta refers to how how much the entropy i.e. the randomness of agent decisions changes with each update in learning. Below, one can observe how entropy decreases as a result of this beta value.*
5. **Learning Rate Schedule: Linear -** *As time increases, our learning rate linearly decreases. A linear schedule gave me the cleanest and most obviously visualizable divergence to optimal policy.*
6. **Num_Layers: 2 -** *Num layers refers to how many hidden node layers are present in the generated neural network model.*
7. **Sequence Length: 64 -** *Sequence length refers to how many previous sequence's learnings the current generation should remember and look back at. Since I wanted my mouse to always remember the past generations when it hit traps, I set this fairly large.*

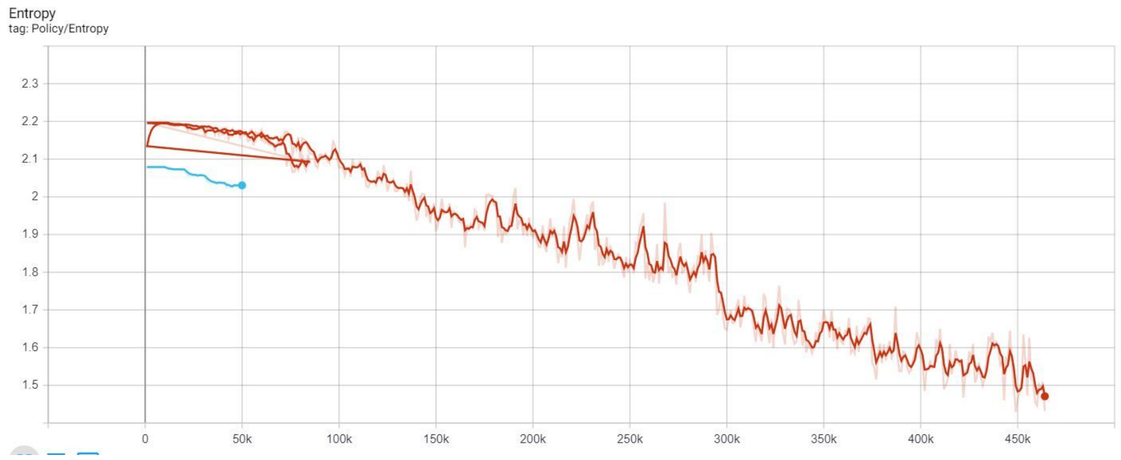
TRAINING VISUALIZATIONS

The following graphs were obtained using Tensorboard. Red indicates the smoothed curve, whereas orange is the actual measured data.The tuned parameters can be found is a yaml file for input to tensorflow in trainer_config.yaml

1. **Cumulative Rewards per Generation vs Time**
   This is the mean cumulative reward given to the agent per generation over time.

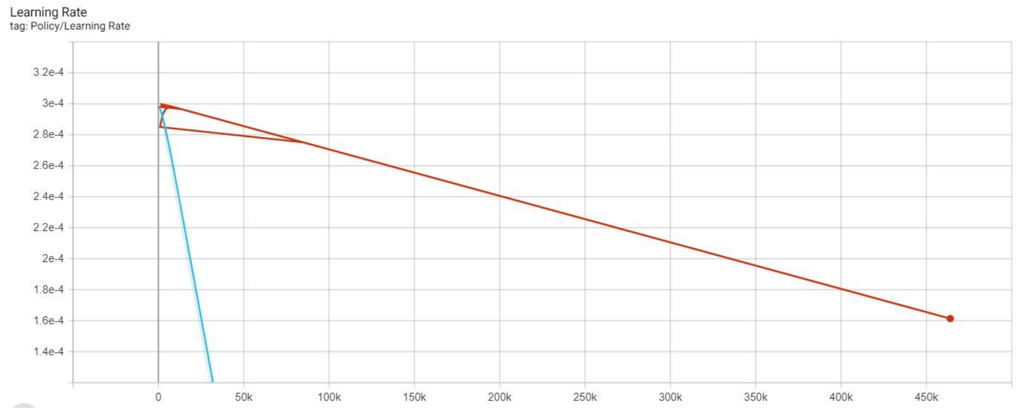


2. **Entropy vs Time**
   Entropy refers to the randomness with which the agent makes its decisions. In the beginning, the agent moved completely randomly, whereas towards the end of the learning process, it has learnt how to make decisions to maximise its rewards.



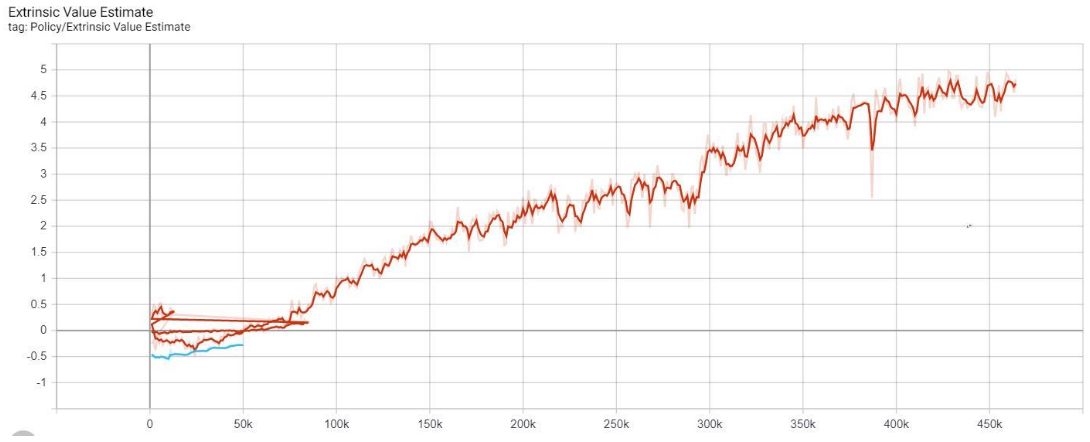
3. **Learning Rate vs Time**
   Learning Rate refers to how much the learning algorithm is influenced by the current runs results in influencing its future decisions. I found that I got the fastest and best outputs

with a linearly decreasing learning rate.



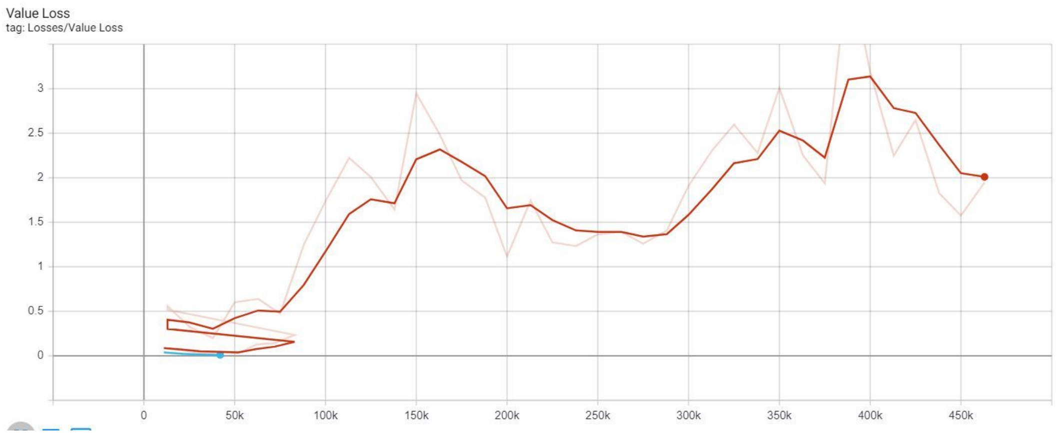
4. **Mean Value Estimate vs Time**

Value Estimate refers to how much reward the agent believes it will receive in the future, at a given point in time i.e. in the current state.



5. **Mean Value Loss vs Time**

Monitoring the value loss curve is what led me to infer that my model had finally learned. Value loss is a measure of how the neural network is able to predict the goodness or badness of a state. When the mean value loss stops rising and begins to fall, it is a reasonable indicator that the reinforcement model has started expecting and earning a stable reward i.e. has learnt the optimal policy.



PERFORMANCE

In the attached zip is a video of the final generation of the mouse running around the cheese board.

OBSERVATIONS

After all this fine tuning and fitting the model to my application by fixing its input vector space and output vector space to the environment, I finally noticed that the mouse started getting the cheese and avoiding traps. It seems to have figured out a fairly easy system, but is still unable to halt its brakes i.e. deal with the physics of deaccelerating in time to not run into traps. Maybe the current velocity could be a vector input in a future generation? Attached in the zip file is also an executable (MouseEatCheese) with the cheese board simulation and the final generation of the mouse.

MY WORK AND FINAL THOUGHTS:

A majority of my work was in designing and setting up the environment. This involved writing the attached C# scripts, extending existing Unity heuristics, and writing scripts to create an appropriate playground to generate the input data and output vector space for the neural network. Additionally, I worked extensively on the hyperparameter tuning, and used Tensorboard and Python on Jupyter in order to tune the parameters to fit the scenario. Unity ML-Agents is a fairly new piece of software and getting it to work properly was an exciting challenge. This was a very interesting visualization problem, and it was very exciting to watch the learning process and note the change in vectors/scalars influencing the movement of the mouse as it came alive on my screen. The reinforcement learning approach is extremely valuable in the Unity Environment since it is a great technique to create video game AI opponents, non-player characters or in other capacities. Using tensorflow as a method to visualize reinforcement learning and simulate a mouse coming to life was extremely interesting and satisfying for me. I wish I had an understanding of the Tensorflow libraries to the extent that I could go into ML-Agents Reinforcement Algorithm and tweak the nn generator manually rather than just as a product of different parameters and environment variables as a further step in order to better fit the model to my application.

Works Referred To

https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/
https://medium.com/machine-learning-for-humans/reinforcement-learning-6eacf258b265
https://github.com/Unity-Technologies/ml-agents/tree/master/docs
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md
http://incompleteideas.net/sutton/book/RLbook2018trimmed.pdf

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., Lange, D. (2018). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. https://github.com/Unity-Technologies/ml-agents.