Everglades

Battle Bots AI with Reinforcement Learning

Group 19 - Team Gold

ALAN PERROW BAO HONG BENJAMIN LE HEUP

JACE MIXON MICHAEL LAAGER

Lockheed Martin

REBECCA BROADWAY TANNER LINDBLOOM

September 2020 - April 2021

Table of Contents

Executive Summary

Everglades is a strategy game developed by Lockheed Martin and presented to us by our sponsors Rebecca Broadway and Tanner Lindbloom. The game consists of two opposing teams fighting to attain and retain control over various capture points on the map, awarding their team with points, with the premise that whichever team has more points at the end of the game, wins. Each control point is considered a node on the map and is directly connected to other nearby nodes, facilitating a method of transportation between them.

Alan Perrow, Bao Hong, Benjamin Le Heup, Jace Mixon, and Michael Laager, the five members of our team, have been assigned to work on the project presented by our sponsors, representing ourselves as Team Gold. We are one of two teams assigned to this task, with the other being Team Black. Both teams have been tasked with the same project, focusing on implementing artificial intelligence into the game via reinforcement learning battle bots. The two-team dynamic is used as a means to observe differences in solutions, alternative methods of arriving at those solutions, and the challenges and restrictions discovered along the way pertaining to this project.

The project itself entails the creation of four separate agents that are developed to play within the Everglades game environment. This will be done by using creative, innovative, and potentially state-of-the-art policy gradient methods in order to train the agents into achieving the highest win rates as is feasibly possible. The creation and training of these agents will be done with OpenAI Gym, a toolkit for reinforcement learning that provides a working environment to be used by the game and our developed agents.

In order to successfully complete this project, each member of our team must conduct research on various topics related to artificial intelligence in order to obtain a fundamental skill set. This includes but is not limited to information on machine learning and reinforcement learning, OpenAI Gym, and Everglades itself. Working as a collaborative whole, our team, Team Gold, will be undertaking this task and all it entails with the goal of fully completing the development of four agents.

Project Narrative

The Everglades Battle Bots AI with Reinforcement Learning project is a task delivered by sponsors from Lockheed Martin to be taken on by a group of students, with the goal of teaching in-depth and applicable knowledge about a wide range of reinforcement learning models available for artificial intelligence training. In the implementation of this project, these models will be used to develop agents which demonstrate varying degrees of success in performing the desired behavior. Agents will be chosen in an attempt to maximize the success rate of a team of AI-driven battle bots against an enemy team, where the enemy battle bots will use either a random action selection strategy, or a discernible strategy via state machine agents.

Statement of Motivation

Alan Perrow

From the very beginning of my venture into the field of computer science, I have always had an interest in artificial intelligence. In those earlier times, my only knowledge of the field was that this technology was something exciting and promising for the future, as the transition to a digital life was clearly already in place. I later learned more about the process by which artificial intelligence implements machine learning to perform certain tasks. These types of tasks would normally be completed inefficiently by humans, such as predicting or discovering patterns from large inputs. However, although this is definitely the most practical and applicable example, there were other alluring demonstrations that caught my eye at the time. Specifically, videos on implementing machine learning to beat basic video games, and later, generalizing the model to be able to beat many different kinds of NES video games. That was the first time where I really thought in an academic sense and considered going into this field in order to understand more of the underlying processes behind this technology. Today, I am continuing along that path, and hope that working on this project will further develop

my skills not only as a computer scientist but also increase my knowledge in the prosperous field of artificial intelligence.

Bao Hong

I love Civilizations. It’s one of the best turn-based strategy video games in the market, famous for its complexity and its portrayal of different civilizations throughout history. In a game of Civ, I can win the game through different means such as be a warmonger and capture all capital cities, or get rich enough and buy a rocket to live on the moon, or stay alive and get the highest score on the last turn. However, the AI of the game is notorious among the players for being bad and very repetitive, even though the game has different difficulty levels to choose from. Higher difficulty Ais are simply received more starting bonus by the developers to make the AI more aggressive to the human player, especially during the early game, since the huge bonus makes them progress to the tech tree faster early on. But if a human player makes it to the mid and late game, the AI will make the same set of moves every time in every game because they don’t know how to strategize since they were implemented by a classic state machine AI that’s been used in most games on the market. It makes the game only fun and challenging during the early game, but really boring during mid and late game in any difficulty.

This makes me wonder, why didn't they use Reinforcement Learning? As a Computer Science student, I have always had a great interest in AI and Machine Learning in general, and I have heard that Reinforcement Learning algorithms are commonly used for improving video games AI. AlphaGo is one the well-known Reinforcement Learning AI that beat the Go world champion, by playing against itself thousands of times and learning from its mistakes. When I heard about the Everglades project, I immediately made the connection between Everglade and Civilization, as both are turn-based strategy games, and that sparks my interest. I can see the potential of Reinforcement Learning in different areas, especially in strategic video games, where in the future, most video games AI can strategize like a human being. However, given that classical state machine and decision tree Ais are still popular today, I realize that Reinforcement Learning AI still has a lot of limitations that cause developers to not use them. I hope through this project, I can explore and learn more about Reinforcement learning and its limitations, to improve, if I can, and apply this technique to different problems.

Benjamin Le Heup

Growing up, I've always had a passion for Mathematics and how it has been used to benefit mankind. As I got older, I got more and more curious about AI particularly ones that were created to challenge a player in video games. At the time, I was under the preconceived notion that these "computers" were created through software engineers programming responses to any event or at least looking at possible outcomes to moves after being proposed a situation making this heavily reliant on software engineers to uncover strategies on their own. This all changed when I came across a new chess AI named AlphaZero in 2018, which was an AI that was created through trial and error by playing against self. What sparked my interest is how this AI made moves that broke many chess conventions that have been formed throughout the last couple of centuries. This got me thinking that machine learning may give us a fast track way into finding an efficient solution to many problems that may not be obvious at first. Machine Learning being a huge beneficiary of Mathematics as well as being an alternative way to solve solutions has made it a field that I would like to get more involved in the future.

Jace Mixon

When it came to topics within Computer Science, I always understood them generically as something that would be done "algorithmically" rather than having any kind of semantics or heuristics in order to speed up computation. Hence, when it came to Artificial Intelligence, I was puzzled at the idea that a machine could categorize or predict new results within the future based on hidden details from previous data. However, I could see the potential it has and how it has evolved over the years, ranging from something as humorous as creating an AI that would produce an extension of a verbal or written speech in somewhat coherent English sentences or do something as uncanny as creating Deep Fakes of celebrities onto other people's faces.

My true fascination to AI, however, came from witnessing how Artificial Intelligence could be placed within a videogame environment and become so well-versed within the game they were playing that the AI would start adapting to the opponent's behaviors and would ultimately beat the human opponent. It was interesting to me

because it showed a machine developing techniques to first learn the basics and then eventually learn more complicated strategies, much like how a human opponent would learn the game. However, since the machine can hone in on being precise and making quick decisions on the fly, it demonstrated how it can be a fierce competitor and an overall enigma as to how it can master complicated techniques within a matter of a few weeks but yet not quite understand how the English language works. My motivation for this project is to gain a better appreciation to the technologies that goes into developing Artificial Intelligence and understand some of its more practical implications in the real world.

Michael Laager

I have always had a curiosity about the exact details in which the world worked. This was my motivation for being in computer science; an implementable way of learning the barebones architecture of something so fundamental to our lives. I always wondered, though, how something so complex can be modeled by a deterministic machine. How can my own thoughts and processes possibly be computed by “if” and “else” statements? It just didn’t make sense. Then, I learned about the field of machine learning (and AI in general) and finally found a path to begin looking down for my answer. The blend of mathematics, statistics, and computing, along with this golden era of machine learning, gave me tremendous interest in this field. Though I’m not particularly well-versed in reinforcement learning, I see this project as a way of expanding my horizons and being able to learn more about how computers are able to deal with the chaos that is the real world.

Goals and Objectives

In the scope of the project itself, our team has been tasked with implementing a reinforcement learning agent which will play the Everglades AI Wargame provided by our sponsors from Lockheed Martin. This agent will be developed by our team using OpenAI Gym, with the goal of maximizing the success against an agent with either a random action selection strategy, or a state machine agent with discernible strategy. Against said random agent, our agent should demonstrate a win rate greater than 60%. In order to do this, our developed agent should demonstrate a consistent, noticeable increase in reward returns across training epochs.

These goals listed above are considered “Must Have”s in our project description, and therefore must be completed by our team, at a bare minimum, in order to have successfully completed this project. Compounding upon these goals are two subsequent lists of goals, each with a higher standard, which reward teams who demonstrate greater accomplishments.

The “Nice to Have” goals are as follows:

- Agent consistently achieves >75% win rate against an agent with a random action selection strategy.
- Teams should implement training via agent self-play.
- Implement four different reinforcement learning techniques.
- Characterize agent performance against different types of script-based AI action selection strategies.

The “Home Run” goals are as follows:

- Agent consistently achieves >95% win rate against an agent with a random action selection strategy.
- Agent should be able to beat all agents from other universities.

Although obviously very ambitious, our team’s objective is to not only meet the “Nice to Have”s, but exceed them, completing the “Home Run” goals provided above.

Alongside the project's technical goals, there also exists a more personal objective which can be considered independent from the rest. As this is a Senior Design project, the knowledge and experience we gain by working towards (and eventually, hopefully, completing) all of our objectives is something which will, in some shape or form, assist or influence us in the future. Whether that means academically, professionally, or perhaps even just personally, the outcome of participating in this project should impact each of us individually. By developing strategies and implementations for creating reinforcement learning agents through OpenAI Gym, we will learn more about machine learning and training models, as well as artificial intelligence in general, and we should be able to apply this knowledge in the future wherever it may be useful or applicable. We should also become more affluent in the technicalities of working in a team environment in the computer science field, as we will be consistently meeting with the sponsors of our project in order to update our progress, stay on track, and continue development in a satisfactory manner.

Specifications and Requirements

During the kickoff meeting with our sponsors, we discussed what were our required specifications for the Reinforcement Learning agent. An agent is a decision model for interacting in a specific environment. In Open AI, an agent interacts with an environment in this case the Everglades game and makes observations and then receives rewards that help shape future decisions that the agent will make. By repeating this process an agent's decision will continuously converge to making specific types of actions in certain states of the environment. Our agent will have at least a 60% win rate against a random action selection strategy. This agent will be able to know how to handle multiple strategies, such as a rush strategy where a player will have their battle bots occupy as many nodes at the beginning to maximize what a player will see with the fog of war. We will do this by training our agents against itself and other agents to get consistent gain in rewards to increase its win rate over time.

Broader Impacts

When an AI is trained with reinforcement learning, the objective is for the AI to quickly learn and understand how the game is played while also being able to choose the “correct” response a majority of the time. It is possible that it can simply imitate what a human opponent can do, but that would mean that the AI can never be better than human opponents since it would inherit the same flaws that a human opponent would perform. Instead, reinforcement learning aims to create an AI that will be trained to not only be on the same level as the human opponent, but also make better choices during critical situations in order to achieve a better outcome a majority of the time.

This same concept can be implemented in a broader way by having the same techniques applied within a real-world context, where important choices need to be decided and any flaws within those choices could ultimately lead to a failure. For instance, in terms of health diagnosis, an AI can be implemented to analyze issues with a patient and can not only through means of scanning for any anomalies but by making decisions based on what it has observed in order to provide the best treatment for the patient. Another example is an AI that practices in the stock market, where not only it can provide models to predict future outcomes for different companies but can also provide its best decision on what stocks to buy and when to buy it. Of course, with all of these examples that reinforcement learning can be a useful asset, the design of the technology would have to be altered in order to be better fitted for the task at hand.

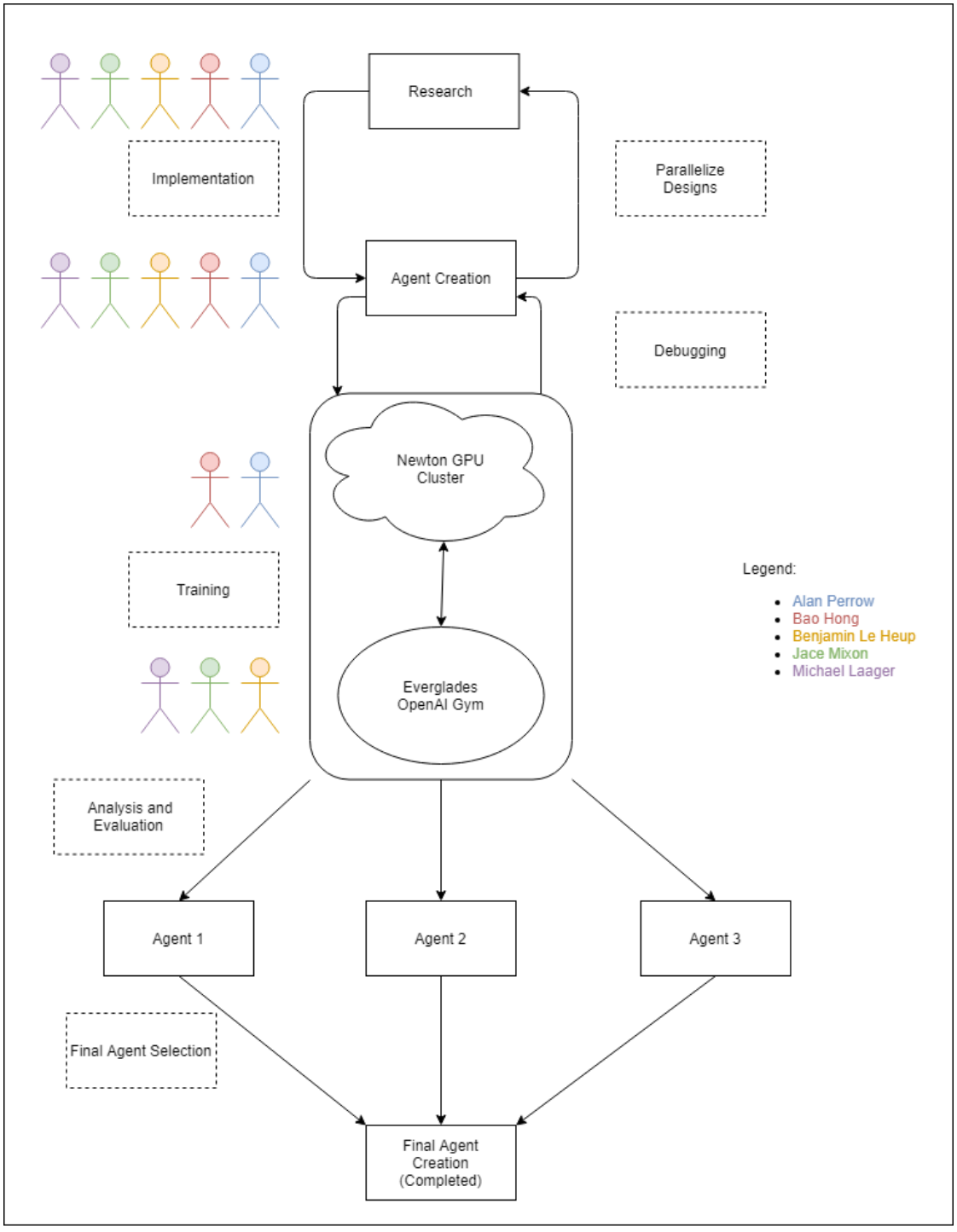
Legal, Ethical, and Privacy Concerns

Though more indirectly correlated to our project, there are a couple of important legal concerns that need to be taken into consideration. The agents that are implemented, if used for other than Everglades, could behave in unexpected ways. In that case, the design could falter, and the blame for the fault is a concern for both the designers, the organization, and the end-user.

What is a more mounting concern is the ethics that surround the agents we implement. These agents are representative of drones that will capture territory and fight against other drones. There is an easy progression from the completed agents to using them in actual drones used in warfare (this being more likely as the project is for Lockheed Martin). There is also, though not as prevalent, a potentiality for the agents to have bias towards different groups of players. If an agent was trained based on matches with real people, the agent could develop biases in playing that could be exploited in public usage. This, however, would be dependent on the groups having different playing strategies/styles of playing the game.

Privacy issues would not be a cause for concern with this project. These agents are designed specifically for Everglades, which does not give the agent any private information. For gameplay, the agent would also be exposed to only elements that the player would have access to, thus not giving them an advantage over opponents.

Project Block Diagram



Project Budget and Financing

When it comes to training AI, it has been notorious for consuming a huge hardware power to train an agent. Because of the nature of this project, it's most likely that we can't just use a single machine to train our models, but a cloud service with a cluster of powerful machines to achieve what we need. There are several options that are available for us right now such as Google Cloud Platform, Paperspace, or a free option like the Newton service that UCF offers. However, training on our own machine is still an option depending on the progress and development of our models.

Personal Machine

Cost of using a personal machine can be tricky depending on the complexity of our models. The cost of training a simple model that can train under 30 minutes is minimal. But if it gets more complex and requires a powerful GPU to train, depending on our state of our hardware right now, it can cost at least $500 to acquire a recommended GPU for machine learning (RTX 2070). Other than that, training on a personal machine can cost a high electric bill depending on the amount of time a model needs to train. If a model needs days to train, a cloud service is preferable.

Google Cloud Platform

Google offers GPUs to help train AI models. The price ranges from $0.35/hour to $2.48/hour depending on the type and number of GPUs being used.

Paperspace

Paperspace standard offers for using their GPUs virtual machine range from $0.45/hour to $2.30/hour. However, they also offer a free student package that provides a free GPU and ability to run Jupyter Notebook on their web interface, suitable for learning and quick development.

Amazon SageMaker

Amazon offers AWS SageMakers that helps to develop Machine Learning and AI applications through their SageMaker Notebook studio. They have a wide variety of

virtual instances from non-GPU to accelerated GPU. The cheapest option for GPU instances on AWS is $0.7364/hour to the most expensive at $6.0928/hour.

Microsoft Azure

Azure also offers a virtual GPU environment for training with the price range from $0.9/hour to $3.60/hour for their NC-series, which is the cheapest one. However currently they are offering NC-promo which is a discount for NC-series with prices range from $0.396/hour to $1.742/hour

UCF Newton Cluster

This option is free for UCF students. It's powerful, and according to professors, they can handle Machine Learning models.

As of right now, with the available options, we preferably choose to use our own machine or UCF Newton Server to train our model if it requires since those are the costless options that we have. Paperspace is also a good option with their student package, and their standard package is more affordable compared to the others. Google also has a similar price tag for their GPU service in the lower end, which is also a viable option. SageMaker and Azure offer powerful virtual machines but come with a hefty price tag. We don't expect our model to require the immense power that Amazon and Microsoft offer.

Newton — UCF's GPU Cluster

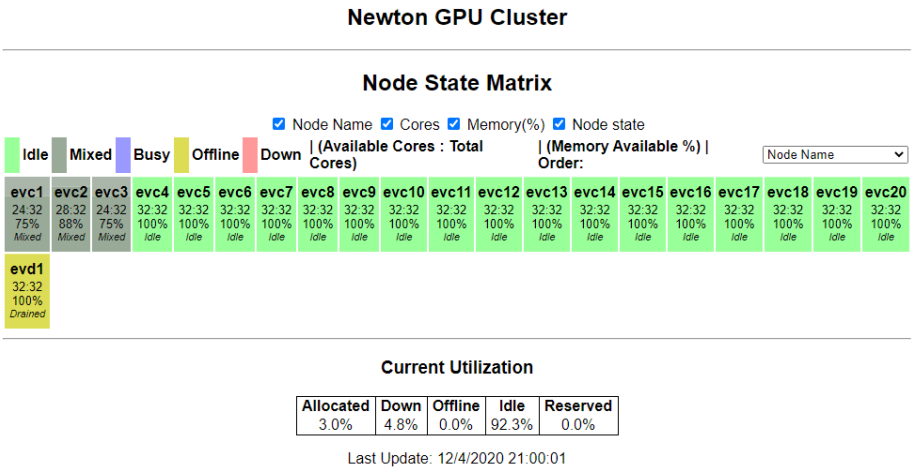
To streamline the process of training the agents for this project, we have decided upon using a GPU Cluster known as Newton, provided by UCF's Advanced Research Computing Center (ARCC).

Technical Description

Newton is a small GPU-based compute cluster, suited mainly for HTC (high-throughput computation) and GPU-based computation. Being an HTC-suited cluster, the environment in which tasks are assigned, worked on, managed, and completed is very demanding, requiring computational resources over potentially long periods of time to accomplish a given task. As one can imagine, the hardware and network requirements are quite overwhelming, but luckily these are well-covered by UCF in our case.

The cluster operates not on a single computer, but on many, with support for some users concurrently. The hardware is listed below:

- 20 compute nodes
- Each node has 32 cores and 192 GB of memory per node
- Each node has two Nvidia V100 GPUs
- In total, 640 cores and 40 GPUs
- Half the nodes have 16 GB GPU cards, the other half have 32 GB cards



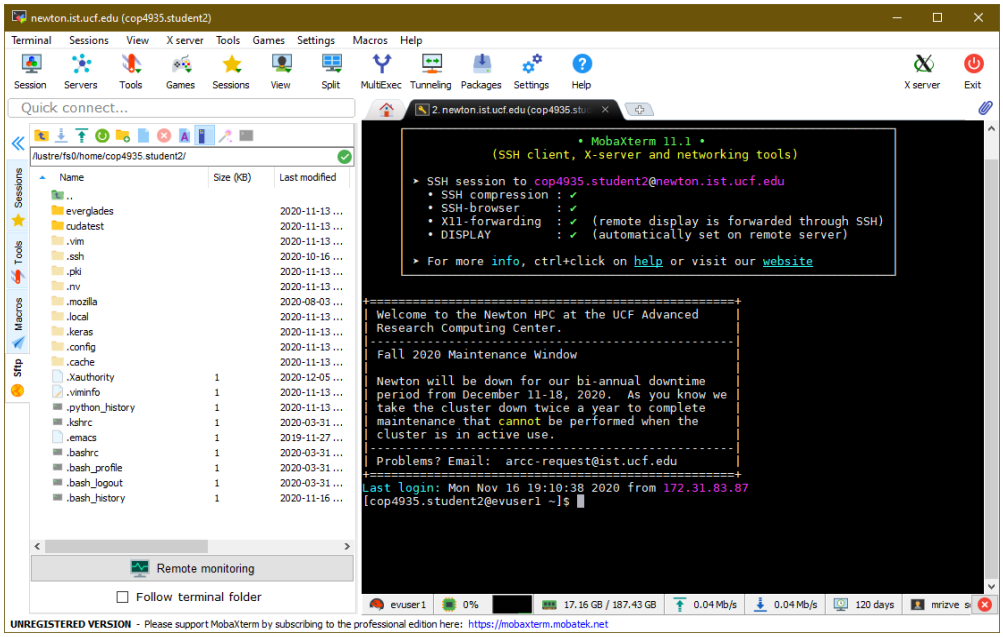
[A16] *Real-time status visualization of Newton, courtesy of the ARCC.*

Accessing the Cluster

Before being able to use Newton, a username, a public and private key, and a passphrase must be provided by a higher-up at UCF — remember that this is expensive technology that is being offered essentially for free to us students! In our team's case, our professor Dr. Heinrich provided us with the necessary credentials as well as a small amount of documentation for assistance in getting us started.

The Newton cluster can only be accessed with authentication via SSH keys, which are encrypted with a passphrase. As stated above, a private key must be used in order to gain access to the cluster. Also, since the ARCC is provided by and for UCF, the only way to access Newton is when connected to the UCF network. This can be done either with a VPN connection or directly through a local UCF network, such as a WiFi connection to UCF_WPA2.

After connecting to the remote host with a provided key, you will be presented with your user's personal directory on the cluster. This local directory is used to edit, manage, and otherwise use in any way one sees fit to assist in their GPU-accelerated computational needs. It is very important to only work within your local directory, as there is restricted access for users outside of this scope.

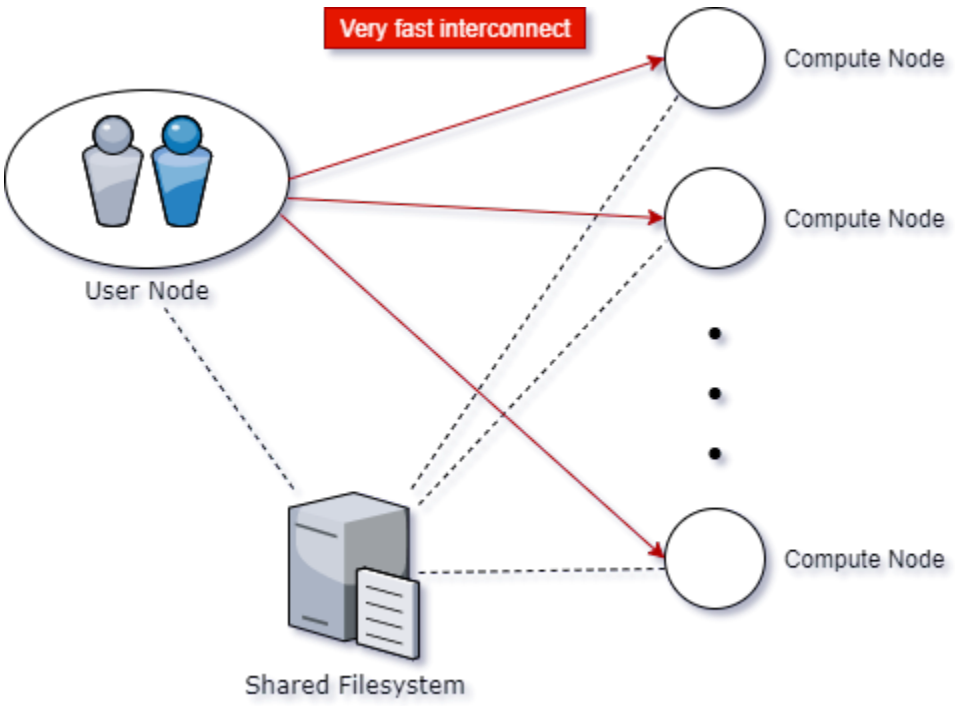


Directory for "student2" on the Newton cluster. Extra folders were added by our team.

To get better familiarized with using the job submission system, there are example scripts located in the `/examples` folder, with GPU-specific examples in the `/examples/cuda` folder, both found in the main home directory. For our team, we are almost entirely focused on using GPU-acceleration in our training, so those examples are the most useful. Before getting to more complicated scripts, it is important to understand how to properly use and take advantage of what the Newton GPU cluster has to offer.

Overview of the GPU Cluster

As stated previously, Newton is suited for HTC, and it is important that the workflow used by the user reflects this. Workflow on this type of cluster (as well as HPC (High-performance computing) clusters, such as the ARCC's alternative cluster, Stokes) typically involves submitting jobs to a batch workload manager, which will then wait for resources to become available. Once sufficient compute resources are available, they will be allocated to the job before beginning execution.



An HPC cluster is a collection of many separate servers (computers), called nodes, which are connected via a fast interconnect.

As a logical view of the cluster, the interaction can be broken down to those between users and the system. Users can:

- Log into a user node.
- Copy data to the user node.
- Manipulate data, scripts, and code on the user node.
- Submit jobs from the user nodes that will run on compute nodes.
- Copy the results back when the job is complete.

Meanwhile, the system:

- Determines when your job can be run.
- Decides which compute nodes will be used.
- Executes user's job on a subset of compute nodes.

Change in Workflow

The process of job scheduling, resource management, and accounting of usage on Newton is handled by the Simple Linux Utility for Research Management (SLURM). Jobs must be created on a user node and will then be computed on the compute nodes of the cluster. Creating a job is typically done by writing a submit script. These submit scripts are like any other unix shell script, except they also contain directions for the resource/workload manager that tell it how many resources you need, how long you will need them, etc. The submit scripts must then be submitted to SLURM, which will handle the task of assigning it a job ID, then subsequently put the job in a queue, or a *partition* in SLURM terminology. [A17]

This creates a workflow distinct from that which one may be used to, as there is no longer a simple, sequential process to follow. Creating, debugging, and running parallel, distributed programs on a cluster is more difficult than dealing with the individual, sequential programs typical when programming in a local environment. Linux command-line proficiency, knowledge of SSH and SCP, an understanding of job queuing and scheduling, and experience in parallel/distributed computing are all needed to properly manage this task. Instead of running and testing programs at your own leisure, using a cluster is instead more like using a shared printer, where you must expect to wait on other potential users before having any needed resources at your disposal.

Utilizing the GPU Cluster

When requesting resources for a job, the most important resources to specify are:

- *compute* — how many nodes, or compute cores, that the job will need
- *memory* — how much memory the job will need (/core or /node)
- *time* — how long the job will run (by “wallclock”)
- *gpu* — how many GPUs the job will need
- *gres* — “generic” additional resource (GPUs may also be requested)

In general, it takes a little bit of tuning to get these right. If the amount of resources requested is far more than actually necessary, then a longer wait will be required for the job to complete. On the contrary, if there are too little resources requested, then the job won’t get everything it needs: a lack of compute cores will cause the job to run very slow, a lack of memory may cause the job to crash, and a lack of time will cause the job to be cancelled prematurely.

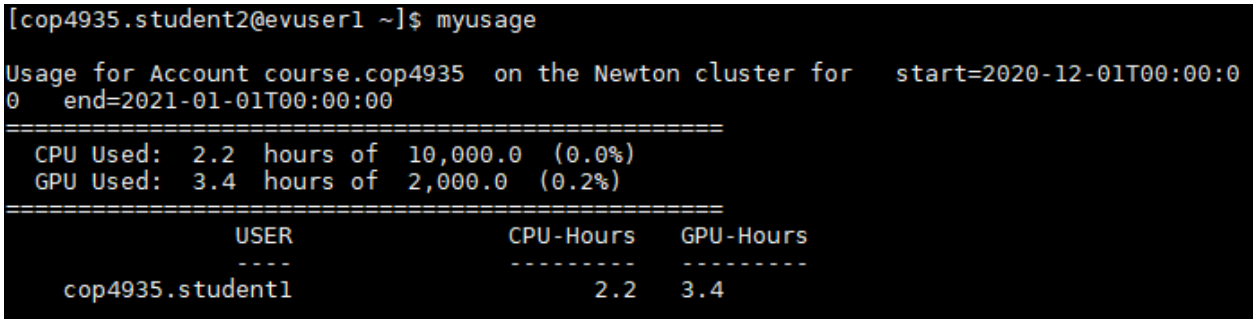
When creating a program that will be run on the GPU cluster, there is bound to be libraries/packages/modules that need to be imported to help assist in the programming process. Newton has a lot of different pieces of software, including different versions of the same software, software compiled using different compilers, software linked to different libraries, etc. The module system can help set up specific software to run correctly and should always be used, to the user’s advantage! You can load/unload modules, list loaded modules, or find which modules are available to use by using the commands provided with Newton:

`module list`	Lists all currently loaded modules
`module avail`	Lists all available modules
`module load abc/abc-1.0-gcc-7.1.0`	Loads the 1.0 version of “abc” compiled with GCC v7.1.0
`module unload xyz/xyz-2.0-gcc-7.1.0`	Unloads the 2.0 version of “xyz” compiled with GCC v7.1.0
`module purge`	Unloads all modules

Restrictions

Unfortunately, managing and providing upkeep for a GPU cluster is not free! At least, not for the school it isn't. This means that users must have some restrictions placed upon them so that they don't hog resources, intentionally or not. One of the most important restrictions to keep track of is compute usage. When requesting and executing jobs, the resources used are tracked and accounted for on the user which is accessing Newton. Be aware of the fact that ARCC specifically keeps track of resources *used* — that means you are charged for what you consume, not what you request.

At the time of writing this, each account is allocated 10,000 CPU hours and 2,000 GPU hours per month, shared among its associated users. That is a very large amount of time, and should be more than enough for anyone simply wanting to train a Gym reinforcement learning model, like our team is. Even so, if there is ever a worry as to how much of a user's monthly resources have been used, it can be displayed at any time by using the `myusage` command:

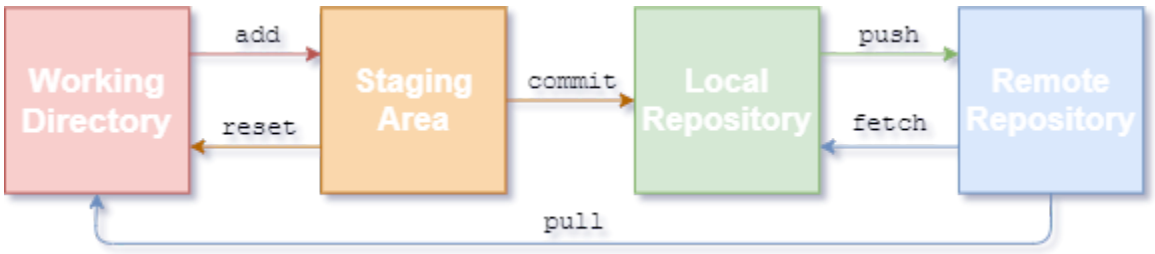


Source Control

To effectively work on a piece of software collaboratively, it is essential to set up some form of source control so that each change can be monitored and tracked before and after it may be implemented. Arguably the most popular source control software for open-source programs at the moment is GitHub, and is what our team will be using to create the agents for this project. Before diving into GitHub, however, there first needs to be a solid understanding of the underlying driving software, Git.

Git

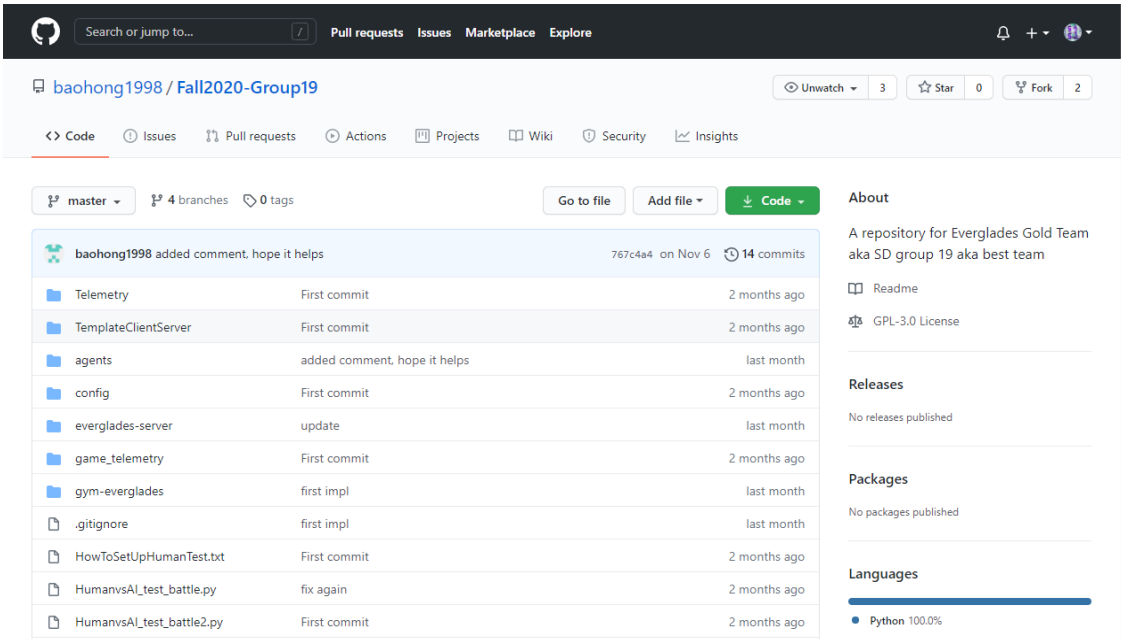
Keeping track of software changes manually can easily become a nightmare, and programs have been developed to help with this. Git is a source control system that can handle any type of project, small or large, with impressive speed and efficiency. It works by managing the file structure of a given project, keeping track of any changes it finds in the structure or in its underlying files. Working entirely inside the console, the process of using Git can be intimidating at first, but is very recommended in order to experience and understand the general workflow when using a source-controlled system firsthand. It's better to experience difficulty and possibly catastrophic error on your own local test repository than with an important repository shared amongst a team! There are plenty of tutorials online about how to set up a Git repository and just as many examples of how to work with them. Git also supports remote repositories, which is what software such as GitHub or Bitbucket expands upon.



An example of a simple Git workflow, command names included.

GitHub

Once a foundation of knowledge has been built by using and becoming more familiar with Git, it is strongly recommended to move onto a service which implements Git and expands upon it for easier usability. GitHub is one of these services; it is a user-friendly and interactive repository management system that fits the needs of a vast majority of programmers' needs. First of all, GitHub is cloud-based, so remote Git repositories can be created on the service for online storage. By being online, the repositories are essentially "published" as open-source software that anyone can look at and edit if they want to, but obviously only the owner or owner-privileged collaborators of that repository can accept potential changes. Adding certain people as collaborators (whether they be friends, team members, or anyone else you know) to work on your project will streamline the process so that you can better keep track of who is adding to the repository.



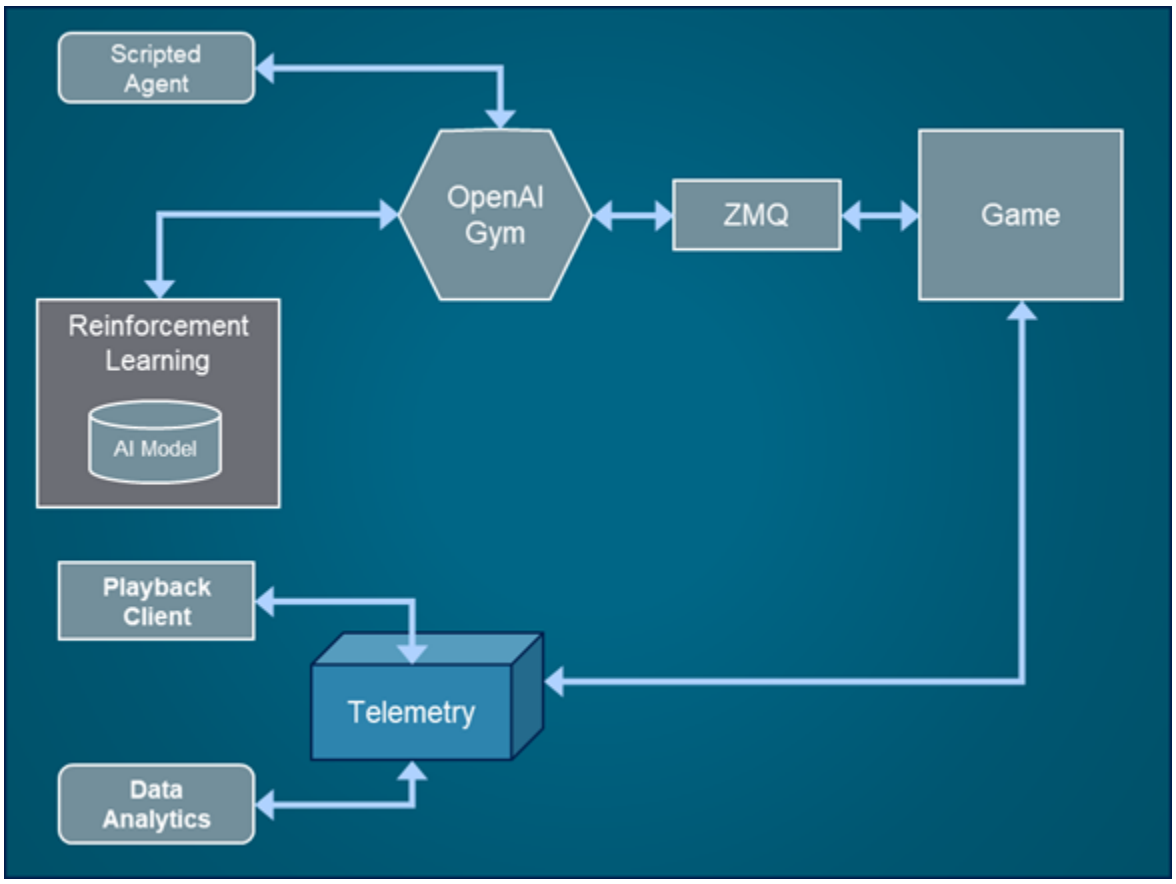
A visual of our team's remote repository hosted on GitHub.

GitHub will aid in our team's software development as not only can we see what each team member is doing, but we can also add or modify source code to better suit everyone's needs when we see fit. Particularly, this aspect will aid us in creating multiple agents independently if desired, with the option of assisting team members when they have troubles with their agents and vice versa.

Everglades Game Overview

Summary

Everglades is a synchronous, turn-based 1v1 competitive strategy game designed to be “played” by AI players. The primary objective is to capture the opponent’s base, and the secondary objectives are capturing nodes and eliminating enemy opponents.

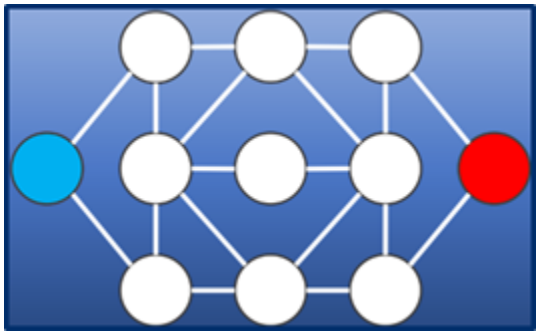


High level flow of Everglades game stack

The game has an Unreal front-end and an OpenAi Gym backend for AI agents. For developing an AI agent, data and game logic can be viewed independently without an Unreal front-end in the provided back-end python scripts or game telemetry, which is a collection of csv files that contain the data about everything of every turn in an Everglades game.

Objective

The main objective and winning condition for the game is to capture enemy base or have a higher score after 150 turns. There is also a tie condition that happens when both teams lose all units. If only one team happens to lose all their units, the other team still needs to capture the base to win the game, unless the game reaches turn 150 before that can happen. In that case the game will break ties through scores.
Team can score by capturing nodes on the map.



Conceptual map layout. Note: Distance between nodes can be configured

Rules

Units

Each team will start the game with a total of 100 units each and will be broken into 12 different groups with 3 different classes: tank, striker, and controller.

Group	Loadout
Group A	8 Strikers
Group B	8 Controllers
Group C	8 Tanks
Group D	8 Strikers
Group E	8 Controllers
Group F	8 Tanks

Group	Loadout
Group G	8 Strikers
Group H	8 Controllers
Group I	8 Tanks
Group J	8 Strikers
Group K	8 Controllers
Group L	12 Tanks

Starting unit group loadout

	Movement	Armor	Damage	Capture Speed
STRIKER	+50%	-50%	+50%	*base*
CONTROLLER	*base*	*base*	*base*	+50%
TANK	*base*	+50%	*base*	*base*

Class attributes

Controllers are great for contesting nodes because of their superior capture rate. Strikers are good for attacking with high damage and movement speed but can receive more damage because of reduced armor. And tanks are best for defense with their high armor stat.

Nodes

There are 9 contesting nodes for both teams to capture and 2 bases. Each node is worth 100 points, partial points given depend on the capture percentage (80% will give 80 points). Because of that, points can be lost when a team loses a node. There are 2 special node types: Fortress and Watchtower. Fortress gives an additional bonus for defending, while Watchtower will give the controlled team extra vision to the next node by reducing Fog of war. Each node also has a defending multiplier, which gives a slight bonus for units that are already inside the node to defend.

- Two nodes connected to base has 1.5x defend bonus

- Base has 1.0x defend bonus

- Other nodes each has 1.25x bonus

Fog of war

There is a fog of war mechanic in this game. A team only has vision for nodes that they have units deployed into. Extra vision can be gained by controlling the Watchtower nodes.

Combats

Combats are automatically resolved immediately after a player moves if there is any node that has opposing units in it. There are a few things that affect combats:

- Units class base attributes.

- Node defense multiplier and fortress bonus

Movement

The only action an agent can perform is move, which will be represented as a 7x2 action space array. And in one turn, only up to 7 out 12 groups can move for each team. An ith element of the array can be denoted as [node id, destination]. For example:

[[2, 3], [5, 7], [1, 2], [3, 6], [6, 2], [9, 10], [1, 4]]

Ideas

Prioritized Experience Replay (PER)

This is the method where we sample particular states of the game our agents have played in and train the Agent from that position forward. This is useful when we see a particular situation come up often and will help train our reinforced learning AI to better develop the “End Game”.

Hindsight Experience Replay (HER)

This method helps in dealing with sparse reward systems where, instead of ignoring episodes that did not get us closer to the end goal, we establish the final state of the episode as a virtual goal and save the episode. This creates a more dense sample space for the agent to be able to learn towards the actual end goal.

Asynchronous Advantage Actor-Critic (A3C)

This method judges how good the actions that our agent performs by following a policy (actor) and updates the policy accordingly. This not only helps the agent to continuously improve how it chooses the actions (policy) but also utilizes parallel programming to independently train multiple agents simultaneously (asynchronous) to reduce our training time and still have a diverse set of results.

Rainbow DQN

This method is a combination of many independent advancements in reinforcement learning, including PER, to drastically improve a single learner. Though advanced, an implementation of an agent using this method would be expected to yield very good results.

Deep Deterministic Policy Gradient (DDPG)

This method takes a combination of approaches from two sources, actor-critic approach and DQNs, and relies on the deterministic policy gradient that can perform its functions on a continuous action space. The ideas from DQN that are implemented are that it is trained off-policy and with a Q network for minimal correlation and consistent targets during temporal differences respectively.

Machine Learning Frameworks

When developing software for machine learning, the large amounts of computation required can become a serious undertaking without the proper tools. Thankfully, there are quite a few libraries and frameworks that have been created and distributed exactly for this purpose! Deciding which of these to use can be tricky, as each option has its own benefits and downfalls, and it is totally up to the user to determine which best suits their needs. Our team researched the most popular frameworks used in order to choose which one will be the right fit for our project.

TensorFlow

At a conceptual level, TensorFlow is a framework resulting from the combination of multiple different machine learning and deep learning models and algorithms. Applications developed with TensorFlow are built through the use of a Python-based front-end API, which are then executed in C++ for better performance. An important note is that TensorFlow is made for more than just deep learning; there are tools to support other algorithms and learning methods, like reinforcement learning. This is a very important feature in relation to our team's decision, as our work entails developing multiple reinforcement learning models. TensorBoard is also an inviting technology, tracking and visualizing metrics such as loss and accuracy as well as the model graph.

As far as efficiency is concerned, TensorFlow is not the top contender. For example, TensorFlow does not support so-called "inline" matrix operations, but forces you to copy a matrix in order to perform an operation on it. Copying very large matrices is costly in every sense. TF takes 4x as long as the state of the art deep learning tools. Also, TensorFlow runs dramatically slower than other frameworks such as CNTK and MxNet. This is not a particular issue with the software itself, however, as Google's acknowledged goal with Tensorflow seems to be recruiting, making their researchers' code shareable, standardizing how software engineers approach deep learning, and creating an additional draw to Google Cloud services, on which TensorFlow is optimized. [A20]

Keras

Building upon TensorFlow, Keras is a deep-learning framework that provides a high-level abstraction for its underlying processes. With an extremely simple and intuitive workflow, developing algorithms and models with this framework is designed to be almost embarrassingly simple. Its front-end API is entirely built upon this, being inspired by the intuitive API of PyTorch.

This is the choice that made the most sense for our team. None of our team members are by any means experts in the AI field, and although we all have high aspirations for ourselves, our scope needed to be kept within realistic bounds. For this reason, we decided that working with Keras, an extremely popular framework, and quite possibly the best Python API used for neural networks, would provide us with the most benefit weighed against all our other options. Also, in the case that Keras is not enough to handle our needs for training a particular agent, we figured that since it is a framework built right on top of another, we could always take a step back and try to work from the ground up using TensorFlow. Efficiency also would not always be of the highest concern (although always kept in mind!), since we planned on utilizing the UCF Newton GPU Cluster and not run every training session on our own local systems.

PyTorch

As a framework focusing more on complex architectures, PyTorch allows for variable-length inputs and outputs with their dynamic computation graphs. This feature is especially useful when working with models that take advantage of this, such as when developing recurrent neural networks.

In terms of efficiency, PyTorch holds up very well with the amount of computational power necessary for dealing with complex architectures. This framework is designed to be used with GPUs in order to accelerate the learning process, such as with their implementation of tensors for faster computations. This increase in speed is a very inviting feature for users dealing with any kind of neural network. Another strong reason to use PyTorch for training is automatic differentiation: it is possible to use functions that numerically evaluate the derivative of a function via backward passes in the neural network. One of the only downsides is that training code is usually written on a case-by-case basis, as it is not as modular as the rest of the features in this framework.

Machine Learning Libraries

Alongside the framework used when programming for machine learning models, libraries are also used to efficiently work with the considerable amounts of required data. Unlike frameworks, however, there is no need to decide upon a certain library to use, as they can be used whenever it is most convenient. There are three popular libraries that assist the most with machine learning.

NumPy

First and foremost, NumPy is a practically essential and fundamental package for all kinds of computing in Python. The documentation explains itself best: “It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more” [A21]. Many frameworks and libraries directly support the usage of this library, as it is becoming a staple for machine learning with Python.

Through the use of NumPy arrays, advanced mathematical operations can be performed on large amounts of data, typically executed more efficiently and with less code than is possible when simply using Python. When working with machine learning, efficiency is key, as large amounts of resources are always in demand. As such, NumPy helps in reducing the demand of resources by increasing the efficiency with which computations are made.

Pandas

Useful for reading all types of information into a Python environment, this library is another essential tool, used for analyzing, manipulating, and preparing data.

Matplotlib and Pyplot

Visualizing data is always important, whether it be for debugging or showcasing, and that is exactly the purpose of this library. Displaying figures or trends over a dataset is made simple with the usage of built in functions.

Machine Learning Concepts

Regression and Classification

In the case of basic machine learning algorithms, the two popular implementations of a neural network are the cases of regression and classification. In terms of regression, the strategy is to create a model that can explain the relationship between two or more independent variables and a dependent variable. This is much like with the implementation of regression within statistics, as both aim to create a plot or equation that can accurately determine the relationship between the variables and then that same plot or equation can be used to predict further values. This is a departure from the normal ideas of regression within natural and social sciences, where the objective of understanding the relationship between separate variables is to be able to characterize those variables to each other, whereas in machine learning, the objective of regression is to be able to predict future outcomes based on those relationships. With classification, the objective of a neural network is still to be able to predict a certain outcome, but instead of predicting future outcomes based on previous knowledge of the data, classification deals with being able to learn different objects and once given a particular object, it can correctly identify the label of the object. To simplify the statement of the differences between regression and classification, regression deals with continuous data points while classification deals with discrete data points, and as such, there are appropriate formulas for both cases.

When creating a regression model, a linear or logistic regression model are a couple examples of the classical approaches to create such models. For linear regression, the model takes on the intercept form y = mx + b, where ‘y’ is our output predicted value, ‘x’ is our independent variable in which the model aims to find a relationship with ‘y’ and other independent variables, ‘m’ can be considered the weight of the independent variable and how much of an impact does it have on the output value, and ‘b’ can be considered our bias value that has a net impact on the entire model itself to guide it towards the predicted value. A more generalized form of the linear regression model is:

$$\hat{y} = w_1 x_1 + \ldots + w_d x_d + b.$$

where the ‘m’ value has been more appropriately labeled ‘w’ for weights and shows that there can be a different set amount of linear combinations of weights and inputs, depending on the context of the problem. For instance, a mapping of bird migration count in relation to temperature of a particular region narrows down the quantities to only one independent variable, that being temperature, and one dependent variable being the number of birds migrating from a certain region. However, in a more practical application of machine learning such as object identification in the case of different fruits, determining the different kinds of fruit within an image can be a matter of determining the shape of the object and the color of an object, which involves two independent variable and one dependent variable, being the prediction of what fruit is within the image. Typically, for the best results possible, more independent variables are introduced into the equation to make each object as distinct as possible to avoid errors in the prediction values.

Due to the linear model being able to plot a line-of-best-fit for a majority of given data sets, linear models are useful in the context of creating regression models that allow the machine to forecast future outcomes given the data points relating to the training the machine has undergone. For classification modeling, logistic regression is best at creating those models due to its modeling not having to be restricted to solely direct relationships between independent variables and dependent variables. While it does not have a concrete formula much like its counterpart with linear regression, the logistic regression model bases its principles on probability and the Odds ratio.

With probability, it can be determined the possibility of a predicted outcome of a value based on a linear combination of independent variables and their weights, which is similar to how the linear regression model establishes itself, with the different being that the linear combination directly impacts the predicted result, whereas logistic regression gauges a probability in which a certain predicted result occurs based off of the combination. Therefore, if we create a linear regression model that produces a certain predicted value, the logistic regression model uses that as an input and calculates the probability of that specified predicted value, which is calculated in our first consideration of an activation function: sigmoid.

Activation Function

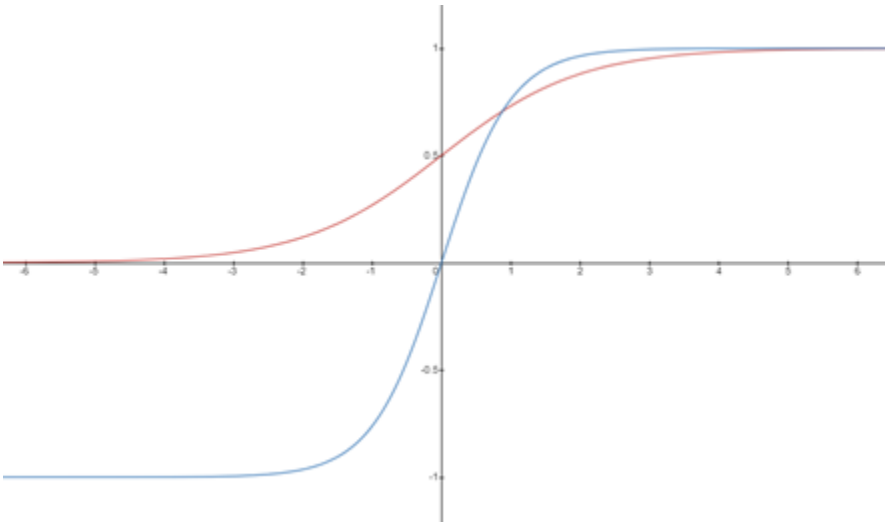
As described from the previous section, logistic regression is calculated by taking the sum of the linear combination of weights and values along with their biases, which is then calculated as the predicted value of the linear regression model, and then calculates the probability of such a predicted value to occur in a standardized equation. In this instance, the classical activation function that is used initially for neural network construction is the sigmoid function, which is defined as:

$$f(x) = \frac{1}{1 + e^{-(x)}}$$

where ‘x’ is the predicted value from the linear regression model. The activation function’s value distribution is structured where the ranges are between 0 and 1 and the intercept value is at 0.5 or 50%. This activation function goes nicely with the concept of probability because it lies in between two values, 0 and 1, that can be thought of as 0% and 100%. In the use of sigmoid, positive values are regarded as values that are more favorable to obtain since the return on the sigmoid values will be 50% and above, whereas negative values are not that desirable since those values fall within the range of 50% and below. Typically, users will define their outcomes of the activation functions to round to one value or another, based on if the return of the sigmoid function was above or below 50%. For instance, if the model were to determine a pass or a fail state classification based on the number of independent variables provided, the sigmoid function may return values such as 0.653 or 0.317, so the user may state that any value above a certain threshold (for instance, 0.5) will be considered a pass state while any value below the threshold will be considered a fail state.

The sigmoid function is not the only activation function that exists for neural networks. Another example of an activation function is the softmax function. This function is a generalization of the sigmoid function that takes advantage of the sigmoid function’s limitation in that it is useful for primarily binary classification. In terms of identifying multiple classifications within a given instance, the sigmoid function’s sum of probabilities do not exactly add up to 1 all of the time since the function is mainly capable of handling one input at a time. With softmax, however, it calculates its sum of

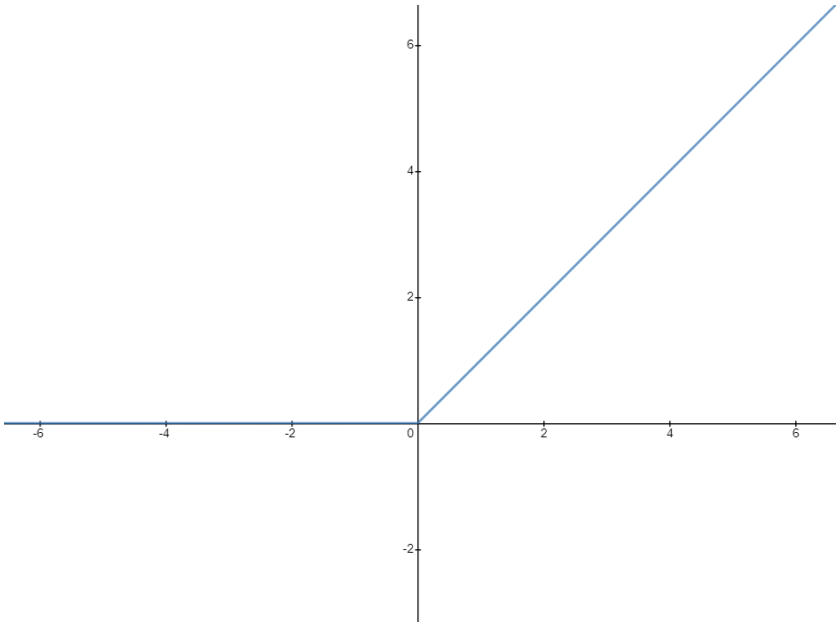
probabilities to ensure that it always equals 1 and determines which of the classes has a higher probability than the others. This is especially useful in the case of identifying hand-drawn digits, where a particular hand-drawn digit may look like a 1 or a 7, but the softmax will determine that one of the digits has a larger probability rating than another and thus makes the larger probability rating the final decision for the classification. Additionally, another activation function that can be used for classification is tanh. The advantage of using tanh over sigmoid can be seen within a graphical depiction of the two functions:



Comparison between the Sigmoid and Tanh activation functions.

where the red graph depicts sigmoid and the blue graph depicts tanh. With sigmoid, while the domain goes from negative infinity to positive infinity, the range is closed between 0 and 1. In contrast, the range for tanh extends itself to go between -1 and 1, which can be used to punish negative values within the domain in a more aggressive way than what sigmoid can accomplish.

Lastly, the most used activation function within modern neural network architectures is the rectified linear unit function, also known as ReLU. With this function, the domain still retains from negative infinity to positive infinity, but the range is determined as follows: if the ‘x’ value is greater than or equal to 0, then the function takes on a linear y = x form that extends from 0 to positive infinity. If the ‘x’ value is less than 0, then the return value is always 0. The graph for ReLU can be visualized as:



Graphic depiction of the ReLU activation function.

The ReLU function takes on a different approach than the other activation functions introduced in this section, as it does not give a direct probability rating based on the input value it receives, but rather promotes the direct relationship between the independent and dependent, predicted variable whenever the independent variable is a positive integer. Should the independent variable be a negative integer, it is squandered to be just 0. This new approach sounds simple to implement but it is a means of solving a particular problem that is discussed within the Gradient Descent section of the machine learning concepts.

Much like how softmax had similar approaches to the sigmoid function, the ReLU function has derivations of itself that sees its applications in different network architectures. For instance, the Leaky ReLU variant proposes the same approach with ReLU in terms of positive integers for the input values, but for negative integers, instead of being reduced to just 0, Leaky ReLU suggests a linear line for negative values that are a fraction of itself. For instance, while the positive integers are represented with $y = x$, the negative integers can take on a similar linear form but the x values are multiplied by a percentage of itself, such as $y = 0.01x$. This will allow the function to be able to take into consideration any kind of input values instead of

completely disregarding values that are negative, while still discouraging the existence of negative values.

Another activation function for consideration is the maxout function, which is known as a learnable activation function and ReLU is a special case of maxout. With the maxout function, it takes in a certain number of elements within a group as input and provides the max output based on those provided parameters. This is considered a learnable activation function because the function is adaptable to the number and kinds of elements it can expect, making it resemble a piecewise function that can graph out different output solutions and then deciding the maximum output based on the inputs. While there are several more activation functions that can be used for any given neural network architectures, there are more appropriate functions to consider for certain cases and some activation functions have their disadvantage that other functions aim to solve as part of their calculation processes.

Loss

While users wish to create a neural network that can be accurate in predicting future results or any sort of categorization problem with 100% precision, the reality is that all networks will make mistakes when it comes to calculating their predictions based on the inputs they were given. The objective with creating a neural network is to create one that can minimize the amount of mistakes they make, and quantifying the amount of mistakes a network makes in a trial through the given data set is known as “cost” and “losses”.

Much like with making a line-of-best-fit manually, a network aims to make the best predicting model as possible by minimizing the margin of error between each data point it observes. In terms of plotting a graph of a line-of-best-fit, that idea of minimizing the margin of error can be observed with how the line closely follows each point and the network attempts to find a line plot that is as close to each point in the graph as possible.

To calculate the error within the predictions that the network has made, there are two approaches that can be adopted for linear regression models: mean absolute error and mean square error. For mean absolute error, the function can be defined as follows:

$$\frac{1}{m}\Sigma_1^m|y - predict(x)|$$

where for every data point ‘m’, the difference between the predicted value and the actual value ‘y’ per data point is summed and then averaged to find the network’s average error in the first iteration of the training process. For mean square error, the function is as follows:

$$\frac{1}{m}\Sigma_1^m\big(y - predict(x)\big)^2$$

where the process is the same as from before, but instead of taking the absolute value of the difference per data point, the square value of the difference is taken for each data point. This approach is a more desirable approach for calculating the error in a linear regression model because when the difference is squared, the error values that are large become amplified and it makes the error more apparent, suggesting the network to urgently rectify those errors. Additionally, the mean square error is easier to differentiate, as that becomes important when it comes to calculating gradient descent for the neural network.

While the mean square error is more appropriate for calculating the error value in a linear regression model, the logistic regression model is more known to adapt a different error function than mean square error, which is a cross-entropy error function. The cross-entropy error function takes on the form:

$$\frac{-1}{m}\Sigma_1^m\big(ylog\big(predict(x)\big) + (1-y)\log\big(1 - predict(x)\big)\big)$$

The function can be analyzed into two components. The first component is the product of the expected ‘y’ value and the log of the predicted value in comparison. The second component is the product of 1 minus the expected ‘y’ value and the log of 1 minus the predicted value in comparison. In this instance and the instances of the previous error functions, it is assumed that the ‘y’ values are binary, meaning that they are either 0 or 1. For this error function, no matter what the ‘y’ value is, there is only one component to compute since the other component disappears from the calculations. For instance, if the ‘y’ value was 1, then the second component would disappear because (1-y) would evaluate to be (1-1) which equals to 0 and negates the second component

completely. Additionally, if ‘y’ was 0, then the first component would disappear because the entire first component would be multiplied by 0, making the entire first component negligible.

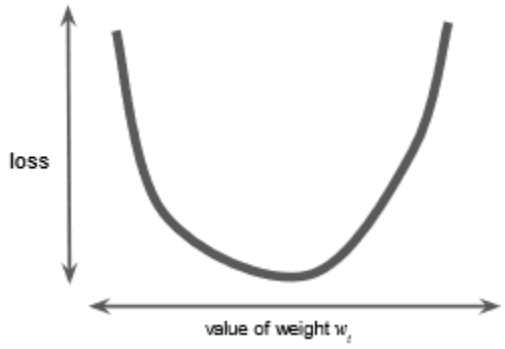
Regarding the first component, our ‘y’ value equals to 1, therefore we expect that our predicted value should be as close to 1 as possible. If it does equal to 1, then the log of 1 would just be 0 and there would be no error in that value to calculate, which is a positive outcome for the network. However, if it is not 1, then it must be less than 1, which means that an error does exist. The further the predicted value is from 1, the larger the log value will be and thus the larger the error value will be for that one data point. Since the log operator is taking the log of a decimal value less than 1, it provides a negative number, so the -1 on the outside of the summation helps in getting rid of the negative values in the error function.

For the second component, the approach is similar to the first component with the ‘y’ value now equaling 0. Should the predicted value also equal 0, then all that is left is the log of 1 and it will equate to have 0 error. However, if the predicted value is not equal to 0, then it must be greater than 0. Therefore, if the predicted value is off, then the difference between 1 and the predicted value will start to accumulate and will then lead to an error cost.

Much like with mean square loss for linear regression, the cross entropy error formula is useful for logistic regression in that it is a sound way of calculating error based off of how far the predicted values are from the actual ‘y’ values and it is also differentiable, which will be useful in calculating the gradient descent for the neural network.

Gradient Descent

For training a neural network to optimize its evaluation to be as accurate to the labels as possible without increasing the amount of errors accumulating in the training process, it can be reasoned that determining the lowest losses possible for the network can be achieved by manipulating the weights for each input. This relationship between the loss of the network training progress and the weights can be visualized as so:



[J18] *Graph depiction of the relationship between weights of the network and loss*

At some point, the weight decided for a given input value will yield the network the lowest possible loss that it can achieve for that one value and is favorable for the network to retain that weight for the singular parameter. However, the appropriate weight for each value may be different when compared to each other in order to determine the combination of weights that will yield the lowest loss possible for the network. This is a sound idea because there are parameters that can be included within the combination of independent variables for the network that do not have a tremendous impact on the predicted output of the network, whereas other independent variables have a large impact on what the predicted output of the network should be. For instance, when trying to classify different animals in a given region, a feature that might be added as an input to the network to help with classification may be the color of the animal, and while it may be helpful in some instances, it may not be an overall helpful parameter to have when differentiating animals from each other.

To determine what weights should be used, the network must first choose arbitrary weight values to test out and see what prediction values it obtains, and the error values associated with it. To accomplish this, the network will train off of the data set that is provided by the user and the network, using the random weights selected initially, will attempt to make a prediction for every single data point given. While it might be a sound idea to train the network on every single data point collected, the issue with that approach is that it is computationally expensive to go through every single data point collected since the network will need to go through those data points several times known as “epochs”. To remedy this situation, a method of training samples for the

network to learn from is known as Stochastic Gradient Descent, also known as SGD. With SGD, a random, single sample is chosen for the network to train from and perform its calculations and since it is only a single sample, the amount of time spent on computation is greatly reduced for the network. However, just training the network off of one sample per epoch is generally not a reliable way of doing training, so whenever SGD is referenced, a “batch” is considered for training, where a batch is a small percentage of randomly selected samples from the entire data set for the network to train from. This implementation of batch training allows the network to reduce the amount of time computing its weights and cost while still having enough samples to be more accurate in the network’s measurement.

After attempting to make its prediction and calculating its errors from the actual expected values for each data point, the gradient is calculated based on the error functions selected from previously. With the gradient calculations, the results provided determine the direction and magnitude at which the weights should move in order to mitigate the amount of errors the network accumulates after predicting its values. So, if the error function that was chosen was the mean square error which is as follows:

$$\frac{1}{m}\Sigma_1^m\big(y - predict(x)\big)^2$$

which can be rewritten as:

$$\frac{1}{m}\Sigma_1^m\big(y - (wx + b)\big)^2$$

then the gradient for the network with respects to the weight ‘w’ is the partial derivative of the error function with respects to ‘w’:

$$\frac{-2}{m}\Sigma_1^m\Big(\big(y - predict(x)\big) * x\Big)$$

For the cross-entropy error calculation, which is defined as the following equation:

$$\frac{-1}{m}\Sigma_1^m\big(ylog\big(predict(x)\big) + (1-y)\log\big(1 - predict(x)\big)\big)$$

the gradient for the network with respects to the weights, which are part of the predict function that was done through a linear combination of weights and values passed

through an activation function, is still the partial derivative of the error function with respects to the weights:

$$\frac{1}{m}\Sigma_1^m(predict(x) - y) * x$$

With these gradient calculations, it allows us to determine the magnitude of the recommended change to our weights based on how large of an error the network has made at each data point. So, in a case where the error was not that large, the gradient calculations will depict a very minor change recommended. Conversely, if the error was large, then the gradient will suggest to the network a very large change in weights. Additionally, in both instances of little or large changes, it will indicate whether the weight change needs to be an increase or decrease in change, identifying a sense of direction in which the weights need to be altered.

It is also in the best interest of the network to update the weights of the network in a steady fashion rather than immediately update the weights as soon as possible. This is because if the weights are to be modified directly by the gradients, then the weights would bounce in all kinds of directions and never fully converge to a local minimum, where the loss is low at an appropriate weight setting. To adjust how much the weights can change, a learning rate is applied to the gradient that determines how much of the gradient is to be used when updating the weight. For instance, if the learning rate was small, then the weights are only updated by a small fraction of the gradient calculation, which guarantees a convergence on a local minimum but at an incredibly slow place. If the learning rate is large, then a large majority of the gradient is used to update the weights and thus leads to the weights having difficulty converging at a singular point. The best approach for learning rates is to have an adaptable learning rate that starts with a large learning rate to allow for larger steps in the direction of a local minimum and then gradually decrease the learning rate so that it properly converges at that minimum.

While the approach to training a network seems straightforward at this point where the network selects random weights initially, makes its predictions, calculates its cost, and then adjusts the weights accordingly through gradient calculations and then repeat the process for a certain number of epochs, the gradient calculation portion is still something that requires optimization because each neuron in the network contains a

linear combination of the inputs and weights and having to calculate the gradients for every weight would take too long for the network to figure out. To put the number into perspective, if the number of inputs was 100 and there are 400 neurons in the network, that is 100 weights for every neuron in the network that needs a gradient calculation, totaling to 40,000 gradient calculations per epoch.

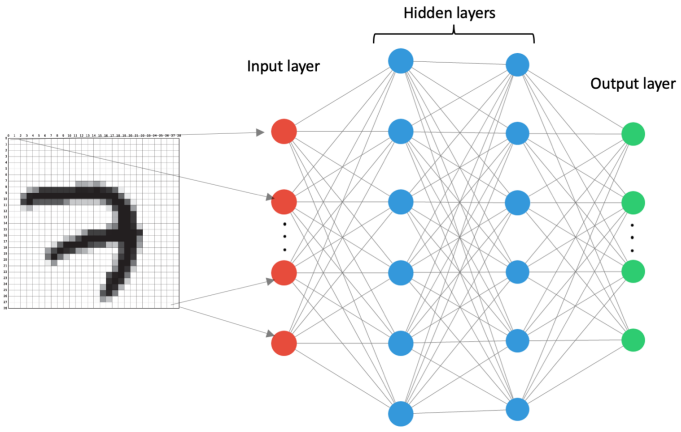
To optimize the gradient calculations for the network, a dynamic programming approach can be implemented to the network that exploits the fact that the neurons already contain the cost values when the network does its attempt at predicting the values to the data set. This algorithm approach is known as backpropagation. With this approach, after the network has done its “forward pass” where the calculations go from one layer of neurons to the next, each neuron already contains their predictions and by the time the final layer makes its prediction, the network can start calculating its error costs for the neurons. This can be accomplished by all neurons in the network that have already gone through the forward pass of the epoch. Once the final layer has computed their costs, the network works “backwards” and extracts the information from the error costs and uses that provided information from each neuron and calculates the gradient for each weight in each neuron until it reaches the front of the network, where it is ready for the next epoch of training and the cycle continues.

Unfortunately, backpropagation does have a disadvantage in that it poses a problem known as the vanishing gradient problem, where the weights towards the tail of the network update normally but as the algorithm reaches towards the front of the network, the gradients for the weights at the front are minimal and causes the weights to not change, leading the network to not adapt to the data set and continue to make more errors in the future. This is mainly caused by the activation function that is selected, such as sigmoid and tanh, where the range values are bounded between either 0 and 1 or -1 and 1 respectively and the change in values is very minimal. A solution to this problem is through the use of the ReLU activation function, which has a larger range than sigmoid and tanh that extends from 0 to positive infinity.

Neural Networks: Overview

As a basis for the prediction-based modeling we will use for this project, our data will be processed with neural networks in order to recognize patterns and relationships that will help us succeed in our goal of developing a team of bots with a game-winning strategy. However, before we create our solution for this goal, it is important that we clarify how using this technology will assist us in our development at a high-level.

At a glance, neural networks seem like incredibly complicated tools, almost like a black box in which it takes some input data and magically returns a way to draw conclusions from that data. But the reality is, once broken down into its key components, the complexity dissolves and you are left with a clearer picture: neural networks are models that simply calculate and provide a prediction based off of its input. By performing sets of calculations in a particular order with learned weights and biases, the model creates a (hopefully) accurate prediction as its output. Importantly, this prediction can be made even when changing the input to something the model has not seen before, constituting a fundamental aspect of neural networks known as learning.

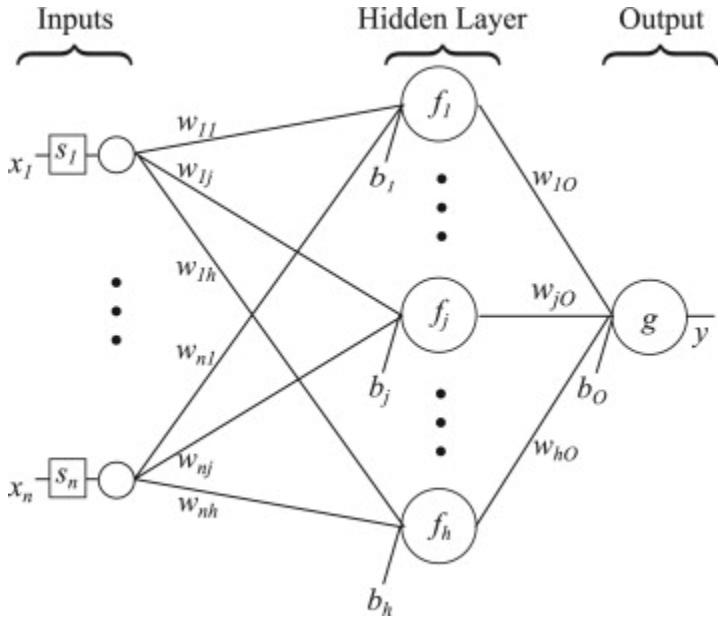


[A2] *Some notable parts here are commonly shared among neural networks.*

Neural Networks: The Multi-Layer Network

On the right-hand side of the diagram above, we can see a forward-feeding **multi-layer network**, which consists of an input layer, any number of hidden layers, and an output layer. The **input layer** is what takes in all of the data provided in the current sample. Each neuron in the input layer represents one or more *features*, pieces of information

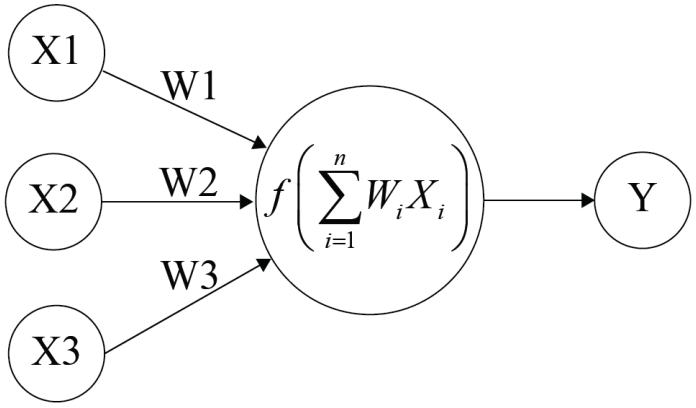
extracted from the sample of data. In the case of this diagram, our feature would simply be the grayscale value of a cell (the value may range from 0-255, 0-1, etc). It is important to note, however, that features can also be more complicated than this if the type of data calls for it. A dataset with color input, for example, may have multiple features within a single vector, each representing their respective red, blue, or green channel. In the case of the Everglades game, a neuron might input features such as location, bot type, or current score. It is important to only select from features which are useful for making a prediction in order to avoid potentially overfitting data with unnecessary information.



[A7] *A multi-layer network with an input layer, a hidden layer, and an output layer.*

Moving on from the input layer, **hidden layers** take the value from the input neurons and, simply put, modify them and pass them on to the next layer. This modification is done with multiple components: weights, biases, and activation functions. **Weights** are constant values that are multiplied by the values of the incoming feature(s), scaling them as they see fit. This gives either more or less significance to a given feature, and is assigned via learning. Once the features are weighted, a bias is added for even more flexibility. **Biases** are constant values added onto the weighted feature and are unique for each hidden layer, having their own unique weight, similar to a neuron. In fact, a bias could even be considered an additional neuron that is appended to each neuron

of the hidden layer, as it is weighted along with the other features and used as an input all the same. Lastly and arguably the most important feature of hidden layers, and perhaps even neural networks as a whole, we have activation functions. **Activation functions** are used in each neuron of the hidden layer to map linear combinations (derived from applying weights and biases to input values) to nonlinear functions, the most common of these being the sigmoid function. The inclusion of these functions allows for greater control of the output of each neuron, as it effectively scales and limits the possible range of output values. For example, in the case of the sigmoid function, its output will always fall within the range of 0-1, no matter how large or small the input.



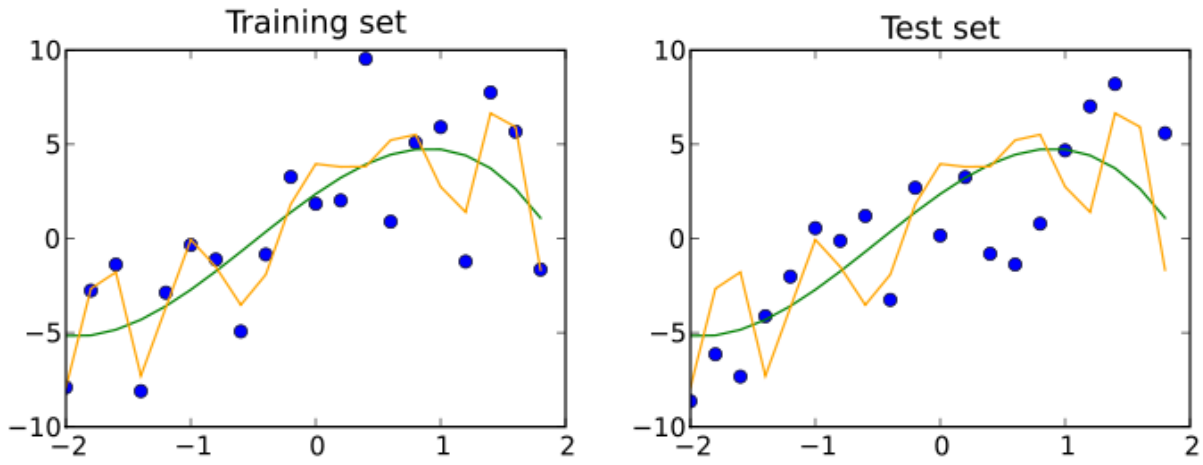
[A8] *A neuron with three weighted inputs, an activation function, and an output.*

The final layer of the multi-layer network is the output layer. Being the most simple and straight-forward of them all, the **output layer** simply takes in input from the previous layer and performs some form of scaling or tuning in order to produce an output within a desired range. A common way to achieve this tuning is using the softmax function (similar to the sigmoid function), which translates the output from a real value to a normalized probability distribution in the range of 0-1. The resulting output from this layer is also known as the prediction of the model.

Neural Networks: Input Data

Moving past the multi-layer network, we can see the data itself on the left-hand of the diagram, which is, in this case, a grid of cells, each presumably containing a value that represents how black or white the cell is. As explained above, the provided data will be fed into the input layer of the multi-layer network, and a prediction for what that data may suggest will be calculated as the output.

There are multiple types of input data to consider for neural networks. **Training data** is used as the initial input to a model in order for it to both fit the data, and to learn what to look for in data that it will see in the future, whatever patterns or other relationships that might be. The amount of training data used is, as a best practice, practically an order of magnitude greater than the rest, as this training is generally the most important step of creating a model. Training data is used for almost all branches of machine learning, as it gives a great foundation with which the model will build upon, and is hand-curated to provide a set of inputs and expected outputs that the model will be expected to accurately predict after training. An exception is in reinforcement learning, as instead of using training data to train a model, it uses a reward-based system to pick up on patterns or other positive behaviors.



[A9] *On the left, a set of training data is shown, and on the right, a set of testing data is shown. Two predictive models are fit to both datasets.*

Validation data is another form of input to neural networks, and is used primarily when choosing between multiple models for a specific task. This set of data is withheld from training, but is directly connected to it by the fact that once a model is done training, it should then be run with the validation data in order to measure the effectiveness of the newly learned model. In this way, validation data can be used to validate that a certain model is fit for the task it is being presented with. It can also be used to determine whether certain features are necessary to include, by observing their influence on the outcome. If a model is not up to the desired standards, some options are to either tune the hyperparameters (parameters applied to the model which are manually set before training) or to select a different model in an attempt to obtain a better result.

After going through both training and verification data, the model may look ready to be used for all future cases (and it is in fact almost ready), but it cannot be considered complete until it is determined to work at the desired level when presented with a set of testing data. **Testing data** is the data withheld from training which is used to measure the effectiveness of a final model. While similar to validation data, note that this data should be used only when a finalized model is selected and is ready to be used for testing some set of input data.

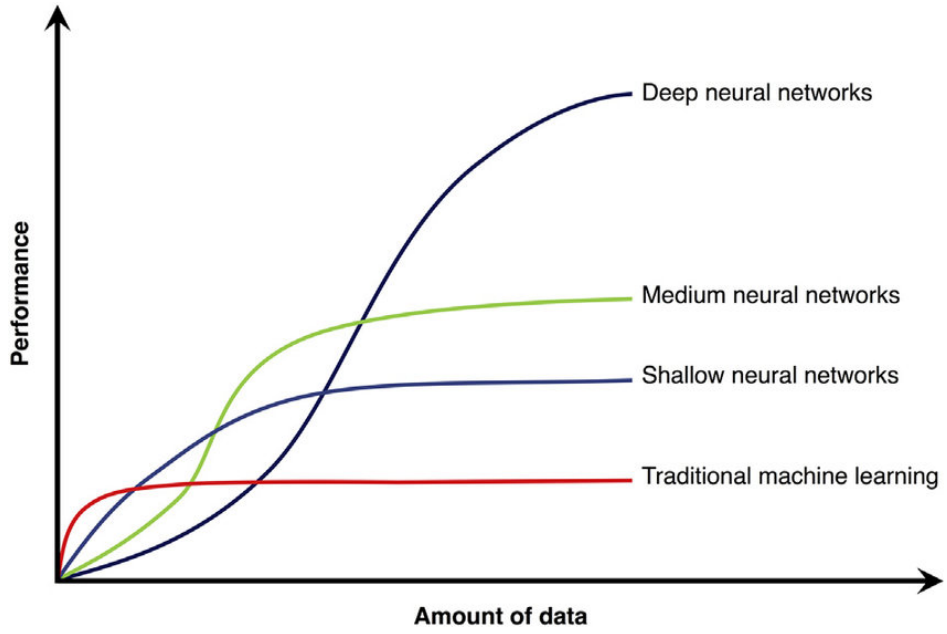
Once all of these types of data are incorporated into a neural network, they will be used in the order specified above to iteratively create an optimal form of the network. This finished model will then be able to be applied for any future input and predict outcomes at a level which is, ideally, similar to those seen when it was applied to testing data.

Neural Networks: Learning

Lastly, there is a third, inferred component of this model. In order for a neural network to be effective, it must first be trained using training data, with each sample of data having a predetermined, expected outcome. If data is just thrown into a network without it having any expectation of what to receive or what to return, how could it possibly make consistent (let alone, correct) predictions? That is where a fundamental aspect of neural networks makes its entrance: the ability to learn. If given proper training data, the model will adapt itself to produce results that succeed in matching

each sample of data with its expected outcome. Once this process of learning how to correctly analyze provided datasets is complete, the model is ready to be applied to testing data, hopefully having success in predicting the correct outcome from data which it has yet to see.

The way in which neural networks are programmed to learn varies slightly from model to model, but it is primarily done by gradually changing the values of weights and biases across iterations. A model will compare its output prediction with the expected outcome and, depending on the error, adjust these weights and biases to try and achieve a better prediction in the next iteration.



[A10] *Different methods of machine learning and neural networks are shown with their performance measured against the amount of data they are given.*

Neural Network Architecture

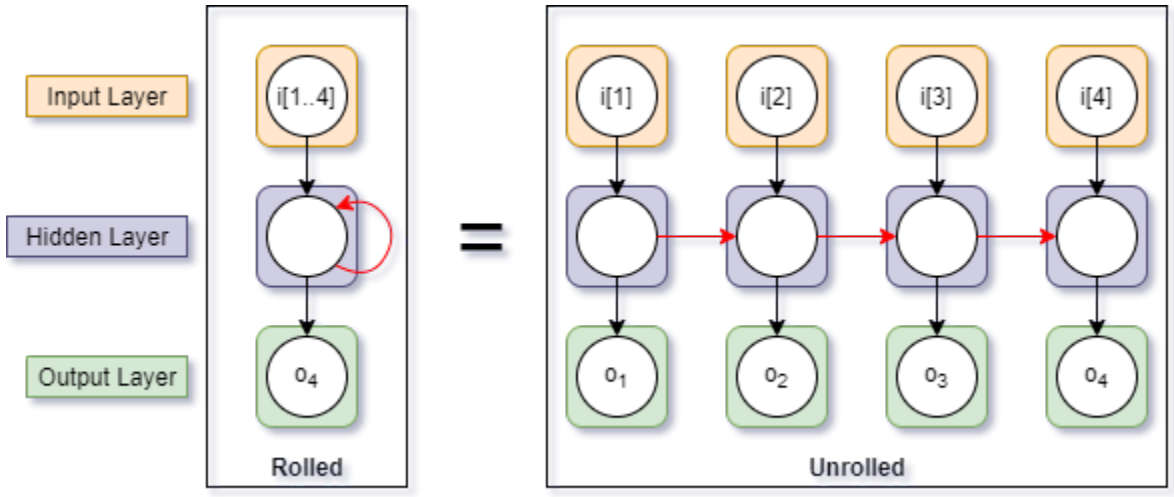
The structure of a neural network gives it its own unique identity, with several of these having their own respective branches within machine learning itself. As a demonstration of the example in the Neural Network section, where an input image is given to a neural network in an attempt to classify what the image represents, we use an architecture known as deep learning along with a feed-forward network.

Feed-Forward Networks

Stemming from the category of deep learning frameworks, feed-forward networks are the foundation for a majority of models using this technology. These networks are very straightforward in the sense that they will take a given input, perform some function upon that input, and return a result as output. Note that this is a direct transformation from input to output; if a network is implemented in a way such that feedback loops are generated via connecting a function back onto itself, the network is further classified as a recurrent neural network.

Recurrent Neural Networks

These types of networks are more complex than a typical feed-forward network, but can also prove to be very powerful when trained effectively. This is due to a combination of two fundamental properties all recurrent networks possess: a distributed hidden state, and non-linear dynamics available to update the hidden state. The usage of states leads to a large variance in the effectiveness of this model, as its performance depends entirely on its implementation in this regard. Similar to a forward-feed network, weights are applied to hidden layers of neurons, but in these recurrent networks, these weights are continuously applied over time without the requirement of moving onto the next layer of neurons.



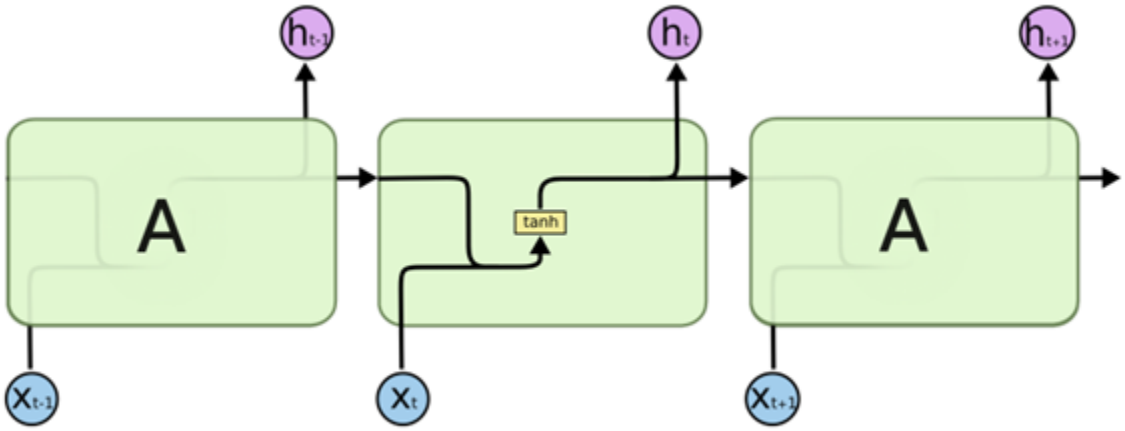
A recurrent neural network can be viewed as many copies of a feed-forward neural network executing in a chain.

Recurrent networks can be used to infer a prediction of the next item of an input sequence without knowledge of the target sequence. This method of prediction borderlines between supervised and unsupervised learning, as it uses supervised methods but without any teaching signal. The power that these networks hold is matched with an equal difficulty in its training requirements. The vanishing gradient problem is very prevalent in these types of models, along with having multiple methods to provide input to the network, each having a different deterministic outcome.

LSTM Networks

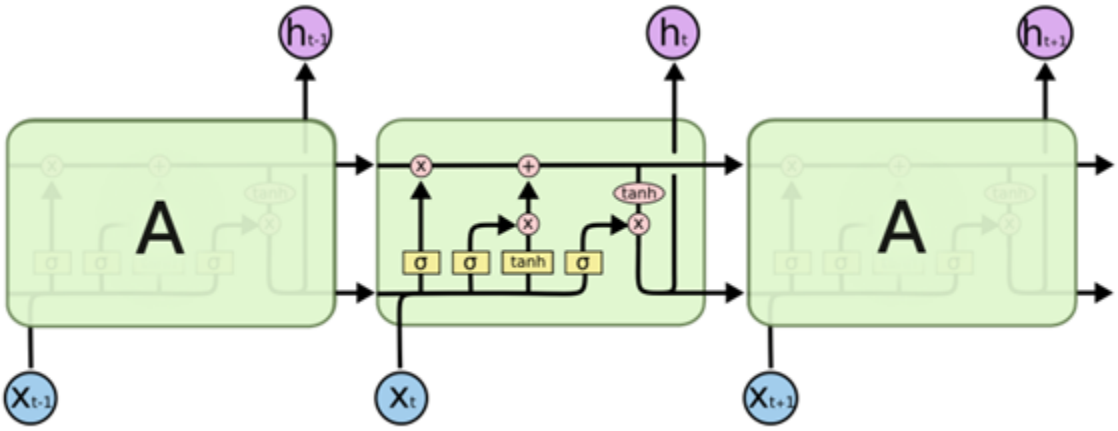
Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They work tremendously well on a large variety of problems and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem.

Remembering information for long periods of time is practically their default behavior, not something they struggle to learn! All recurrent neural networks have the form of a chain of repeating modules of neural networks. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer [B1].



The repeating module in a standard RNN contains a single layer.

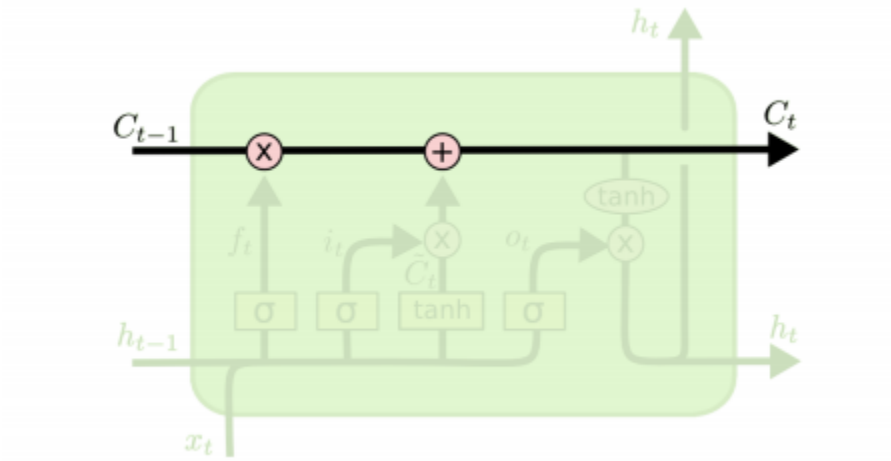
LSTMs also have this chain-like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way [B1].



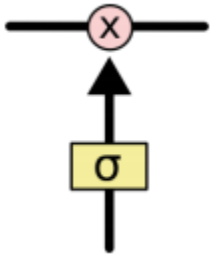
The repeating module in an LSTM contains four interacting layers.

The Core of an LSTM network

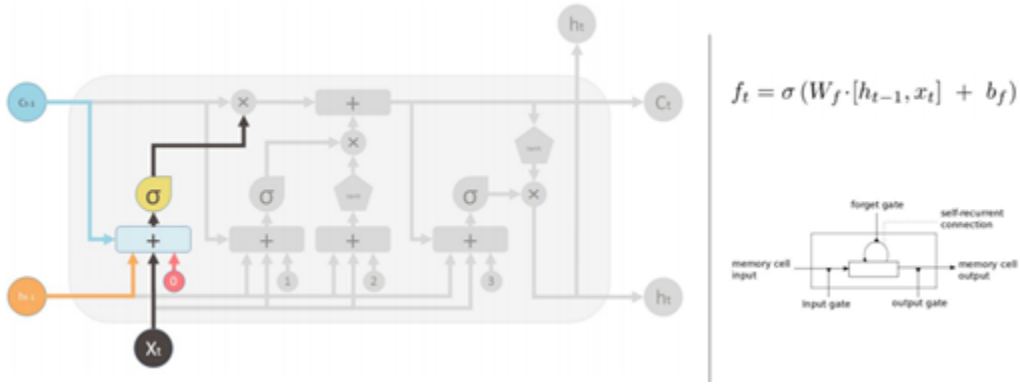
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



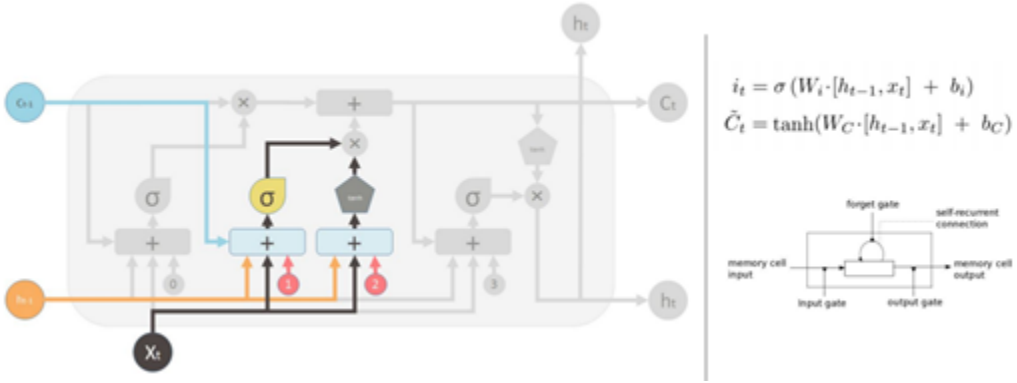
The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



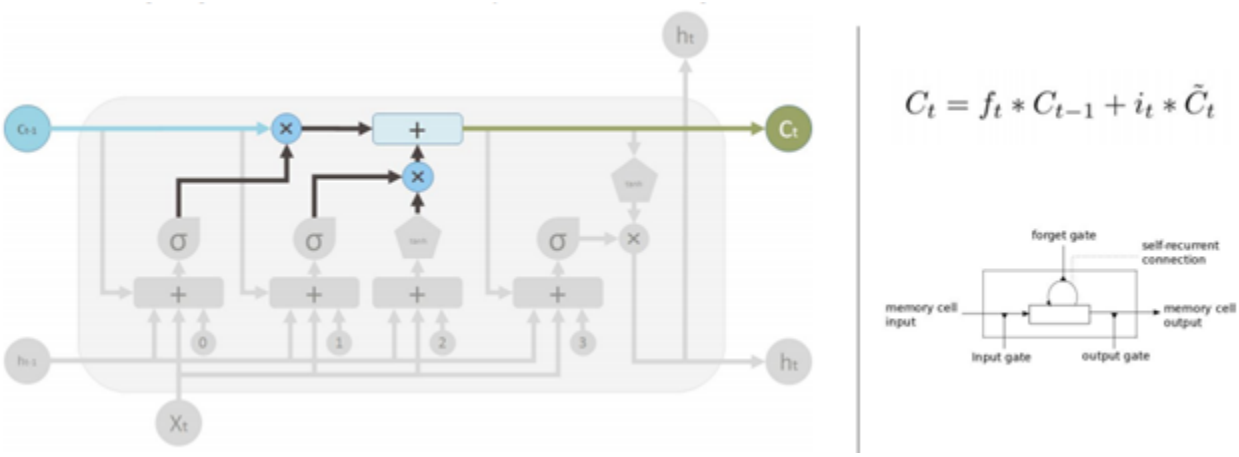
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”
An LSTM has three of these gates, to protect and control the cell state.



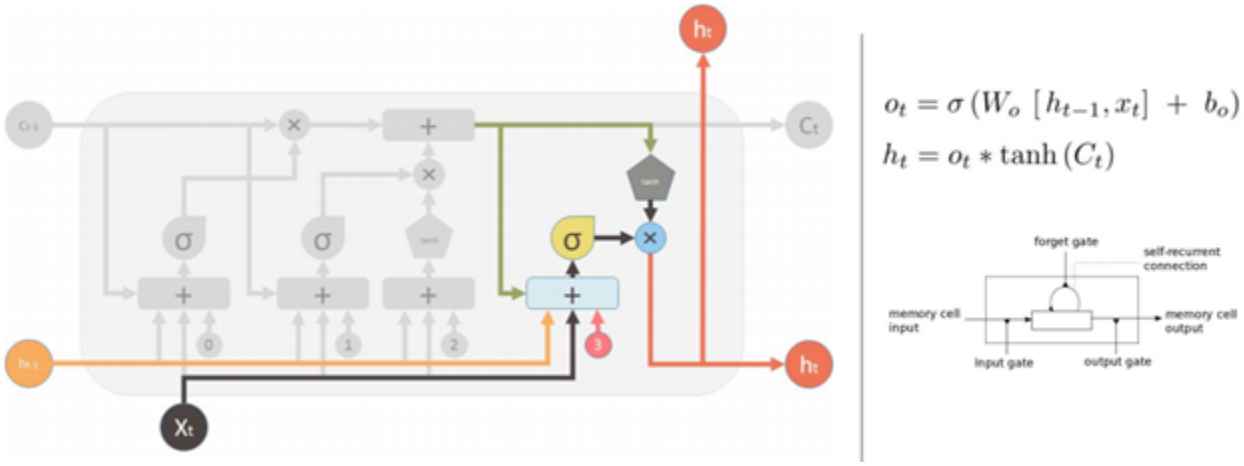
The first gate is called the Forget Gate, which controls what information needs to be forgotten.



The second gate is the input gate, controlling the new information being added to cell state from the current input



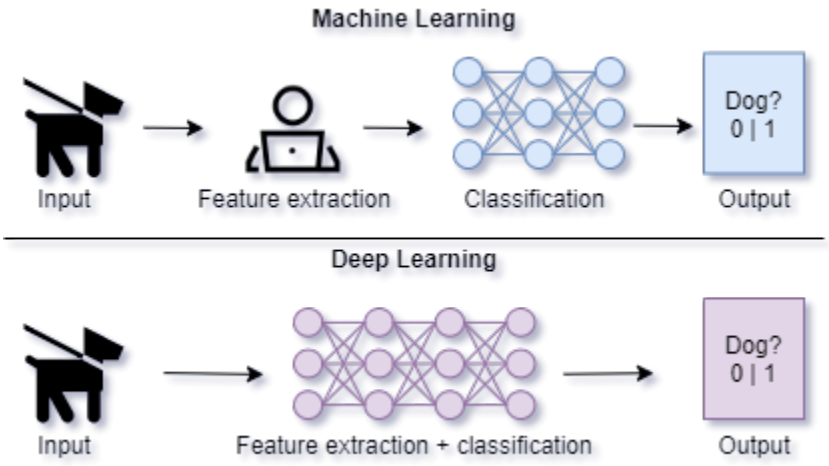
The cell state vector aggregates the two components (old memory via the forget gate and new memory via the input gate)



And finally, the output gate decides what to output from the memory

Deep Learning Networks

These types of networks use large, vast training sets to thoroughly train a model step-by-step, layer-by-layer. Multiple hidden layers are implemented to extract features from the input data, with each layer obtaining more detailed or specific information than the last. These layers compound upon each other and work towards predicting some classification of the input data, all still within the processes of the deep learning network.

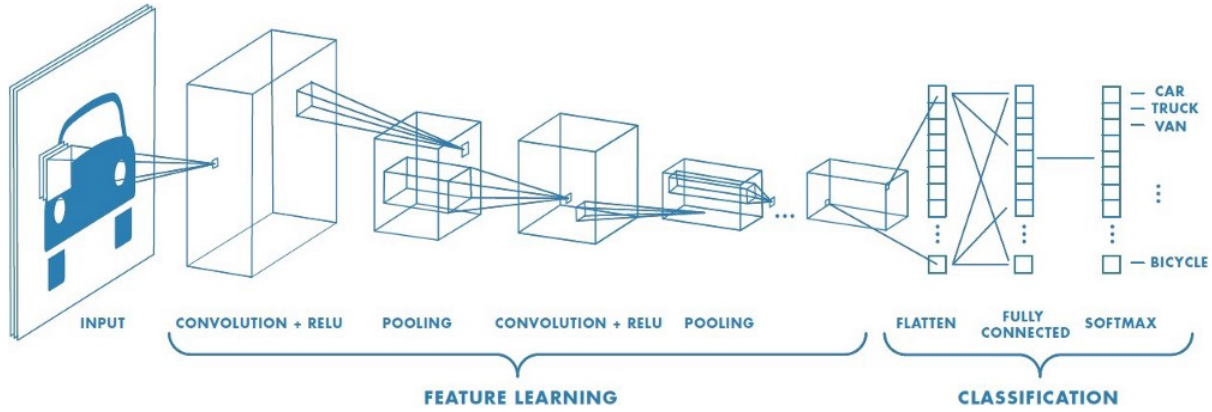


Comparing deep learning with machine learning.

This type of neural network is the most common, as it is very generalized while also consistently providing good results. There exist subsets of this network of deep learning, further building upon its strengths and either generalizing or providing more specification when needed. An example of this would be the convolutional neural network model.

Convolutional Neural Networks

As a subset of deep learning, convolutional neural networks also utilize multiple hidden layers with large amounts of input in order to achieve an optimal resulting model. The name of the network itself actually derives from a fundamental operation known as convolution, which creates feature maps from input data. The combination of a deep learning network and an implementation of convolution leaves it as a popular choice for the task of identifying images or objects from a scene. This task is something trivial for humans, yet normally incredibly difficult for computers as there are just too many variables to account for (lighting, rotation, 3D space, etc). What sets this type of neural network apart from others is the implementation of the replicated features approach, also known as parameter sharing. This approach employs the usage of many different copies of the same feature detector in order to reduce the number of free parameters that must be learned by the model. In other words, it simplifies the learning process at the low-level by allowing the sharing of parameters for similar features across an input.



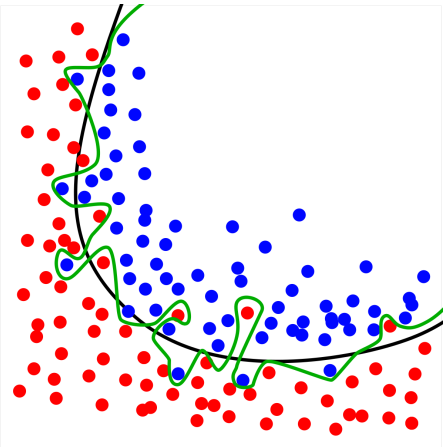
[A6] *An example of a convolutional neural network classifying an image of a car.*

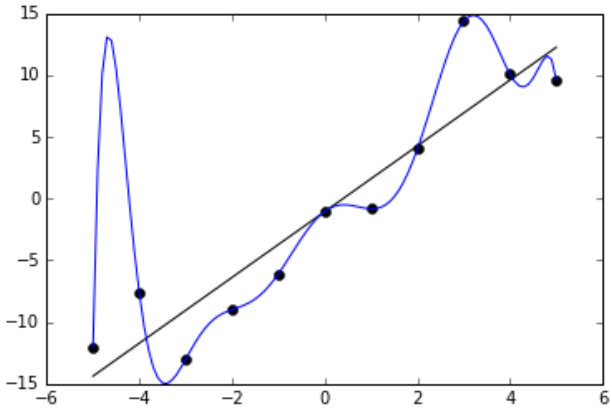
Autoencoders

On the simpler side, another type of neural network known as autoencoders are used in unsupervised learning frameworks to learn how to efficiently create representations, or encodings, for a set of input data. A rough comparison could be made from this type of network to a sort of data compressor. When given input data, an autoencoder will deconstruct the data into a simpler form within the hidden layers of its network, and then subsequently output the important pieces of information it could grab out of the compression process. By using a small number of neurons in the hidden layer, the model creates a bottleneck, which forces the model to not simply memorize and regurgitate the data it is receiving, but instead pass it through the bottleneck, squeezing out the useful parts. The main appeal of autoencoders is this ability to reduce the dimensionality of a set of data, learning to ignore any unwanted, noisy, data it might receive as input and only process the important parts.

Overfitting

When a model learns from a training set, it picks up on patterns and other relationships from that data in an attempt to generalize it. Sometimes, however, this generalization fails, and all that is left is a model which simply makes predictions based on the training data and the training data only. In terms of a graphed function, this may result in a model which returns a graph that nowhere near represents the testing data, but instead is still trying to fit the training data. As another example, a set of training data might try to teach a model to recognize cats from dogs, but in testing it might match anything as a dog if it is brown, simply because brown dogs were far more prevalent in the training set than brown cats.





These figures [A13] shown above demonstrate the concept of overfitting data perfectly. In both graphs, there is an approximated and an overfit curve. Although the overfit curves, being the green line (left) and the blue line (right), accurately matched all of the data points correctly, this type of curve would never work outside of this specific set of data, as it is far too specific and complex for the type of data given. Instead, the approximated curve would be more suitable for the generalized processing of data that is required for machine learning. There are a few ways of dealing with overfitting, such as obtaining a larger or more generalized training set, or applying regularization such as early stopping or using dropout techniques.

Regularization

Obtaining larger sets of training data isn't always the most useful advice when trying to handle overfitting, and even still, it decreases in effectiveness past a certain point in most networks. Regularization is introduced as a way to help combat this issue.

L2 Regularization

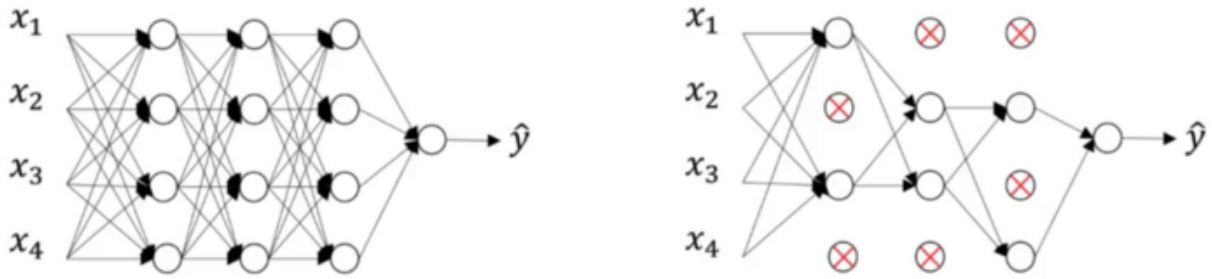
In this form of regularization, an extra term is added to the model's cost function, namely, the Frobenius norm of the weight matrix, which is equivalent to the squared norm of a matrix. In this term, lambda is a hyperparameter referred to as the regularization parameter. This extra term can be seen in the function below on the right hand side, added to the cost function we are used to seeing.

$$J\left(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|w^{[L]}\right\|_F^2$$

Since the regularization parameter is a hyperparameter, it can be tuned before the model starts training. When lambda is large, larger weight values are penalized more. Similarly, when lambda gets smaller, the regularization effect diminishes. When lambda is zero, no regularization takes place.

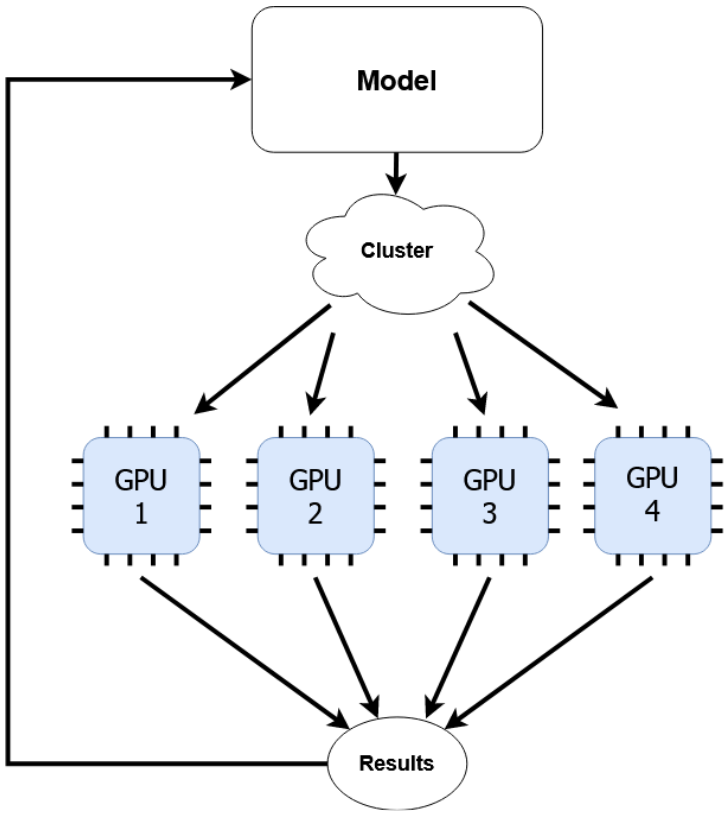
Dropout Regularization

To reduce the chance of overfitting, this form of regularization assigns a probability for each neuron in a hidden layer to be kept in the model. This means that each neuron in a hidden layer has a chance to be completely removed from the model for that iteration. A threshold is set as a hyperparameter that decides what this probability should be, with a value of 0.8 representing a 20% chance to remove a given neuron. In this way, the model learns not to weigh any given neuron too heavily, as there is always a chance for that neuron to be removed.



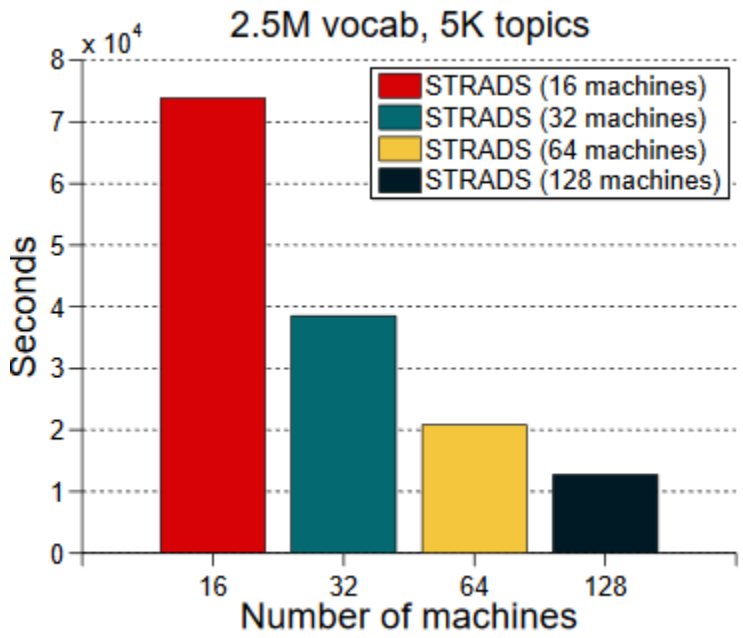
Parallelization

One of the most impactful time-saving tools that exist is parallelization. With this, it is possible to perform independent calculations throughout different machines. This can potentially be very cost-effective, reducing the time it takes to perform a set of calculations by orders of magnitude. Parallelization is a broad subject, and covers much more than machine learning. With this power, though, one can create models that converge much quicker than they would have otherwise.



A diagram showing how a cluster would simultaneously operate multiple iterations of an algorithm on different GPUs.

In reinforcement learning specifically, not every algorithm is fit to handle being processed in parallel. Usually, iterations of a machine learning algorithm tend to use information gained from previous iterations (it uses memory of past runs to achieve better performance). Asynchronous, parallel runs of an algorithm tend to not apply well when attempting to learn, as the learning iterations would be independent and thus could not utilize the experience learned on other GPUs. In places where



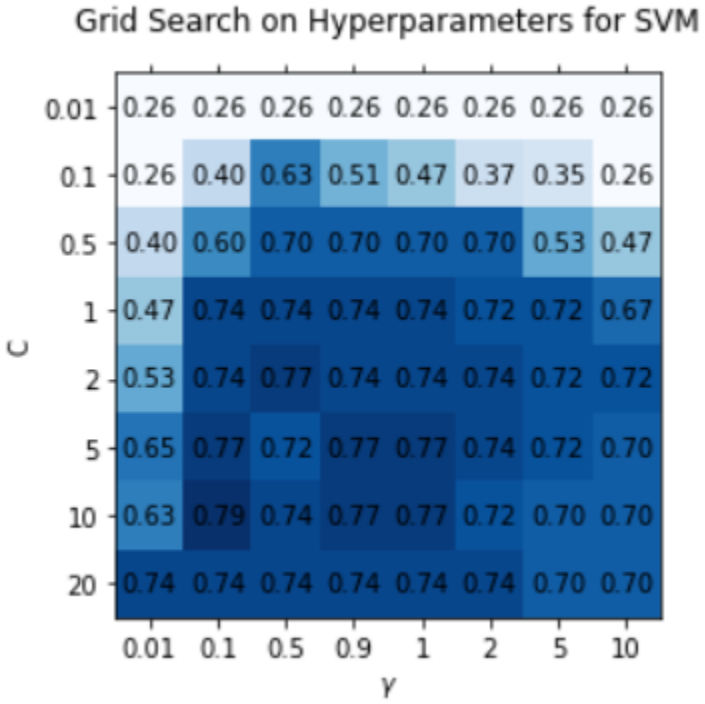
[ML21] *The effect that running multiple machines in parallel has on time for a machine learning algorithm.*

One scenario that would benefit any algorithm though is to apply it to meta-analysis. Meta-analysis in machine learning can come in many different forms. The most common of those are hyperparameter tuning and testing different models in parallel. Not only does this automate the learning process, it also reduces time by doing different training runs on different machines.

Hyperparameter Tuning

Typically, hyperparameters are chosen with former intuition on how those values will affect the model. This, however, leaves out the potential to algorithmically find the best values for the model to work with. Even with the vast differences in hyperparameter types and model variants, there are still ways to search for the best ones.

A grid search is the simplest of these methods. For a hyperparameter h_i, a set of possible values is given to that hyperparameter. The range can either be defined in the parameter itself (ex. the amount of dropout for nodes in a neural network is in the range $[0, 1]$) or by prior knowledge of useful ranges. A grid with dimensionality $|h|$is made, and all possible hyperparameters will be used in model training.



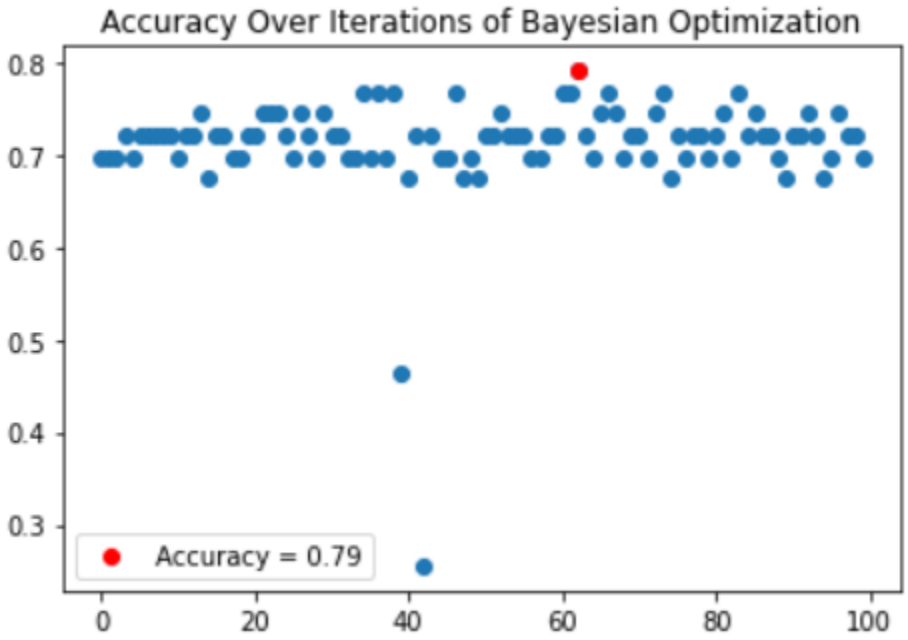
An example of grid search performed on an RBF-based SVM. The values are accuracies, so C=10 and γ=0.1 were the best values.

Grid search is an exhaustive method, i.e. brute force, so it is extremely time-costly. However, grid search is trivial to parallelize, which can greatly decrease that costliness. For Everglades specifically, the Newton Cluster allows relatively simple parallelization of the GPUs. Despite being brute force, performing a grid search may be useful. One optional change to grid search is to randomize the search, which can give a better understanding of the hyperparameter values that yield the best result. Randomizing the search has drawbacks, though. An exhaustive search would take the same time as grid search, and randomizing also makes parallelizing more difficult.

The main problem with the search algorithms so far is that they are slow. This is because they are not *informed searches*. This means that there is no knowledge gained from tuning the hyperparameters in a particular way. Bayesian optimization fixes this issue by creating a probabilistic model, known as a *surrogate function*, for determining what values of hyperparameters work best.

Bayesian optimization can be described as having four elements: the objective function, the domain space, the optimization algorithm, and the results. The objective function is the model who's hyperparameters need to be tuned. The domain space is all of the possible values that the hyperparameters can take (these can either be categorical or numerical). The optimization algorithm is chosen to create and fit the surrogate function with hyperparameters that it chooses, and returns the results that are saved into a history of results.

There are many optimizer functions that exist, such as the Tree Parzen Estimator (TPE), gaussian process surrogate, and random forest regression. The goal of these surrogate functions are to create an estimate of the objective function using the hyperparameters that are being tuned. This function then makes predictions about the parameters, which are then applied to the objective function. Over time, the surrogate function converges closer to the objective function, which causes the minimum loss.



An example of Bayesian optimization, using TPE, performed on the same SVM for 100 iterations. It found C=16.749 and γ=0.046 to be the best values.

As seen in the above graph, this technique balances exploring the domain space and exploiting knowledge gained to achieve the highest accuracy result. In this case, grid search and Bayesian optimization were given roughly the same amount of iterations, and they found near identical optimal values. In higher dimensions of the domain space, however, grid search would increase exponentially. Bayesian optimization would not suffer this explosion in computational resources.

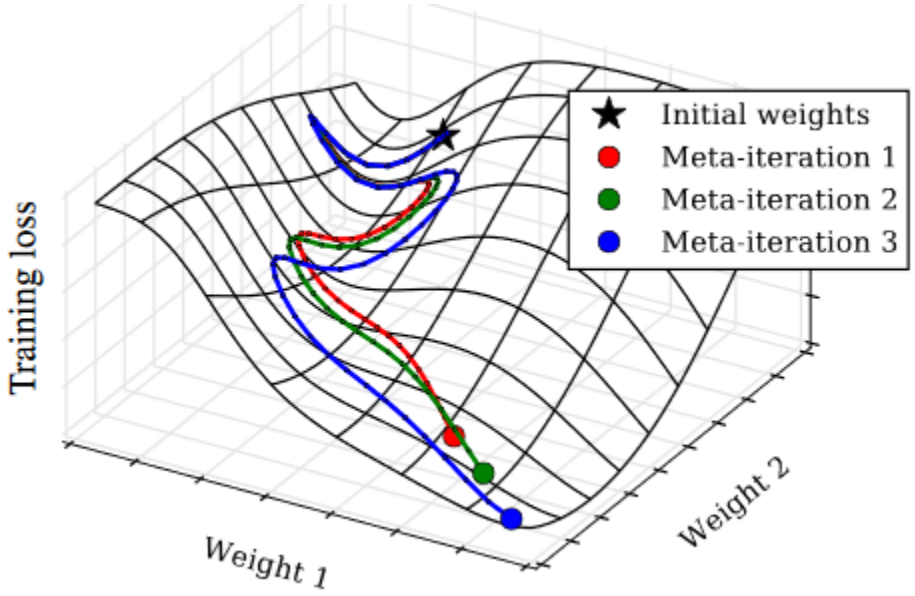
Bayesian optimization is most certainly a highly valuable tool that can be implemented into the Everglades project for hyperparameters whose values are uncertain. The time that is saved using this method is worth the computation overhead of making the informed choice of hyperparameters, and may even overcome the parallelization advantages that grid search offers.

One of the most popular tools for implementing Bayesian optimization is a library known as `hyperopt`. This library allows for many optimization algorithms to be implemented for a wide amount of machine learning models with relative ease.

```
from sklearn import svm
from sklearn.metrics import accuracy_score
from hyperopt import fmin,tpe,hp,STATUS_OK,Trials,rand

trials = Trials()

# Defines the objective function
def tuner(params):
    model = svm.SVC(**params,kernel='rbf')
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    rbf_acc = accuracy_score(y_pred,y_test)
    # Minimizes function, therefore -accuracy
    return {"loss": -rbf_acc, "status": STATUS_OK}

space = {
    "C": hp.uniform("C",0.01,20),
    "gamma": hp.uniform("gamma",0.01,20)
}

best = fmin(fn=tuner,
        space=space,
        algo=tpe.suggest,
        max_evals=100,
        trials = trials)
```

The code used to run a SVM through Bayesian optimization; the results are given in the above chart (all iterations of the meta-analysis is stored into the trials variable).

For some applications, there is yet another set of tuning algorithms that are commonly used. Gradient-based optimization allows for hyperparameters to be tuned in accordance with gradient descent.



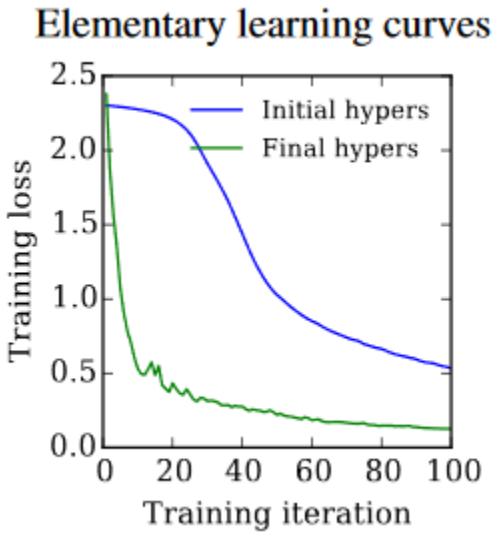
[ML20] *Iterations of gradient-based hyperparameter optimization ran on two parameters.*

Most importantly, for any gradient-based optimization technique to work, the hyperparameters must exist in the proper space such that minute changes are reflected in the parameters. This means that categorical data or integer data would not work as hyperparameter inputs, and would need to be optimized via another method.

Similar to normal weight updates in a neural network, the gradients for learning through meta-iterations occur through a forward and a backward pass of the entire training process. This includes all the passes of a single meta-iteration to the network to gain valuable insight into what the gradients are, which increases memory storage requirements drastically. Optimizations have, however, been made to this approach, and has made gradient-based optimization possible to do on normal datasets.

The main advantage to a gradient-based approach is that many hyperparameters can be efficiently tuned at once. In grid search, and even Bayesian optimization, parameter search increases dramatically as more hyperparameters are added, especially in cases where there are a lot of categorical parameters or large search spaces.

The lack of ability to handle discrete values is the most negative aspect about gradient-based optimization, though even this can be relieved through parameterizing the discrete space into a continuous one for the algorithm. However, this method shares all of the same downfalls as normal gradient-based learning. There is still the issue of the exploding gradient, overfitting the validation set is still a possibility, and it is complex. The results though are pretty good:



[ML20] *An example of the loss on a model with initial hyperparameters and the ones learned from gradient-based tuning.*

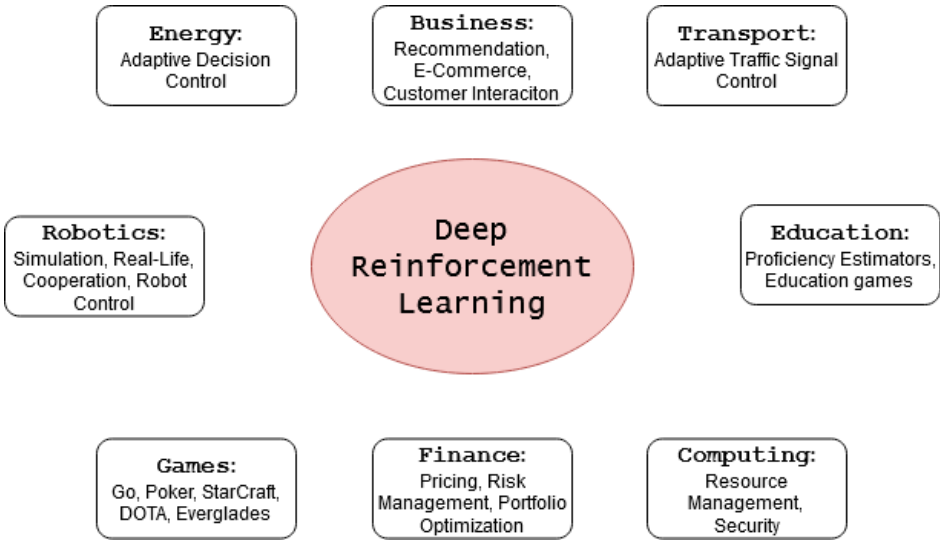
For implementing any of these algorithms for an agent, a full training session would not take place. The agent would instead train using sets of hyperparameters for relatively short periods of time, and then the best parameters would then be used in full training sessions. This would further reduce any temporal overhead caused by the tuning process. Most likely, Everglades will use Bayesian optimization as it provides the simplest, most efficient optimization while also allowing for categorical parameters to be included in the search.

Reinforcement Learning Concepts

Summary

Reinforcement learning is the process of using an agent to directly learn in its environment. The agent is able to use the knowledge from the environment to either learn the *dynamics* of the environment, or the best way to navigate it. Reinforcement learning uses the MDP framework to build the fundamentals, and attempts to find the most efficient way to navigate the space of the environment, some of which could be orders of magnitude or infinite in size. Reinforcement learning is incredibly useful, and

For example, think of a game of Super Mario. Humans don't learn how to play by examining every possible action in every possible scenario… that's simply not possible. Humans are more complex, and learn much more intuitively. Reinforcement learning is trying to accomplish the same goal. Besides video games, there are many other applications of reinforcement learning being researched and utilized currently.



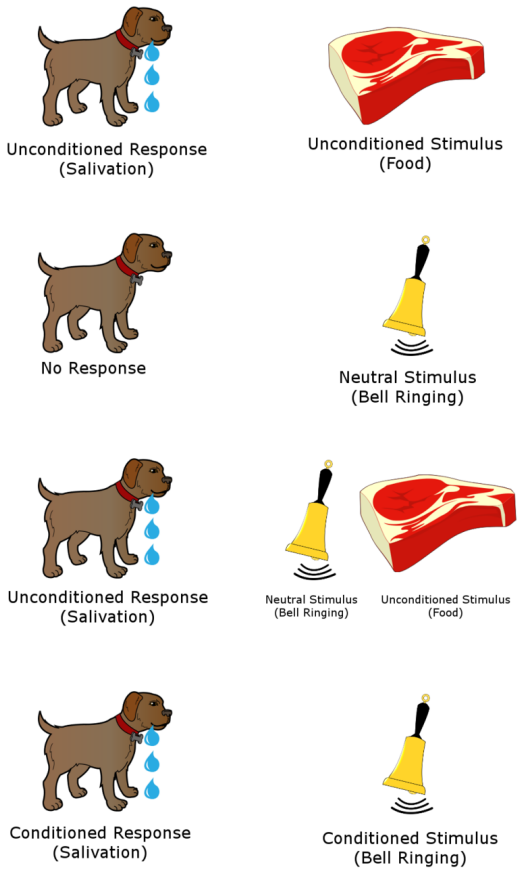
A non-exhaustive list of applications of (deep) reinforcement learning.

Before getting into the specifics of reinforcement learning, it's good to get a background on the history and inspiration for this field.

Background

Reinforcement learning, like other machine learning, is largely rooted in statistical and probability theory. RL is special, however, in how it was created through independent contributions from psychology and control theory.

Looking at the former, research into how learning is a consequence of reinforcement goes back to the 1850's, where learning was phrased as "groping and experiment" by Alexander Bain. The term "reinforcement" was coined in 1927 by Pavlov, who was studying conditioned reflexes on animals. This is the same Pavlov who's most memorable for the 1890's Pavlov's Dog experiment. In the '27 monograph, Pavlov introduces the idea of reinforcement learning as a temporal relationship between a stimulus and a response to that stimulus.



[ML3] *Classic conditioning on Pavlov's Dog.*

The other major contributing body to reinforcement learning is control theory. From Wikipedia, control theory is defined as “dealing with the control of dynamical systems in engineered processes and machines. The objective is to develop a control model for controlling such systems using a control action in an optimum manner without delay or overshoot and ensuring control stability.” In the 1950’s Richard Bellman extended previous works of Hamilton and Jacobi to formulate what has become known as the Bellman equations. The Bellman Equations typically follow the following format, and as you continue reading, you will recognize formulas that are Bellman equations.

$$V(x) = \max_{a \in \Gamma(x)} \{F(x, a) + \beta V(T(x, a))\}.$$

The methods used to solve Bellman Equations are known as dynamic programming. The process of dynamic programming is known to be the only way to solve stochastic optimal control problems. Since these control problems are directly correlated to reinforcement learning. The only major difference is that the dynamics of the system that is to be controlled needs to be completely known for dynamic programming. In reinforcement learning, this is not the case.

Like reinforcement learning, dynamic programming tends to approximate through iterations. Thus, dynamic programming is still considered to be a reinforcement learning algorithm.

The following diagram gives a general overview of how reinforcement learning is correlated to these other fields:

[ML4] *Correlation between RL, DP, and classical conditioning.*

During the rise of reinforcement learning, there was already another popular framework for creating optimal routines, genetic algorithms. In 1950, Alan Turing proposed a learning machine that would mimic evolution. Since then, other scientists have worked to create such systems, and over time genetic algorithms have evolved to become a great asset in machine learning. With a direct inspiration from nature, like reinforcement learning, these algorithms focus mainly on direct competition between many groups of potential solutions and using “natural selection” to determine which candidates would live on.

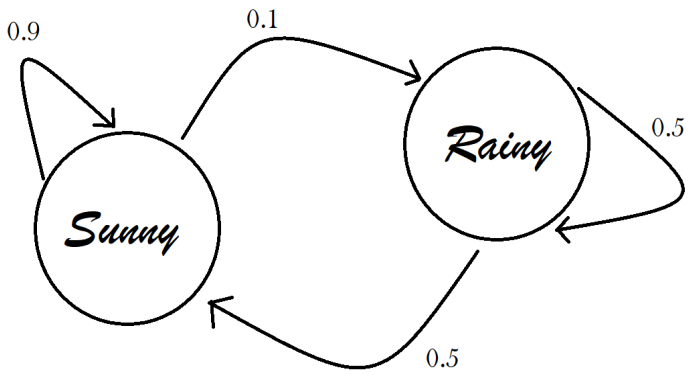
Genetic algorithms, though very much separate from reinforcement learning, are used in a lot of similar scenarios: learning games, optimizing searches in an environment, and much more. In recent times, genetic algorithms are starting to be combined with reinforcement learning to better optimize the algorithms. Though not new, reinforcement learning is expanding rapidly and still has many goals to achieve.

Markov Chains

A Markov chain is fundamentally a transition from a previous state to a future state, in which the transitions are stochastic and follow what is known as the *Markov property*. This property holds that, for any state X_n, the next state X_{n+1} is only dependent on the current state. The previous states hold no influence over what the next state is. This can expressed as such:

$$P\{X_{n+1}|X_0, X_1, \ldots, X_n\} = P\{X_{n+1}|X_n\}$$

This may sound like a very specific scenario that is not likely to occur, but let's see an example that would hold the Markov property. If you observed that it was raining today, you would know that the likelihood of it raining tomorrow would be higher. The current state (raining today) has a direct influence on the next state (raining tomorrow), without needing to know the probabilities of previous days.



A Markov chain for weather states.

The transitions are usually shown in the form of a *transition matrix*, which shows the probabilities of entering the next state from the previous one. Using the example above, the transition matrix would look like this:

$$\mathbb{P} = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

Markov Decision Process (MDP)

The concept of Markov chains can be extended to allow for actions to be taken and rewards to be collected from those actions. Thus, a MDP is defined to be a 4-tuple

$$(S, A, P_a, R_a)$$

with the states ***S*** [1], actions ***A***, probability of an action *a* leading to state *s'* from state *s* ***P_a(s,s')***, and a reward for *a* on *s* that led to *s'* ***R_a(s,s')***. In fact, as this is just an extension to Markov chains, having only one action per state and setting all rewards to 0 would reduce the MDP to a Markov chain.

How are actions chosen in this framework? This is done through the use of a policy π that determines the action $\pi(a, s)$ to be taken.

$$\pi : A \times S \to [0, 1]$$
$$\pi(a, s) = \Pr(a_t = a \mid s_t = s)$$

In MDPs, the goal is to make this policy optimal. One way to do this, which is later discussed in the Rewards section, is to maximize the *discounted sum of rewards*. A policy that accomplishes this is said to be optimal, and is denoted by π^*.

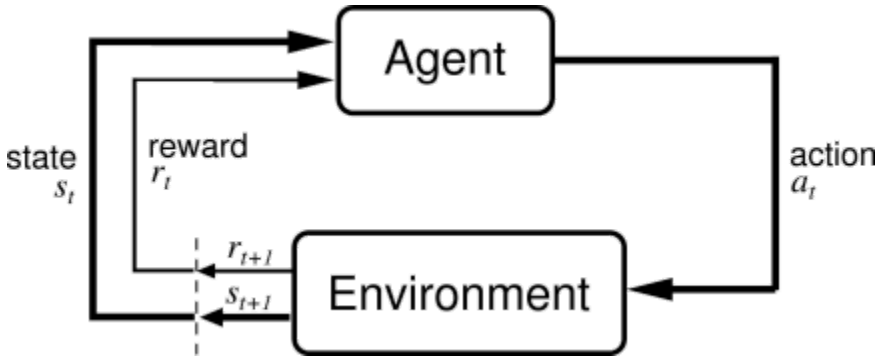
Reinforcement learning is an extension of the MDP framework, where either the policy and the rewards are unknown. If a state *s* was not assumed to be completely visible to π, then this would be reinforcement learning using the *Partially Observable Markov Decision Process* (POMDP). For Everglades, the MDP framework is sufficient.

[1]: Notice how the notation for states altered from the notion of Markov chains. This is because of the transition from probability theory to reinforcement learning, in which the notation of *S* makes more sense.

The Agent and Environment

In the MDP framework, the most widely used notion of the learner is referred to as the *agent*. In any particular state *S*, it will take some action *a* and receive a reward *r* for the action that it took. Over time, this agent uses the *rewards* given (or taken away) from it to learn the dynamics of the *environment* that it is in. Using this information, the agent hopefully learns and is able to improve the rewards given to it.

The environment is everything that is not the agent. It controls the state and uses actions from the agent to determine what the next state is[2]. It also determines the reward for any action taken and gives this to the agent to learn from.



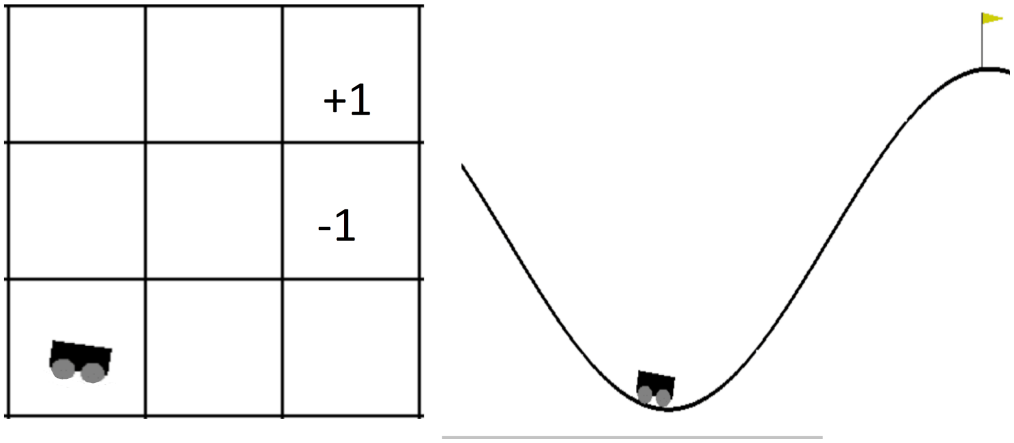
[BL4] *The interaction between an agent and the environment for a timestep t.*

The agent and the environment interact for a discrete amount of time in this circular fashion. Though simple, the framework provided by having these interactions under the MDP framework allow for many situations to be modeled well. Even continuous-time cases can be extended to from these notions.

[2]: Not all environments base the next state solely on the actions that the agent provides it. For an example of this, look at the frozen lake environment [ML9].

State and Observation Spaces

A *state space S* comprises all the possible states in an environment. The state space is visible to the agent, who is able to use that information to find the optimal policy. However, sometimes the state space is not completely visible to the policy. As mentioned previously, a reinforcement learning agent would use the POMDP framework to observe all parts of the state possible. Hence, the name for this space that the agent would look at is the *observation space*.



An example of mountain-car, which demonstrates the difference in discrete (left) and continuous space (right) In this case, the observation space is equivalent to the state space.

Action Spaces

An *action state A* comprises all the possible actions that the agent is able to do on a given state/observation space. As noted in the example above, action spaces are divided into two separate cases: discrete and continuous. In robotics, the action space is continuous, as movements of limbs or changes in momentum are made within $\mathbb{R}$. However, in most turn-based games, the action space would be discrete.

What if the valid actions an agent could take change over time? The action space can still comprise all actions possible throughout the entire episode, and the agent would need to decipher when to use valid actions, as invalid actions would be processed by the environment. Knowing this simplifies the notion of *A* and allows for easier generalizations regarding the action space.

Rewards

An agent's only goal is to maximize the total amount of reward that it is given to by the environment. Using this, it is possible to implicitly give the agent a goal to accomplish by giving rewards for doing "good" actions. Good actions are those that either get the agent to the goal or closer to it. To be more specific, at any timestep *t*, the agent would want to take an action that maximizes the *expected reward* G_t.

A simple way of finding the expected reward is by simply summing rewards expected at future timesteps:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

This, however, means that the agent equally takes into account all actions taken after an event. For some environments this works fine, but in others where $\mathcal{T}$(the final timestep) is very large or infinite in size, this method becomes infeasible. In reality, we would like to introduce temporal locality into the equation, and that is done by introducing the *discount rate* $\gamma \in [0, 1]$. Over time, the discount rate will lead to subsequent rewards being less impactful. Choosing a smaller rate means locality is more important, and vice versa.

This new way of calculating expected return can be formulated as such:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

This function is also known as an *objective function*, and is most commonly used to find the optimal policy π^*. Even though the latter formula is required in games that can last forever (Everglades is *not* one of those games), the concept of temporal locality in actions taken by the agent is something to consider in teaching the agent the importance of its moves.

Reward Shaping

With the current definition of reward, an agent would only receive a reward if the immediate goal was met (finished a course, won the game, etc.). Otherwise, the agent would receive some non-positive reward. This seems fine on the surface, an agent gets a reward for completing the objective, but this can be detrimental to the learning process.

For example, look at the Atari game Montezuma's Revenge. Though deceivingly simple in design, there are many aspects to the game that would be extremely difficult for an agent to complete. There are required keys that need to be grabbed, obstacles to jump over, and puzzles to solve. It is highly unlikely that the agent would be able to figure out the correct path and steps to take to reach the goal. The agent will lose every single time, and so it will never be rewarded for actions that actually progressed the agent towards the goal.

In this screen alone, there are many obstacles the agent has to overcome to get the key: ladders, ropes, jumps, fall damage, moving platforms, and skulls.

In other terms, the probability of an agent getting a positive reward $R$is equivalent to the probability that the agent surpasses different learning goals:

$$P(reward) = \lim_{k \to \infty} (\prod_{i=1}^{k} P(obstacle_i)) = 0$$

So how can the agent learn? One way to accomplish this is through the concept of *reward shaping.* Instead of passing *k* obstacles to receive a reward, the goal is to get intermediate rewards for completing objectives along the way. These rewards would be positive, but smaller than the reward given when reaching the goal. A formal definition of reward shaping can seen as adding the reward shaping function $R_f(s, a)$ to the original reward function $R(s, a)$in the MDP. Andrew Ng et al. [ML11] showed that under certain conditions, $R + R_f$is still optimal for the MDP.

For the Montezuma's Revenge example, reward shaping can be used to guide the agent towards the key; the agent can get rewarded for passing specific checkpoints designated by the designer. Since the agent will take less time to reach one of these checkpoints, it will move the agent towards the key.

There are some major caveats to reward shaping, however. The most noticeable is that it causes a loss in generality for the agent. Now that the agent is being guided to the goal, the original goal of a self-learning algorithm is somewhat lost (depending on how much reward shaping takes place). The second major disadvantage is that the agent may become less incentivized to explore when there is a more immediate reward to be gained. The agent's goal will not be to complete the original objective; instead, it will only try to gain the intermediate reward as much as possible. In Montezuma's Revenge, for example, suppose that an intermediate reward was made so that the agent got a reward whenever climbing a ladder. This may seem reasonable at first, the agent needs to climb at least one ladder to reach the goal, but this can quickly backfire. Instead of heading towards the objective, the agent now will climb up and down the ladder to get the immediate rewards.

Reward shaping can be of varying difficulty to include, depending on the environment and objectives, but it can be a valuable tool in helping agents that are having trouble reaching the original goal.

Value-Based Reinforcement Learning

In reinforcement learning, it can be difficult to create a function that determines what move an agent should make given a state and a value. That's where value-based learning can play a role in creating an agent that can make moves by predicting the rewards it can receive in the future. These value-based reinforcement learning agents typically reference a table based on states and possible actions and finding the max value among all the choices. After making an action the function updates the table based on rewards it receives helping the agent make a better decision in the future when the agent reaches the state in the future. This is the basis of how agents learn how to play a game with Q-Learning which is one of the simplest forms of value-based reinforcement learning.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

[BL4]

Q-Learning is a combination of how good a position is as well as a discount on how good future events can be. The learning rate determines how often we will replace the old value given our new value. Note that this does not take into account values whether they are higher or lower it simply is a chance whether a Q value is getting updated or not. How much the discount factor or learning rate is dependent on what the agent is being trained to do. A weakness of Q-Learning is that the more states and actions you have the bigger the table can be and the more episodes it will take for an agent to be able to assign. This is because in order to get an accurate sequence of good actions an agent must go down a sequence of moves multiple times. Furthermore, it also has to do this many times to fill in as many branches of the table as possible. Value-based learning is useful when there are few states and actions that have to be considered in order to train an agent in a reasonable amount of time but it can be useful in conjunction with a policy function to solve more complicated environments which will be discussed later.

Policy-Based Reinforcement Learning

A policy is a mapping from a states to probabilities of selecting each possible action. (Sutton 58) Policy-based agent takes a state and looks at the probability of making each particular action ‘a’. More specifically π is a probability distribution of $a \in A$ over states $s \in S$ denoted as π(a|s) where A and S are the action-space and state-space respectively. Typically, when you start training a particular policy function you set every action to have equal probability in every state. After every step, a policy function updates the probability of possible actions taken based on the expected reward received which is given by the value function. Policy-based reinforcement learning algorithms also use value functions to determine “how good” a state an agent is in or how good the value functions think an agent will perform in a given state this is known as the state value function. (Sutton 58) The other value function that Policy-based algorithms used is known as an action-value function (also known as Q function) which determines how good a particular action is given a state under π.
State-value Function:

State-value Function:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

Action-Value Function:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

[BL4]

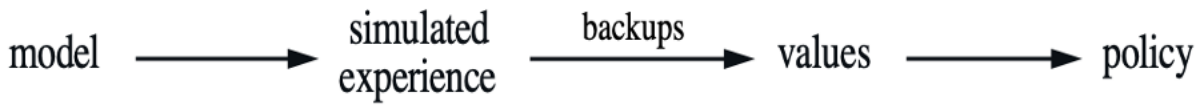
Model-Free Reinforcement Learning

In many cases, we do not know how to instruct our agents with any information about the environment. In these situations, we have our agent learn through trial-and-error by making random moves until they are able to get rewards and work to a solution. This is known as model-free reinforcement learning in which an agent is relying solely on learning and does not use any models which will be defined in the next section.

Because this type of reinforcement learning does not use a model the agent requires more episodes to be able to create a solution oftentimes this solution is not optimal without the use of some exploration.

Model-Based Reinforcement Learning

When creating a reinforcement learning agent sometimes we know how an environment will react to any action made given a state which can be useful for an agent to learn. Anything an agent can use to predict how an environment will respond is referred to as a model of the environment. Models are essentially used to simulate an experience of the environment. Given a starting state and action, a model produces a possible transition and a distribution model generates a possible transition weighted by their probabilities occurring. (Sutton 160) For example, say we have a model that considers rolling a pair of dice before making an action. The model could describe the distribution of the numbers that can be obtained by the sum of the two dice. In this model, it would give the higher probabilities are around 7 and reduces the father away a number is from 7. By weighing the outcome of rolling a pair of dice the model would advise the agent that getting a number between 6-8 is a lot higher than numbers like 1 or 12 and should make decisions according to those predictions. This process is known as planning which is a computational process that takes a model as input and produces or improves a policy for interacting with a modeled environment. (Sutton 160) There are two distinct approaches to planning, State-space planning, and plan-spaced planning. State-space planning refers to searching a state-space for an optimal-policy or optimal path to a goal. Plan-space planning is similar but searches through a space of plans to help an agent make decisions and adjusting policy accordingly. In state-space planning method involves simulating an experience then computes updates and backups from the experience with the help of a value function and uses that information to update the policy an agent is using.



Source: Reinforcement Learning by Richard S. Sutton and Andrew G. Barto page 160

Exploration vs Exploitation Dilemma

Unlike other forms of machine learning, reinforcement learning often does not have information about what is the correct action for a given state. The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions. [BL4] What this means in general an agent that interacts in the environment only can make decisions based on the information that it has and not based on being given an action to make. Because of this reinforcement learning agents have to deal with a choice at every time step. Do they decide that their knowledge of the best move and go down a path that has been taken before or do they try and take a detour and see if the alternative path yields a better result? This is known as the Exploitation vs Exploration problem where the agent decides to save time and make a decision based on what it knows or use that time and try to expand its knowledge in the hope of a better solution at any given timestep.

We will discuss the K-armed Bandit problem to explain concepts of the Exploitation vs Exploration problem. Imagine a slot machine but instead of one lever the slot machine has k-levers. The goal of the problem is to be able to maximize the reward by choosing the lever that has the greatest expected return. Each lever does not give a consistent return when pulled but does yield a mean value of return depending on the time step t of when it makes the action a_t. Because of this when deciding whether an action is “good” enough there must be a sizable enough sampling of that state to determine what the mean value return for that action is adequate. Let call this return the reward or R_t. Let defined this reward equation $q_*(a)$ as follows

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a] .$$

[BL4]

q_* (a) represents the state of the Q-value function to know what the best move is. Meaning it can accurately choose the best lever and get the maximum reward from the slot machine. Every state of the Q-value functions up until this state is denoted as q_t (a) meaning after t time steps of experience the function can only determine the best move based on reward gain in t time steps.

Let's consider the situation where the agent would like to preserve the current state of the q function. What would the reinforcement agent have to do? The reinforcement agent would have to look at all available actions and based on previous experience choose the action that yields the highest expected rewards. By doing this the q function is not acquiring any new knowledge of the environment. In the k-armed Bandit problem, let consider the player of the slot machine chooses lever number 3 100 times and receives a reward r. The player gains knowledge of what happens after 100 lever pulls of lever 3. If the player repeats these actions, it is likely the player does not receive another result. It is sometimes the case that at any given state of an environment there is only one good action and preserving the current state of the Q-value function is necessary to save time. This is known as "exploiting" an agent's current knowledge of the environment or choosing a greedy action.

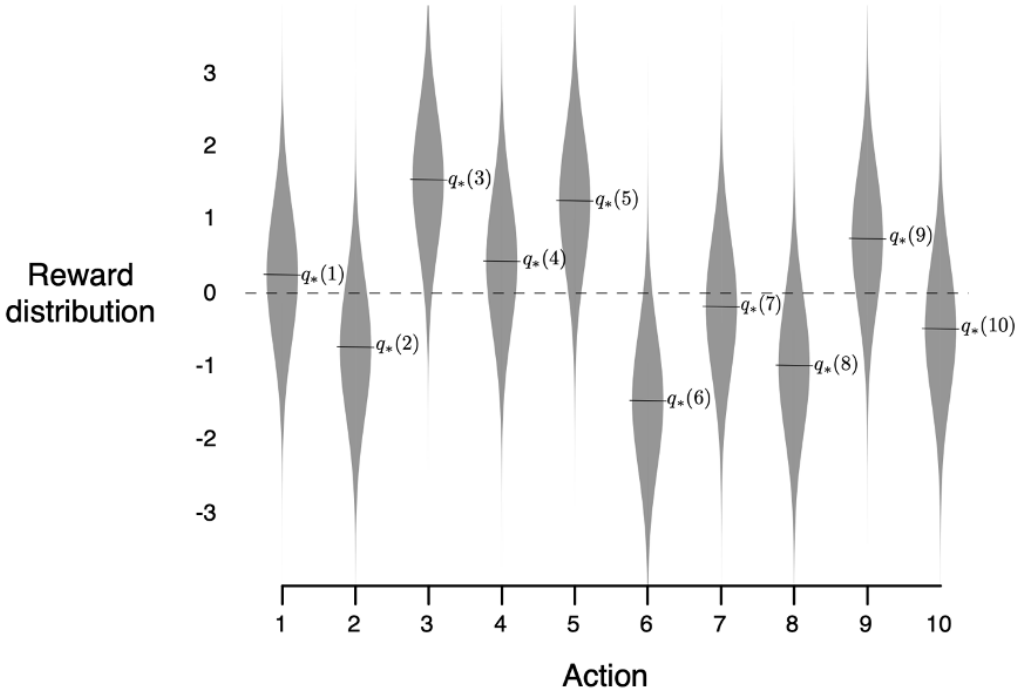
The contrary to this is sometimes there is no way to know what the best action is given the state of the environment. This uncertainty is the reason why exploration may be needed to properly understand the state of the environment. Later in the paper, there will be discussions about effective ways to explore new states. How to properly explore an environment is heavily dependent on the environment. The trade-off is an opportunity by choosing to explore options in a particular state that might not yield better results than the current max expected reward at that state. Let's defined the action value or Q-value function Q_t (a) as follows:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

[BL4]

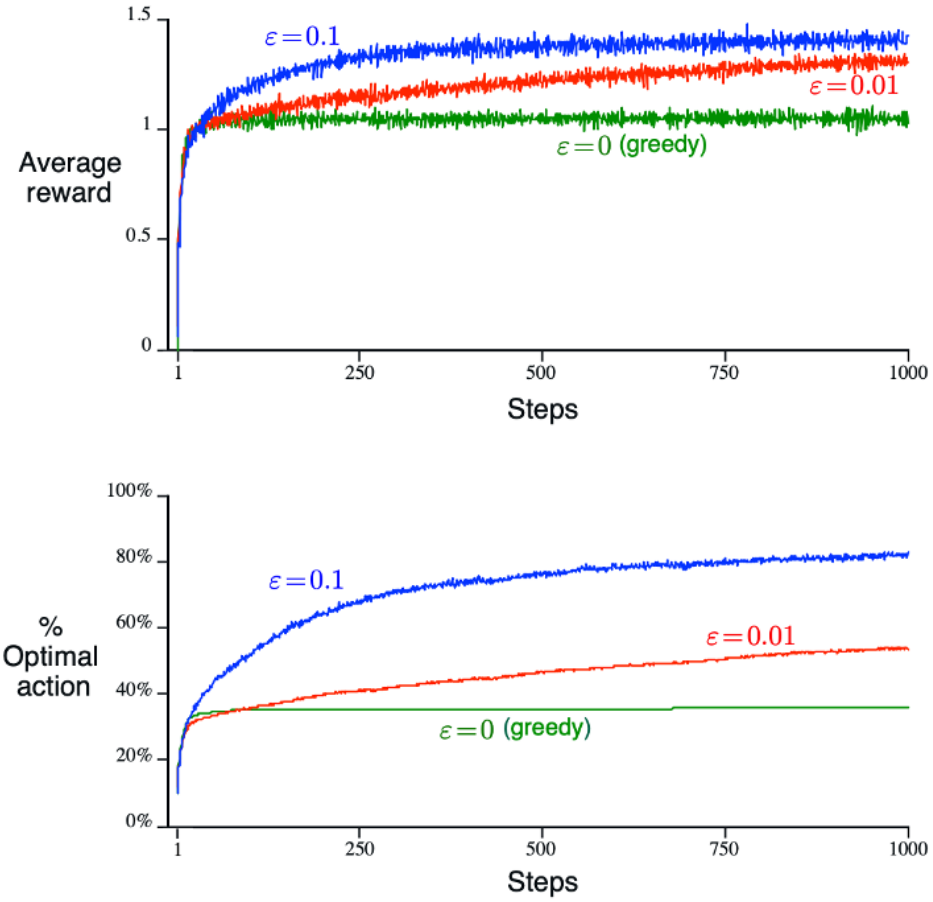
In the equation above you will see that this equation measures average reward over time. The sum of the rewards R_t for an action ‘a’ divided by the number of times that action has been made. It is obvious that as the denominator approaches infinity the formula converges to $q_*(a)$ or the optimal Q-value function. Even though this is true oftentimes it is impractical for reinforcement learning agents to sample a huge number of actions in given environments due to large states and action spaces.

In the textbook Reinforcement Learning an Introduction by Richard S. Sutton and Andrew G. Barto, the authors illustrated an experiment to show the different performances between greedy and epsilon greedy action value methods. Epsilon greedy method refers to making the most of the current knowledge the Q function has by choosing the best action it thinks it has 1-epsilon of the time where 0 < epsilon < 1. For example, let epsilon = 0.1 then the probability that an agent uses the Q_t (a) to make an action would be 0.9, and with a probability of 0.1 the agent will make a random action. This is the most basic way of exploration and it is not an optimal way of exploration.



[BL4]

Above is an example of a 10-armed bandit problem where our “slot machine” has 10 different levers which have a normal distribution around a mean value. The chart above is the result of what you would have as an expected value of a 10-armed bandit problem as the number of trials of each lever being pulled approaches infinity. In this 10-armed Bandit implementation, the agent is to choose levers 1000 times and then yield the reward at the end. This is considered to be one run. This experiment has had 2000 runs in order to adequately have the reinforcement agent gain experience. In many environments it takes time for the agent to understand what the expected return of an action in a given state of the environment. In this test, there are no states but nevertheless we will treat each time step as an essential timestep to the k-armed bandit problem.

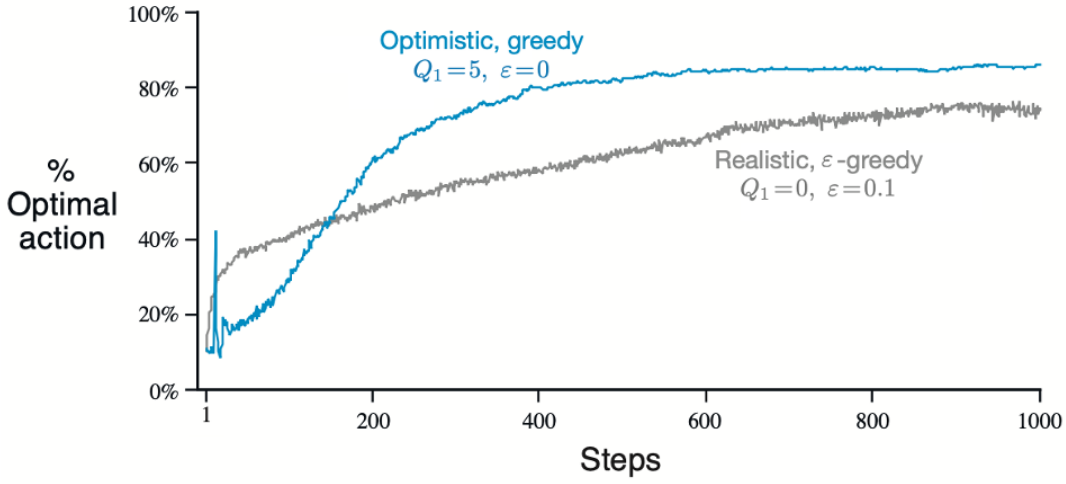


[BL4]

Above is a graph measuring the performance for average reward and optimal actions of the greedy, epsilon-greedy with epsilon=0.01 and epsilon-greedy method with epsilon = 0.1. Among the three methods, the greedy algorithm performed the worst because it was content with its current knowledge of the K-armed bandit problem. As previously mentioned, if an agent does not explore an environment it will automatically default to the first solution it gets. Meaning after it receives a positive reward for solving a problem it will often time default to the same path in every run. In the above graph, it appears that the greedy algorithm was able to find its plan within 100 time steps. As a result, it plateaus at about 35-40% optimal actions and consistently receives 1 reward. This is because it is happy with just 1 reward and does not try to improve its knowledge. The reason why it has such a low optimal action percentage is that the actions it has are just the actions that it so happened to use at the beginning to solve the problem it was posed with. In the epsilon-greedy method with epsilon=0.01, we see a dramatic improvement compared to the greedy method. Within 1000 time steps it can achieve about 50% optimal actions. The increase of the optimal action percentage would eventually get closer and closer to maximize as the number of times steps approaches infinity. Because the exploration only happens 1/100 of the time at any given time step it simply needs to explore certain states to converge to a more optimal Q-value function. The third method is also an epsilon-greedy method but instead of exploring 1/100 of the time, it explores 1/10 means it explores 10 times more often than the previous method allowing it to gather more information about the environment. As a result of this, it almost doubles the results of the other two methods by 2 at time step 250. Because it constantly is exploring 1/10th of the time it takes more time for the graph to plateau as it is more likely that this epsilon-greedy method chooses random actions in states that need to be explored. As a result, it achieves 80% optimal choices by 1000 time steps and has an average of about 1.5 rewards per timestep. The optimal reward for this set-up is 1.55 reward per timestep and given enough time this epsilon greedy method will eventually achieve this. In general, epsilon-greedy methods are better in helping reinforcement agents learn how to use the method optimally. In some situations, when the reward isn't very variated and simply solving the problem is good enough then greedy methods are sufficient to solve those types of environments.

Exploration method – Optimistic Initial Values

It is often the case that when training a reinforcement agent that some states of an environment never get explored because the reinforcement method determines that actions leading up to those states have low expected rewards. By giving action-states an optimistic initial value, you can guarantee exploration of every state and action given enough time steps. Let Q_0 be the initial state of the Q function and let it initialize all states and actions to have a reward r_{max} such that the reward yielded by the environment cannot be greater than r_{max}. In the first run through the states the agent will virtually be making random actions because every estimation of the state-action function will yield the same value r_{max}. In the second run of the environment, the agent will not take the same series of actions since the reward of those actions will be lower than the optimistic value we set for the other actions. Because of this, the reinforcement learning agent will constantly sample different actions in every state until every action in that state has been sampled at least once. By doing this we guarantee that every action in every state is explored because of the optimistic value set to every action. In general, this is not a very useful method of exploration when it comes to non-stationary environments, where states and actions are required to determine a Q value. But in non-stationary environments this proves to be more effective than epsilon methods. Below is a graph of the optimistic, greedy method vs an epsilon-greedy method for the k-armed problem.



[BL4]

As you would expect since this method focuses more on exploring instead of reaching optimal states. But because it adequately explores it reaches to a solution much faster than an epsilon-method given that the environment is stationary. It's because in many environments that the state of an environment plays a role in a decision that this method does not yield any useful results for most problems. One use for this method is instead of setting all the states to an optimistic value you can set this optimistic value to all actions at one state that you would like to know more about and get a good sample of data from it. This could be helpful if this is a very common state in a reinforcement learning environment and it would be time well spent to understand every branch of that state.

Exploration method - Upper Confidence Bound

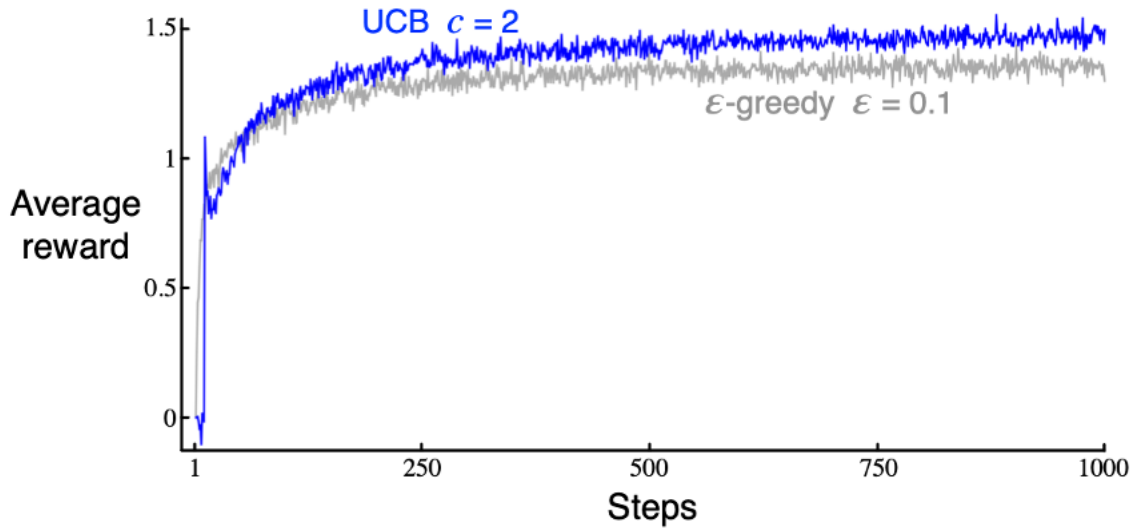
In the previous section, we discussed exploration as every time step there is a small probability that we explore by choosing a random action. Often it's the case that it is a huge waste of time to explore states that have been explored many times before and have no potential to be better. Upper Confidence Bounds is a method in which it prioritizes sampling actions that haven't been taken as frequently as others. Below is the equation that represents this action-method:

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

[BL4]

Action A_t is determined by the Q value of all the actions added by a constant c multiplied by the square root of the natural log of time step t divided by the Number of times that action has been made at time t. In this equation, the based decision of the action as always is dependent on the Q-value of the action at time t but in this equation, we add a number that larger for actions that haven't been taken very much and lower for actions that have. The numerator will constantly be increasing but the

denominator of the fraction stays the same as long as that action has not been taken. Even if an action is determined to yield a small reward and as a result, the action hasn't been taken eventually the numerator increases to a point where it negates the low reward return and takes the action anyway. The constant c > 0 represents how important exploration is for learning. The larger c is the more effects the ratios have and the less Q values at time t matter for calculation of the action A_t.



[BL4]

In the above graph, the test of Upper Bound Confidence method versus the better performing epsilon method where epsilon= 0.1. In this graph, we see that the Upper Bound Confidence method learns marginally faster than the epsilon method over 1000 time steps because of the Upper Bound Confidence for the 10-armed bandit problem. In general, the Upper-Bound Confidence method will perform better the bigger the action-space is given an adequate amount of time to train.

Exploration with NoisyNets

In traditional RL, exploration is important to discover for multiple hidden solutions that could be optimal. However, most of the heuristic exploration methods, despite being heavily researched, still strongly relied upon random exploration, or entropy regularization, to induce novel behaviors. To add more efficiency to exploration, the concept of Noisy Net was introduced [B5]

The NoisyNet differs from the conventional networks by directly using its weights as the exploration factor to discover new behaviors. A single change to the weight vector and induce a consistent, and very complex, state-dependent change in policy over multiple time steps. The perturbations are sampled from a noise distribution. The variance of the perturbation is a parameter that can be considered as the energy of the injected noise. These variance parameters are learned using gradients from the reinforcement learning loss function, alongside the other parameters of the agent. [B5]

NoisyNets are neural networks whose weights and biases are perturbed by a parametric function of the noise [B5]. More precisely, let $y = f_\theta(x)$ be a neural network parameterized by the vector of noisy parameters θ which takes the input x and outputs y.

Consider a linear layer of a neural network with p inputs and q outputs, represented by

$$y = wx + b$$

where $x \in R^p$ is the layer input, $w \in R^{q \times p}$ the weight matrix, and $b \in R^q$ the bias. The corresponding noisy linear layer is defined as:

$$y \overset{\text{def}}{=} (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + \sigma^b \odot \varepsilon^b,$$

The parameters $\mu^w \in R^{qxp}$, $\mu^b \in R^q$, $\sigma^w \in R^{qxp}$ and $\sigma^b \in R^q$ are learnable whereas $\varepsilon^w \in R^{qxp}$ and $\varepsilon^b \in R^q$ are noise random variables.

Now the origin paper explores two options: Independent Gaussian noise, which uses an independent Gaussian noise entry per weight and Factorized Gaussian noise, which uses an independent noise per each output and another independent noise per each

input. The main reason to use factorized Gaussian noise is to reduce the compute time of random number generation in our algorithms. This computational overhead is especially prohibitive in the case of single-thread agents such as DQN and Dueling. For this reason, we can use factorized noise for DQN and Dueling and independent noise for the distributed A3C [B5].

- Independent Gaussian noise: the noise applied to each weight and bias is independent, where each entry of the random matrix is drawn from a unit Gaussian distribution. This means that for each noisy linear layer, there are pq +q noise variables.

- Factorized Gaussian noise: by factoring each entry of the epsilon matrix, we can use p unit Gaussian variables for noise of the inputs and q unit Gaussian variables for noise of the outputs. Each entry can be written as:

$$\varepsilon_{i,j}^{w} = f(\varepsilon_i) f(\varepsilon_j),$$
$$\varepsilon_{j}^{b} = f(\varepsilon_j),$$

Since the loss of a noisy network is an expectation over the noise, the gradients are:

$$\nabla \bar{L}(\zeta) = \nabla \mathbb{E}\left[L(\theta)\right] = \mathbb{E}\left[\nabla_{\mu,\Sigma} L(\mu + \Sigma \odot \varepsilon)\right].$$

Then we use a Monte Carlo approximation to the above gradients, taking a single sample at each step of optimization:

$$\nabla \bar{L}(\zeta) \approx \nabla_{\mu,\Sigma} L(\mu + \Sigma \odot \xi).$$

Deep reinforcement learning with NoisyNets

NoisyNets have a wide application across different algorithms that require to use exploration. In the original NoisyNets paper, they demonstrate how to apply NoisyNets to some popular RL algorithms such as DQN, Dueling DQN, and A3C.

For DQN, a NoisyNet can be applied by first removing the greedy epsilon parameter. The policy will greedily optimize the action-value function. Secondly, the fully connected layers of the value network are parameterized as a noisy network, where the

parameters are drawn from the noisy network parameter distribution after every replay time step. For replay, the current noisy network parameter sample is held fixed across the batch. Since DQN and Dueling take one step of optimization for every action step, the noisy network parameters are re-sampled before every action.

When replacing the linear layers by the noisy network, the parameterized action-value function Q(x, a, e) can be seen as a random variable and the DQN loss becomes the NoisyNet-DQN loss:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma \max_{b \in A} Q(y, b, \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right]$$

Then we need to compute the unbiased estimate of the loss for each transition to the replay buffer, one instance of the target network and one instance of the noise in the target network and one instance of the online network. We generate these independent noises to avoid bias due to the correlation between the noise in the target network and the online network. For the action choice, the original paper generates another independent sample for the online network and act greedily with respect to the corresponding output action-value function.

Similarly, for the DQN Dueling, we have a loss function for the NoisyNet version of the NoisyNet-Dueling:

$$\bar{L}(\zeta) = \mathbb{E}\left[\mathbb{E}_{(x,a,r,y)\sim D}[r + \gamma Q(y, b^*(y), \varepsilon'; \zeta^-) - Q(x, a, \varepsilon; \zeta)]^2\right]$$
$$b^*(y) = \arg\max_{b \in \mathcal{A}} Q(y, b(y), \varepsilon''; \zeta).$$

For A3C, the NoisyNet version was also modified in a similar fashion to DQN. Firstly, the entropy bonus of the policy loss is removed [B5]. Secondly, the fully connected layers of the policy network are parameterized as a noisy network, and the chosen action is always drawn from the current policy. For this reason, an entropy bonus of the policy loss is often added to discourage updates leading to deterministic policies. However, when adding noisy weights to the network, sampling these parameters corresponds to choosing a different current policy which naturally favors exploration. Because of direct exploration in the policy space, the artificial entropy loss on the

policy can thus be omitted. New parameters of the policy network are sampled after each step of optimization, and since A3C uses n step returns, optimization occurs every n step. [B5]

$$\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}; \zeta, \varepsilon_i).$$

As A3C is an on-policy algorithm the gradients are unbiased when noise of the network is consistent for the whole roll-out. Consistency among action value functions Qˆ i is ensured by letting the noise be the same throughout each rollout. [B5]

On Vs. Off Policy Learning

On and off-policy methods arise as two possible solutions to the dilemma of exploring the entire state space. Since the agent may otherwise not find the optimal policy, there must be a way to allow for an agent to guarantee exploration.

The first method is to use on-policy control, which updates the policy that is being used by the agent to make decisions. The policies that use on-policy methods are *soft*, as $\pi(a|s) > 0$ for all $s \in S$ and $a \in A$. In other words, the policies that the agent uses to make decisions allow all actions to be made with a nonzero probability. πwill increasingly move towards being deterministic, but will never reach that point.

Off-policy control, on the other hand, uses two separate policies. The *target policy* is being improved, similar to on-policy. However, there also exists a *behavior policy*. The behavior policy is what controls the agent's actions, and the improvements update the target policy. Off-policy takes longer to converge and is more variant because of the multiple policies, but is a more generalized control method that can be applied to many different situations.

The best way to illustrate the differences between the two policies is through an example: take Q-Learning and SARSA (two reinforcement learning algorithms that update q-values to find the action-value function).

Q-learning uses the following to update q-values,

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \mathbf{max}_a Q(S',a) - Q(S,A)]$$

whereas SARSA uses

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$$.

Q-learning updates the q-values using the next state *s'* and a *greedy action a'*. This makes Q-learning off-policy, as the greedy action taking is the behavior policy and the q-values are the target policy. SARSA, on the other hand, uses the current q-values to decide the action that updates themselves. This makes SARSA on-policy.

Tabular Methods

All reinforcement learning problems have the optimal policy π^*, but only some are feasible to reach. For these cases, the state space can be modeled in a *tabular* format. This also would hold true for the action space. Though these methods will prove to be infeasible to use for Everglades, understanding that solutions that eventually reach π^* exist will help when attempting to approximate it for Everglades.

The most popular tabular methods that exist for finding optimal policies/ optimal value functions are also follows:

1. Monte Carlo Methods
2. Dynamic Programming
3. Temporal Difference Learning (ex. Q-learning and SARSA)

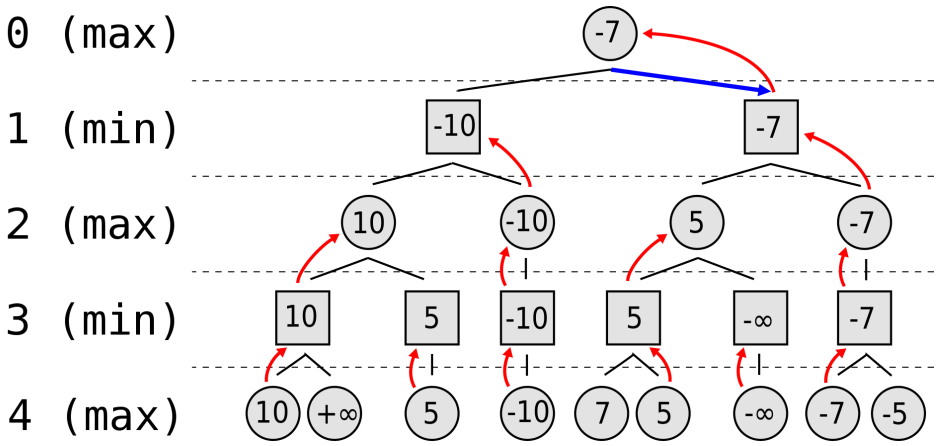
Each of these methods have their respective pros/cons, which will be discussed later in this paper. To first understand how these tabular methods work, it is best to first start off with an initial explanation of the minimax tree search algorithm and its optimization form known as Alpha-Beta.

Minimax

Minimax is a recursive, back-tracking algorithm that searches the game-tree to determine what is the optimal move for the agent. This algorithm requires there to be players fighting an opponent (usually named MAX and MIN). These two opponents

fight each other, and assume that the other is making the optimal move. The difference is that MAX will pick a maximized value, whereas MIN will pick a minimized value. The concept behind this structure is to create the scenario where the MIN agent needs to minimize the potential loss, and MAX will create the optimal gain. The game is going to be played in reverse, analyzing alternating moves between MIN and MAX.

As a backtracking algorithm, the best representation of the algorithm is as a tree, where you can visualize how the game is being played in reverse. An example of such a game is shown here:



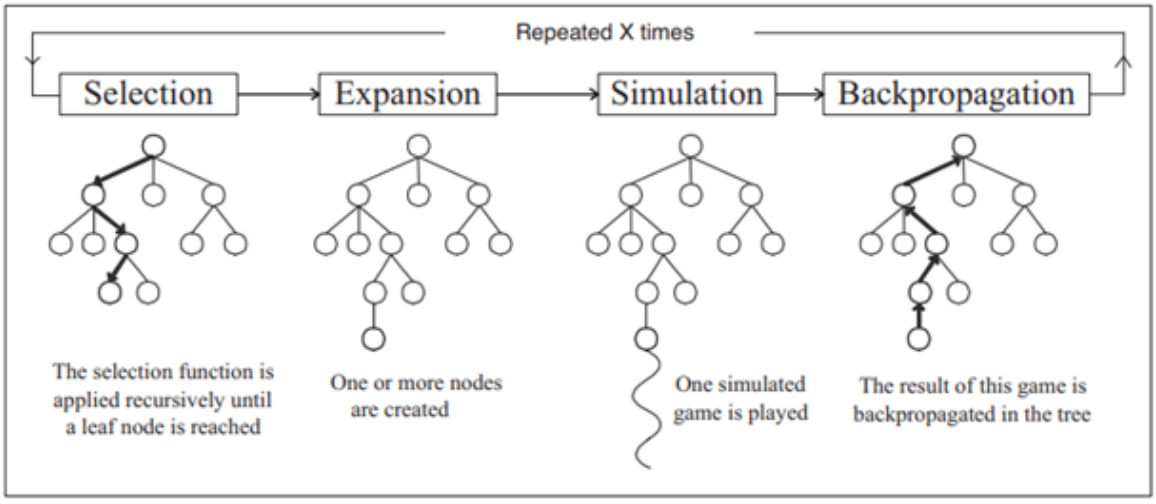
[ML22] *An optimal game-tree that was created using minimax. In this case, MAX was the first player to make a move.*

After recursively reaching the top of the tree (the initial state), the agent would then pick the maximum out of the children. This thus *minimizes* the *maximum* loss, creating an optimal game (hence minimax). This algorithm is optimal and complete, as it searches the entire state. As with tabular methods, however, this algorithm is subject to the massive time complexity that it takes to make a complete walk through the game-tree.

Alpha-Beta

The Alpha-Beta approach is an algorithm based on the minimax algorithm but does not attempt to reinvent the algorithm. Instead, it aims to optimize the current algorithm with a new understanding of how the minimax algorithm calculates the possible paths available. For this algorithm, it passes two new parameters to configure: Alpha and Beta. Alpha is used to represent the best value that the maximizer currently can guarantee on a given level of the game tree or above, and Beta is the placeholder for the best value that the minimizer currently can guarantee on a given level or above [J7]. This optimization path with the alpha and beta parameters help the current minimax algorithm in that it avoids looking at different branches of the game state tree that the algorithm already knows does not yield a better value than a previously seen branch. For example, with the previous example mentioned in the minimax tree search algorithm section, if there existed a third option to traverse in the game state tree but it has already been observed that one of the other two paths already contains a large value for either alpha or beta, then the alpha-beta algorithm ignores the third option because it has already determined that it would not be worth the computational time to investigate the additional third path since it will not yield a better result than what it has already seen in the other options [J8].

Monte-Carlo Tree Search (MCTS)



[J10] *Outline of a Monte-Carlo Tree Search.*

The Monte-Carlo Tree Search, also known as MCTS, was first introduced in 2006 by French Computer Scientist Rémi Coulom. The motivation for using this tree search was a simple objective: given a game state, can we select the most promising next move. To be able to complete this objective, we first should define our game to be a “finite two-person zero-sum sequential game [J2].” The term itself has several parts that can be analyzed as such:

- “finite” indicates that there only exists a finite amount of ways to interact between the players at any given time step. This also implies that there can not be an infinite amount of selections for one player at any given time step.
- “two-person” indicates that there are only two actors interacting with each other.
- “zero-sum” indicates that the two actors have opposite goals in the game that will intentionally intervene with each other.
- “sequential” indicates that the game is turn-based, meaning that one actor will take an action and then the other actor will respond after the previous actor has finished taking their action. This repeats throughout the entire interaction.
- “game” indicates the interaction between the two actors.

While it is nice that we can use this term to summarize our games for consideration when using this approach, the problems that game AI encounters with these games is that in a complex game-environment, adequate evaluation of the game state requires domain-dependent and complex tasks [J10]. When the agent plays throughout the game, ideally it should know a sufficient or perfect amount of information provided to it so that it can make the best decision possible. However, the games themselves may contain large action space or immense action-state pairings that it becomes impractical to analyze every single action or action-state pairing to identify which is the better solution for every single time step possible in the game.

We can address this issue by first taking our game and representing the game state as a game tree instead. At the root node, it can be initialized as the initial game state, where no actors have selected an action yet. The transition from any node to a children node, if it exists, can be considered a move that the actor selects. Terminal nodes, or leaf nodes, are the final states of the game, where a winner of the game can be declared, and no more actions can be made between the two actors. Being able to

traverse any segment of the game tree from the root node all the way towards a terminal node is considered a single game being played.

In the previous section of the paper where minimax was discussed, it was understood that the minimax algorithm will try to maximize our rewards in the game while understanding that the opponent will try to minimize it. This is a fairly straightforward task to accomplish, especially for games that have smaller game tree representations such as Tic-Tac-Toe. However, the main weakness that the minimax algorithm has is that for more in-depth games such as Chess and Go, the game tree representation becomes much larger and for the minimax algorithm to be effective, it needs to expand out the entire game tree, which is computationally expensive and leaves the agent to hang on making a decision for quite some time. A potential remedy to this situation would be to consider a threshold, in which we only analyze parts of the game tree and not the entire tree itself. This will allow us to observe the tree in distinct parts and maintain a level of effective computation. However, using a threshold may lead to a situation where the observed tree state may not have any terminal nodes to backpropagate information back to the root node, therefore not much analysis can be done from there. This is where MCTS becomes the effective approach for this situation.

MCTS is a probabilistic and heuristic-driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning [J4]. The algorithm itself has different parts that will be explored in this section, but the main difference that MCTS has in contrast to the previously mentioned minimax algorithm is that MCTS will traverse several branches of the game tree until it reaches a non-explored node, in which the algorithm will simulate the game from that node onwards an propagate information back. The algorithm terminates either after some set amount of time or after a certain amount of computation resources has been expended, which then becomes the “threshold” for the MCTS algorithm.

Monte-Carlo techniques have emerged in games such as Poker [J16] and Scrabble [J17], so the concept of using Monte-Carlo methods has existed for previous games before, especially games that may not contain perfect amounts of information. In a previous section of the paper, it was discussed that the Alpha-beta framework was used as an optimization path for the minimax algorithm so that the algorithm does not

compute paths of the tree that do not need evaluations since a better solution has already been discovered. The alpha-beta framework works well in games such as Chess and Checkers, two games that can be considered having complex game trees, but only because there exists an adequate evaluation function for both games and the game has a low branching factor [J10]. If such a feature existed for every possible game, then the alpha-beta framework would be a viable option to use to establish a policy network for the agent to make the best decisions possible. However, most classical and modern board games and video games lack these two distinctions. Additionally, as modern games become increasingly more complex and feature larger action spaces or larger action-state pairings, an evaluation function and low branching factors may become impractical for such games. The Monte-Carlo Tree Search has been used as an alternative to alpha-beta, but was mainly criticized because of its somewhat random approach to evaluating the game state, but ended up creating strong foundations for game programs, such as the different Go-programs that exist nowadays. Additionally, the Monte-Carlo Tree Search approach was used in DeepMind's AlphaGo agent in 2016. Current video games grow a more complex environment than classical and modern board games that require a thorough and adequate evaluation function, which has become a difficult task to accomplish; the Monte-Carlo Tree Search method sets out to resolve those issues.

In terms of the algorithm for the Monte-Carlo Tree search, the algorithm attempts to strike a balance in the exploration-exploitation trade off, where it will decide based on the value of the rewards it has obtained and how many different game scenarios it has traversed through [J4]. The exploration-exploitation trade off is a common occurrence for reinforcement learning algorithms because when the agent learns the proper actions in the environment it is in, we want the agent to be able to select the best choices possible at those possible states but at the same time we want the agent to explore the action space so that it does not converge at a local minima quickly and can lead to interesting discoveries about the game itself.

With that idea in mind, MCTS can be divided up into four phases: Selection, Expansion, Simulation, and Backpropagation. In the selection phase, the algorithm will traverse several branches of the game tree until it reaches a non-explored node. This allows the algorithm to play a part in exploring the action and state spaces. Once it

reaches the node that has not been explored yet, it goes into an expansion phase, where it analyzes the different options it has available to it and takes note of those available actions it has. Once it has noted the new children nodes of the unexplored node (assuming that the unexplored node is not a terminal node), then it begins the simulation phase, where it will mark the unexplored node as a root node in a recursive section of the algorithm and will play out the game using different actions as it traverses down the tree in a singular game. After it has reached a terminal node, it marks the nodes and transitions it has taken and then enters the backpropagation phase, where the information about the actions, states, and rewards are propagated back up to the root node that was originally discovered as an unexplored node. The information at that node based on the simulation that the algorithm has traversed is then updated and the algorithm continues throughout the game tree.

The part that has a bit of obscurity in terms of the Monte-Carlo Tree Search algorithm is within the simulation phase, where the algorithm simply selects actions to make as traversals to different nodes of the game tree until it has reached a terminal node. The question that is raised is how exactly the algorithm decides which actions to take as it moves down the tree. The move selection in the simulation is done through a function known as a rollout policy function, which will consume a game state and produce the next action [J2]. The rollout policy function in the MCTS algorithm is typically custom-made depending on the environment the agent or algorithm is interacting with, but to allow for fast simulations, the default rollout policy function is uniformly random. This then means that the actions that are selected for a default MCTS algorithm are random.

Additionally, some added notes about the Monte-Carlo Tree Search algorithm is its consideration of a node to be visited and fully explored, as that can have a bit of obscurity to the meaning. For MCTS, a visited node is a node where a simulation occurred starting at that node. If all of the children nodes of that node have been visited, then the node is converted to a fully expanded node. Note that this does not mean that a node that has been reached by means of a simulation are considered visited. For example, if we have a node n with children nodes n¬1, n2, and n3 and node n has not been explored yet, then we mark n as visited and we play out a simulation from that node and we consider it as a root node for this simulation. However, if our

rollout policy states that we should take the transition to n1 as our action, when we visit that children node, n1 is not considered a visited node because it was selected by a rollout policy which could be defaulted to the random selection policy, thus making our selection of the node random and the policy decided to not explore n2 or n3.

After a game simulation has been played out, the information about the action transitions, the states visited, and the rewards accumulated are backpropagated back to the starting node. These pieces of information are used to decide the next node to visit in terms of priority. While we would like to visit all of the possible game states and their corresponding game simulations, the algorithm's threshold may restrict us to only playing a select few simulations, so it is important to have priority over which nodes to visit in a short amount of time and resources. For that, we have a function known as Upper Confidence Bound for Trees, also known as UCT. The function is a tool that lets us choose the next node among visited nodes to traverse through for MCTS and can be described using the given formula:

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log\big(N(v)\big)}{N(v_i)}}$$

In the UCT function, it takes in two parameters v and vi, which both represent the node and child node, respectively. Q(vi) describes the total simulation rewards at the given child node and N(vi) represents the total number of visits at the child node. Providing a ratio between the two values returns a value that can best represent the exploitation factor at the child node, since a high yielding value means that the child node displays a good amount of rewards for the amount of times it has been visited. However, just using this exploitation factor is flawed because it will have the agent quickly converging to a local minimum that may be adequate in some game states but not an effective solution for the overall game itself. Additionally, the ratio can be skewed to misrepresent the true status of the child node. Suppose the child node's ground truth of rewards is very minimal and may have one or two actions that yield rewards. In a scenario where the simulation allows us to travel to the child node and randomly choose the path that does contain a reward just after that initial value, the exploitation factor will display a large and favorable number for the agent to be encouraged to travel to the child node continuously, even though it is not the most optimal path to be

traveling in the game tree. To prevent this problem from occurring, a second term is included named the exploration factor that increases in size if the node has not been visited that much. This skews the result of the UCT function so that even if a child node has a high yielding value of rewards, if another node has not been visited that much, the exploration factor may become dominant in the function and can yield a larger value for UCT.

The overall advantages of using MCTS is due to its simplicity in terms of its implementation, since the main premise of the algorithm deals with recursive tree descent and keeping track of information below each node, much like how an AVL tree establishes itself as a data structure. Additionally, the algorithm is known as a heuristic algorithm, meaning that it can operate effectively without any knowledge in the particular domain except for the rules and end conditions of the game. The MCTS algorithm can also be saved in any intermediate step during the algorithm and can be used in future cases of the game, and also supports asymmetric expansion, in the off chance that the game tends to have a more in-depth game tree towards using one set of actions than another.

While the advantages of the MCTS being portable and malleable to a vast majority of different classical and modern board games and videogames, it does come with some setbacks. For instance, while the algorithm does support asymmetric expansion of the game tree, it still requires a large amount of memory to be allocated after rapid tree growth expansion due to the algorithm needing to keep track of which nodes have been visited, which nodes have been fully expanded, which of the children have been visited and how often they have been visited, and the amount of rewards accumulated for each node. This can lead to a major overhead problem since there are those key details to maintain while also upholding a tree structure to represent the game state, which contains a vast amount of pointers to different nodes with expansive trees, especially in the cases that the games have complex rules and large action spaces to consider. Additionally, due to the constraints on the nodes it can visit in a short period of time, some node traversals might lead to failure. This means that some simulations may end abruptly, or some parts of the tree may not be fully explored due to the limited threshold that is placed on the algorithm. Lastly, much like with any given action policy

network, MCTS requires a large number of iterations to be considered effective at deciding the best actions to take at each game state.

Temporal Difference Learning

To explain the approach, consider the following analogy [J1, BL4]: Imagine you were driving to work and had a GPS in your car. The GPS would provide you with an estimation about the arrival time to your destination based on available statistics at the given time. However, as you drive, you experience traffic jams, road construction, and other means of which the travel will slow down or speed up and thus the estimation arrival time changes. At each point of the trip, there was always an update on the estimation until you finally arrived at the destination.

Now, suppose the same scenario occurred, but the difference is that the GPS would not give an updated estimation as the travel went on. Instead, when you arrive at your destination, the GPS provided a detailed summary of the trip, such as when there was a traffic jam and how much time was lost because of the slowdown. The question then becomes how effective the two representations of the information to the driver are, and the answer is that it depends. Sometimes the ending summary may be more informative and is completely accurate since it knows the details of the trip than the former method, whereas the former method gives more helpful feedback to the driver, even if it may not be accurate all of the time.

This analogy is used to give an intuitive understanding of how Monte-Carlo Tree Search differs from Temporal Difference Learning. The Monte-Carlo Tree Search algorithm is only effective when it simulates a procedure and then provides the appropriate information about the simulation afterwards, much like how the GPS provides information after the driving trip, giving a detailed summary of what the trip provided. In contrast, Temporal Difference Learning provides more immediate results that the agent can use in their evaluation, much like how the GPS provides estimations of the trip while the car is still in motion.

For Temporal Difference Learning, the name itself is derived from the use of changes in predictions over successive time steps to drive the learning process [J3]. Temporal difference learning algorithms are often used in reinforcement learning to predict a

measure of the total amount of rewards expected over the future, much like how the Monte-Carlo Tree Search algorithm looks into different simulations of the game in order to find the best path to take in the game tree to yield the most amount of rewards. While MCTS and Temporal Difference Learning does have their differences, that is not to say that the two algorithms are mutually exclusive. As a matter of fact, Temporal Difference is made up of both Monte-Carlo Tree Search and Dynamic Programming. Like the MCTS algorithm, Temporal Difference Learning learns from raw experience without a model of the environment's dynamics [BL4]. Additionally, much like Dynamic Programming, Temporal Difference Learning methods update estimates based in part on other learned estimates without waiting for a final outcome [BL4]. For the rest of the explanation of Temporal Difference Learning, the notation "TD(0)" is used to show that the algorithm looks one step ahead, which is a particular case of TD(n).

As a refresher, the Monte-Carlo Tree Search algorithm plays an entire episode until it reaches a terminal node in the game state tree and then calculates the statistical values, such as the number of states the episode has visited and the number of rewards it has accumulated for its simulations. MCTS updates its values as such [BL4]:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

based on the state it currently finds itself in at time step t, MCTS uses the value function and adds a second term, which is the difference between actual return value Gt and the value function at the current state, all of which are multiplied by the alpha term, which is also known as the learning rate or the step-size parameter. The issue with this evaluation function is that it has to wait for a simulation of an episode before it can be able to calculate the value function at the state. Therefore, the value function is left hanging with no values to calculate until the simulations have been concluded and it leads the agent to stall when it comes to deciding the best course of actions. This makes the Monte-Carlo method's evaluation not as helpful since it becomes a slow process to calculate and it only takes into consideration just the states and not any of the actions that are made.

Another hindrance that is made is that for Dynamic Programming evaluations of the game states, there becomes an issue of how it deals with the exploration-exploitation

dilemma of the tree search. All Dynamic Programming does initially is that it initializes all of the states to have minimal rewards and then calculates based on surrounding states until no further improvement can be observed. In order for Dynamic Programming to be effective in handling reward optimization, it still needs to explore different states of the game tree in order to have the most accurate telling of the best rewards located in the tree.

What makes Temporal Difference Learning different from the other methodologies is that TD(0) looks only at the first state ahead in order to make its calculations, which makes it different from Monte-Carlo Tree Search because it does not need to wait for an entire simulation of an episode to play out before making a calculation of the value function and is different from Dynamic Programming's method since it doesn't need to exhaustively look at several other states in order to make the calculations. The Temporal Difference Learning algorithm uses a different formula of the value function V(St), where it only looks one time-step ahead. The following formula is used to calculate the value function for Temporal Difference Learning [BL4]:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Much like with the previous Monte-Carlo Tree Search value function, it takes in the current state but it also uses a future state to calculate the value function along with the rewards it can expect in the next timestep, represented as R_{t+1}. The gamma factor is the discount factor used to curb the results of future rewards and the alpha term is used as the learning rate or step-size factor.

The way Temporal Difference was first understood came from an exploration of a different problem stated, which was simply wanting to predict future rewards, especially if it depends on a gamma factor. The way rewards were thought of was in the original form [J3]:

$$Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots = \Sigma_{i=1}^{\infty} \gamma^{i-1} y_{t+i}$$

This represents a summation of rewards for each time step into the future and each one having a magnitude of gamma applied to it. The issue with this algorithm is when gamma approaches the value of 1, in which case it becomes a summation of rewards, but may lead to the agent getting into a cyclic loop in the game tree if it notices that it can get a large sum of rewards by repeating the same processes and avoids the overall end goal of the game, which is a noticeable problem when it comes to reward shaping. This is often known as the infinite-horizon discounted prediction problem [J3].

Another issue is that there needs to be a probability function that balances the relationship between the signals x and y, which corresponds to the actions made at time-step t and the rewards the agent earns by making those rewards, respectively. If no balance exists between the two signals, then the evaluation function is simplified to being a randomized function, which does not benefit the agent's learning process. The best way to find this relationship between the two signals is by assuming a Markov chain relationship.

From here, the simplest form of the TD(0) algorithm is one where the value of gamma was set to 0, therefore the distribution of rewards in the previous function is simply equal to y_{t+1} since none of the other terms exist in the function once gamma is equal to 0. This makes it so that the reward function is primarily concerned with only the immediate future reward, much like how the concept of TD(0) works, in that it only looks at one time-step ahead. From there, the error function is denoted in a supervised way by checking to see what the expected outcome was at state S_t and follows this equation format [J3]:

$$p_t + \alpha(y_{t+1} - p_t) = (1 - \alpha)p_t + \alpha y_{t+1}$$

where alpha is used as the learning rate once more. If alpha is set to 1, then the entry is simply set to y_{t+1}, which denotes the future reward expectation in the next time-step ahead. However, if alpha was set at any other learning rate, the entry becomes an approach to an expected prediction.

While it is helpful to have a function that can train our agent in a supervised manner, the issue is that it is based on the premise that the gamma term is equal to 0, therefore nullifying a majority of the reward expectation in different time steps, which is not practical in a real demonstration of an agent training and learning in an environment. So, the question becomes, “What if the gamma value wasn’t 0?” If this were the case, then we use the original reward function Y_t as stated from before, but it requires us to know the rewards at every single time step in the future of the game, which ultimately mirrors a Monte-Carlo Tree Search. However, we can bypass that issue with the following evaluation of the reward function [J3]:

$$\begin{aligned} Y_t &= y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots \\ &= y_{t+1} + \gamma[y_{t+2} + \gamma y_{t+3} + \gamma^2 y_{t+4} + \cdots] \\ &= y_{t+1} + \gamma Y_{t+1} \end{aligned}$$

This new understanding of the reward function conserves the one-step-ahead prediction that TD(0) establishes and simply applies the gathered calculations into further calculations. With the new understanding of the reward function, the new Temporal Difference error calculation can be written as [J3]:

$$\delta_{t+1} = y_{t+1} + \gamma P_t(x_{t+1}) - P_t(x_t)$$

where the error function now takes into consideration the next move the agent should play to maximize the rewards gained in the future. From the evaluations made, the simplest TD algorithm with a lookup table is as follows [J3]:

$$P_{t+1}(x) = \begin{cases} P_t(x) + \alpha\, \delta_{t+1} & \text{if } x = x_t \\ P_t(x) & \text{otherwise,} \end{cases}$$

There are concerns about the algorithm’s accuracy since the predictions are not necessarily accurate, but over the course of the algorithm’s operation, later predictions tend to become more accurate due to there being more “training” that occurs on the algorithm’s part and most of the calculations done at a broad-based level have already

been completed initially towards the beginning of the game. Much like with Dynamic Programming and the Monte-Carlo Tree Search algorithms, the estimations made from the Temporal Difference Learning algorithm will not be accurate solely on one episode alone; several iterations of the same iteration will improve the accuracy of the estimation.

The policy control that is implemented in Temporal Difference Learning is seen in approaches such as SARSA and Q-Learning. With SARSA, it resolves the issue of considering future actions and states by choosing a future action a' based on an epsilon-greedy method at state s' and once it arrives at the new state s', it will perform the expected actions [J1]. With Q-Learning, a lookup table is established with the action and states used as the dimensions of the table and the method attempts to maximize the Q-values based on the information that is provided from the table.

Temporal Difference Learning has shown to be a better solution than Dynamic Programming since it does not require a model of the environment nor reward and probability distributions in order for the method to be effective in any environment. Additionally, Temporal Difference learning is advantageous over MCTS since it does not have to wait for calculations to be made in a separate thread in order to proceed with its own calculations. While it does have those advantages over MCTS, both have their strengths and weaknesses. While TD(0) is known to converge to an optimal value based on any fixed policy much like MCTS, it raises the question: Which gets to the value function first: MCTS or TD(0) [BL4]? Currently, it still remains to be an open question, and some speculate that it may not even be the most appropriate question to ask for now.

SARSA

For the introductory approach, it is first considered to follow a Generalized Policy iteration (GPI) using Temporal Difference Learning methods for the evaluation or prediction part [BL4]. Additionally, the algorithm must decide the trade-offs between exploration and exploitation and the approaches of on-policy and off-policy. However, for this approach, instead of just isolating the states themselves and identifying the values of the states based on transitions much like how simple MCTS or TD(0) would resolve the value function, the method will look at transitions between pairings of states

and actions and calculate values based off of those transitions from pairings. Whenever the algorithm transitions to a nonterminal state of the game tree, we use the regular Q-value formula as described from before [BL4]:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

Whenever the algorithm moves to a terminal state, Q(S', A') is defined as zero, since there are no additional states or actions to choose from at a terminal state. This rule is used for the quintuple (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}) for every transition from one action-state pairing to the next action-state pairing. This is the basis of the name "SARSA".

SARSA is known to be a straightforward design implementation. Based on the on-policy method that SARSA uses, it updates q_π based on the policy π and converges towards epsilon, which is used to describe the exploration-exploitation factor of the agent's learning process, respective of the algorithm's epsilon-greedy approach. The overall convergence of SARSA is dependent on the Q-values it determines, which means that how epsilon is implemented and used, the optimal policy and action-value function will converge as long as all state-action pairs are visited an infinite number of times until the greedy policy converges.

Q-Learning

Q-learning, or quality learning, is attempting to learn the action value function. It does so by updating a matrix (or table) of q-values, which represent the quality of an action per a given state.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

[ML14] *A Q-table showing the value of actions (columns) on states (rows).*

The formula for updating such q-values is given in the On vs. Off Policy section, but here we will give a more conceptual overlook for what q-learning is. Q-learning initializes all the q-values arbitrarily (this does not affect convergence). It will use a greedy policy to choose actions, and will update the q-values based on the result of the greedy policy.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until s is terminal

[BL4] *Pseudo-code for implementing Q-learning.*

This algorithm will converge to the correct action-value function, as the target policy (remember, q-learning is an off policy learning algorithm), still allows for exploring the entire state space.

What are some issues for this algorithm? Well, this algorithm, being a tabular solution method, requires a finite state space and action space. Even then, an action or state space that is too large leads to tables that are far too large to deal with on hardware. Being able to approximate the q-values, rather than storing all of them, would be much nicer. Thus, we now will transition into approximating the policy.

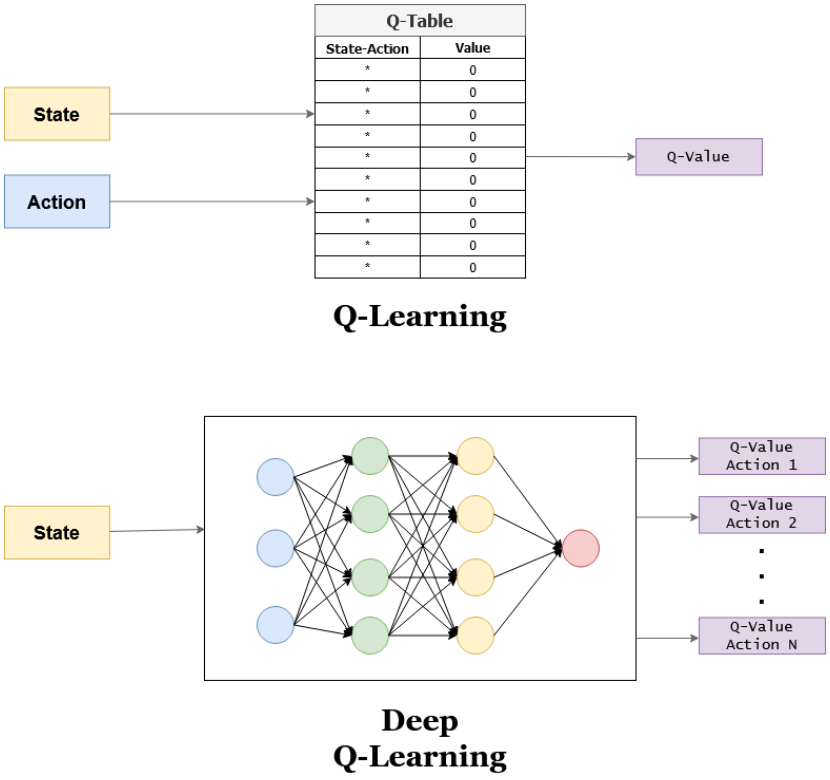
Approximation Methods: Neural Networks

What about the case where approximating the policy is the only feasible way? Nonlinear function approximations are handled very well by neural networks, and so utilizing them in this context seems to be the solution. For the most part, this is the case. In general, reinforcement learning that utilizes deep neural networks is known as

deep reinforcement learning. Let's look at the case of a *deep Q-network* that utilizes a NN to approximate the value function.

Deep Q-Learning

One of the most notable uses of using neural networks in reinforcement learning was when DeepMind decided to approximate the Q-value function using a neural network.



The difference in structure between Q-learning and Deep Q-learning.

This network, rather than outputting a single q-value, outputs all the q-values for the entire action-space, which greatly increases computation. The computation is saved on the loss function that the network builds off of. The loss function is based on the Bellman equation for q-learning which was presented previously.

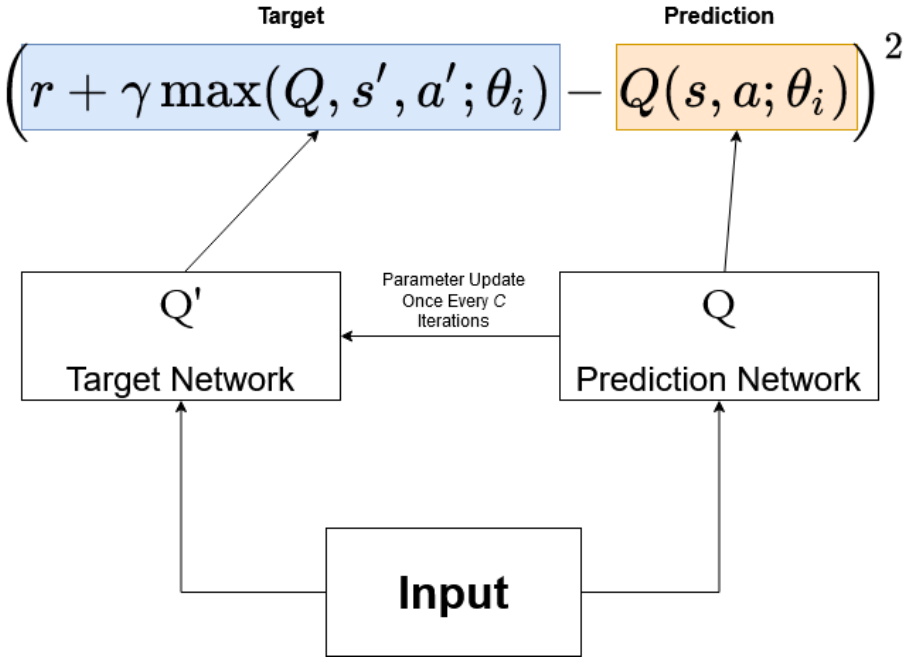
$$L = \frac{1}{N}\sum_{i}^{N}(r_i + \gamma \max_a Q_{\phi'}(s_{i+1}, a) - Q_{\phi}(s_i, a_i))^2.$$

This loss function is essentially a mean-squared error that is computed using the network and a greedy policy. This is then used to optimize the network through SGD.

This method alone seems fine at first, but when you look deeper, you realize that the function trying to be approximated by the neural network is one that alters over time. Neural networks are not designed to handle a continuously changing function, and thus there needs to be something changed to combat this. More formally, if the target function y_i and $Q(s_{i+1}, a)$ are updated in the same direction on each iteration, then divergence will occur.

$$y_i = r_i + \gamma \max_a Q_\phi(s_{i+1}, a)$$

To combat this issue, the *target network* is introduced. This allows for stability to be reintroduced to the network by not using the same network for the selection of actions, rather the target network is a frozen copy of the original network that is updated on every *C* iterations.



The target and the prediction (original) network cooperate to stabilize learning.

Multi-Agent Learning

Some tasks in reinforcement learning are not always feasible to accomplish with one agent. Imagine a game where multiple entities must work together to accomplish a goal (like Everglades with the groups of drones and goal of capturing the enemy base). If this situation was modified to where no entity knew the location of the other ones (a partially observable MDP), it would not make sense to use one agent to account for every entity. Rather, an agent for each of the entities would make much more sense, as they would be able to independently from one another. This is an example of the most basic form of a multi-agent system – one in which the agents do not communicate and cannot coordinate tasks. Usually, these agents respond to one-another and can coordinate tasks.

A model for multi-agent learning is by expanding the MDP to account for multiple agents and their changes in environments. This multi-agent model is known as a *stochastic game*. A general overview overview of this expansion is as follows: for each agent *i* there is a set of observations O_i that is sent to the agent. These observations are individual and do not reflect the entire environment state. Each of these agents will be able to choose from a set of actions A_i to take given an observation. The transition functions for this POMDP will be as follows,as shown by Ryan Lowe at al. [ML18]:

$$T: S_t \times A_1 \times A_2 \times \ldots \times A_i \times \ldots \times A_n = S_{t+1}$$

Each O_i, A_i does not necessarily need to be of the same size. Take, for example, drones in the Everglades game. If there were different classes of drones that had different sight ranges, then the observation space for each drone would be of different sizes. If a drone had different movements, i.e. being able to move between a larger amount of nodes, then the action space would differ. Thus, it is appropriate to say that these agents are fundamentally different from one another, and thus the required space for each agent is also different. The agents' rewards will also be individual, rather than cumulative between the agents. Therefore, each agent *i* will receive each reward as follows:

$$r_i: S \times A_i \rightarrow \mathbb{R},$$

and a return the set of observations for each agent:

$$o_i : S \rightarrow O_i.$$

The agent's individual goals remain the same, attempting to maximize the expected return for itself.

Typically, the agent structure in multi-agent systems is formulated as one of three groups: fully cooperative, fully competitive, or mixed. In a fully cooperative game, the reward function for each of the agents is the same: $r_i = r_j$. This means that any positive or negative reward impacts both agents equally; they would be trying to accomplish the same goal. If we were to use multi-agent learning in Everglades, then this would be the reward function that the agents would go by, as any of the groups of drones have the same goal as all the others. On the other hand, if $r_i = - r_j$ for some pair of agents,they are said to be fully competitive. They would be attempting to accomplish the opposite goal as the other - like two sports teams pitted against one another. If neither of the two cases above hold, then the agents are said to be mixed. In this scenario, either this is ambiguity in how the agents are interacting, or their goals are inconsequential to one-another.

Multi-agent reinforcement learning has the ability to allow autonomous control between different actors within the model, which can breed various complex strategies. However, this architecture has some major limitations that would make it less favorable for use in the Everglades environment. The most impactful one is the exponential increase in computational complexity. The agents would need to interact directly with the other agent to coordinate strategy, but this makes it more difficult to train. Another issue arises from viewing the environment from the perspective of a single agent. As other agents are independently updating their policies, the environment is non-stationary. This violates one of the Markov assumptions that were made to guarantee convergence with some reinforcement learning algorithms.

More than likely, Everglades would not need a multi-agent system, as the environment is sufficiently small and a single optimized agent should be able to control all of the troops. However, in future scaling of the game, multi-agent learning may become more reasonable.

Introduction to OpenAI Gym

OpenAI is a company focused on artificial intelligence, providing a significant amount of frameworks, APIs, and other tools to assist in their development. Our team is focused on a specific project of theirs known as OpenAI Gym: a toolkit for developing and comparing reinforcement learning algorithms. Gym supports the teaching of agents in an environment, allowing them to learn over time. What is learned by these agents can be anything, ranging from simple to complex, depending entirely on what is expected of them in a given environment. Most commonly, the agents will be trained with the goal of optimally solving any and all problems or tasks presented to them.

Gym focuses on a branch of machine learning known as reinforcement learning, where an agent receives rewards based on what is deemed as good behavior and over time learns to perform in a way that maximizes the reward obtained. The behavior of the agent is classified by performing an action, and then obtaining an observation and reward from the environment resulting from that action. The environment consists of an action space, describing the format of valid actions, and an observation space, describing the format of valid observations.

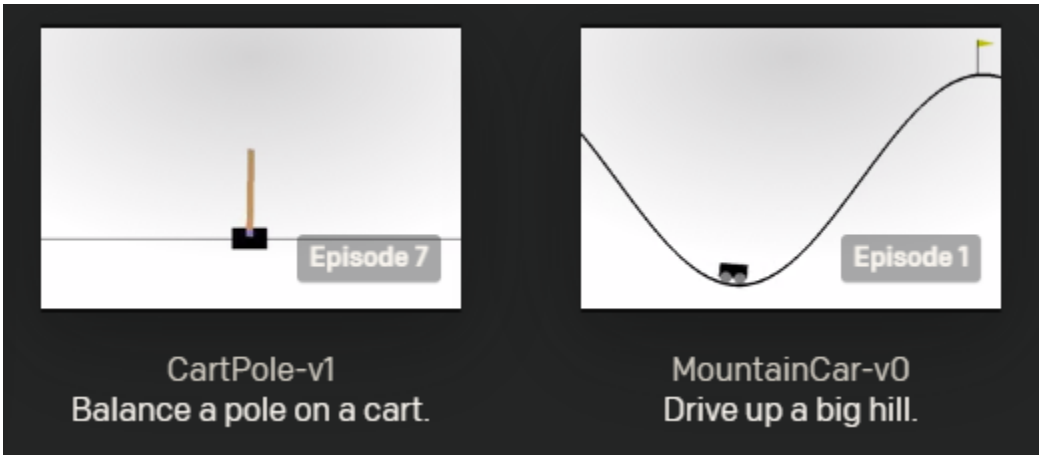
Reinforcement learning is an important field of machine learning as it encompasses an expansive field of problems and their potential solutions. With Gym, reinforcement learning models can be designed to not only solve an exact problem, but also a generalized problem that may be present in other, similar environments.

[A18] *An example of many different Atari games compatible with OpenAI Gym Retro.*

The ability to create models that can be implemented in multiple environments is extremely beneficial to the machine learning field. Not only does it allow for improvements in computational intelligence for those interested in improving a particular algorithm, it also generalizes that improvement, making it available for anyone interested in advancing their models that find themselves in a similar situation. This allows for an extreme broadening of the potential application of this technology, able to be used in practically anything involving a sequence of decisions. This applies to such topics as controlling robotic movement, making financial business decisions, or learning to play video games.

A First Implementation

Gym is both a well-documented and widely-used software, making it very easy to find information about how to create an agent, even if completely new to the idea of reinforcement learning. Following the general tutorial found in the Gym documentation, each member of our team worked on an algorithm that implements a simple provided environment, such as a cart-pole simulation that balances a pole on a cart, or a car simulation that controls a driveable car stuck in a valley between two mountains. The goal for each implementation was to successfully “solve” the challenge present in the environment that was used, such as keeping the pole balanced on the cart, or successfully driving the car out of the valley. This was a good way to introduce the methods used in Gym that would help us train our agents in the future.



[A19] *Visuals from two of the “Classic control” environments provided by Gym.*

Notation

Probabilities:
$P(x)$→ probability of x
$\mathbb{E}(x)$→ expected value of x

Markov decision processes (MDPs):
$s,\ s'$→ states
S→ set of all states
a→ actions
A→ set of all actions
r→ rewards
R→ set of all rewards
π→ policy
π^*→ optimal policy
t→ timestep
G_t, F_t→ expected (discount) reward at time t

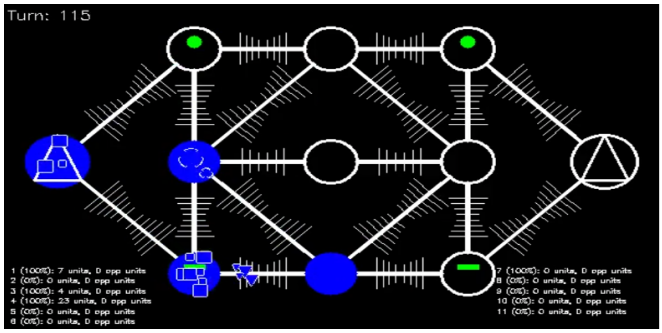
Partially observable Markov decision processes (POMDPs):
$o,\ o'$→ observations
O→ set of all observations

Specific Functions:
$Q(S, A)$→ quality function
$Q'(S, A)$→ target quality function
$UCT(v_i, v)$→ upper confidence bound

Everglades Game Environment

Gym Environment

The environment of Everglades analyzes how many nodes are currently being occupied by what team and with how much control. Everglades is a game where every turn the environment sends observations and rewards based on what has happened in the Everglades Environment. At the end of every game if an agent wins the game it receives +1 reward, 0 for a draw, and -1 when it loses. This means an agent is only getting an indication of how well it is doing after it has completed a game. This means that the agent can only start learning after going through many episodes. In this case, an episode is defined to be when a game of Everglades concludes in a win, draw, or loss for the reinforcement learning agent. The Everglades environment also returns a Boolean done parameter that indicates when the game has concluded. The three conditions that would conclude the game are when one of the teams captures the other team's base, when 150 turns have passed, and when all one hundred units from one side have been eliminated. The Everglades Gym Environment currently does not offer an info parameter that could give useful diagnostic information such as probabilities of the last state of the environment after an action is made. This information is typically not used directly by the agent but can help create a model. The Everglades environment offers two forms of observations for the agent to analyze. One being a 105-index array that describes the state of the environment which will be discussed in detail in the next section and the other being a visual representation of the environment that is shown below:



[BL3]

Lastly, the Everglades environment offers an action-space for the agent to work with. In everglades, all actions can be fed to the env.step() method including moves that are illegal at the time the action is being made. Details about the action space will be discussed in greater detail later in this section.

Observation-Space

As described previously, an observation space is a set of parameters that are given by the environment to the agent in order for it to make decisions. The observation space describes the state of the environment.

In the Everglades game environment, the observation space is given to describe the current turn number, the state of the 11 nodes in the environment, and the 12 groups belonging to the receiver of the observation.

Observation index [0] describes the turn number of the game.

Observation index [1-44] represents information about each of the 11 nodes in the environment where each node has 4 indexes. The indexes are broken into 1-4, 5-8, 9-12, 13-16, 17-20, 21-24, 25-28, 29-32, 33-36, 37-40, 41-44. The first index of each one of the groups indicates that it is a fortress with a 1 and 0 if it is not. The second index represents if it is a watchtower with a 1 and a 0 if it is not. The third index shows how much control a team has over a tower. This number ranges from -100 to 100. The number in the range represents percentage control over a node. A negative number indicating that the enemy has control and a positive number indicating that the receiver of the observation has control. The fourth and last index of the group represents how many enemy bots are at that particular node and the range is 0-100.

Observation index [45-104] describes the state of the 12 groups belonging to the receiver of the observation. Each group has 5 indexes worth of information. The first index represents the node that the group currently is located in. The second index in the group represents the type of group it is 0 for controller, 1 for striker, and 2 for tank bot. The third index represents the average group health ranging from 0-100. The fourth indicates if the group is currently in transit with a 1 and if it is not in transit it will be a 0. The fifth index indicates how many units are remaining in the group ranging from 0-100.

Although the limit is 100 in the current implementation of the game the hugest group is group H with 12 tanks.

Action-Space

The action-space is the options an agent has been given as possible moves that it could make at any particular state. In Everglades, an agent is allowed to move 7 groups of battle bots from one node to another node that is adjacent to that node in the environment. Once a group has been moved to another node it will be in transit for a number of turns depending on what node it is moving to and from what node it is coming from. The actions that the environment will request for will be a 7x2 array indicating what groups go to what nodes. For example, if one of the indexes is [3,2] in the 7x2 array then it wants to move group 3 to node 2. If illegal moves are made the agent will lose its move for that particular index in the action array. This can be very useful for an agent if it decides to only want to move fewer than 7 groups at any given point. Illegal moves can be considered any node a group cannot be moved to because it is not an adjacent node to the node that group is currently located as well as moving nodes that are currently in transit.

Action Space Size

In the Everglades environment, there are many actions to consider when the neural network has to approximate the best moves to make at any given state. As mention before each player has 12 groups of battle bots. If we assume that we can move each group of bots anyway we like not considering only illegal moves, we know that we can move 7 of our 12 groups to 11 different nodes. Because illegal moves result in the agent not moving a battle bot group, we know can consider these as cases for us to not move agents. So that means in any given state the agent is allowed to move 1-7 groups. One consideration when setting up an output layer for a Deep Q-Network is naively set the output layer to be of the size of the number of possible moves and simply choose the biggest seven positive Q-values and make that the decision for an agent to make. This approach is not sufficient because what is being done here is considering 7 separate decisions instead of 1. It is the case that moving 7 battle bot

groups without consideration of the other groups being moved can yield a sub-optimal result.

This can be resolved by organizing our output layer of the neural network to be the size of every possible combination of group movements. As a reminder the equation for finding the number of k combinations of n objects is

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Also, assuming that the battle bots are positioned on nodes that have the most legal movements then every battle bot has up to 5 adjacent nodes that they can move to. As a result, the agent's action space size is equivalent to

$$Legal\ Action\ Space\ Size \leq 5\sum_{k=1}^{7}\binom{12}{k} = 5[\binom{12}{1} + \binom{12}{2} + \cdots + \binom{12}{7}]$$

$$= 5\sum_{k=1}^{7}\frac{12!}{k!\,(12-k)!} = 5 * 3302 = 16510$$

Due to this large action space, it is very difficult to implement a basic Q-Learning implementation to effectively train an agent. As result, our reinforcement agent needs to use many alternative tools to fully explore all the possible actions at any given time step t. By using Neural networks to approximate a Q-value table, agents can feasibly be able to effectively learn the environment.

Policy Gradient Algorithms

REINFORCE

REINFORCE is a family of algorithms that originally came as a direct consequence of wanting to approximate the policy function directly (rather than implicitly using a greedy algorithm on a learned value function). The policy function has weights that are updated according to learned experience in the form of a learned value function. Monte Carlo methods allow these algorithms to estimate the expected value by using samples of learning.

At the time, this algorithm was revolutionary for allowing stochastic environments to be learned and guarantee convergence to a local maxima, but has some major limitations. First, using policy gradients alone means that the policy weights are adjusted very slowly. Thus, training time is increased drastically. These algorithms also have the issue of being *sample inefficient*, as the learned experience is lost on each iteration of the algorithm.

Below is the steps necessary to run one iteration of the REINFORCE algorithm [ML17]:

The REINFORCE Algorithm

1. Sample trajectories $\{\tau_i\}_{i=1}^{N} from \pi_\theta(a_t \mid s_t)$ by running the policy.
2. Set $\nabla_\theta J(\theta) = \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i))(\sum_t r(s_t^i, a_t^i))$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Actor Critic

In the Actor-Critic algorithm, there are two components that it relies on to be successful as an Actor and a Critic. An actor makes actions based on a policy function where the critic analyzes the state of the environment (value-based) after an action has been made by the actor. After the critic has done its analysis of the action it updates its value-based table with a new Q-value and then gives the Actor information about how good or bad the action that it made was, this is shown in the figure below. By doing so this helps the Actor's policy function converge quicker. Over time, the actor will make better decisions which will intern will have the critic get better at analyzing and vice versa. Another advantage in doing this is unlike traditional policy-based algorithms it updates its policy function after every action instead of the end of the episode which also greatly reduces training time.

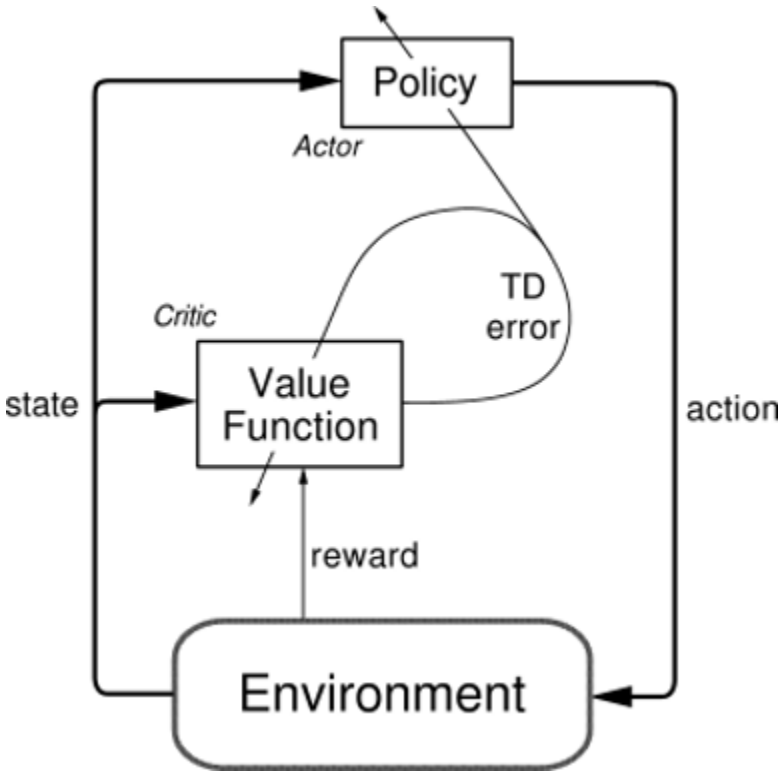


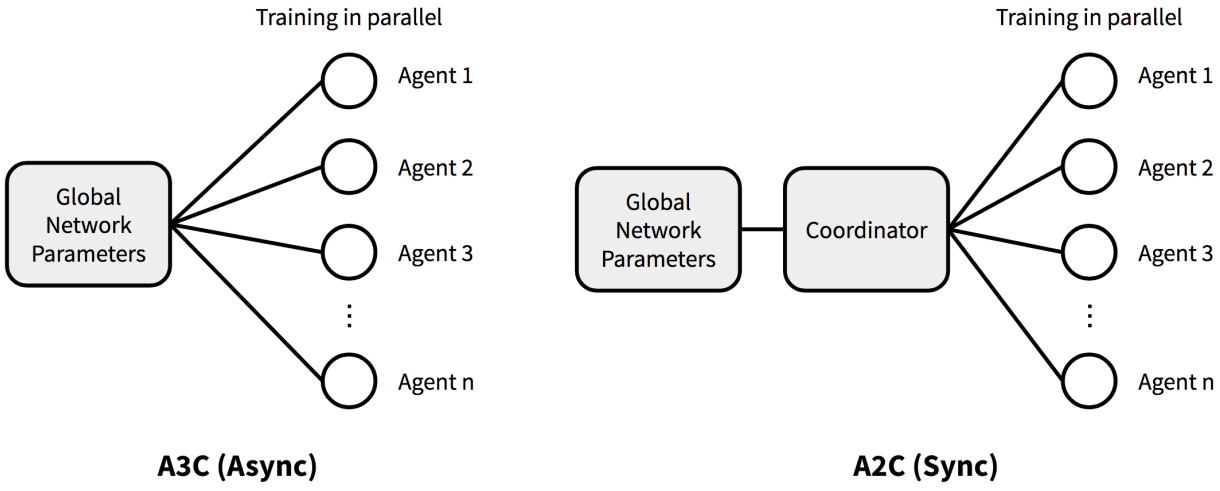
Figure 1

Advantage Actor-Critic (A2C)

As a synchronous, deterministic version of A3C, the gradient updates are performed together which keeps the training more cohesive and potentially allows for a faster convergence. To do this, a coordinator waits for the completion of all parallel actors, and only then updates the global parameters, allowing all actors to perform the next iteration with the same updated policy.

Part of the on-policy family, this algorithm permits updating the network only immediately after collecting experiences from the environment, as the data used was generated from the same policy which is being updated, and is thus discarded after each update.

A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C. [A14]



Asynchronous Advantage Actor-Critic (A3C)

With a particular focus on parallel training, this algorithm employs critics to learn the value function while multiple actors are trained in parallel, sometimes becoming synchronized with global parameters from time to time. Hence, A3C is designed to work well for parallel training. It is described by the authors as “a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous

gradient descent for optimization of deep neural network controllers.” (Volodymyr Mnih, et al.) [A15].

An outline of the algorithm [A14] is given here:

1. We have global parameters, θ and w; similar thread-specific parameters, θ' and w'.
2. Initialize the time step $t = 1$
3. While $T \leq T_{\text{MAX}}$:
    1. Reset gradient: $\mathrm{d}\theta = 0$ and $\mathrm{d}w = 0$.
    2. Synchronize thread-specific parameters with global ones: $\theta' = \theta$ and $w' = w$.
    3. t_{start} = t and sample a starting state s_t.
    4. While (s_t != TERMINAL) and $t - t_{\text{start}} \leq t_{\text{max}}$:
        1. Pick the action $A_t \sim \pi_{\theta'}(A_t|S_t)$ and receive a new reward R_t and a new state s_{t+1}.
        2. Update $t = t + 1$ and $T = T + 1$
    5. Initialize the variable that holds the return estimation
    $$R = \begin{cases} 0 & \text{if } s_t \text{ is TERMINAL} \\ V_{w'}(s_t) & \text{otherwise} \end{cases}$$
    6. For $i = t - 1, \ldots, t_{\text{start}}$:
        1. $R \leftarrow \gamma R + R_i$; here R is a MC measure of G_i.
        2. Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|s_i)(R - V_{w'}(s_i))$;
        Accumulate gradients w.r.t. w': $dw \leftarrow dw + 2(R - V_{w'}(s_i))\nabla_{w'}(R - V_{w'}(s_i))$.
    7. Update asynchronously θ using $\mathrm{d}\theta$, and w using $\mathrm{d}w$.

Soft Actor-Critic (SAC)

Another variant in the Actor-Critic family is the Soft Actor Critic algorithm. However, unlike its siblings such as A2C and A3C, which use the actor-critic framework to train and make improvement of the agent using an on-policy approach, which can lead to a problem with sampling complexity and hyperparameter sensitivity. SAC was designed to address this problem by combine with some off-policy characteristics such as using a replay buffer similar to Q-Learning to increase sampling efficiency by learning using past experience, introducing maximizing entropy method to maximize expected return, while still keep the robustness of an actor-critic framework by using policy network (actor) and value function network (critic) separately.

In tradition RL, we have this notation below that represents for maximizing the expected return sum of reward:

$$\sum_t E_{(s_t,a_t)\sim p_\pi}\left[r(s_t,a_t)\right]$$

The new SAC framework, which introduce new entropy maximizer component, will change the mathematical notation above to:

$$\sum_t E_{(s_t,a_t)\sim p_\pi}\left[r(s_t,a_t)+\alpha H(\pi(\cdot|s_t))\right]$$

The temperature parameter a determines the relative importance of the entropy term against the reward, and thus controls the stochasticity of the optimal policy. The maximum entropy objective differs from the standard maximum expected reward objective used in conventional reinforcement learning, though the conventional objective can be recovered in the limit as a → 0. [B4]

This new framework has several conceptual and practical advantages compared to the traditional RL models. First, the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues [B4]. Second, the policy can capture multiple modes of new optimal behavior. In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions [B4]. Lastly, this framework can also extend to infinite horizon problems by introducing a discount factor to ensure that the sum of expected rewards and entropies is finite.

Derivation of Soft Policy Iteration

The soft actor critic begins by deriving from soft policy iteration, a general algorithm for learning optimal maximum entropy framework policies that alternates between policy evaluation and policy improvement in the maximum entropy framework. In the original paper, the derivation is based on the tabular setting, to enable theoretical analysis and

convergence guarantees [B4]. There are 2 steps involved in this derivation. First is the policy evaluation step of soft policy iteration, which will compute the value of a policy according to the maximum entropy objective [B4].

$$\mathcal{T}^{\pi} Q(\mathbf{s}_t, \mathbf{a}_t) \triangleq r(\mathbf{s}_t, \mathbf{a}_t) + \gamma\, \mathbb{E}_{\mathbf{s}_{t+1} \sim p}\left[V(\mathbf{s}_{t+1})\right],$$

where

$$V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi}\left[Q(\mathbf{s}_t, \mathbf{a}_t) - \log \pi(\mathbf{a}_t|\mathbf{s}_t)\right]$$

The equation below is the soft state value equation. We can obtain the soft value function for any policy pi by repeatedly applying the equation for T pi.

The second step is the policy improvement step, updating the policy towards the exponential of the new Q-function. This choice of update can be guaranteed to result in an improved policy in terms of its soft value. Since in practice we prefer policies that are tractable, the original paper also adds additional restrictions with some set of policies II. In other words, in the policy improvement step, for each state, the policy will be update according to

$$\pi_{\text{new}} = \arg\min_{\pi' \in \Pi} \mathrm{D}_{\mathrm{KL}}\left(\pi'(\cdot|\mathbf{s}_t) \,\middle\|\, \frac{\exp\left(Q^{\pi_{\text{old}}}(\mathbf{s}_t, \cdot)\right)}{Z^{\pi_{\text{old}}}(\mathbf{s}_t)}\right).$$

The partition function normalizes the distribution, and while it is intractable in general, it does not contribute to the gradient with respect to the new policy and can thus be ignored.

The full soft policy iteration algorithm alternates between the soft policy evaluation and the soft policy improvement steps, and it will converge to the optimal maximum entropy policy among the new policies.

Next, we will approximate the algorithm for the continuous domains, as mentioned in the original paper, where we need to rely on a function approximator to represent the Q-values as running the alternation between the two steps would be computationally too expensive. The approximator gives rise to a new practical algorithm called the soft actor-critic. [B4]

Soft Actor-Critic

As discussed above, after deriving from the soft policy iteration, we need an approximator to represent the Q values, as alternating between the two steps of evaluating the soft policy and policy improvement would be too expensive. This approximator will be for both the Q-function and the policy.

The state value function approximates the soft value. This quantity can be estimated from a single action sample from the current policy without introducing a bias, but in practice, including a separate function approximator for the soft value can stabilize training and is convenient to train simultaneously with the other networks. The soft value function is trained to minimize the squared residual error. [B4]

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right] \right)^2 \right]$$

Where D is the distribution of previously sampled states and actions in a replay buffer. The gradient of the equation above can be estimated with an unbiased estimator

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(\mathbf{s}_t) \left(V_\psi(\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) + \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) \right),$$

Where the actions are sampled according to the current policy instead of the replay buffer. The soft Q-function parameters can be trained to minimize the soft Bellman residual.

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right],$$

with the equation:

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right],$$

Which can be optimized with stochastic gradients

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right).$$

Finally, the policy parameter can be learned by directly minimizing the expected KL-divergence D_{KL} from the new policies equation:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_\phi(\cdot|\mathbf{s}_t) \,\middle\|\, \frac{\exp\left(Q_\theta(\mathbf{s}_t, \cdot)\right)}{Z_\theta(\mathbf{s}_t)} \right) \right].$$

The next step is to minimize the value of the equation above. Since the target density is the Q-function, which is represented by a neural network and can be differentiated, it is convenient to apply the reparameterization trick, resulting in a lower variance estimator. [B4] The equation thus can be rewrite as:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[\log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t)|\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t)) \right],$$

Then, we approximate the gradient for the equation above with:

$$\begin{aligned} \hat{\nabla}_\phi J_\pi(\phi) = {} & \nabla_\phi \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) \\ & + (\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)) \nabla_\phi f_\phi(\epsilon_t; \mathbf{s}_t), \end{aligned}$$

The detailed pseudo code for every step of a soft actor critic from the original paper [B4] is included below:

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

Everglades and SAC

One distinctive characteristic of the Everglades environment is that it has a discrete observational space instead of a continuous space, which the SAC algorithm was designed to work with. However, during the initial implementation using a Deep Q Learning algorithm, there are a lot of current limitations of the current agent can be improved upon by incorporate some of the SAC characteristics such as improved replay buffer, a different way to implement actor-critic networks, and using the concept of maximizing entropy to improve the expect return rate. All of those reasons make the SAC very compelling as a possible improvement to our required A2C and A3C agents as well as Q-Learning since it inherits the best of both worlds.

Deep Deterministic Policy Gradient (DDPG)

In reinforcement learning, there are cases where the action spaces are continuous meaning the actions being made are over a set of real numbers rather than a set of discrete numbers for making actions. In a situation like these, reinforcement learning methods such as Deep Q-Network (DQN) are not very effective because of the tubularization of actions. In these situations, DQN attempts to create a network where it creates intervals to solve these problems and have values within those intervals map to one action. The issue with this is that it could be impossible to reach the target network without making each interval small enough and by doing so there are more state-action pairs to calculate. For example, let's consider an action that is mapped to the set {-3, 0, 3} where a is the value that will be mapped to an action a3. Now let's say 3 such actions are of this form each that shifts increments the number by 0,1,2. By this logic, we will have 33 such observations mapping to 3 actions. This is just a discrete example that shows that the number of observations can yield different outcomes. In the situation where we would consider actions in such a way that we can account for the discretization of real numbers, it would have that many actions mapped to it and therefore take very long for DQN to reach a target network. In situations such as these, we should consider using some kind of policy that yields a real number value that represents an action.

Deep Deterministic policy gradient (DDPG) is a model-free, off-policy, actor-critic algorithm that solves some of the problems that come with using DQN. Because of

these attributes, it can handle environments that rely on a continuous action space. Furthermore, DDPG will use some features of DQN, such as a replay buffer to reduce similarities between episodes as well as use a target network θ' to train the neural network θ that we are updating Q values in.

Like in many other approaches to reinforcement learning DDPG makes use of the recursive relationship know as the Bellman Equation:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}\left[Q^{\pi}(s_{t+1}, a_{t+1})\right]\right]$$

[BL1]

As a reminder, the Q value is updated at each timestep t when it is given a state s and action a. The q value is calculated by getting the current reward value and adds it to a discount value of the subsequent expected Q values. In this basic form of the Bellman equation, we consider the Q value of this network improving based on a target network that it is working against which in many cases is a previous version of the Q network that we are currently training. This target Policy is represented as follows:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))\right]$$

[BL1]
To prevent inner expectation, we calculate the action based on the target policy m instead of just looking for the best-case action in a particular state. We do this so that the Network that we are currently being updated is consistently learning against a policy instead of an ever-changing Q network which can cause the network we are training not to converge.

As previously mentioned, for continuous action spaces, it is very difficult for DQN to be able to effectively be able to map states to actions without optimizing each action a_t, at every state in every timestep t. For that to happen it would require very fine

discretization of actions and this would lead to a very huge network to train to require an unfeasible amount of training time. DDPG solves this problem by using a policy m(s|q) which maps states to a specific action. This is known as the actor which is what is being trained to make better decisions. After this actor makes its decision then it is assessed by the critic-model which is a targeted Q-network that takes in the state and the action made by the actor and yields a Q value. This Q value is used to make adjustments to the policy m so that it can make better decisions in the future. Note that the policy can only be as good as the target network so we must have a critic model that is constantly updating in order but only update the target network after the policy starts converging on the old target network. The equation that updates this policy is the following:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t} \right]\end{aligned}$$

[BL1]

When training reinforcement learning agents it is often the case that some states are reached more often than others. Because of this when training an agent in an environment it is important to look at those states and look at as many branches from those states as possible to be sure that the Bellman equation can take into consideration the best move when adding a discounted value of the preceding states. As previously mentioned, DDPG uses a replay buffer to take advantage of these situations. This drastically reduces the time it takes to train an agent because we no longer have to wait for the agent to reach a particular state to assess it again. In general, a replay buffer is a finite-sized cache of previously reached states. The optimal size of the replay buffer is very dependent on whether or not the algorithm your using is On- or Off- policy. Since DDPG has a DQN-like component in the critic model it is an Off- policy reinforcement learning method it can rely on a huge replay-buffer as it will not affect a target network when training. At every given timestep we can add a new state of the environment to the replay buffer for training later. The information that is required to be stored in this replay buffer is the current state of the environment, the action the actor model made at the said state, the reward that it is projected to receive,

and the next state of the environment after the actor model makes the action at the current state of the environment.

Generally speaking, implementing Q-learning with neural networks for reinforcement proves to be very unstable. (Lillicrap Timonthy Continuous Control with Deep Reinforcement Learning) Because of this, it is necessary to make copies of current networks to train them and wait until a convergence before updating the actor-critic model. This copy is used to calculate future target values to determine the quality of each move. Changes to the target network are made after tracking the changes in the current network where starts to converge. The changes are shown below.

$$\theta \leftarrow \tau\theta + (1-\tau)\theta' with\, \tau \ll 1$$

[BL1]

Agent's training in environments with continuous action spaces tend to have problems with exploration with policy-based learning. Another advantage of being an off-policy algorithm, is that DDPG can explore independently from the learning algorithm by introducing noise to the target actor-policy network m'. By adding noise to the actor policy network, the range of actions a policy can yield at any given states expands giving the critic model more data to look at and speeding up the process of training the actor. The equation is shown below:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

[BL1]

In the experimental tests, that Timothy Lillicrap and his team in every environment that they tested DDPG with they were able to determine a reliable agent within 2.5 million steps in each environment. This is roughly a difference of a factor of 20 of how many steps it required for a DQN to reach a solution of about the same level. The environments that they tested on were environments that had low dimensional

observation spaces meaning that there are fewer variables to consider in the environment. This makes it easier for the actor to learn as it will not need to adjust a large number of weights given to each one of those variables. The environments also had continuous action spaces and therefore were not very friendly to the DQN environment. The pseudo-code for DDPG is as follows:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

[BL1]
In the Everglades environment, we have a discrete action space where an agent can move up to 7 groups of battle bots into 11 nodes across the game map. Although this isn't a continuous action space it is still a very large action space to consider and it would be very beneficial to have an actor-critic algorithm to help the reinforcement agent learn. There are other aspects of this algorithm that can be used to improve learning in the Everglades environment. If we decide to do an off-policy implementation of the reinforcement agent, then we can use a huge replay buffer for the exploration of previously arrived states. Another feature that can be implemented in our actor-critic reinforcement agent is to wait until learning has stabilized before updating our target actor-critic model. This may slow down learning overall, but it guarantees that learning

is consistent and converges properly based on a current target network. Lastly, we may consider creating a noisy-network model to further encourage exploration. With both a replay buffer and a noisy network implemented the agent maximizes the usage of each experience, it has interacted with the environment. Overall, DDPG might not be optimal for our Everglades environment but it does have many aspects to it that can be used in our custom implementation of the Everglades reinforcement learning agent.

Trust Region Policy Optimization (TRPO)

An issue with a normal policy network is that as the agent continues to explore the environment and select their actions based on their current state and rewards from previous actions, the same policy network will not be adequate enough to be able to reward the agent properly based on the new state they currently find themselves in. To make sure that the policy network is as up-to-date as possible to keep up the reward system with the evolving agent, a policy gradient is put into place that will modify the policy network as the agent continues to venture through the environment.
The formula for a common gradient estimator in a policy gradient algorithm has the following form:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

Where π_θ is the stochastic policy network and A_t is an estimator of the advantage function at time t. With this formula, we take the log of the probability of a particular action given the state the agent is in and multiply the relative value of the action that was chosen. The expectation E_t is the empirical average over a batch of samples. The advantage function is calculated by taking the difference of the discounted sum of rewards and the value function's prediction of rewards from state s onwards. The advantage function is described as follows:

$$\sum_{t=0}^{\infty} \gamma^t r(s_t)$$

Where $r(s_t)$ is our reward function and γ is our discount factor which is bounded by [0, 1]. The gamma factor indicates how immediate the rewards are going to be since it is favorable for the agent to receive rewards that are more immediate rather than somewhere in the future.

With this policy gradient, we can look at different actions within a given state and decide how likely we want this action to be for this given state based on the rewards that were received. If the advantage function states that the action we took was positive which indicates that the sum of discounted rewards was more favorable than what we had expected from our value function, then we want those actions to be more probable and frequent with our agent. Otherwise, if the advantage function was negative and the sum of discounted rewards was not in the agent's favor when comparing it to other future rewards from our value function, then we want that action to be less probable with our agent.

The problem with this approach, however, is that the gradient method does not consider the magnitude at which the policy can change. What this means is that since this is a policy gradient algorithm, all of the inputs to the network are based on the agent's self-progress through the environment, so there isn't any replay buffer that the agent can simply train off of reliably and thus is fully dependent on the policy network to make the decisions. However, with an uncontrolled policy gradient determining the actions of the agent, the policy network can change drastically to the point where it is outside the scope of the original input and all the actions decided are mainly incorrect.

Trust Region Policy Optimization (TRPO) aims to solve the issue of creating a policy gradient algorithm that limits how much the policy network can change during any given action and state pairing [J12]. John Schulman (et. al) devised an algorithm that modifies the policy gradient to a form that is as follows:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t[\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

The first line is the proposed policy gradient which appears identical to the original policy gradient, but the log operator has been replaced with division with the previous policy network. This subjects the new policy network to be updated in relation to the previous policy network to ensure that the magnitude of which the network is to be updated does not diverge too far from the original network. Additionally, a new constraint, the KL constraint, is an added feature to the algorithm that directly compares the new policy network and the old policy network and only validates a change that is underneath a certain threshold in order for the new policy network to be updated. Should the new network attempt to update further than the provided threshold, the gradient will be truncated to where the threshold value is the upper limit on how much the network can change. This new approach to a policy gradient algorithm allows the policy network to update within a reasonable amount to make sure that the agent can decide on an action within a given state and understand their boundaries within the state.

Proximal Policy Optimization (PPO)

While Trust Region Policy Optimization attempts at moderating the magnitude of the policy gradients per time step, the process to update the new policy network becomes incredibly complex and leads to overhead issues with optimizing the policy gradient calculations due to the need to calculate a second term that checks if the new policy network changes within a certain threshold or not. With Proximal Policy Optimization (PPO) proposed by John Schulman (et. al) [J11], the goal for the algorithm is to simplify the calculations of the TRPO algorithm to avoid the overhead issue and make the implementation of the policy gradient more portable to different network training programs.

Before the algorithm can be discussed, Schulman (et. al) [J11] first simplified the TRPO algorithm to be in a more readable form:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

This is the same probability ratio that was mentioned in the TRPO algorithm and will be later used in the PPO algorithm implementation. From here, the TRPO algorithm can be reduced to the following form:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]$$

The TRPO algorithm can thus be described as the product of the probability of the action with the state pairing in respect to the previous policy network and the advantage function that will determine how likely the agent wants to perform those actions in the given state pairings. For $r_t(\theta)$, the value of the function will be greater than 1 if the action and state pairing was very probable in the old policy network and the value will be between 0 and 1 if the action and state pairing was not probable in the old network. The *CPI* superscript refers to a conservative policy iteration, which states implicitly that the gradient is not to change too much in regards to the previous network, otherwise a lack of constraint will lead to drastic changes in the policy network that will lead to the original problem of regular policy gradients.

All that is lacking from the recent equation altercation is the KL constraint that decides how much is the new policy network allowed to change, which is a large contributor to why TRPO is an optimization from the original policy gradient algorithm. With PPO, the algorithm implements the KL constraint along with the general TRPO algorithm as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

The policy gradient algorithm takes the minimum of two terms over a batch of samples of states and actions with an agent in a learning environment. The first term is the normal policy gradient algorithm that was shown in the simplification of TRPO as discussed from before. The second term within the minimum operator is similar to the first term, but it contains a clipping function with the r_t function and the two bounds: 1 – epsilon and 1 + epsilon. The bounds are in place to prevent the r_t function from updating too much and thus flattens out if it reaches or goes beyond that boundary.

The difference between the two boundaries is that 1 – epsilon is meant for decisions where the action, according to the advantage function, was negative and thus it was a bad decision for our agent to take, while the 1 + epsilon boundary is meant for decisions where the action was positive and it was a good decision for our agent to take. These boundaries serve to be our threshold values, where it is determined by the value epsilon uses to limit the amount of change the policy network takes on to update itself. The minimum function is in place for this algorithm because in the case where the advantage function predicts a negative outcome of rewards and r_t is rather high (meaning that the probability for the action to occur is high), then it is necessary for the gradient to move to a less probable chance of the action to occur. This only happens with the unclipped version of the product than the clipped version because the unclipped product creates a lesser value than the clipped product and thus is more favorable in how the agent should adapt to the situation.

The final PPO algorithm itself is performed as follows:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

[J11] *Pseudo-code for the PPO algorithm.*

The algorithm runs on two threads and incorporates an Actor-Critic Style approach for its loss and gradient descent calculations. Within the first thread, it will use the old policy network to run various states and actions, and it will compute the advantage function for each time step in the iteration. Afterwards, a second thread calculates gradient descent using the proposed algorithm of PPO and modifies the new policy network as prescribed from before. The threads can further push for optimization by not being linked together and instead work independently from each other. While the first thread calculates the different episodes and different state-action pairings and the

corresponding advantage values per pairing, the second thread can create the new policy network based on the previous results of the first thread.

The overall loss function that is used to train and evaluate the performance of the agent is with the following formula:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

For $L_t^{VF}(\theta)$, the term is responsible for updating the original network based on the rewards the network is expecting to see within the later future. This allows the network to update within a certain direction rather than updating aimlessly and evaluating if the network is in a preferred state through random updates. For $S[\pi_\theta](s_t)$, the term is the function's entropy term which determines if the agent has explored enough of the environment, since the agent can find a local minima of loss and weights to the input but it might not be the preferred solution to the environment and that is partly due to the agent's lack of exploration of other possibilities that can lead to better minima values.

The PPO algorithm sets to be a simplified version of TRPO while still retaining the goals of the original algorithm which allows the agent to update its policy network in a considerable but not a drastic amount and retains the principles of TRPO within its calculations of the policy gradients.

State of the Art

AlphaGo DeepMind

AlphaGo is a notable project that represents a breakthrough for Reinforcement Learning in recent years.

Go originated from China and has been around for over 3000 years. The game is a turn-based strategy board game that requires a lot of multilayer strategic thinking with infinite possible game states. Because of the complexity of the game, developing a good state-based AI was very challenging.

However, in 2015, a team from DeepMind Technologies, later was acquired by Google, researched and developed a state of the art Go AI at that time and defeated the Go world Champion at that team by utilizing Reinforcement Learning.

The AlphaGo AI was developed using a Monte Carlo tree search, which was not a new method, but it was accompanied with the power of Machine Learning. It was trained to use the previous knowledge using a neural network hence deep learning and was trained extensively through thousands of matches in a dataset.

Dota 2 OpenAI Five

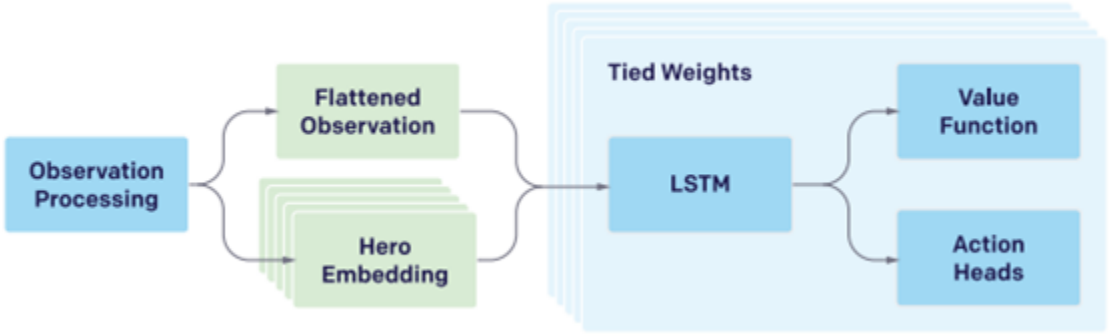
A more recent development in reinforcement learning. Starting from 2016, the researchers developed AI that can play competitively in a real-time, team-based video game Dota 2. Compared to Chess and Go, which has seen success in AI development, Dota 2 has a different and more complex environment because it is a real-team strategy game, instead of turn-based.

However, with the public debut in 2017, OpenAI Five surprised the Dota community by winning a one-on-one game against a Dota world champion at that time. A year later, with more development, the AI attempted to challenge professional teams in a conventional five-to-five game but lost both games that it was playing. However, it was still considered a success.

Then in 2019, the team at OpenAI Five attempted one more time with their more developed AI. This time they successfully defeated the world champion in a five-to-five game with a best-of-three format. [B3]

The team used a complex, state of the art policy gradient method called Proximal Policy Optimization (PPO) to develop, train, and improve their AI. By using this method, the team successfully develops an AI that can perform well in an environment that requires a large number of actions to be calculated. Since the AI calculates new actions every 4 frames in an average Dota game of 45 minutes running 30 frames per second, it needs to calculate 20 000 moves every 33 ms, compared to 250 moves for the whole of Go or 40 moves in a game of Chess. [B3]

In its paper, they used a 4096-unit LSTM as the core component for their training system [B3] instead of the traditional feed forward network.



Each of the five heroes on the team is controlled by a replica of this network with nearly identical inputs, each with its own hidden state. The networks take different actions due to a part of the observation processing's output indicating which of the five heroes is being controlled. The LSTM composes 84% of the model's total parameter count [B3].

AlphaStar

On December 19th, 2018, the DeepMind team achieved yet another accomplishment to broadcast the progression that the Artificial Intelligence community has gained when their latest agent, dubbed “AlphaStar”, beats professional StarCraft player Grzegorz “MaNa” Komincz and Dario “TLO” Wünsch in the Blizzard Entertainment video game StarCraft II. This is one of the accomplishments that the Artificial Intelligence community has achieved over the past several years. Different achievements that have been gained include Deep Blue Chess’s AI agent beating the world champion back in 1997 and DeepMind’s previous agent creation AlphaGo beating grandmaster Go player Lee Sedol in the board game Go back in 2016.

Comparison

What makes AlphaStar different from its predecessors is that the game that the agent plays, StarCraft II, contains special rules that makes the game vastly different from games such as Go and Chess. To summarize, the video game stipulates, “Real-time play, partial observability, no single dominant strategy, complex rules, and a particularly large and varied action space [J9].” Real-time play means that every single time step that the agent must evaluate the best solution possible was every single millisecond that the game was being played out, meaning that the agent doesn’t wait for their opponent to make a decision before the agent can be allowed to make their decision as well. Partial observability means that the game contains imperfect information, which means that the agent at any given time step may not have all of the critical information provided to it in order to make the best informed decision at that point. No single dominant strategy means that the agent must have a varied amount of solutions available, complex rules means that the agent must take more time in order to decide the best decision possible, and a large action space means that a Q-value table is definitely not considered for evaluating the agent’s policy network.

Additionally, AlphaStar was given unique restraints that seems odd to implement into an Artificial Intelligent agent but was used to create a reasonable and fair agent for other human players to compete against. As the AlphaStar blog mentions [J5], the agent plays with the same constraints as the human opponents, the agent can play one-on-one matches as and against all three races (Protoss, Terran, and Zerg); each

being a single neural network, league training is automated and starts with supervised learning instead of playing against previously trained agents from past experiments, and the agent played on the official game server instead of a hand-crafted game to support the needs of the agent. The interesting point is that the agent plays with the same constraint as the human players, which includes having the same camera views that a human player would observe and not being able to input more keystrokes than an average professional player could achieve. This makes the agent have a more human-playing style that many people have cited about the agent.

Implementation

Based on the report that the AlphaStar team has reported [J6], the central policy that AlphaStar uses is based on the following figure:

$$\pi_\theta(a_t \mid s_t, z) = \mathbb{P}[a_t \mid s_t, z]$$

where π_θ is the policy function with current weights θ that represents a probability distribution of actions given the state space, which is the recorded observations and all subsequent actions. The state space can be represented by the following:

$$s_t = (o_{1:t}, a_{1:t})$$

where $o_{1:t}$ represents the observations made from the beginning to the current timestep and $a_{1:t}$ uses the same time stamps but with the action pairings with each observation it made. The last input for the policy function is a statistic z that summarizes a strategy sampled from human data, such as a build order in the game.

For the training process of the AlphaStar agent, based on the report and the blog post [J5] made by DeepMind, the training was based on a combination of temporal difference learning, clipped importance sampling, and a new self-imitation algorithm that the team creates solely for this agent. As a start for the agent, the team gathered several samples of human play, in which the agent would use to replicate through means of supervised learning and then improve its gameplay through reinforcement learning. The reinforcement learning algorithm was based on a policy gradient algorithm that had similarities to the advantage actor-critic algorithm, otherwise known as A2C. The reinforcement learning algorithm was known as “self-play”, where the

agent would play against itself in order to further improve its game play, which is a simple and effective idea initially, but there were problems with using self-play as a model for reinforcement learning for AlphaStar. The main issue highlighted is that the agent could potentially "forget" its optimal solutions and then find itself going through an infinite loop of playing rock-paper-scissors and never fully converge. In StarCraft II, there exists units that have their strengths and weaknesses, which are exploited by other units with strengths and weaknesses, but those units are also exploitative with other units and thus begins a cycle of rock-paper-scissors, where the agent may opt to use a set of units, find that it can be exploited by other units, so it then opts for the new set of units, but those have weaknesses and then finds another set of units to use. This is the problem with there not existing a single strategy that can win the videogame and means that the agent will have a difficult time trying to converge to an optimal solution.

The best approach that the AlphaStar team can do for this problem is their creation of the "fictitious self-play", where instead of the agent going against just a single set strategy, learning from its mistakes, and then finding itself looping through the same strategies and never converging, fictitious self-play makes the agent compete against a mixture of all previous strategies so that the agent can fully diagnose the exploitative features of the units it is using as a whole and can evaluate the pros and cons of different strategies as a whole.

While the use of fictitious self-play was a convincing way to solve the problem of not being able to converge to an optimal strategy, it still was not a powerful enough solution to address all of the flaws and weaknesses that the agent's strategies contains since fictitious self-play was more concerned about individualized weaknesses rather than broad-based weaknesses in the agent's strategies. To combat this, the AlphaStar team utilized an interesting approach with something known as the "League". The League contains a group of agents used to exploit the AlphaStar's weaknesses to improve their play fictitious self-play and the objective to simply maximize the winrate had its flaws. The League was composed of three different groups of agents. The first group of agents are the main agents that represent AlphaStar's gameplay and what most people would think of the agent since it facilitates the action decisions and is therefore the primary learner in the League. It uses Prioritized Fictitious Self-Play (PFSP) that adapts the mixture probabilities proportionally to the win rate of each

opponent against the agent. The first group of agents contains three main agents, one for each race (Protoss, Zerg, Terran). The second group of agents are known as the main exploiter agents that have the primary objective of looking at the three main agents from the first group and try to identify any exploits that they might have and expose the agents to it so that they can learn from those exploitative features. The second group contains 3 agents, one for each race, much like the first group of agents in the League. The final group of agents are known as the League exploiter agent that uses a similar PFSP but its purpose is to exploit the entire League itself for a more broad-based analysis of the agents' effective performance.

Each of the Terran, Protoss, and Zerg AlphaStar agents were trained off of the official Battle.net online matchmaking in different iterations and the main hardware that the AlphaStar agent used for the training process was done across 32 third-gen TPUs per one agent in the League. Overall, the initial supervised training of the AlphaStar agent to establish a baseline for the agent, dubbed AlphaStar Supervised, was done through the Battle.net online matchmaking service. Afterwards, there was a 27-day league training to create the midpoint agent known as AlphaStar Mid, and finally a 44-day league training for the final iteration of the AlphaStar agent known as AlphaStar Final. That means for the final iteration of the AlphaStar agent, it took the model 44 days off of the 32 TPUs to train and have the best possible iteration of the League as they could.

The agent was trained with human-like approaches and fictitious self-play as the main focus for its training in order to address issues with discovering novel strategies and game-theoretical challenges respectively. For novel strategies, ideas where there was a clear solution to a certain problem would not be obvious for the agent and it would have a difficult time trying to find the best solution possible, considering that the action space was vast and that the agent could end up being lost as to where to converge. Additionally, the game was complex enough that certain challenges may become too abstract or convoluted for the agent to be able to fully analyze. To explore the vast action space, it relied on learning human strategies initially and then was introduced a latent variable and a form of distillation to bias exploration towards human strategies, which helped preserving high-level strategies and preventing any conditioning to any opening moves [J5], [J6].

Evolutionary Algorithm Perspective

While AlphaStar had an interesting approach to combat unique challenges to learn a complex game, the agent has been shown to be an excellent study when considering the uses of evolutionary algorithms. In the context of evolutionary algorithms, AlphaStar used Population-Based Training [J13], which is a memetic algorithm that uses Lamarckian evolution, which contains an inner loop to do backpropagation to hypertune the parameters while and outer loop selects the best network based on several selection methods [J9]. The advantages of using PBT is their effective use of distributed and asynchronous resource management [J14] to reduce overhead and its use of steady state [J15] which is contrast to generational evolutionary algorithms in that steady state "allows optimizations and evaluations of individual solutions to proceed uninterrupted and hence maximize resource efficiency [J9]." This means that the agent effectively allocates resources to its training where it does not produce an overhead and yet additionally allows something that mimics parallelization with the training process using steady state since processes in training go uninterrupted as different parts of the agent completes different tasks at the same time.

Additionally, it has been observed that the first consideration for training the agent was through the use of self-play but has shown to be not the most effective strategy by itself. Instead of simply doing self-play, the agent can improve itself using a superset of self-play known as competitive co-evolutionary algorithms, where it takes a solution and evaluates it in comparison to a population of solutions [J9]. In StarCraft II, with no one best strategy to claim victory, the agent contains a set of solutions it can use that it obtains through the reinforcement learning algorithm and the League strategy that the AlphaStar team implemented. This would mean that for each time step, the agent requires having a diverse set of solutions to evaluate from. Once it obtains the set of solutions, the League will go through those solutions and choose which ones do not have apparent weaknesses and which have flaws that should not be considered due to its exploitative nature. AlphaStar is an observation of different Artificial Intelligence research thus far, including research Lamarckian evolution algorithms, competitive co-evolutionary algorithms, and quality diversity algorithms.

Evaluation

AlphaStar Final, after the end of training and evaluation, was ranked above 99.8% of ranked human players and at Grandmaster level for all three races. Additionally, AlphaStar Supervised, the baseline model used to build up AlphaStar Final, had reached an average rank above 84% of human players alone. When compared to OpenAI Five, the Dota 2 agent was able to achieve success but only after certain limitations were met, such as a restriction on heroes that the human players and agent could play, the hard-coded sub-systems for certain aspects of the game, and having no limitations on the camera perceptions. What makes AlphaStar a unique accomplishment in comparison to OpenAI Five is that it played similarly to a human opponent, with full access to all three playable races in the game, a restricted camera view that all players must use, and 22 non-duplicate actions per 5-second window and 110 ms of delay for “reaction timing”, which mirrors that of a human player. Thus, at no point did the human opponents feel like they were competing against an agent with an unfair technical advantage; the agent played much like how a human would play StarCraft II.

Gantt Chart

Senior Design 1

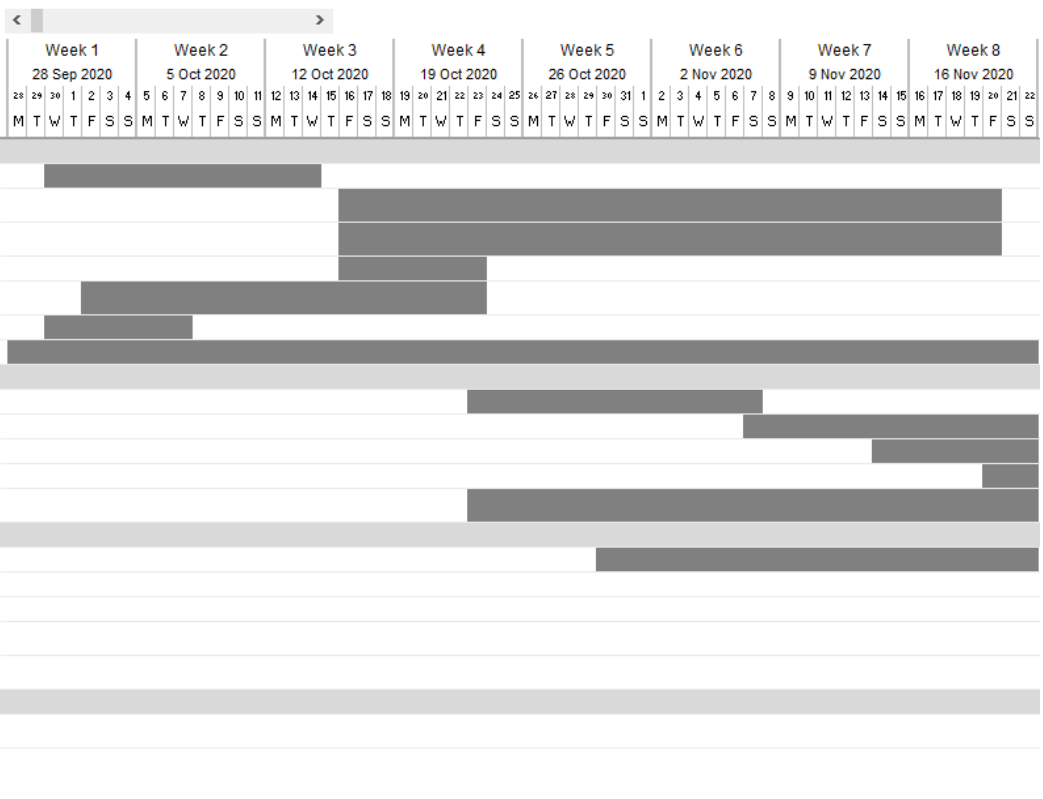
Everglades

Group 19 Gold Team

Gantt Chart Template © 2006-2013 by Vertex42.com.

Project Start Date: 9/30/2020 (Wednesday) Display Week: 1

WBS	TASK	LEAD	START	END	DAYS	% DONE	WORK
1	**Document**			-			-
1.1	Project Overview	Jace	Wed 9/30/20	Wed 10/14/20	5	100%	11
1.2	Machine Learning Overview	Michael	Fri 10/16/20	Fri 11/20/20	5	100%	26
1.3	Policy Algorithm Overview	Ben	Fri 10/16/20	Fri 11/20/20	4	100%	26
1.4	Open Gym AI	Alan	Fri 10/16/20	Fri 10/23/20	4	100%	6
1.5	Everglades evironment set up	Bao	Fri 10/02/20	Fri 10/23/20	5	100%	16
1.6	Budget	Bao	Wed 9/30/20	Wed 10/07/20	7	100%	6
1.7	Final Design Doc	Ben	Sat 2/10/18	Mon 12/07/20	7	100%	736
2	**Research**			-			-
2.1	DQN	Bao	Fri 10/23/20	Sat 11/07/20	4	100%	11
2.2	PPO	Michael	Sat 11/07/20	Mon 12/07/20	3	100%	21
2.3	A3C	Ben	Sat 11/14/20	Mon 12/07/20	3	100%	16
2.4	Newton cluster	Alan	Fri 11/20/20	Mon 12/07/20	6	100%	12
2.5	Alternate policy gradient algorithm	Jace	Fri 10/23/20	Mon 12/07/20	3	100%	32
3	**Implementation**			-			-
3.1	DQN	Bao	Fri 10/30/20	Mon 12/07/20	4	80%	27
3.2	PPO	Alan	Mon 12/21/20	Fri 1/08/21	3	0%	15
3.3	A3C	Michael	Fri 1/08/21	Fri 1/29/21	3	0%	16
3.4	Increase winrate to > 70%	Jace	Fri 1/29/21	Fri 2/12/21	6	0%	11
3.5	Increase winrate to > 90%	Jace	Fri 2/19/21	Tue 3/09/21	3	0%	13
4	**Feedback**			-			-
4.1	Get initital feedback from sponsor	Michael	Tue 3/09/21	Fri 3/12/21	1	0%	4
4.2	Finalize and send the AI to compete with other teams	Michael	Tue 3/16/21	Fri 4/02/21	1	0%	14



Senior Design 2

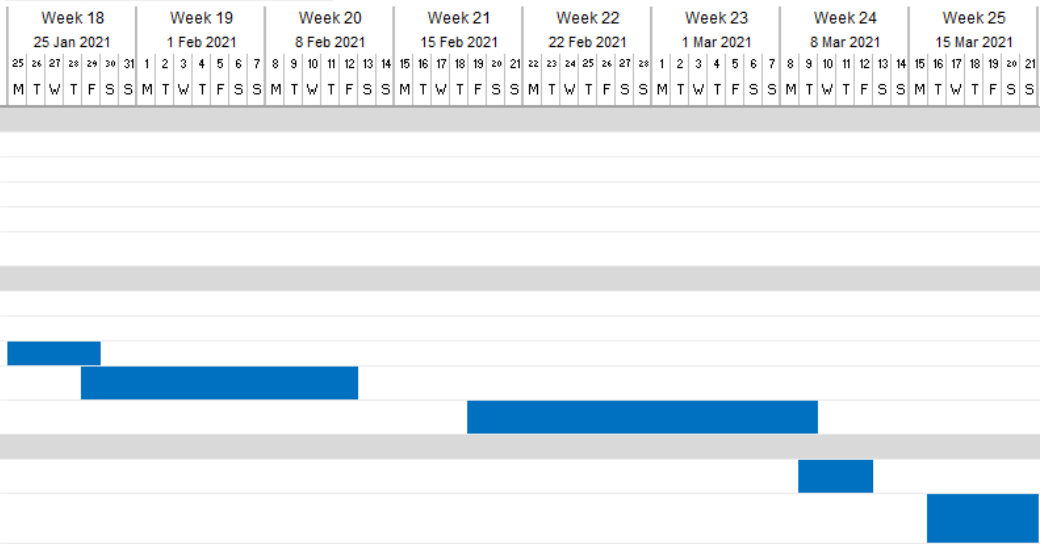
Everglades

Group 19 Gold Team

Gantt Chart Template © 2006-2018 by Vertex42.com

Project Start Date 9/30/2020 (Wednesday) Display Week 18

WBS	TASK	LEAD	START	END	DAYS	% DONE	WORK
2	**Research**			-			-
2.1	DQN	Bao	Fri 10/23/20	Sat 11/07/20	4	100%	11
2.2	PPO	Michael	Sat 11/07/20	Mon 12/07/20	3	100%	21
2.3	A3C	Ben	Sat 11/14/20	Mon 12/07/20	3	100%	16
2.4	Newton cluster	Alan	Fri 11/20/20	Mon 12/07/20	6	100%	12
2.5	Alternate policy gradient algorithm	Jace	Fri 10/23/20	Mon 12/07/20	3	100%	32
3	**Implementation**			-			-
3.1	DQN	Bao	Fri 10/30/20	Mon 12/07/20	4	80%	27
3.2	PPO	Alan	Mon 12/21/20	Fri 1/08/21	3	0%	15
3.3	A3C	Michael	Fri 1/08/21	Fri 1/29/21	3	0%	16
3.4	Increase winrate to > 70%	Jace	Fri 1/29/21	Fri 2/12/21	6	0%	11
3.5	Increase winrate to > 90%	Jace	Fri 2/19/21	Tue 3/09/21	3	0%	13
4	**Feedback**			-			-
4.1	Get initital feedback from sponsor	Michael	Tue 3/09/21	Fri 3/12/21	1	0%	4
4.2	Finalize and send the AI to compete with other teams	Michael	Tue 3/16/21	Fri 4/02/21	1	0%	14



Project Milestones

Senior Design 1

Initial Design Proposal Paper - 09/30/2020

The team as a whole will have an initial 15-page document discussing topics outlined in the Critical Design Review Outline. These topics include project requirements, specifications, goals, block diagrams, objectives, budgeting, Legal, Ethical, and privacy concerns.

Setup Game Environment - 10/02/2020

Every team member will have an understanding of how the Everglades Environment works as well as have Open AI Gym and the Everglades package installed on personal machines.

Researching Open AI - 10/16/2020

Team members will gain a fundamental understanding on how Open AI Gym can be used to create reinforcement learning agents by working on personal projects and following basic tutorials.

Initial Agent Design - 10/23/2020

An initial agent design has been decided and implementation has started. First design will be either DQN or PPO as required by the project must-haves.

Second Paper Submission – 10/23/2020

Every team member will be assigned different portions of the final project diagram. Team members will do research on various topics and contribute 15 pages.

Agent 1 DQN implementation – 10/30/2020

Initial agent implementation has been completed. All technologies required such as PyTorch and Newton clusters will be set up for training the reinforcement agent. The agent will be put through small training sessions.

Agent 1 DQN Complete - 11/13/2020

The agent will go through extensive training sessions. Team members will identify any shortcomings with reinforcement training algorithms.

Agent 1 DQN Redesign - 11/27/2020

Team members will brainstorm on how to fix all shortcomings that have not been fixed. Continue research will be done if needed. Benchmarking will be done after every training session. Variable adjustments will be measured and plotted on a graph for use in future reinforcement implementations. A detailed report of DQN performance will be completed for the paper.

Winter Break

Short meetings will be held during winter break to discuss what policy gradient algorithms we plan on implanting for our third and fourth agents. These algorithms will be more complex and will require more research which will be done during winter break.

Senior Design 2

First Meeting before the semester - 01/01/2021

Team members will decide what are the 2 policy gradient algorithms to use for the Agent 3 and 4.

Agent 2 PPO Implementation – 1/07/2021

Team members will do research on PPO for implementation for initial PPO implementation. The reinforcement agent will be put through some training sessions and reward accumulation will be measured. Team members will identify the shortcoming of the initial implementation.

Agent 2 PPO Training – 01/21/2021

The reinforcement Agent will be put through extensive training sessions. Performance measuring will be done for the final presentation. Small fixes and adjustments will be made to the reinforcement agent if needed.

Agent 2 PPO Redesign - 02/28/2021

Any agent shortcoming will be addressed, and any variable changes will be measured and plotted for the final paper. Continued research may be required.

Reward Shaping For High Score- 03/07/2021

Reward shaping methods to reward agents to capture many nodes to get a higher score then the opponent has been implemented.

Reward Shaping For Defending Base Node - 03/14/2021

Reward shaping method for rewarding the agent to send battle bots to the base node when it is being attacked has been implemented.

Discernible Strategy Training and Testing - 3/21/2021

Training has begun to defeat discernible agents. The two current implementations of the agents are being trained against Cycle, Base Rush, Same Command and Swarm strategies.

Discernible Strategy Training and Testing Completed - 3/28/2021

Training is now completed. DQN and PPO have achieved 60% win rate on average against discernible strategies. Conclusions have been made that the environment is unfavorable to players who play defensively against base and cycle rush strategies.

Agent 2 PPO Complete - 04/01/2021

Agent has completely been implemented with reward shaping.

Agent 1 Update ℅ Rainbow DQN Complete - 04/07/2021

Agent has been completely trained with reward shaping. Data collection has begun for the final documentation.

Agent 3 (SAC) Implementation - 04/07/2021

Initial Implementation of a policy gradient algorithm will be completed. The reinforcement agent will go through some training and team members will identify issues that the agent runs into in the training process and document them.

Agent 3 (SAC) Training - 04/15/2021

The reinforcement agent will be put through extensive training sessions. Benchmarking comparisons will be made with DQN and PPO implementations to measure overall performance.

Agent 3 (SAC) Complete - 04/18/2021

Final implementation of agent 3 will be finalized. All changes from previous versions will be documented for future presentation. Final report on performance of the reinforcement agent will be finished. Agent was not fully implemented with reward shaping.

Methods for Improving Performance

One-Hot Encoding

In Everglades, there are a few instances in the observation space where they used integer values to describe categorical data. This can cause some problems when training a neural network to try to learn how to play a game. In a situation where these indexes would represent some numerical value, such as the health of the battle bots, the neural network will be able to crunch numbers, and when it calculates some number such as 11.5 it will know that the health is between 11 and 12 and will be able to make the right decision with this information. Let's consider integer representation for categorical data in Everglades. As an example, in Everglades' observation space, the environment uses integers to represent locations of where each bot group is. If a bot group is in location 1 then the neural network will attempt to calculate what the best move is for that bot group. Now let's say after the neural network crunches numbers it says a bot group is at location 3.5 and tries to make a decision based on that. This is where the issue lies as 3.5 means it is between locations 3 and 4, but 3 and 4 have distinct characteristics that do not translate into non-integer values. The nodes adjacent to nodes 3 and 4 are completely different and therefore non-integer values do not have any meaning.

Let's take a real-world example of this. Let's denote 3 countries that are Vietnam, China, and Thailand as integers 1, 2, 3 respectively. Then say that the neural network calculates a value in one of its layers that represents country 1.5. This means the country is in between Vietnam and China which has no meaning in the real world. This problem can be solved by the method of One-Hot Encoding the observation space. One-Hot Encoding is the process of changing categorical data to a vector of 1s and 0s as booleans to determine whether it is a particular category or not. Let's consider the example previously mentioned for the three countries. In that initial example we were using one integer to represent all three countries; now we will use a vector of 3 indices to describe these three countries. Let's describe it as [isVietnam, isChina, isThailand].

So if I am in Vietnam then the vector is [1,0,0]. By doing this, a neural network can precisely know what type of data it is dealing with.

In the Everglades' environment, for the one-hot encoded observation space, we converted the type of bots a group is composed of to a vector of size 3 and converted the location index to a vector of size 11. If we converted these indices to just more indexes to represent the vectors, then we changed the original 105 index observation space into 249 to take into account all the conversions. There are some implementations out there that convert these indexes to just an array to represent the vectors in which case the observation space will remain 105 in this situation. But we decided to go with the former in our implementation.

Legal Moves Masking

Generally in reinforcement learning, especially when using deep training techniques, ensuring AI agents always make legal sets of moves is unnecessary. Usually when using deep training technique, because the goal is generalization of the environment, or in other words, form a general behavior for an AI agent, we want not to involve in the generalization process as much as possible so that we can see how well an agent learning on its own with the raw observation info. In a lot of environments, deep AI agents should automatically learn the difference between legal and illegal moves after a period of generalization.

However, for Everglades, we observed that our agents do not produce a good result when we let it make whatever moves. Without knowing which moves are illegal, the general behavior of our agents during testing is that, oftentime, the agents don't make a lot of actions, or decide to do nothing. That is because in Everglades, there are 132 moves that we can make at any given turn, and depending on the observation input, there are a lot of turns that most of the moves are illegal. For example, if all groups are in transit, all moves are illegal. If we let the agents do whatever, the agents would try to fit a lot of these moves into the network, and in Everglades, if an agent makes an illegal move, it will be translated into doing nothing.

Therefore to ensure that our agents always make progress during training, and generalize well enough so that it would produce a good result when we test it, we support our training agents with an illegal detection method.

It's quite simple. It takes in the original observation array, then we process the illegality of the moves based on the location of each group. All agents that are in transit are not supposed to make any action, so the actions belonging to those groups will be considered illegal. Actions that would move a group to the neighbor nodes are legal. Move that leads to the current node, or in sort, stay and defend, is also considered legal. The rest sare illegal moves. The output of this function is a boolean table of size 132 that marks every legal move as True and illegal move as False.

During training, it's also important to apply the legal move mask to the random actions during exploration, and make sure the agent only updates its network from learning the legal actions.

Rewarding Shaping Methods

Since the Everglades environment only offers rewards at the end of the game with a +1, -1, or 0 if the agent won, lost, or tied respectively, it creates a Sparse Reward Problem within the environment, since the agent does not know if it has selected a "good" move or not due to the lack of feedback the environment gives at each time-step. The agents could still learn from this reward distribution, but it would require the agents to reach 150 time-steps at most per game to learn if it made the right choice at the end, making the training sample-inefficient and too focused on the last time-step of the game over any of the other time-steps of the game.

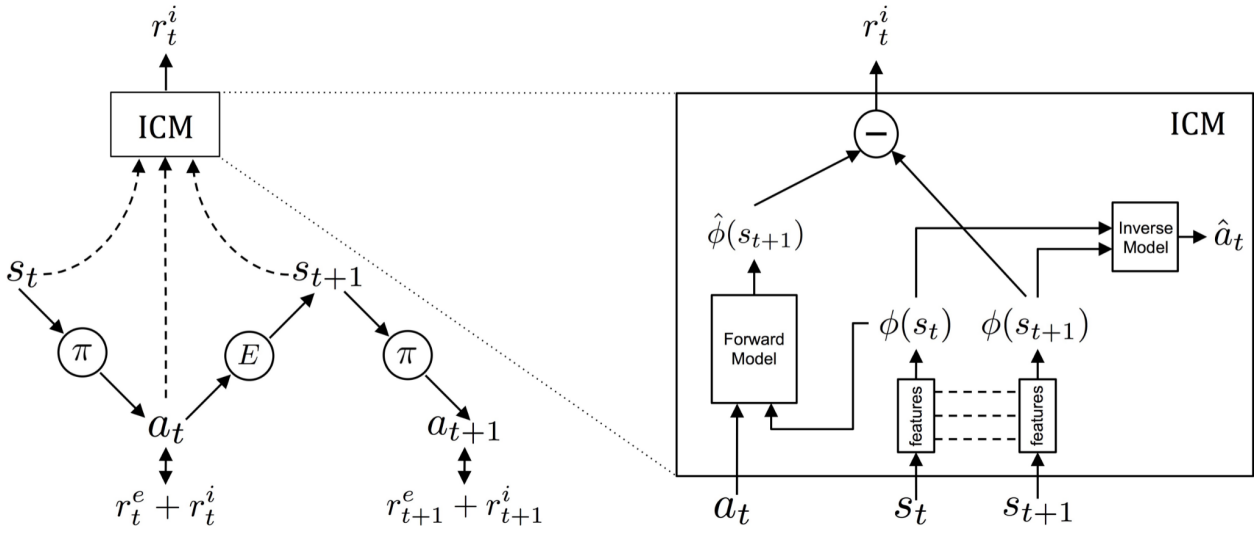
One approach to solve this situation is for the agent to play an entire game until it receives a reward and then adjust all of the previous time-step rewards to match the reward the agent received at the end of the game. For instance, if the agent plays all 150 time-steps of a game and receives a +1 reward for winning the game, then the agent can modify the previous 149 time-steps to also be +1 reward, since all decisions in the game led to a victory for the agent. Additionally, if the agent receives a -1 reward at turn 150 indicating a defeat, then the agent can change all 149 previous time-steps

to be -1 since those time-step actions led to a defeat for the agent. The issue with this approach is that a majority of actions per time-step do not guarantee a win or a loss for the agent. This means that there are certain actions at a given time-step that can be indicated as leading to both a win and a loss, which conflicts with the agent's expected reward function per game state and therefore the agent cannot converge to a reward model for the agent to follow and exploit.

For the purpose of this project, it was important to solve the Sparse Reward Problem for this environment so that the agents can have a stable training session and can adopt certain behaviors based on how rewards are distributed by a unique reward-shaping function instead of relying solely on the environment's rewards. For this project, three different reward-shaping methods were implemented and tested for improvements in performance for our agents: Intrinsic Curiosity Module, Maximize Score, and Defensive Play.

Intrinsic Curiosity Module

The Intrinsic Curiosity Module [J19] is a self-supervised convolutional neural network that rewards the agent for exploring the state-action space. The weights for the convolutional neural network are constructed by observing the previous state and current state and attempts to predict what set of actions have been taken to transition between the two states. Once the encoding weights for the network are trained, the module will attempt to predict what the next observed state will be in the environment based on the current observed state and the action taken by the agent. The "error" that the module receives will be the intrinsic reward the agent will obtain at each time stamp. For instance, if the module can successfully predict the next state, the error for the module is minimized, thus the rewards the agent receives is also minimized. This encourages the agent to explore more of the state-action space to make the ICM network less capable of predicting the next state, which then increases the error of the ICM and thus maximizes potential rewards for the agent.



[J19] Diagram of the Intrinsic Curiosity Module.

This module is helpful for on-policy algorithms, such as Proximal Policy Optimization, that do not have explicit functions to encourage exploration of the environment such as epsilon for DQN and additionally suffers from receiving low amounts of rewards from the environment. However, the module does not explicitly encourage the agent to exploit winning behaviors but rather explore all possible state-action combinations. To get desired behaviors from the agents in the Everglades environment, another reward-shaping function is required to promote winning behaviors. Additionally, having the agent observe a majority of the state-action space is computationally expensive temporal-wise and thus it could not fit into the project's timeline, so it was discarded after training and testing with the PPO agent for 5 weeks.

Maximize Score

An observation was made in the environment that the score for each player is calculated at each time-step of the game. This is because the game can end after 150-turns have been reached and none of the bases have been captured by the other player nor have any of the total units on one player's side been reduced to 0. In this state, the winner is determined by the player that has the largest score, which is calculated based on the number of nodes that are controlled on the field plus the number of units still alive.

Since the score for each opponent is calculated at each time-step of the game for each opponent, the reward-shaping method labeled “Maximize Score” is calculated as the subtracted difference between the score for the agent and the score for the opponent. The advantage of using this method is that if the agent has a lower score than the opponent at a given time-step, then the reward given will be negative. Otherwise, if the score for the agent is greater than the opponent, then the reward will be positive. Additionally, the larger the difference of score is, then the more negative or positive the reward outcome will be. The drawbacks to this reward-shaping method is that it rewards based on how well the states are instead of the actions, but the agents should be interpreting how well the actions are based on the rewards since the agent can still make good actions even if the state suggests that the agent will “lose”. Additionally, the reward-shaping method is not effective against base-rush opponents since the opponents do not prioritize health of the units or capturing nodes but instead try to catch the other base in a short amount of time. This makes the reward-shaping method prone to reward the agent heavily in the beginning of the game since it captures nodes and takes out a few of the enemy units, but the enemy units are currently located in the agent’s base and the score drastically drops in the last few time-steps of the game.

Defensive Play

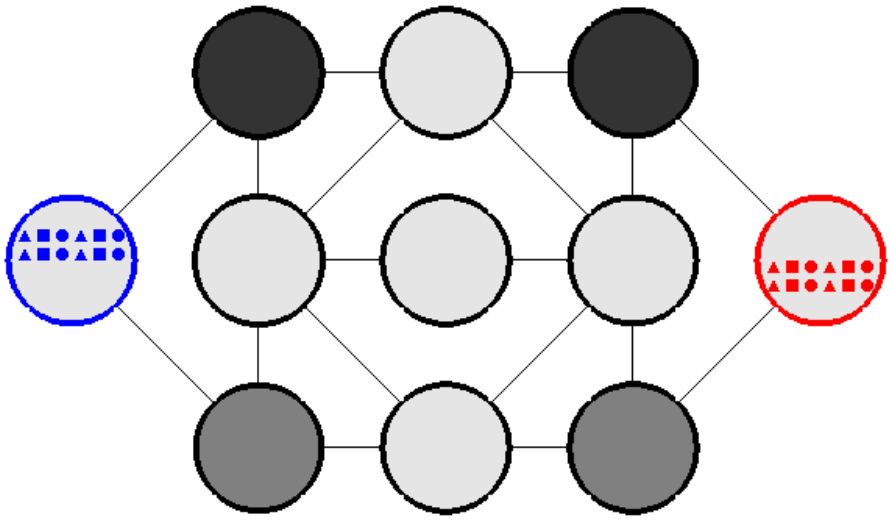
To adjust the issues that are found in the Maximize Score reward-shaping method, a secondary reward-shaping method is implemented to encourage “defensive actions” to prevent base-rush opponents from stealing the agent’s base within a short amount of time-steps. This method was labeled Defensive Play and the function rewards based on state configuration that the agent can observe and the actions that it took. The reward-shaping method observes nodes that are closest to the base node and analyzes the number of enemy units there are at each node in comparison to the number of agent units there are at the same node. Preferably, there should be an equal or slightly larger amount of agent units contesting at the same node than there are enemy units at the same node, which will guarantee the slowdown or elimination of enemy units going towards the agent’s base. Additionally, if the agent has selected actions that move units from non-contested units over to contested units or closer to contested units, then the function gives positive rewards.

The issue that this reward-shaping method poses is that if the rewards are not configured properly, then there could be too many rewards given for playing too defensively and thus the agent will move units to specified nodes and will not play offensively. Additionally, the environment's observation space can be misleading since enemy units can be observed to be occupying a node but are in transit, therefore the units cannot be attacked and are on the move towards a node that the agent cannot observe or predict.

Game Environment Renderer

After the initial DQN and PPO agent were implemented, it started to become necessary to understand the interactions that were happening on the map. Knowing troop placements throughout the course of the game would better allow for different reward shaping approaches to be revealed. It would also allow for easier debugging. Thus, we decided that implementing a renderer for the Everglades environment would be worth the extra time spent.

The renderer was made to act almost exactly like how traditional renderers within gym environments work: the user passes in the complete observation space and it visualizes and/or returns the RGB array for collection. The main difference, though, is that the Everglades renderer is not embedded into the environment; instead, it is a separate class that must be first instantiated. This modular approach was done for ease of use and allows for the original environment code to remain intact.



The first state of a game visualized using the Everglades renderer.

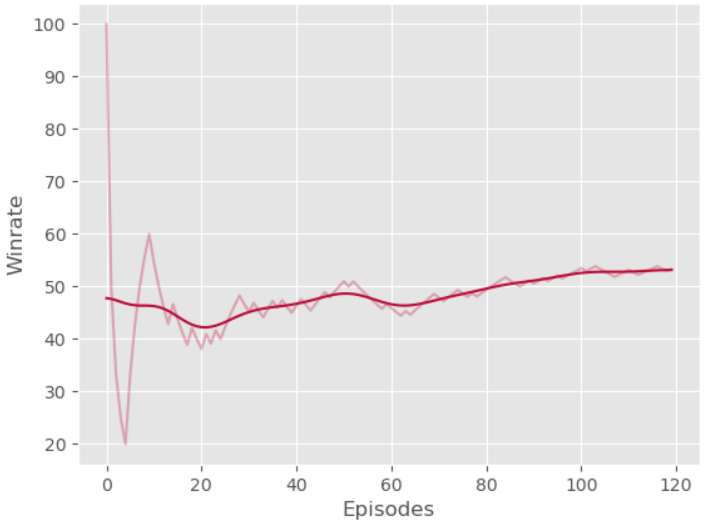
The visualization element shows each team as two separate colors, blue and red. The drone groups are seen as small polygons moving. Their shape corresponds to their type: triangles are strikers, circles are controllers, and squares are tanks. When the groups are in transit, they become slightly transparent. When they are killed off, their color darkens. The nodes are colored blue or red depending on who controls them (it currently does not take into account the percentage of control when showing color, but this feature should be simple to implement). The fortresses are dark gray, whereas watchtowers are a slightly lighter gray. Other textual information currently would be difficult to implement within this renderer, as it uses a front-end interface that OpenAI used for their “Classic” reinforcement learning environments’ renderers. However, reimplementing this via a lower-leveled approach would not be too difficult. For reference, OpenAI used pyglet as their window creating tool.

Self-Play

One of the focuses of this project was how to optimize training of the agents. Through this we discovered curriculum learning and self-play as means of varying the training regimen. Though we were able to implement both, only curriculum learning was able to really be used as an improvement tactic due to time constraints. However, we were

able to conduct a simple experiment with self-play on an older D3QN model that, due to being able to repeat actions and a poor implementation of PER, had trouble moving within the environment.

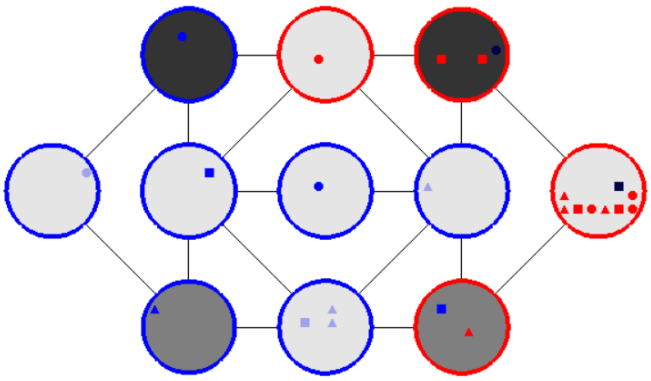
The way that self play was introduced to the agent was through a single-learner regimen, where only one of the two players was learning. Player A, the learner, would begin by playing an exact replica of itself. Then, after every 10 games, a copy of Player A was saved to an opponent pool. To encourage learning, most of the time the current copy of the agent is used for training. However, in some cases (a percentage chosen, in this case 20% of the time) the opponent pool would uniformly select Player B from all previous copies of Player A. This is done to prevent forgetfulness.



D3QN self-play for 120 episodes, winning 53.3% overall.

In this setup, the expected wins as training progresses is expected to be some amount above 50%. Since the agent is expected to beat itself half the time and there are previous iterations weaker than the current agent chosen to battle randomly, there is some overage to be expected.

Since this regimen did not include the random actions agent, it is impossible to determine how well this particular agent would perform against random actions without battling against it. In a set of 100 matches, D3QN lost every game. This is not inherently due to self play, however. As stated previously, this agent was at a major disadvantage because of the implementation of the action selection. Even through this disadvantage, self-play allowed the agent to learn primitive movement throughout the game board.

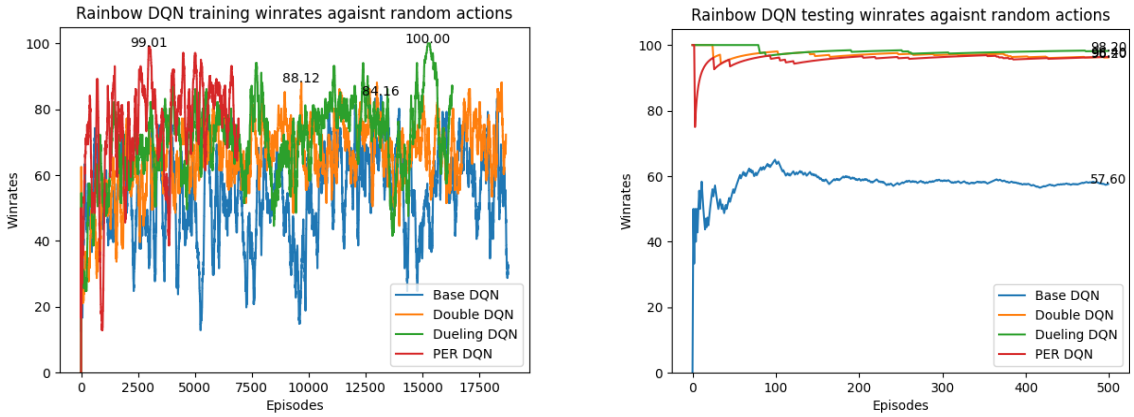


Self-play taught D3QN (red) learning movement within 100 episodes.

Self-play is still a promising learning technique that should be explored further for learning in Everglades. The original plan for self-play was to introduce it after performing curriculum learning on both random actions and opponents with discernible strategies. This would allow for knowledge of these opponents to be inherited, and push an agent that has already learned to beat those opponents further by learning its own flaws and defending against its own attacks. It may even be beneficial to introduce the original set of opponents as being part of the initial opponent pool, allowing the self-play learner to not forget how to beat them.

Results

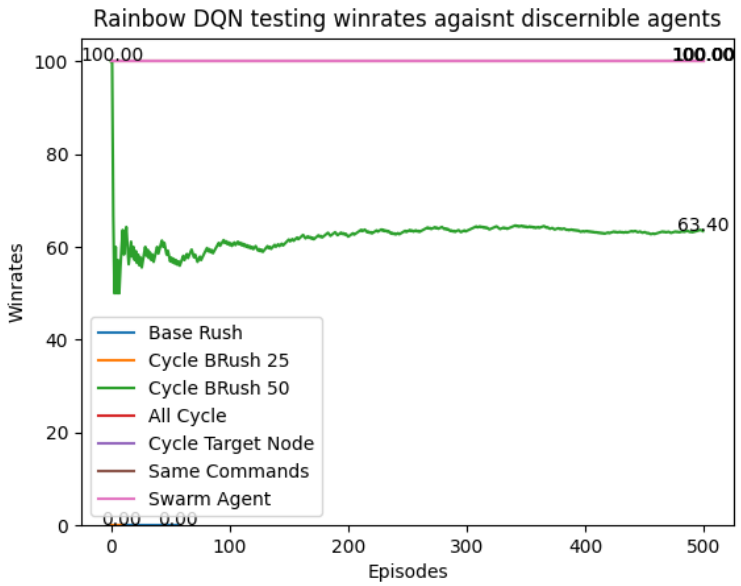
DQN



Our half rainbow DQN agents perform really well against random actions. During training, all of the 4 DQN agents we included here found peaks over 90% win rate. During testing, excluding base DQN, all of the other extensions have high win rates (>95%). Notice that we didn't include the other half of the rainbow DQN, namely Noisy Net, Multi Steps, and Distributional. The reason is because we are already satisfied with the current results with only the first 4 DQN extensions. But we also test with Noisy Net and Multi Steps, and both were producing worse results. Specifically, Multi Steps produced a 80% win rate during testing and Noisy Net had only 16% win rate during testing.

We hypothesized that the reasons for poor performance for Noisy Net was the way it's updating how the agent explores the environment. Using a Noisy Net is to make the agent learn how to explore overtime instead of using epsilon, which, theoretically, should help our DQN agents improve the win rate. However, none of our training and testing data produced good results (average below 20%).
For multi steps, the results are decent, but not offer any increase in win rate so we also didn't include it in the final version. Our hypothesis for multi steps to produce worse results is because it tries to learn multiple steps at a time, the model is more susceptible to miss the local optima.

The data in the graph is from the agents with reward shaping supports. However, for DQN, we found no difference between with and without reward shaping, except when using PER. Since PER prioritized better states, it makes sense that with a reward shaping, which quantifies how better a state is to another, PER performs much better with the support of a reward shaping method. Our non shaping PER agent only has a 56% win rate.



Our results against some given discernible strategies for DQN are not as exciting as against random actions. Our DQN agent found it impossible to defeat base rush and cycle rush at turn 25. However, with a cycle rush at turn 50, the agent started to win with a win rate of 63.40%. For other strategies, the agent found itself to win 100%.

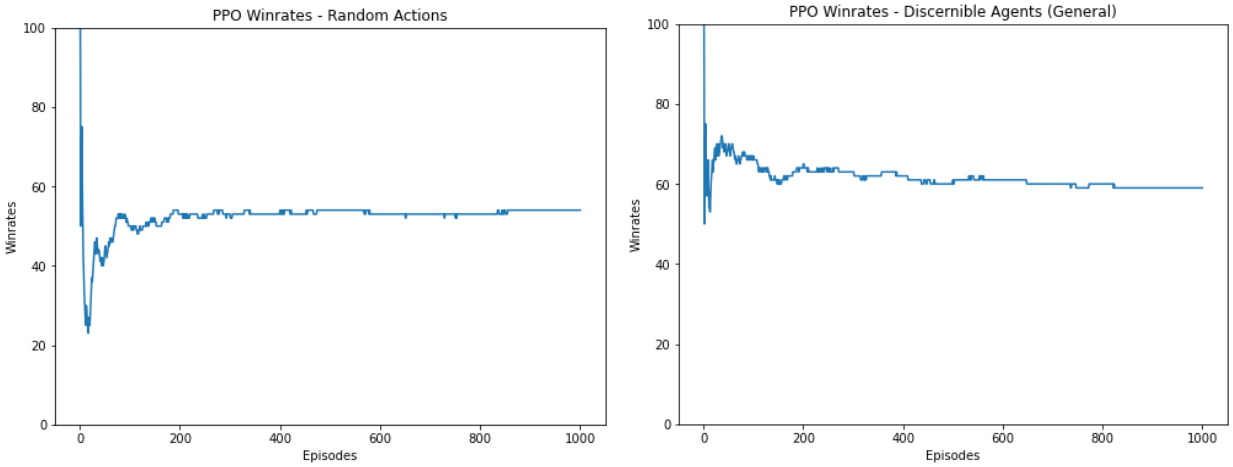
PPO

Since there was a higher demand for training against discernible agents, more efforts were put into getting the PPO agent trained efficiently against discernible agents and will implicitly be trained against random action opponents. The method used for training PPO against discernible agents is called “Cumulative Learning”, where PPO trains against weaker discernible agents and gradually trains against harder opponents.

To decide which discernible agents are “weaker” and “stronger”, all discernible agents played the game against each other for 10 games and have a “difficulty” ranking per discernible agent. For each win that the discernible agent gets, their difficulty ranking increases by one point. This leads to creating a sorted roster of discernible agents based on their difficulty rankings, and the final roster that was created is the following, from least difficult to most difficult discernible agent: same_commands, swarm_agent, all_cycle, cycle_rush (25/50), base_rushv1, and cycle_target_node11. The training will start with a random_actions opponent to explicitly have training and testing results against a random_actions opponent for the project.

After creating the roster of discernible agents, the PPO agent goes through training starting with the least-difficult opponent and works up to more difficult opponents. For each opponent in the roster, the PPO agent and the opponent train for a total of 20,000 episodes, and at the beginning of each episode, the training algorithm randomly selects between the current opponent and an opponent that PPO has previously trained against as to ensure that the training process is generalized across all opponents. The probability distribution is still skewed towards the current opponent since the agent needs sufficient training against the current opponent. After every 5,000 episodes, the agent is tested for 1,000 episodes to get the winrate percentage. If the winrate is greater than 75%, then the training against the opponent is terminated since it has found a well-defined strategy against the opponent. After each opponent, the probability skew is updated towards the next opponent.

After performing the cumulative learning with PPO, the results are as follows:



PPO winrates after conducting cumulative learning against Random Actions and Discernible Agents.

The average winrate against random_actions after 1,000 episodes is 54% and the average winrate against discernible agents is 60%. PPO struggled with finding a balance between offensive play and defensive play while ensuring that the total score is maximized at a given state. While having a winrate of 60% against discernible agents does meet our goals, we still would have preferred to have a slightly higher winrate as to have more confidence that the goal has been met.

As for a dissection of the performance for each discernible agents, the performance are shown below:

PPO winrates against each discernible agent individually.

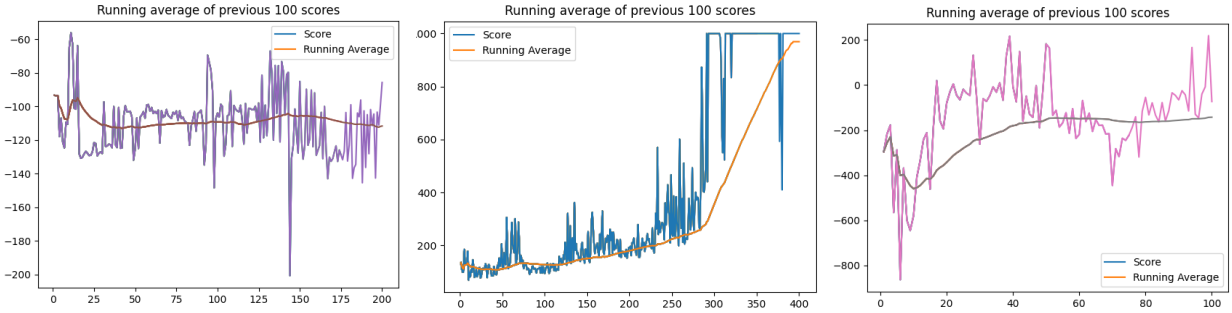
Against least-difficult opponents, such as same_commands, swarm_agent, and all_cycle, the agent successfully scored a 95% winrate whereas against harder opponents, such as cycle_rush (25/50) and base_rushv1, the winrate is around 20%. PPO inherited behaviors for defensive play at the beginning portion of the game, but after turn 30, the agent could not achieve full defensive play at the base node and thus the aggressive opponents became overwhelming for PPO to handle. We partially attribute this fault to be due to the complexity of the Defensive Play reward-shaping method, since the function had planned modifications that could not be satisfied before the project's deadline. An anomaly, however, is the performance against cycle_target_node11, which PPO achieved a 51% winrate. This could be due to PPO being able to play defense effectively before its behavior became random in the later half of the games.

SAC

During March of 2021, our primary objective was to focus on further optimization of our DQN and PPO agents, but we were also interested in reaching our stretch goal of implementing a third agent. We researched some algorithms, especially those written and described above in our "Policy Gradient Algorithms" section, and decided upon Soft Actor-Critic (SAC) as the algorithm to use for our final agent.

The first task was completely implementing SAC as it was initially presented in its debut research paper, "*Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*". This implementation uses a modified actor-critic algorithm to maximize entropy vs. reward while acting in a continuous environment.

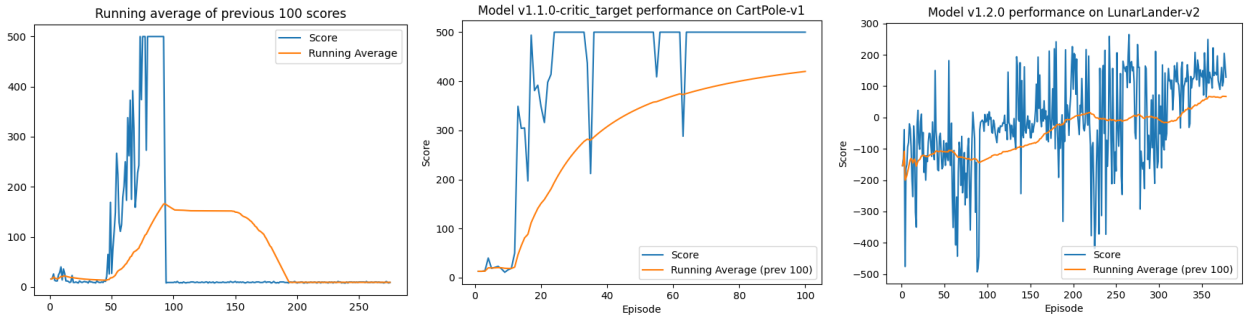
The next goal was transcribing the algorithm's usage within a continuous space to instead work within a discrete environment. This was necessary in order to work with Everglades, as the game environment is not compatible with a continuous action space. So, we first created a benchmark version of our continuous agent along with its performance on several basic, popular continuous environments provided by OpenAI Gym, with some of the results shown below:



Continuous SAC agent performance on BipedalWalker-v2 (left), InvertedPendulum-v1 (center), and LunarLanderContinuous-v2 (right).

The agent performed well enough to be considered satisfactory, even without any time spent tuning hyperparameters or other types of behavior shaping. In particular, the performance on the BipedalWalker-v2 environment would most definitely have been improved if more rewards were obtained from exploring random actions further, as that environment requires the agent to learn how to “run”, maintaining its balance whilst doing so, which requires a very complex, human-like pattern of motion input.

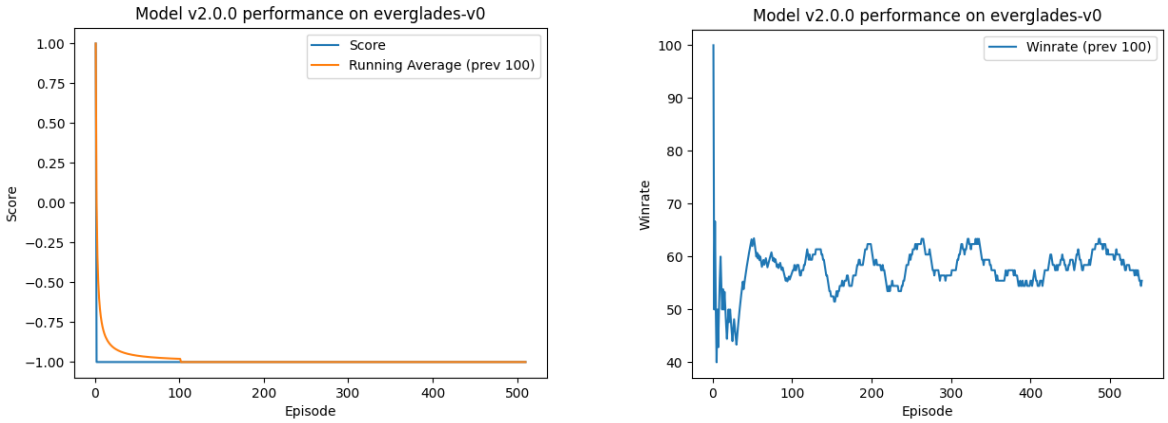
With these benchmarks recorded, we then focused on the previously mentioned transcription of continuous SAC to a discrete SAC. There were not too many resources available for this modification, although even in the original paper there was mention of the possibility of a continuous-to-discrete transformation. The resource we primarily used was another research paper dedicated specifically to this topic, aptly named “*Soft Actor-Critic for Discrete Action Settings*”. The hope was that the code bundled along with this publication could be referenced and used as a crutch when creating our new modified agent, but unfortunately a lot of it was outdated and therefore unusable. The derivations, key changes, and algorithms mentioned in the paper were still of much use, however, and we managed to complete a discrete version of SAC within a few weeks’ time. Just as we did before, we created a benchmark version and recorded its performance against discrete environments similar to the continuous ones above:



Discrete SAC agent performance on CartPole-v1(left; initial performance), CartPole-v1 (center; corrected), and LunarLander-v2 (right).

A primary issue we faced when creating the discrete SAC agent was how to deal with the new probability distributions being generated by our policy network. Our initial implementation led to agent instability upon learning optimal behavior as action probabilities were unclipped and some would begin approaching zero. There's no issue with that in particular, but the agent's loss function also uses log probabilities, and that leads to undefined behavior as $log(0)$ results in a "nan" value which will propagate throughout a network's weights if gone unchecked (see the first performance graph). Once this issue was fixed, we saw the first true success of our discrete SAC implementation! It seemed to be performing just about as well as the continuous version, but was now compatible with discrete action spaces. So now the final step was to implement this agent within Everglades' gym environment.

By this point, we had found ourselves in a time crunch, and any time spent further developing SAC was essentially taking away time from preparing for the presentation of our completed project. Nevertheless, development continued as we wanted to get our third agent to a decent state in order to present it, otherwise all the hard work put into it would be for naught. Once adapted to the Everglades' environment, our first test run was a complete failure, but improvements were subsequently made after debugging and locating the performance issues. The final performance of our SAC agent is display below:



Discrete SAC initial failure in Everglades (left) and final training results (right).

In particular, there are a few improvements that still need to be made in order for SAC to be considered as satisfactory. The first improvement needed is a method of sampling the policy network's output probability distribution without replacement, as TensorFlow (the framework used to develop SAC) does not include this functionality by default—ideas were bounced around about converting to a NumPy array and sampling with their framework instead, but this had some conflicts with the tensor output of the policy network. Another improvement would be the implementation of a legal move mask, as described previously in this paper under the "Legal Moves Masking" section; the agent currently can select any arbitrary action it believes will result in the best reward, but that action may very well be illegal at the current state and so will not be performed and therefore will not net any reward. With these improvements made, we are confident the agent's performance will either come close to or rival that of our Double DQN agent, as the implementations would be similar, just with a slightly different objective.

Citations and References

[A1] T. Yiu, “Understanding Neural Networks,” Medium, 04-Aug-2019. [Online]. Available: https://towardsdatascience.com/understanding-neural-networks-19020b758230. [Accessed: 21-Oct-2020].

[A2] S. D. Silva, “A Beginners Guide to Neural Nets,” Medium, 23-Mar-2020. [Online]. Available: https://towardsdatascience.com/a-beginners-guide-to-neural-nets-5cf4050117cb. [Accessed: 21-Oct-2020].

[A3] “The front page of A.I.,” DeepAI. [Online]. Available: https://deepai.org/. [Accessed: 21-Oct-2020].

[A4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” Advances in Neural Information Processing Systems, 2012. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks. [Accessed: 21-Oct-2020].

[A5] J. Le, “The 8 Neural Network Architectures Machine Learning Researchers Need to Learn,” KDnuggets, Feb-2018. [Online]. Available: https://www.kdnuggets.com/2018/02/8-neural-network-architectures-machine-learning-researchers-need-learn.html. [Accessed: 21-Oct-2020].

[A6] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way,” Medium, 17-Dec-2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed: 21-Oct-2020].

[A7] T. Matias, F. Souza, R. Araújo, and C. H. Antunes, “Learning of a single-hidden layer feedforward neural network using an optimized extreme learning machine,” Neurocomputing, vol. 129, pp. 428–436, 2014.

[A8] G. Ciaburro and B. Venkateswaran, “Neural Networks with R,” Packt, Sep-2017. [Online]. Available: https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788397872/1/ch01lvl1sec12/weights-and-biases. [Accessed: 21-Oct-2020].

[A9] “Training, validation, and test sets,” Wikipedia, 21-Oct-2020.

[A10] A. Tang, R. Tam, A. Cadrin-Chênevert, W. Guest, J. Chong, J. Barfett, L. Chepelev, R. Cairns, J. R. Mitchell, M. D. Cicero, M. G. Poudrette, J. L. Jaremko, C. Reinhold, B. Gallix, B. Gray, R. Geis, T. O'connell, P. Babyn, D. Koff, D. Ferguson, S. Derkatch, A. Bilbily, and W. Shabana, “Canadian Association of Radiologists White Paper on Artificial Intelligence in Radiology,” Canadian Association of Radiologists Journal, vol. 69, no. 2, pp. 120–135, 2018.

[A13] “Overfitting,” Wikipedia, 21-Oct-2020.

[A14] L. Weng, “Policy Gradient Algorithms,” Lil'Log, 08-Apr-2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html. [Accessed: 21-Oct-2020].

[A15] Mnih, V., “Asynchronous Methods for Deep Reinforcement Learning”, arXiv e-prints, 2016. IEEE.

[A16] “Newton GPU Cluster,” UCF Advanced Research Computing Center. [Online]. Available: https://arcc.ist.ucf.edu/status/newton.html. [Accessed: 04-Dec-2020].

[A17] R. P. Wiegand, “Job Submission within the ARCC,” UCF Advanced Research Computing Center, 02-May-2018. [Online]. Available: https://arcc.ist.ucf.edu/index.php/help/tutorials/job-submission-within-the-arcc. [Accessed: 04-Dec-2020].

[A18] V. Pfau, “Gym Retro,” OpenAI, 02-Sep-2020. [Online]. Available: https://openai.com/blog/gym-retro/. [Accessed: 04-Dec-2020].

[A19] “A toolkit for developing and comparing reinforcement learning algorithms,” OpenAI. [Online]. Available: https://gym.openai.com/envs/. [Accessed: 04-Dec-2020].

[A20] C. Nicholson, “Comparison of AI Frameworks,” Pathmind, 2020. [Online]. Available: https://wiki.pathmind.com/comparison-frameworks-dl4j-tensorflow-pytorch. [Accessed: 06-Dec-2020].

[A21] SciPy community, “What is NumPy?,” NumPy v1.19 Manual, 29-Jun-2020. [Online]. Available: https://numpy.org/doc/stable/user/whatisnumpy.html. [Accessed: 06-Dec-2020].

[B1] “Understanding LSTM Networks”, 2015 [Online]. Available: https://web.stanford.edu/class/cs379c/archive/2018/class_messages_listing/content/Artificial_Neural_Network_Technology_Tutorials/OlahLSTM-NEURAL-NETWORK-TUTORIAL-15.pdf

[B2] Akshay Sood, “Long Short Term Memory Lecture” [Online]. Available: http://pages.cs.wisc.edu/~shavlik/cs638/lectureNotes/Long%20Short-Term%20Memory%20Networks.pdf

[B3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, “Dota 2 with Large Scale Deep Reinforcement Learning”, 2019 [Online]. Available: https://arxiv.org/pdf/1912.06680.pdf

[B4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”, 2018

[B5] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, “Noisy Network for exploration”, 2019 [Online]. Available: https://arxiv.org/pdf/1706.10295.pdf

[BL1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” arXiv.org, 05-Jul-2019. [Online]. Available: https://arxiv.org/abs/1509.02971. [Accessed: 01-Dec-2020].

[BL2] L. Weng, “Policy Gradient Algorithms,” Lil'Log, 08-Apr-2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html. [Accessed: 21-Oct-2020].

[BL3] “Project Everglades AI Developer's Guide v1.4.” 2019.

[BL4] R. S. Sutton and A. Barto, Reinforcement learning: an introduction. Cambridge, MA: The MIT Press, 2018.

[BL5] S. Karagiannakos, “The idea behind Actor-Critics and how A2C and A3C improve them,” AI Summer, 17-Nov-2018. [Online]. Available: https://theaisummer.com/Actor_critics/. [Accessed: 21-Oct-2020].

[BL6] T. Lindbloom, “Project Everglades Overview.” Orlando, 20-Jul-2020.

[BL7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” arXiv.org, 05-Jul-2019. [Online]. Available: https://arxiv.org/abs/1509.02971. [Accessed: 01-Dec-2020].

[BL8] M. Ashraf, “Reinforcement Learning Demystified: Exploration vs. Exploitation in Multi-armed Bandit setting.,” Medium, 04-Dec-2018. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-demystified-exploration-vs-exploitation-in-multi-armed-bandit-setting-be950d2ee9f6. [Accessed: 02-Dec-2020].

[J1] Z. Salloum, "TD In Reinforcement Learning, The Easy Way," 27-Nov-2018. [Online]. Available: https://towardsdatascience.com/td-in-reinforcement-learning-the-easy-way-f92ecfa9f3ce. [Accessed 5 December 2020].

[J2] K. Czarnogorski, "Monte Carlo Tree Search - Beginners Guide," 24-Mar-2018. [Online]. Available: https://int8.io/monte-carlo-tree-search-beginners-guide/. [Accessed 5 December 2020].

[J3] A. G. Barto, "Temporal difference learning," 19-Nov-2007. [Online]. Available: http://www.scholarpedia.org/article/Temporal_difference_learning#targetText=An. [Accessed 5 December 2020].

[J4] R. Roy, "ML | Monte Carlo Tree Search (MCTS)," 14-Jan-2019. [Online]. Available: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/. [Accessed 5 December 2020].

[J5] O. Vinyals et al., "AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning," 30-Oct-2019. [Online]. Available: https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning. [Accessed 5 December 2020].

[J6] O. Vinyals et al., "Grandmaster level in StarCraft II using multi-agent reinforcement learning," Nature., Vol 575, pp. 350-369. 30-Oct-2019. [Accessed 5 December 2020].

[J7] A. L. Aradhya, "Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)," 12-May-2019. [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/. [Accessed 5 December 2020].

[J8] "Minimax search and Alpha-Beta Pruning." [Online]. Available: https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm. [Accessed 5 December 2020].

[J9] K. Arulkumaran, A. Cully, and J. Togelius, "AlphaStar: An Evolutionary Computation Perspective," GECCO '19., pp. 1-3. 13-Jul-2019. [Accessed 5 December 2020].

[J10] G. Chaslot et al., "Monte-Carlo Tree Search: A New Framework for Game AI," Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 216-217. [Accessed 5 December 2020].

[J11] J. Schulman et al., "Proximal Policy Optimization Algorithms," pp. 1-12. 28-Aug-2017. [Accessed 5 December 2020].

[J12] J. Schulman et al., "Trust Region Policy Optimization," pp. 1-16. 20-Apr-2017. [Accessed 5 December 2020].

[J13] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, et al. 2017. “Population based training of neural networks”. arXiv preprint arXiv:1711.09846 (2017). [Accessed 5 December 2020].

[J14] M. Nowostawski, R. Poli. 1999. “Parallel genetic algorithm taxonomy”. In KES. pp. 88–92. [Accessed 5 December 2020].

[J15] G. Syswerda. 1991. “A study of reproduction in generational and steady-state genetic algorithms.” In Foundations of Genetic Algorithms. Vol. 1. 94–101. [Accessed 5 December 2020].

[J16] Billings, D.; Davidson, A.; Schaeffer, J.; and Szafron, D. 2002. “The challenge of poker.” AI Vol 134(1). pp. 201–240. [Accessed 5 December 2020].

[J17] Sheppard, B. 2002. World-championship-caliber scrabble. Artificial Intelligence Vol 13(1). pp. 241–275. [Accessed 5 December 2020].

[J18] "Reducing Loss: Gradient Descent", 10-Feb-2020. [Online]. Available: https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent. [Accessed 5 December 2020].

[J19] D. Pathak, P. Agrawal, A. Efros, T. Darrell, “Curiosity-driven Exploration by Self-supervised Prediction”, ICML 2017.

[ML1] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, “Algorithms for Hyper-Parameter Optimization”, nips.cc. [Online]. Available: https://papers.nips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf. [Accessed: 30-Nov-2020].

[ML2] A. Arasanipalai, “How to make Deep Learning Models that Don’t Suck”, 2018. [Online]. Available: https://nanonets.com/blog/hyperparameter-optimization/. [Accessed: 29-Nov-2020].

[ML3] “Classical Conditioning”, Wikipedia. [Online]. Available:]https://en.wikipedia.org/wiki/Classical_conditioning. [Accessed: 16-Oct-2020].

[ML4] F. Woergoetter, B. Porr, “Reinforcement Learning”, 2008. [Online]. Available: http://www.scholarpedia.org/article/Reinforcement_learning. [Accessed: 19-Sep-2020].

[ML5] D. Soni, “Introduction to Markov Chains”, Medium, 05-Mar-2018. [Online]. Available: https://towardsdatascience.com/introduction-to-markov-chains-50da3645a50d. [Accessed: 30-Oct-2020].

[ML6]“Markov Decision Process”, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Markov_decision_process. [Accessed: 31-Oct-2020].

[ML7] “Reinforcement Learning”, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Reinforcement_learning. [Accessed: 31-Oct-2020].

[ML8] R. Givan, R. Parr, “An Introduction to Markov Decision Processes”. [Online]. Available: https://www.cs.rice.edu/~vardi/dag01/givan1.pdf. [Accessed: 02-Nov-2020].

[ML9] “Frozen Lake Environment”, OpenAI.com. [Online]. Available: https://gym.openai.com/envs/FrozenLake-v0/. [Accessed: 7-Oct-2020].

[ML10] “States, Observations and Action Spaces in Reinforcement Learning”, Medium, 19-Apr-2020. [Online]. Available: https://medium.com/swlh/states-observation-and-action-spaces-in-reinforcement-learning-569a30a8d2a1. [Accessed: 19-Oct-2020].

[ML11] A. Ng, D. Harada, S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping”, 1999. [Online]. Available: http://luthuli.cs.uiuc.edu/~daf/courses/games/AIpapers/ml99-shaping.pdf. [Accessed: 01-Dec-2020].

[ML12] L. Mao, “On-Policy vs. Off-Policy in Reinforcement Learning”. [Online]. Available: https://leimao.github.io/blog/RL-On-Policy-VS-Off-Policy/. [Accessed: 03-Oct-2020].

[ML13] “What is the Difference Between Off-Policy and On-Policy Learning”, stackoverflow.com, 2015. [Online]. Available: https://stats.stackexchangse.com/questions/184657/what-is-the-difference-between-off-policy-and-on-policy-learning. [Accessed: 03-Oct-2020].

[ML14] J. McCullock, “A Painless Q-learning Tutorial”. [Online]. Available: http://www.mnemstudio.org/path-finding-q-learning-tutorial.htm. [Accessed: 26-Sep-2020].

[ML15] A. Choudhary, “A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python”, analyticsvidhya.com, 18-Apr-2019. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. [Accessed: 29-Sep-2020].

[ML16] R. Sutton,D. McAllester, S. Singh, Y. Mansour, “Policy Gradient Methods for Reinforcement Learning with Function Approximation”, nips.cc. [Online]. Available: https://papers.nips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf. [Accessed: 13-Oct-2020].

[ML17] D. McNeela, “The Reinforce Algorithm AKA Monte-Carlo Policy Differentiation”, 18-Apr-2018. [Online]. Available: https://mcneela.github.io/math/2018/04/18/A-Tutorial-on-the-REINFORCE-Algorithm.html. [Accessed: 15-Oct-2020].

[ML18] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”, arXiv.org. [Online]. Available: https://arxiv.org/pdf/1706.02275v4.pdf. [Accessed: 02-Dec-2020].

[ML19] L. Busǫniu, R. Babuska, and B. De Schutte, “Multi-agent reinforcement learning:An overview,” [Online]. Available: http://www.dcsc.tudelft.nl/~bdeschutter/pub/rep/10_003.pdf. [Accessed: 03-Dec-2020].

[ML20] D. MacLaurin, D. Duvenaud, R. Adams, “Gradient-based Hyperparameter Optimization through Reversible Learning”, arXiv.org, 2015. [Online]. Available: https://arxiv.org/pdf/1502.03492.pdf. [Accessed: 05-Dec-2020].

[ML21] S. Lee, J. Kim, X. Zheng, Q. Ho, G. Gibson, E. Xing, “On Model Parallelization and Scheduling Strategies for Distributed Machine Learning”, nips.cc, 2014. [Online]. Available: https://papers.nips.cc/paper/2014/file/7d6044e95a16761171b130dcb476a43e-Paper.pdf. [Accessed: 06-Dec-2020].

[ML22] “Minimax”, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Minimax. [Accessed: 06-Dec-2020].