Deep Reinforcement Learning Coursework

1. Environment Creation

1.i. Environment Description:

The environment that I have designed is based on the video game – Pokémon. In my version, the agent has a team of one Pokémon – Pikachu - that they have to use to defeat 5 Pokémon belonging to the opposing trainer.

The trainer's "battle" each other by selecting moves, which then damage the opposing Pokémon. Damage is calculated based on each Pokémon's stats, which are scaled with the Pokémon's level, and type-effectiveness is taken into account – for example, a fire-type attack would deal twice the normal damage against a grass Pokémon (i.e., it is "super effective"), but only half against a water Pokémon.

Stochasticity is built-in by two methods. Firstly, each attack has a level of accuracy associated with it, where moves miss if a random number generated is above the accuracy threshold – when this happens, the move chosen does zero damage. Secondly, for every attack, there is a random chance of a critical hit (dealing roughly double the normal damage). This chance is based on the Pokémon's speed statistic and is on average ~30%.

There are two difficulty modes included as I wanted to test how the performance of Q Learning changed as the number of states and the complexity of the problem increased. The details of how these two environments differ are as follows:

1.ii. Basic Environment:

- ***Environment:*** The trainer's Pokémon is invincible and cannot take damage. In this environment the goal is to defeat the opposing Pokémon in the least possible moves.
- ***Action choices:*** 4 moves (actions) are possible.
    - Focus Energy: Increases the attack stage of Pikachu by 1, up to a maximum of 5 stages. For each stage, the multiplier to Pikachu's attack is increased by 0.5x, from 1.0x (0 stages) to 3.5x (5 stages).
    - Mirror Move: Copies the move used by the opposing trainer.
    - Thunderpunch: A thunder-type attack that deals 50 damage with 90% accuracy.
    - Psychic: A psychic-type attack that deals 70 damage with 90% accuracy.
- ***Pikachu*:** Level 25 (against level ~60 opponents) to ensure this task isn't too trivial and victory can be dramatically sped up by smart play (i.e., boosting Pikachu's attack and using super effective moves).
- ***Opposing Trainer:*** The opposing trainer only has one move to choose from each turn.
- ***Termination:*** When all the opposing Pokémon are defeated, or the agent reaches the maximum moves limit (100).

- ***Rewards:*** There is a -1 reward for each move taken, to discourage slow, inefficient strategies. The agent receives +5 reward every time an opposing Pokémon is defeated, and +100 when all are defeated.
- ***States:*** There are 330 different states; 5 according to the number of opposing Pokémon left to defeat, 6 according to the current attack boost of Pikachu (from 0-5), and 11 referring to the health of each opposing Pokémon (10% buckets from 100-1% and one more bucket for 0% - i.e., defeated).

1. iii. Intermediate Environment:

Unless otherwise specified, these attributes are in addition to those found in the basic environment (1.ii.).

- ***Environment:*** In this environment it is possible for the opposing Pokémon to deal damage to Pikachu. The goal is to defeat all the opposing Pokémon without being defeated.
- ***Action Choices:*** 1 additional possible move.
    - Full Restore (HP Potion): This restores Pikachu's health to its initial value.
- ***Pikachu:*** The level was increased to 70 to ensure Pikachu had enough health and defense to withstand a few hits from the opposing Pokémon before being defeated. Full restores can then be leveraged by the agent to help prolong and/or win the game.
- ***Termination:*** The environment now also terminates when Pikachu is defeated.
- ***Rewards:*** There is now also a -100 reward if Pikachu is defeated.
- ***States:*** There are 3,630 states; 330 are the same as the Basic environment, multiplied by 11 health buckets corresponding to Pikachu.

2. Q-LEARNING IMPLEMENTATION

2. i. Algorithm Description

Q-learning is an example of off-policy learning. It aims to learn the optimal policy (π) by taking actions determined by different policy (μ). In order to do this, a q-table is constructed reflecting the decision space of the environment, defined by the number of possible states and actions – Q [S, A].

The q-values in the table are initialized at zero and updated as the agent interacts with the environment. The q-value, $Q [S_t, A_t]$, is the expected total reward the agent would receive if they started in state, S_t, took action, A_t, and then followed that policy for all subsequent actions.

Updates to the q-values occur through temporal difference learning, whereby bootstrapping allows the agent to learn from incomplete episodes – in this case the agent updates values after every action. This approach has several advantages including being more efficient, working in environments that do not terminate and being lower variance compared to Monte Carlo methods.

The process of q-learning is set out below:

- The next action (A_{t+1}) is chosen according to the behaviour policy μ.
- The reward that would have been received if the action (A′) was chosen by the optimal policy (π), is also calculated.
- The q-table is updated towards the value of the optimal policy's action using the following formula:

$$Q\ (S_t, A_t) = Q\ (S_t, A_t) + \alpha\ (R_{t+1} + \gamma\ \max_a Q\ (S_{t+1}, A') - Q\ (S_t, A_t))^1$$

- Where α is the learning rate of the algorithm and γ is the discount rate.

Policy(π) acts greedily with respect to Q (s, a), meaning it always takes the action with the highest expected reward. Whereas Policy(μ) acts epsilon-greedily whereby the value of epsilon dictates

the likelihood of the policy to select a random action, otherwise it acts greedily. This method helps to ensure some exploration of the environment, and the value of epsilon typically decays over episodes to reduce the exploration as the optimal policy converges towards Q*(s, a).

2.ii. Q-Learning Environment Performance

Hyperparameters:

- ***Epsilon*** – This influences the level of exploration that the behaviour policy (μ) follows. A value closer to 1 encourages the agent to select actions at random more frequently than a lower value. Epsilon value will always start at 1 in my experiments.
- ***Epsilon Decay*** – Dictates how fast the value of epsilon will decay towards 0 over time. A value closer to 1 will ensure that epsilon decays more slowly and therefore there will be more exploration through taking random actions.
- ***Alpha*** – The learning rate determines the step size taken by the algorithm towards new observed values. A larger value of alpha increases this step size and can decrease convergence time; however, it can also increase variance and the chance of converging to a suboptimal solution.
- ***Gamma*** – The discount factor is applied to future rewards; this helps to incorporate the cost of future uncertainty into the current value estimate. A lower value of gamma will discount future rewards more highly (i.e., they will be worth less at time t).

Evaluation Graphics:

I use 3 main graphics to visualize differing performance for Task 2, below are descriptions of what each one represents.

- ***State-Value Matrix:*** A matrix representing the possible states in the environment and the expected value assigned to them by the Q-learning algorithm post-training. A brighter value indicates greater expected reward from being in that state.
- ***Q-table:*** This shows the scaled q-values associated with each action according to the relevant state. A brighter value indicates a greater reward is expected from that state-action combination.
- ***Line Plots:*** Showing the mean and standard deviation of reward throughout training – these values were collected by running the algorithm 100 times using a greedy policy at specified time intervals during training. Super effective moves used and the turn it took Pikachu to reach its maximum attack boost were also tracked. This is a measure of the "intelligence" of the solution as an increase in super effective moves shows it has learnt to exploit the opposing Pokémon's weaknesses, whilst it is most beneficial to use the attack boosts early in the game as then their effect lasts the longest.

Below are charts showing the impact of altering each hyperparameter. In each case multiple values were tested to find the optimal settings, however, only a few are plotted to demonstrate the impact across the range of values searched.

Epsilon Decay Impact (Gamma = 0.7, Alpha = 0.1, Environment = Intermediate)

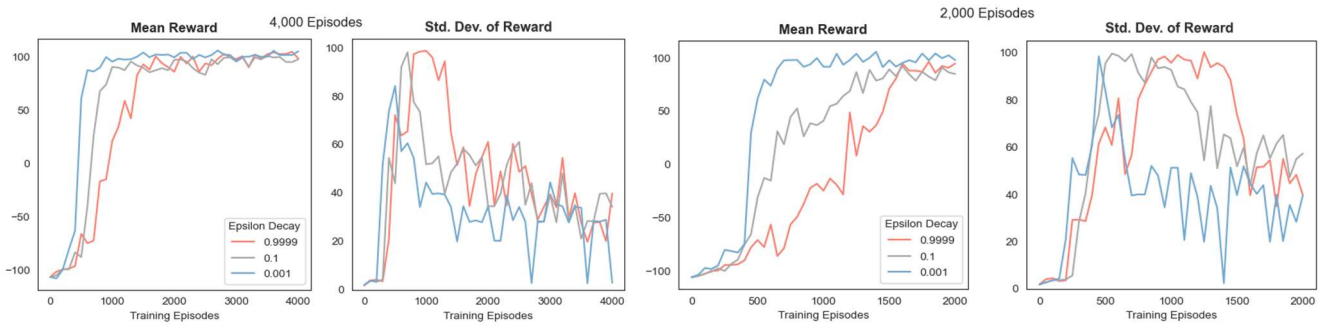

Figure 1: Line plots showing mean and variance of reward using 3 different epsilon decay values across 2,000 and 4,000 episodes

For this problem, a lower value of epsilon decay leads to a faster convergence to the optimum on average. The above chart shows that the relative lack of random exploration doesn't negatively affect the algorithm's ability to improve. After ~400 episodes a lower decay value means that actions are picked greedily more frequently compared to higher decay values which take longer to converge as they are still attempting more random actions. This can be seen in both the speed to reach maximum reward (~90) and lower variance across episodes.

Alpha Impact (Gamma – 0.7, Epsilon Decay = 0.9, Episodes = 2,000)

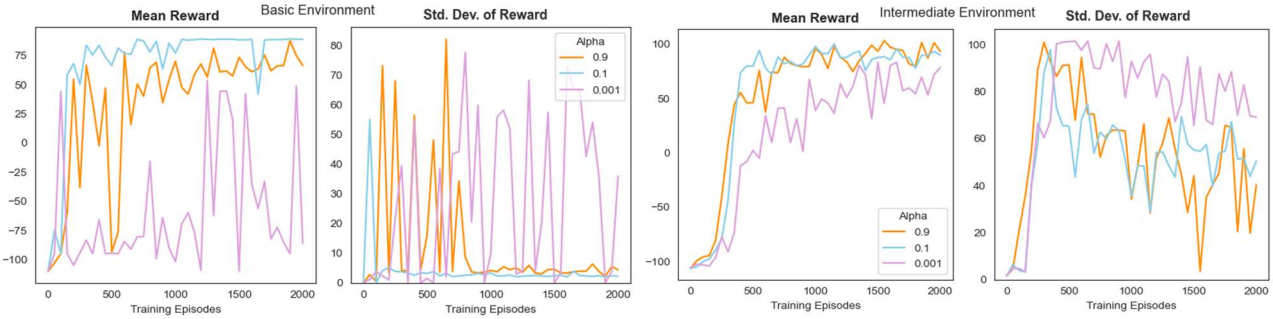

Figure 2: Mean and variance using 3 different alpha values across the basic and intermediate environment setups

The impact of alpha, the learning rate, can be seen most effectively in the less complex – Basic – environment. This shows the importance of setting an appropriate learning rate (i.e., 0.1 for this problem).

A rate that's too high (0.9 above) can cause the model to change too quickly towards a suboptimal solution and the left chart shows the 0.9 rate has converged with low variance to a solution that gets a lower average reward than the 0.1 rate. Conversely, a rate of 0.001 is too low for the Basic environment and the small step size may be causing the algorithm to get stuck at local minima, resulting in worse average performance and much higher variance.

Gamma Impact (Alpha – 0.1, Epsilon Decay = 0.9, Episodes = 2,000, Environment = Basic)

Gamma has a lower overall impact on performance in this environment than both alpha and epsilon decay. However, Figure 3 does show that a low gamma value can cause higher variance in the model as the environment is a linear one and the positive values do not filter back as far, resulting in less certain decision-making in the early states.

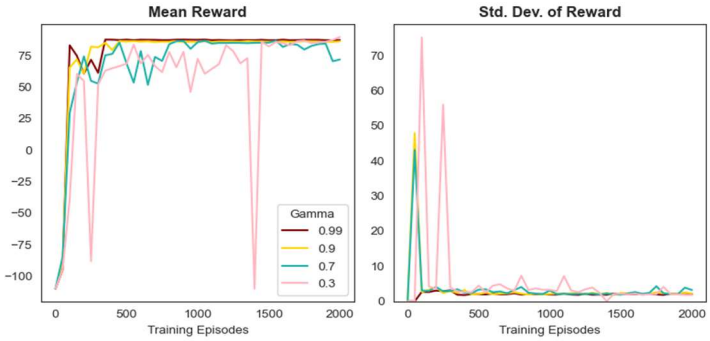

Figure 3: Mean and standard deviation using 4 different gamma values in the Basic environment.

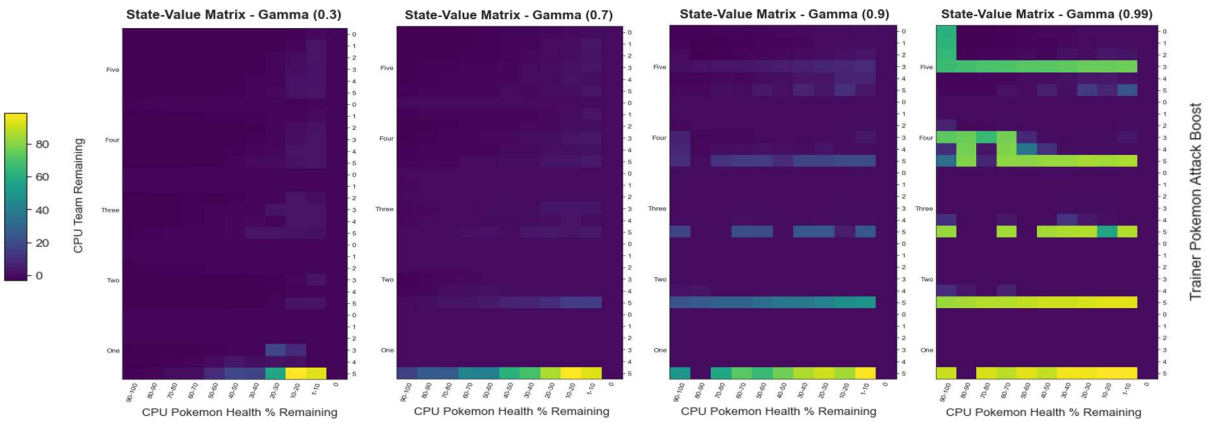

Figure 4: State-Value Matrix for the 4 different gamma values

How these values are propagated back through the states can be seen in Figure 4 where the agent starts in the top left corner which equates to – 5 opposing Pokémon remaining, all at 90-100% health, and Pikachu has 0 attack boost. The agent graduates towards the bottom right, where the last opposing Pokémon is reduced to zero health. A gamma of 0.99 ensures that the positive reward filters all the way back to the agent's initial state, while a gamma of 0.3 only carries back across 5-6 states.

Performance Study at Two Difficulty Levels:

Basic Environment – *Epsilon Decay (0.001), Alpha (0.1), Gamma (0.9), Episodes (1,000)*

The mean reward obtained stabilizes at the maximum value (~90) after ~600 episodes, where the standard deviation also stabilizes at ~1.5 points per run.

The algorithm quickly learns that super effective moves increase the victory speed. However, after 400 episodes, it also learns that quicker times can be reached by immediately using focus energy 4 times to reach maximum attack by turn 4 and then less moves are required to win.

Q-learning outperforms a random policy substantially (see Figure 6), obtaining a 48% better mean reward, with 75% lower standard deviation. Further evidence of the algorithm learning the nature of the game can be seen in Figure 7 whereby the Q-values show the highest perceived valued action for each state.

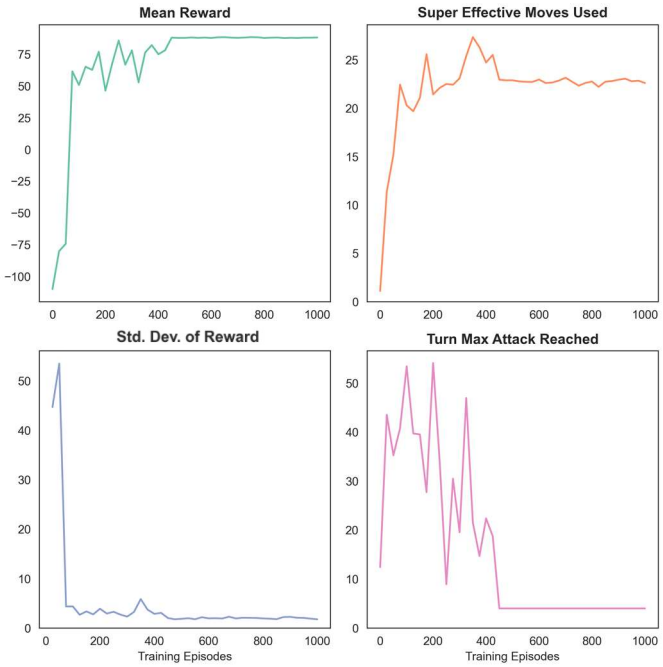

Figure 5: Line plots showing key stats tracked across episodes

Model	Mean Reward	Std. Dev Reward	Mean Super Effective Moves Used	Mean Turn Max Attack
Q-Learning	89.15	1.61	22.89	4.0
Random	60.18	6.46	16.13	19.61

Figure 6: Comparative results from 100 episodes using; a random policy, and Q-Learning after 1,000 episodes training

The Q-table demonstrates the learning that has happened by assigning higher q-values to specific state-action pairs. Focus Energy is most valuable used in the earlier states. It has also learnt to use; mirror move to copy Dragonite's dragon-type attack (which is super-effective against Dragonite), Thunderpunch versus the water (Lapras) and flying (Charizard) Pokémon, and Psychic against Gengar which is weak to psychic-type attacks. Against Machamp there is no strictly optimal action and the Q-table reflects this.

In this example some states are not visited as they are quickly deemed sub-optimal. For example, the attack boost is particularly important to use early and so Pikachu never encounters the last opposing Pokémon without using Focus Energy at least once beforehand.

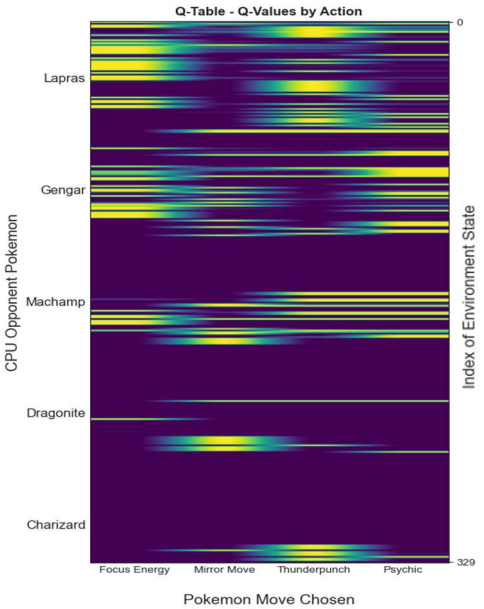

Figure 7: Q-Table showing values (scaled between 0 and 1) assigned to each state-action pair after 1000 episodes of Q-Learning

Intermediate Environment – *Epsilon Decay (0.9999), Alpha (0.1), Gamma (0.9), Episodes (5,000)*

With the added complexity caused by adding an extra possible move, and the stochasticity of the damage caused by the opposing Pokémon, the Intermediate environment took roughly 4 times to number of episodes to converge to the optimal mean reward.

Epsilon decay was increased to 0.9999 as, although convergence to an optimal mean reward was faster with a low decay value, the standard deviation of rewards converged to an optimal minimum more often in my tests with a higher value of decay. This is likely because this allowed for more exploration to occur after many training episodes, enabling the algorithm to continue to improve, lowering variability.

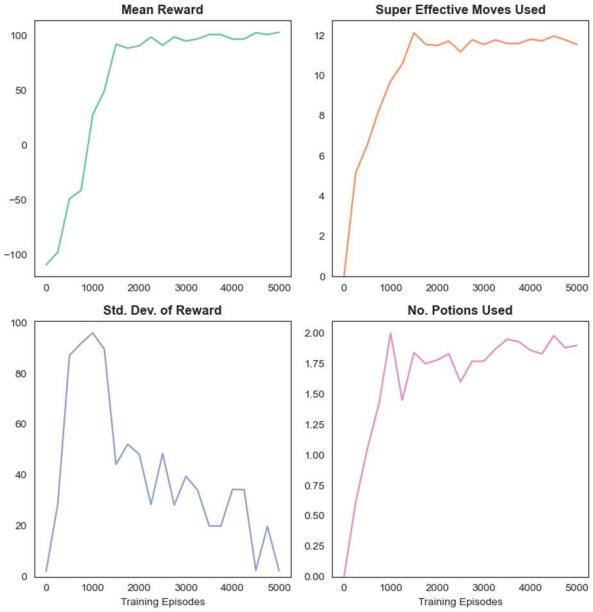

Figure 8: Line plots showing key statistics tracked across episodes

Despite this, some variability remained even after 10,000 episodes and across epsilon decay values (see Figure 10). This potentially reveals some of the limitations of the basic nature of Q-Learning when dealing with environments with stochasticity.

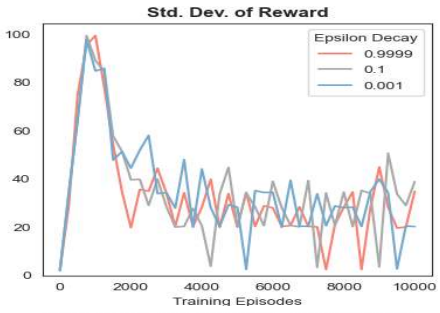

Figure 10: Std. Dev. Variability – 10,000 episodes

Model	Mean Reward	Std. Dev. Reward	Super Effective Moves	No. Potions Used
Q-Learning	90.53	2.81	11.61	2.93
Random	-57.19	85.58	4.37	3.66

Figure 9: Comparative results from 100 episodes using; a random policy, and Q-Learning after 5,000 episodes training

As can be seen from Figure 9, a random agent performs far worse on the Intermediate environment, confirming the increase in difficulty. Interestingly, the number of potions used is similar to the number used by the Q-Learning agent, demonstrating that correct utilization of potions at the right time is key to solving this environment.

This is backed up by the Q-Table (Figure 11) which shows similar trends to the Basic environment Q-Table (Figure 7), although now Full Restore potions are the most valuable action during the fight against Lapras and at the start of the fight against Dragonite.

Overall, the Q-Value-action pairs are less well-separated compared to the lower difficulty Basic environment, potentially showing one reason for the variability of results seen in Figure 10. More states have been visited in attempting to find the optimal solution (a result of the higher epsilon decay), and the optimal route is less clearly signposted by the Q-Values.

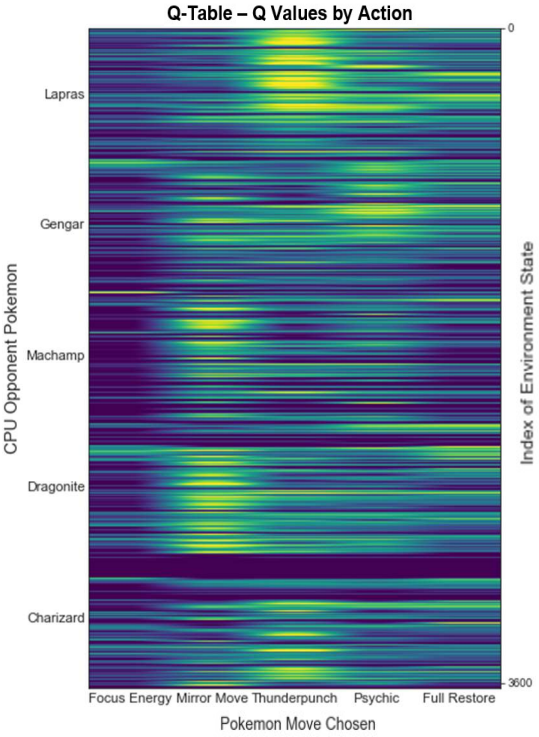

Figure 11: Q-Table showing values (scaled between 0 and 1) assigned to each state-action pair after 5,000 of Q-Learning.

3. Asynchronous Advantage Actor Critic (A3C) Implementation

3. i. Algorithm Description

Asynchronous methods were proposed by Minh et al.[3] as a solution to some of the drawbacks that they observed in experience replay. Primarily these were that experience replay was memory and computationally expensive and that off-policy learning algorithms are required in order to update based on data from an older policy.

Instead of storing an agent's data in experience replay memory, in A3C multiple agents are run in parallel on the same multi-core CPU, across multiple threads. These agents each work on their own version of the environment, thereby decorrelating the data they generate as they are likely experiencing multiple different states. The approach was found to be faster and better-performing than the GPU-based algorithms at the time, and required far less resources than setups utilizing massively distributed architectures to store memory.

By working on a single CPU, in addition to the reduced communication costs of sending parameter and gradient updates between workers, this also enables "Hogwild!"[4] updates to be made. Hogwild! allows processors to access a shared memory in order to update each other's work in parallel. This method drastically outperformed prior methods requiring locking and synchronization.

The best performing application of these asynchronous methods suggested by Minh et al. was the asynchronous actor critic algorithm (A3C). Actor critic methods are on-policy reinforcement learning methods that aim to reduce variance in the reward signal by using a critic. The critic estimates the action-value (Q value) or state-value (V value) function and the actor then updates its policy parameters (θ) in the direction suggested by the critic[2]. The critic aims to evaluate – how good is policy π_θ for the current parameters θ? For A3C, the advantage function is used as the method for this evaluation:

$$A^{\pi}{}_{\theta}(s_t, a_t) = Q^{\pi}{}_{\theta}(s_t, a_t) - V^{\pi}{}_{\theta}(s_t)$$

The advantage tells the algorithm how much *more* reward than *usual* it will get if it takes a particular action. The basic steps taken by the A3C algorithm are as follows[5]:

- A neural network is initialized with random weights for every thread in the multi-core CPU – in my case I have 12 threads and so have 12 workers all operating different "local" neural networks.
- "N" steps are taken in each environment using policy π_θ, saving the state (given by the agent's observations), action, reward (R) and next state transitions. If done, the R is set equal to 0, otherwise $R = V(s_t)$.
- Rewards are discounted backwards, according to the gamma value ($R \leftarrow r_i + \gamma R$), from the end of the episode or the N-step batch limit.
- The policy loss (associated with the actor) and value loss (associated with the critic) are calculated as follows:
    - actor loss (i) = $-\log(\pi_{a|s}) *$ advantage
    - critic loss (i) = advantage **2
    - advantage = $(R - V(s_i))$
- The loss is calculated as the mean of the actor and critic loss and is used to update the gradients of a global model, the parameters of which are then fed back to each worker's local model. These steps are then repeated until convergence.

There are multiple benefits of using A3C compared to Q-Learning for a more complex environment. It is likely to be faster and more efficient at finding an optimal solution as it has multiple agents experiencing the environment compared to one. In a complex environment with many states, it gets extremely memory-intensive or practically impossible to maintain a state-value matrix and associated Q-tables. Furthermore, advantage estimates may be preferable to

discounted rewards, as they focus on how much *better* the rewards were than expected, and therefore can focus the algorithm's learning on the areas where it is most lacking good predictive ability.

3. ii. Environment Description

The environment has been adapted from Tasks 1 and 2 and made more complex in order to test the A3C algorithm. Two versions have been created in order to compare the difference in performance of A3C with added stochasticity, they are described in detail below.

3.iii. Advanced Environment:

- **Environment:** The goal is now to last as long as possible without being defeated, up to a maximum of 500 turns. When the five opposing Pokémon are defeated, another five are generated, in a loop until the game ends.
- **Rewards:** The agent receives a reward of 1 for every turn taken in the environment, and a reward of -1 when the episode ends.
- **Similarity to Intermediate:** Pikachu's level, the opposing Pokémon, and the moves available are all the same as the Intermediate environment.
- **Potions:** Unlike the intermediate environment, the number of Full Restore potions available are limited. The agent is given 3 at the start of the run and gains 1 every time it defeats the 5 opposing Pokémon.
- **Observations:** The agent learns through observations taken from the current state. In this environment, these observations consist of the; agent Pokémon's HP, current opposing Pokémon's HP (both as a percentage of 100), the amount of opposing Pokémon left on the opposing team, and the agent's Pokémon's current attack boost.
- **States:** Compared to the intermediate environment, the advanced environment has roughly 3 million possible states, making Q-learning a much less of an appropriate solution.

3.iv. Extreme Environment:

- **Environment:** In this environment there are now 5 different opposing trainers (one has 3 variants) who all control 5 different Pokémon. After one trainer is defeated, another is randomly generated to battle, the agent's goal remains to reach 500 turns without being beaten. After one trainer is defeated, all of the agent's Pokémon are healed to full health before fighting the next one. Super-effective moves now do 8x more damage if used by the agent and 4x more damage if used by the opposition – it is therefore crucial to avoid being hit by one of these.
- **Pokémon:** Instead of Pikachu the agent now has Venusaur, Blastoise and Charizard (Level 55), all with 4 moves of different attack-types or status effects.
- **Actions:** There are no full restores available, instead, in addition to each Pokémon's four moves, the agent has two options to switch to either of the other Pokémon that are not currently battling (if they haven't fainted).
- **Observations:** In addition to the 4 from the advanced environment, another 15 observations are provided – Categorical binary variables according to which opposing

trainer the agent is facing (7 different variants), the agent Pokémon's current defense and special attack states, and 6 binary variables revealing which of the agent's Pokémon is available for switching by using each switch action ("Switch1" or "Switch2").

- **States:** The possible states in this environment are roughly: 300 (the health percentage of the agent's 3 Pokémon) x 500 (HP of the 5 opposing team's Pokémon) x 36 (the different possible attack/defense/special stats of the Agent's Pokémon, which can now be both positive and negative) x 60 (opposing Pokémon's possible stat combinations) = 108 million.

3. *v. Case Study on A3C Performance:*

Hyperparameter Impact:

Hyperparameter combinations were tested on the extreme environment with a 200k episode sample size. This reduced search took roughly 2 hours per hyperparameter combination and so it was not possible to test all combinations until convergence was witnessed. Despite this, the results were clear enough to be able to gain information about the optimal settings. The "Episode Reward" tracked is a running mean where new observations are added to the existing mean reward with a 0.001 weighting.

Batch Size: This hyperparameter dictates the N-steps taken before an update is communicated from each local network to the global network via the loss function along with subsequent gradient updates. A lower batch size indicates the updates happen more frequently and can enable the algorithm to learn faster. However, if the batch size is too small it might not provide enough information on each update for the global model to update effectively and this can hinder convergence.

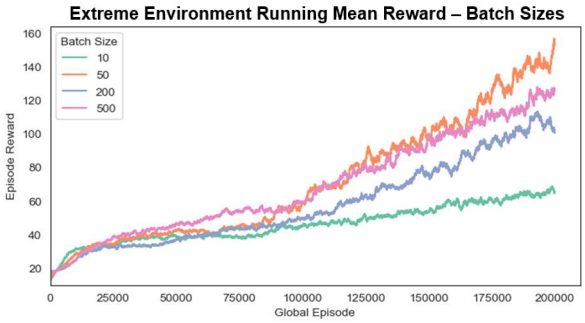

Figure 12: A3C Running Mean Episode Reward with Varying Batch Sizes
Gamma = 0.9, Learning Rate = 0.00001, No. workers = 8

Figure 12 shows the experiments performed with 4 different batch sizes in the Extreme Environment over 200k episodes. 500 is the maximum possible as otherwise the episode is guaranteed to end and communicate the loss due to the 500-turn limit. A batch size of 10 proved to be too low for the algorithm to efficiently learn, whilst above 50 the benefits from batching experiences seemed to diminish, possibly because some important information was being lost.

Learning Rate: Learning rate had a similar effect to that seen with Q-Learning. In the advanced environment, with less complexity, the algorithm was able to converge with a learning rate of 0.001. However, in the extreme environment the learning rate had to be reduced to $1e^{-4}$ or lower to show any improvement in performance. This helped ensure that the step size wasn't too large, thus avoiding converging to a sub-par optimum. In figure 13 it can be seen that there is no improvement at 0.001, whilst at $1e^{-4}$ it learns quickly and then begins to converge at a reward of ~260. The improvement is slower with $1e^{-5}$; however, the progression is steadily upward, indicating over a longer period of episodes it may be the optimal rate.

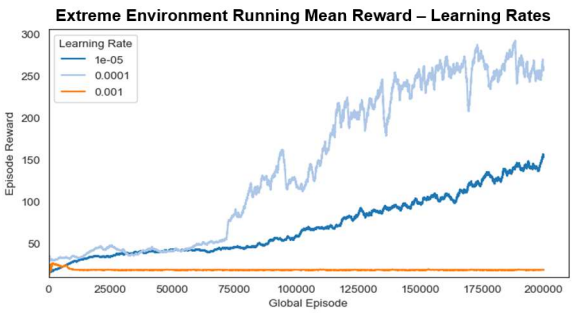

Figure 13: A3C Running Mean Episode Reward with Varying Learning Rates
Gamma = 0.9, Batch Size = 50, No. workers = 8

Number of Hidden Layers and Nodes: For the advanced environment a one-layer architecture with 128 hidden nodes was sufficient to converge to a running mean score of ~240 in 200k episodes. However, with an extra hidden layer, the algorithm converged more quickly to a higher

running mean score (see Figure 14). In the extreme environment the algorithm displayed very little learning – reaching a mean reward of 33 after 200k episodes. This was likely caused by the increase in the number of inputs (from 4 to 19) and the dramatic rise in the stochasticity of the environment, caused by the increased possible stat changes and randomized opponents. For A3C to display learning capabilities in the extreme environment it was necessary to implement two hidden layers with 512 maximum hidden nodes.

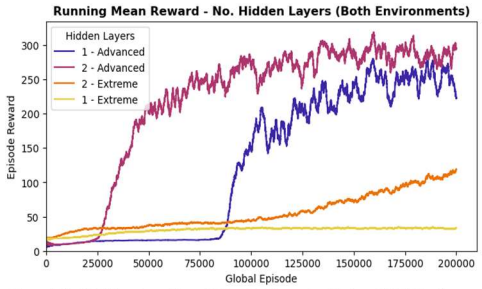

Figure 14: A3C Running Mean Episode Reward with 1 or 2 Hidden Layers
Gamma = 0.9, Batch Size = 50, No. workers = 12, LR = $1e^{-4}$ (Adv.), $1e^{-5}$ (Extr.)

Activation Function: Tanh outperformed ReLU during testing as ReLU led to results with much higher variance. It may have been the case that the weight updates were large enough to result in the dying ReLU problem, whereby the majority of output neurons may have ended up producing 0, resulting in random action choices. The problem persisted with a reduced learning rate and so Tanh was chosen as the optimal activation function for this problem.

Number of workers: The number of workers dictates how many local networks are able to asynchronously explore the environment. The upper bound is limited to the number of threads on the CPU. Figure 15 shows that the algorithm is able to improve more quickly with more workers, as more dimensions can be explored at the same time.

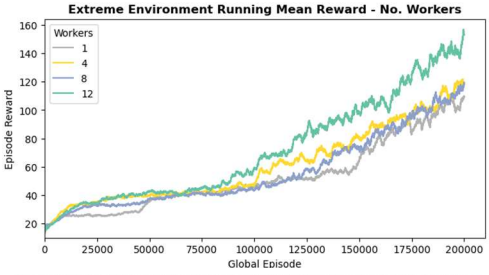

Figure 15: A3C Running Mean Episode Reward with Varying Number of Workers
Gamma = 0.9, Batch Size = 50, LR = $1e^{-5}$, 2 Hidden Layers

Performance: A3C reached good levels of performance on both environments (see Figures 17 and 19), dramatically outperforming a random model, as well as the NEAT algorithm (see Task 4). In the advanced environment the algorithm was able to learn to conserve healing potions and to use them when they were most effective (i.e. when the agent was low on health). In addition, Figure 17 also shows the agent was able to learn which actions would lead to super effective attacks, similar to Q-Learning.

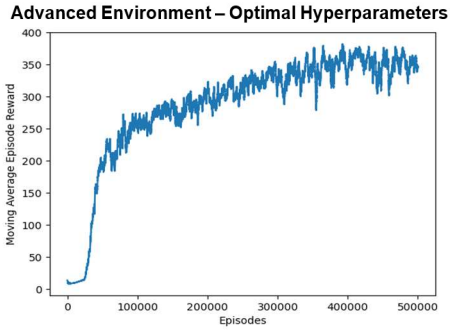

Figure 16: Line Chart of Running Mean Reward – Advanced Env.
Gamma = 0.9, LR = 1e-4, Batch Size = 50, Workers = 12,
Activ. = ReLU, 2 Hidden Layers, 512 Max Hidden Neurons

Given the extreme environment has ~35x the number of possible states the results show A3C has good scalability for this task, as it converged to a solution in ~5x the number of episodes. The agent showed the ability to use the switch actions to both avoid and land super effective hits.

Model	Mean Reward	Std. Dev. Reward	Mean Super Eff. Moves	Mean No. Potions Used
A3C	440.55	157.72	292.29	71.32
NEAT	308.09	204.06	65.59	50.33
Random	11.20	5.38	2.46	1.78

Figure 17: Results from 1,000 Generated Episodes in the Advanced Environment

However, in both solutions the standard deviation remained comparatively high, in future work it would be interesting to compare the performance of other DL models to investigate whether this could be improved.

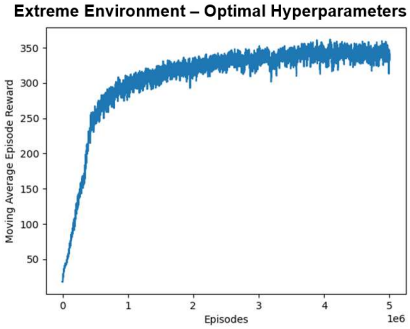

Figure 18: Line Chart of Running Mean Reward – Extreme Env.
Gamma = 0.9, LR = 1e-5, Batch Size = 500, Workers = 12,
Activ. = ReLU, 2 Hidden Layers, 512 Max Hidden Neurons

Model	Mean Reward	Std. Dev. Reward	Mean Super Eff. Moves Used	Mean Super Eff. Moves Hit By	Mean Trainers Beaten	Mean Switches Made
A3C	344.56	170.19	117.89	44.45	24.98	17.97
NEAT	61.80	54.32	7.93	6.69	3.28	0
Random	20.58	9.41	3.06	2.26	0.21	4.08

Figure 19: Results from 1,000 Generated Episodes in the Extreme Environment

4. NeuroEvolution of Augmenting Topologies (NEAT) Implementation

4. i. Algorithm Description

Neuroevolution is the artificial evolution of neural networks using genetic algorithms[6]. The NEAT approach does this by initializing small, simple networks and allowing them to evolve over generations by becoming more complex. Throughout this process, structures and behaviours are tested to find combinations that suit a particular task.

Two different types of "genes" are defined that make up the "genome" of each network – nodes and connections. Node genes consist of sensors, outputs and hidden types. Sensors relate to the number of inputs – in my case this is the observations from my environment – while outputs relate to the number of possible actions. These two types are fixed, however, hidden nodes can be added or removed throughout the evolutionary process.

Connection genes specify how each node is connected to another. The connections carry associated weights that can be altered and connections can also be disabled entirely. The mutation rate controls how often existing connections are updated or new ones are added.

NEAT uses a process called crossover to combine two genomes to produce a new network. If this was performed blindly there would be a very high chance of creating a non-functional network and so NEAT utilizes "historical markings" to identify genomes with similar traits, whereby merging is more likely to produce functional improvements.

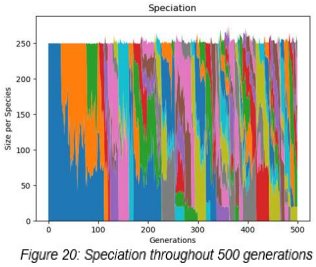

Figure 20: Speciation throughout 500 generations

Lastly, speciation is employed to separate a population into different species (Figure 20). These species are identified according to their historical markings and speciation is used primarily to allow for new structures to develop and evolve without being out-competed early in their life cycle by older, more established networks. "Fitness" is used to assess the performance of each network and is the deciding metric for whether a network should be allowed to continue in the evolutionary process or should be made extinct. NEAT allows fitness to be shared intra-species, allowing for co-operative improvement. In my environments a fitness score of 1 is accumulated for every turn the agent remains undefeated (up to a maximum of 500 turns).

4. ii. Hyperparameters and Performance

I implemented NEAT using the python package[7] which requires a configuration text file that is used to specify 49 different hyperparameters surrounding the evolutionary strategies. The following parameters were altered to investigate their impact on performance within the advanced environment with 200 generations of NEAT networks:

Population Size: This controls the total number of networks possible across the different species in one generation (see Figure 20). A larger size will provide a greater pool of potential parent networks that may have desirable structures; however, a larger population may be unnecessary for certain problems, resulting in increased time to convergence.

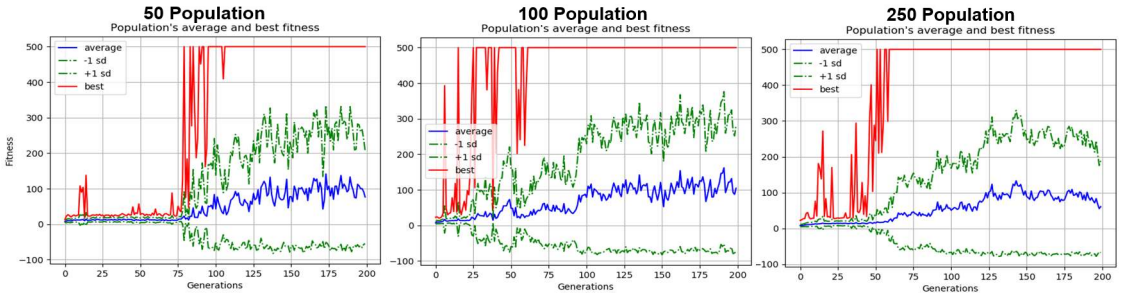

Figure 21: Average and Best Fitness of NEAT networks across different population sizes – Mutation Rate = 0, Elitism = 2, Max Stagnation =20

In Figure 21 above, a larger population size generally produced better networks from a lower number of episodes. However, since parameters like stagnation and elitism are integer inputs (rather than percentages), they need to be scaled with population size, otherwise a larger population can result in more stagnant/ non-functional networks in the overall pool – seen by the declining performance of the 250 Population size after 150 episodes. Larger generation sizes also severely affect speed and so there is a trade-off with performance to be considered.

Mutation Rate: Dictates the likelihood of a mutation occurring. A rate that is too low may result in not enough diversity across the population, whilst a rate that's too high may result in the algorithm failing to converge as new mutations are constantly interrupting evolution. Evidence of this can be seen in Fig 22, as a rate of 0.1 causes the results to become completely unstable, with no evidence of convergence. A rate of 0.05 can be seen to slightly reduce variance in comparison to 100 population size with 0 mutation rate seen in Figure 21.

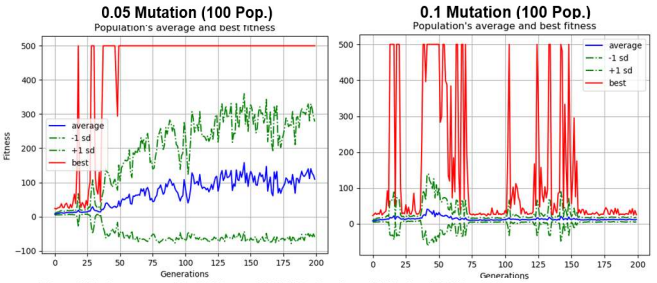

Figure 22: Average and Best Fitness of NEAT networks with Varying Mutation Rates

Elitism: Controls the number of most-fit networks that will be preserved "as-is" from one generation to the next. As can be seen from Figure 23, when elitism is too high there is not enough innovation within the generations and so the algorithm fails to improve. A number too low though may discard functional networks too readily, which could hinder convergence longer term.

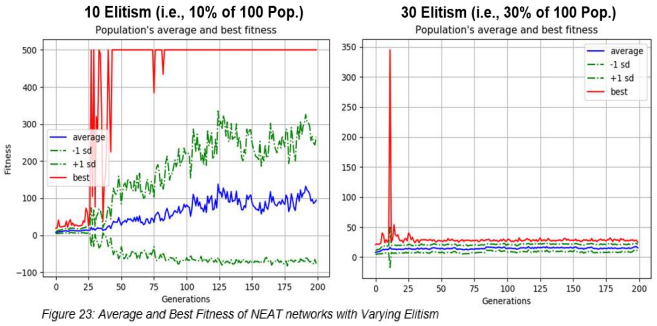

Figure 23: Average and Best Fitness of NEAT networks with Varying Elitism

Activation Function: Similar to my approach with A3C, I compared the performance of the Tanh and ReLU functions. The results showed that, again, Tanh proved to be the better activation function for this environment due to the lower variability seen with results - see Fig 24.

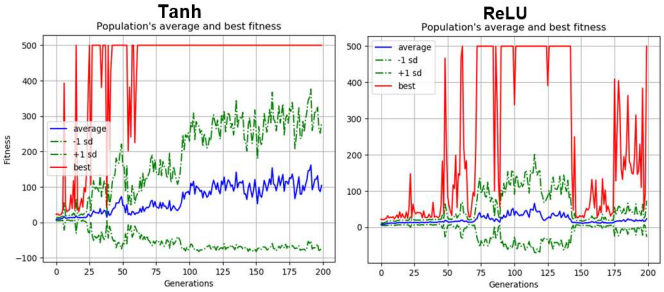

Figure 24: Average and Best Fitness of NEAT networks with Varying Activation Functions

Performance: Evaluation of NEAT convergence has to be performed differently to A3C. This is because not all networks are working towards one unified goal, and therefore a comparable running average performance will never be obtained by the NEAT algorithm, as each generation is a mix of old and new networks. Various "best" networks (i.e., scoring 500 on the environment) were selected and tested on 1,000 generated episodes (see results in Figures 17 and 19 on page 10).

It was found that NEAT performed well, although still worse than A3C, on the advanced environment, while performing not much better than a random model on the extreme environment. NEAT works well at finding a network whose weights are randomly calibrated appropriately for the environment it is facing at the time of training. In the extreme environment, because the opponents are not static and can be completely random, the model that may have worked for a previous run fails on the next. Despite this, for the advanced environment, NEAT found an optimal solution within 17 generations – taking 30 minutes to find - compared to A3C taking almost 3 hours to converge. Therefore, when speed to convergence is priority, and the environment is relatively stationary, NEAT could be a preferable option.

REFERENCES

[1] Silver, D. ‘Lecture 5: Model-Free Control’, p. 43. Available at: https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf
[2] Konda, V. R. and Tsitsiklis, J. N. (2003) ‘Tsitsiklis, “On actor-critic algorithms’, SIAM J. Control Optim, pp. 1143–1166.
[3] Mnih, V. et al. (2016) ‘Asynchronous Methods for Deep Reinforcement Learning’, arXiv:1602.01783 [cs]. Available at: http://arxiv.org/abs/1602.01783 (Accessed: 1 April 2021).
[4] Niu, F. et al. (2011) ‘HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent’, arXiv:1106.5730 [cs, math]. Available at: http://arxiv.org/abs/1106.5730 (Accessed: 7 April 2021).
[5] Silver, D. ‘Lecture 7: Policy Gradient’, p. 41. Available at: https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf
[6] Stanley, K. O. and Miikkulainen, R. (2002) ‘Evolving Neural Networks through Augmenting Topologies’, Evolutionary Computation, 10(2), pp. 99–127. doi: 10.1162/106365602320169811.
[7] Configuration file description — NEAT-Python 0.92 documentation. Available at: https://neat-python.readthedocs.io/en/latest/config_file.html (Accessed: 7 April 2021).