**Distributed Learning Through ML-Agents**

**By Christian Johnston**

**CPSC 4387 Distributed Computing**

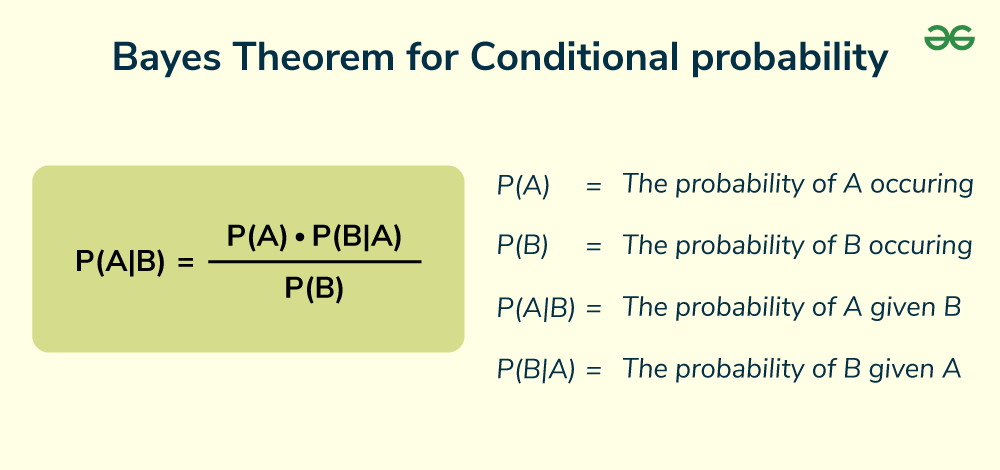
**Fall 2024**

**Introduction**

This project is a test environment for the application of distributed computing to a machine learning ecosystem. Machine learning is a valuable skill that is quickly growing in usage and by utilizing a distributed learning system, performance improvements can be made to support both quicker uptake on information and also improved accuracy due to increased learning rates. While the process of training a model can be slow at times and need lots of training data, this can be improved by adding multiple learning environments that all consolidate their learning into one model.

**Machine Learning**

Machine learning (ML) as a concept has been around since early computing research and development. While computational algorithms have always been designed to store data over time through memory, there have also been ideas about how certain algorithms could potentially “learn” over time and adjust their parameters and output depending on what data the machine stores into memory. The earliest machine learning algorithms were designed using Bayesian statistical inference to calculate probabilities of hypothetical scenarios given prior evidence and update the results as more information becomes available.

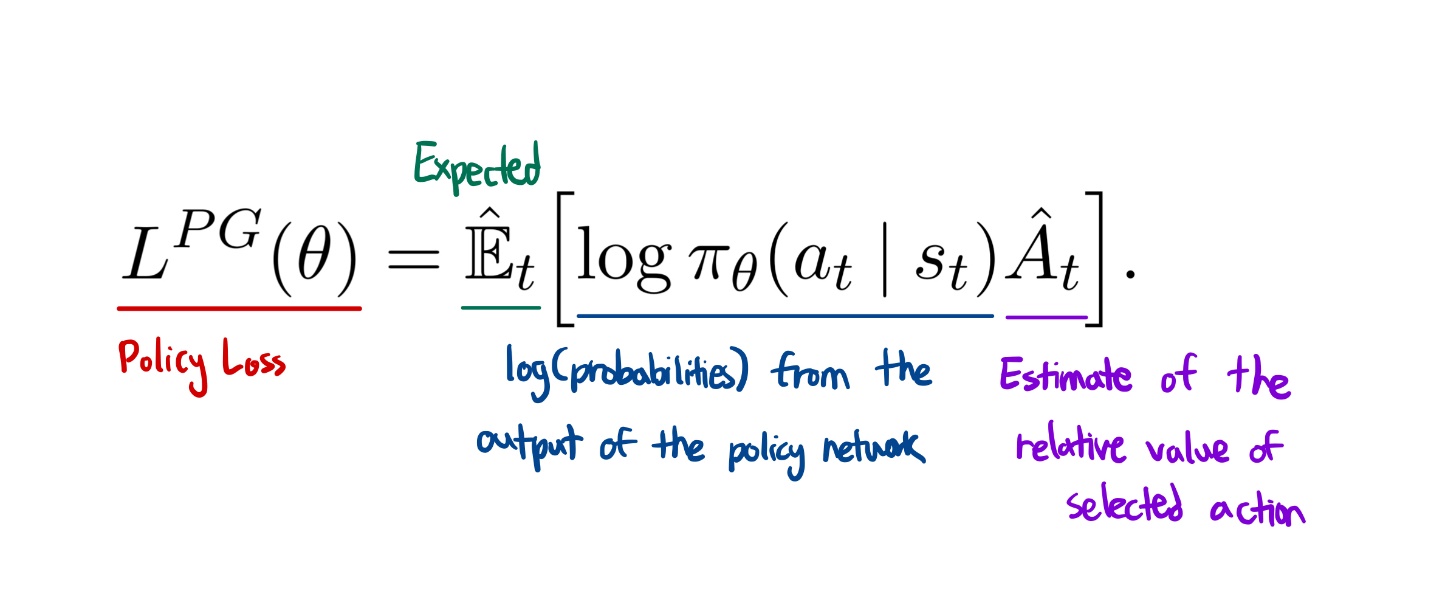


*Figure 1: Bayes Theorem*

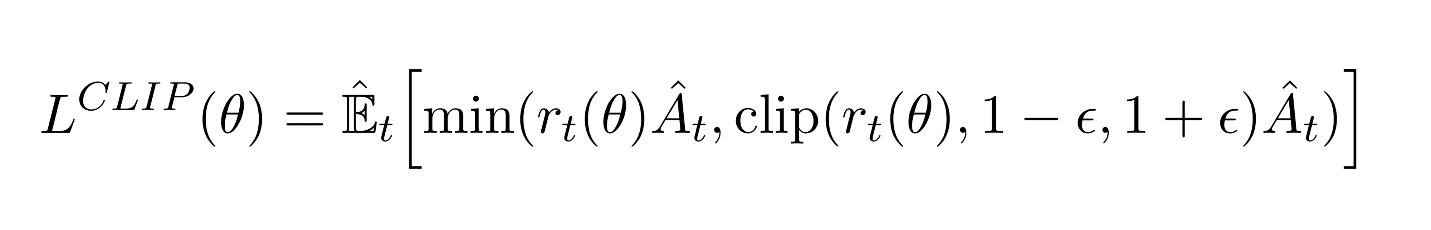
In the 1980’s, the theory of backpropagation enabled computer scientists to start designing and training neural networks through gradient estimations and dynamic programming. Still, at this time computing systems did not have the capacity (outside of highly theoretical supercomputer systems) and processing power to compute these algorithms even though they were highly efficient. Eventually work in ML shifted towards a data-driven approach and new theories and algorithms were designed such as Support Vector Machines (SVMs) and Recurrent Neural Networks (RNNs) while computer systems became more efficient at processing these algorithms. By the mid 2010’s computing systems have enough processing power and algorithms are efficient enough that deep learning becomes a reality and soon generative models such as ChatGPT and Stable Diffusion took over the forefront of ML research. In unity’s ML-Agents package, there are two deep reinforcement learning algorithms being used: Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). Before explaining PPO and SAC, there are a couple terms to define though. Policy refers to a set of rules that an agent follows to decide which action to take in a given state to maximize some sort of reward the agent will receive for performing an action deemed correct. The policy is a function that maps states to actions. The agent in the context of this project is defined as the entity that will be making decisions and interacting with the environment. The agent makes an observation (by checking the state of the environment), then uses its policy to determine an action. After execution of the action, a reward is given to the agent that is either positive or negative depending on the outcome. The agent then updates the policy to reflect the rewards it received and adjusts over time to perform actions that return the highest cumulative rewards. While simple in concept, there are a lot of calculations that need to be adjusted and tracked to optimize the actions of the agent.

**PPO and SAC**

Proximal Policy Optimization is an algorithm that is classified as a policy gradient method for training an agent’s policy network. Basically, small updates or steps are used to slowly reach optimum solutions without taking “too big of a step”. Empirically, smaller updates are more likely to converge to an optimal solution, and a step that is “too big” could result in “falling off the cliff” (getting a bad policy) and either taking a long time or even having no possibility to recover. Instead, the policy is updated conservatively by comparing how much it has changed to the previous iteration. The ratio is then clipped into a range so that the policy does not have incentive to go too far from the previous iteration (which is why it is called **proximal**).

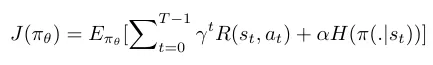


*Figure 2: Policy Gradient Algorithm*



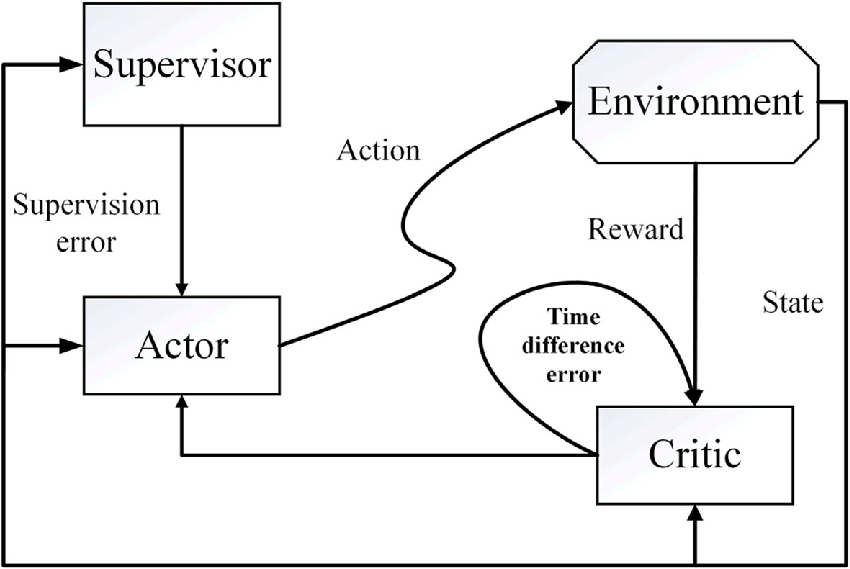
*Figure 3: Clipped PPO Algorithm*

Soft Actor-Critic (SAC) is a state-of-the-art reinforcement learning algorithm based on maximum entropy reinforcement learning where the objective is to find an optimal policy that maximizes the expected long-term reward and long-term entropy. In maximum entropy reinforcement learning, the agent tries to optimize the policy to choose the right action that can receive the highest sum of rewards and long-term sum of entropy. The aim is to find the probability distribution that maximizes entropy.

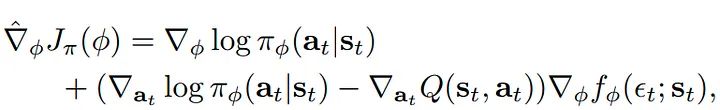


*Figure 4: Maximum Entropy Objective Function*

Actor-Critic systems are a temporal difference (TD) version of the policy gradient function. Basically, there are two neural networks: an Actor and a Critic. The actor decides which action should be taken and the critic informs the actor about how good the action was and how it should adjust. This algorithm is a bit more complex but utilizes two systems to learn and perform actions.



*Figure 5: SAC Framework*

**

*Figure 6: Gradient Equation for SAC (Deltas represent TD calculations)*

**Distribution**

Because Machine Learning can be very intensive due to the complexity of the algorithms and calculations as well as time needed to train and optimize policies, it can be both time and resource intensive for singular computer systems to perform. With a distributed system of learning environments and actors training the model, this process can be sped up and the performance of the models improved. There are two main methods for distributed model training: Data Parallelism, and Model Parallelism.

Data Parallelism divides the data depending on the number of worker nodes present in a system. A single coherent output results from having the exact model available to all worker nodes (either through centralization or replication). In the approach, the data is partitioned into n parts, where n is the total number of workers in the computer cluster that are accessible. Each worker node contains a copy of the model, and each one trains the model using a different subset of data. Training loops are run either synchronously or asynchronously.

Model Parallelism is a machine learning approach for distributing a neural network model over several computers or computing devices. The parameters are divided across several machines in model parallelism, enabling each machine to process a piece of the input data and determine the appropriate output. This is often utilized if the model is too large to fit into a single computer’s memory. The input data is often divided among several computers, with each unit processing a portion of the data.

**Tools and Procedure**

For this project, there will be several options implemented to demonstrate both machine learning and distributed performance improvements. The Unity game engine will be used as a virtual and visual environment for the actors with a sample environment set up for machine learning purposes. In the Unity game engine, a tool package known as ML-Agents will be utilized with built in algorithms and learning environment parameters. The model that will be used is a variation on PPO that utilizes a clipped policy gradient algorithm to perform the actions and calculate the rewards. There may be an example utilizing SAC as well if resources allow. The algorithms are written in Python and several packages must be utilized to train and model these algorithms. The python package PyTorch is a machine learning framework and dataflow package that controls many aspects of data flow between the model and agent including training and policy optimization. C# will also be utilized by Unity to initialize and control the actors and environment. The project plan is to test several PPO environments in the ML-Agents package, find one that fits the parameters of the project the best, and then start testing with a new model to train. After testing and training a new model, a new method will be implemented to adjust the learning rates of the PPO algorithm and implement distribution to test learning rates with a distributed system.

**References**

Zhang, J. (2021, August 27). *An introduction to machine learning with Unity ML-Agents*. Coder One. <https://www.gocoder.one/blog/introduction-to-unity-ml-agents/>

Unity Technologies. (2022). *Background: Machine Learning*. Unity ML-Agents Toolkit. <https://unity-technologies.github.io/ml-agents/Background-Machine-Learning/>

Gill, J. K. (2023, May 17). *Distributed Machine Learning Frameworks and its Benefits*. XenonStack. <https://www.xenonstack.com/blog/distributed-ml-framework>

Galakatos, A., Crotty, A., and Kraska, T. (2017) *Distributed Machine Learning*. Department of Computer Science, Brown University. <https://cs.brown.edu/people/acrotty/pubs/Galakatos2017_ReferenceWorkEntry_DistributedMachineLearning.pdf>

Geeks for Geeks, (2024, September 17) Bayes’s Theorem for Conditional Probability. Geeks for Geeks <https://www.geeksforgeeks.org/bayess-theorem-for-conditional-probability/>

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017, August 28) Proximal Policy Optimization Algorithms. Open AI <https://arxiv.org/pdf/1707.06347>

Simonini, T. (2024) Deep RL Course – The intuition behind PPO. Hugging Face. <https://huggingface.co/learn/deep-rl-course/unit8/intuition-behind-ppo>

Karunakaran, D. (2020, December 2) Soft Actor-Critic Reinforcement Learning algorithm. Medium. <https://huggingface.co/learn/deep-rl-course/unit8/intuition-behind-ppo>