CSC 4444: Artificial Intelligence

Race Car AI

Group 1

Group Members:

Dean Compton	Reed Ladnier	Pacco Tan	Robert Williamson
Jameson Harrington	Hypatia Mills	Zhi-Ruei Tu	Zhenjie Yu
Rohan Kadkol	Ian Nezat	Ruxin Wang	

Report Submission:
December 3, 2021

Abstract – Racing games are very common in today's gaming industry. They can be a true test of skill when racing against the best racers and AI's. This project explores the effectiveness and ability of different reinforcement learning models including Proximal Policy Optimization (PPO), NeuroEvolution of Augmenting Topologies (NEAT), Deep Deterministic Policy Gradient (DDPG), and Deep Q Learning (DQN) to complete a Unity-based racing game built for the project. We compare all these models on training effort (number of epochs to convergence), learning trend (observing by eye the best learning graph), and model complexity (the network complexity if it is using a deep network) and derive conclusions of what model works best in what cases

Table of Contents

Introduction / Motivation

Video games are one of the favorite leisure activities in recent times, especially for the younger generations. What makes video games exciting is the challenge of completing the objective. To give a fun challenge to the human player, game developers train AI models to act as adversarial agents with the same objective as the human player.

Traditionally, these AI agents are trained using supervised learning. Training using the traditional supervised learning approach makes the agent only as good as the player data we train the model with. This caps the skill level of the agent. The reinforcement learning approach on the other hand tries to learn the best moves to achieve the objective but does this all by itself, without using any player data. This approach can produce a model that is better than even the best human player since it does not use any human generated data for training.

The exciting prospect of producing an AI model without using any human data for training, motivated us to build a reinforcement learning model. The high demand and excitement for a self-driving car AI helped give us the topic of race car AI.

Thus, we finally decided to build an AI for race cars in a car racing game.

This project's future will be in the self-driving car industry, with additional features incorporated, such as safety, digital trace, and so on. The majority of the issues that self-driving cars will face will still be related to mortality and human factors.

Problem / Objective

Our objective of this project is to train a race car AI to give a challenging and fun experience for the human player. To best compete with the human player, our AI will aim to:

1. Minimizing the time:

   Minimizing the time required by the car to complete the whole racing track is the main objective of the problem.

2. Avoid collisions

   Avoid hitting the road boundaries to promote staying on the track over going off-road.

Objective Accomplishing Approach

To accomplish our above mentioned objective, we will be implementing reinforcement-learning based AI models. To build the best race car AI and do a comprehensive study for this class project, we decided to implement four different approaches of reinforcement learning:

1. Proximal Policy Optimization (PPO)
2. NeuroEvolution of Augmenting Topologies (NEAT)
3. Deep Deterministic Policy Gradient (DDPG)
4. Deep Q-Learning Network (DQN)

To implement these algorithms, we had to build our PEAS problem specification, custom racing game environment, and a bridge to communicate between our models written in Python and our game environment written in C#.

We will first cover these problems, code, and environment setup tasks to give a better understanding of the implementation of our project. Following that, we will cover our results in training each of the four above reinforcement learning approaches.

PEAS

1. Performance Measure

All reinforcement learning models need "reinforcement" in the form of instantaneous rewards. To determine the instantaneous rewards at each state, we used the following reward function:

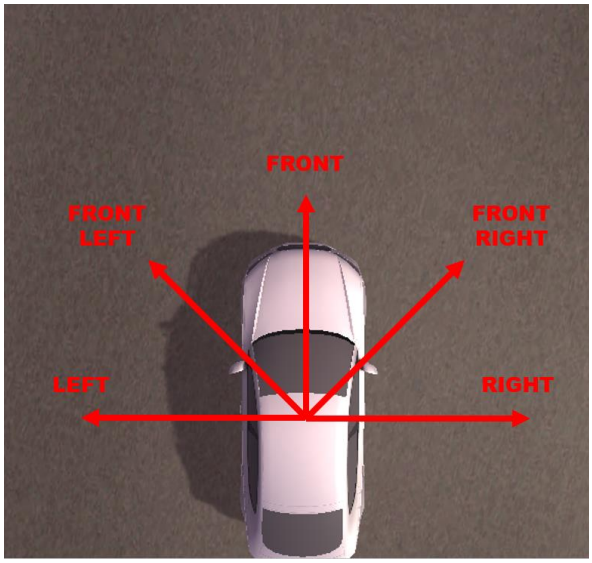
- Hit the wall -20 points
- Pass the next checkpoint +10 points
- Reach the end of the track +20 points
- For all other states -0.05 points

2. Environment

We added 5 ray casts to the car in the following directions:

- Front
- Left
- Right
- Front Left
- Front Right

These ray casts would originate from the car and record the distance travelled till it hits another game object. This will help us detect obstacles like the racetrack boundaries and the race car's proximity to those obstacles.



Range of each ray cast sensor = $[0, 35] \cup -1$

($sensor\ value = -1$ when the ray doesn't hit a game object after travelling 35 $units$)

3. Actuators / Actions

The actuator for our car for the steering wheel.

From a higher view, the main actions taken by our car agent (steering wheel) are:

- Turn left
- Keep driving straight
- Turn right

We chose not to have an action to accelerate or decelerate the car. This is because in our environment, the agent would always have to always accelerate the car to reach the finish line in the least time. Decelerating the car or doing nothing will always give a higher time than if the car agent always accelerates. Therefore, we do not have an action to accelerate, decelerate, or do nothing.

The above actions were from a higher, more general view. From an in-depth, more detailed view, each model further subdivided this general action space into more fine-grained actions.

PPO

The PPO model uses continuous outputs in the range [-1,1], where negative values indicate a degree of left turning, and positive values indicate a degree of right turning.

NEAT

The NEAT model can give continuous or discrete actions. For this project, the action was determined by two outputs with the sigmoid activation function. One is to represent turn right and the other turn left. The left output was subtracted from the right output to give a continuous range of –1 to 1 which is the angle by which the car turns.

DDPG

A continuous action in the range [-2,2]

DQN

Since the DQN approach only works with a discrete action space, we created 5 discrete actions. Our five discrete actions are:

1. Steer car left by 2 units
2. Steer car left by 1 unit
3. Steer straight
4. Steer car right by 1 unit
5. Steer car right by 2 units

4. Sensor values

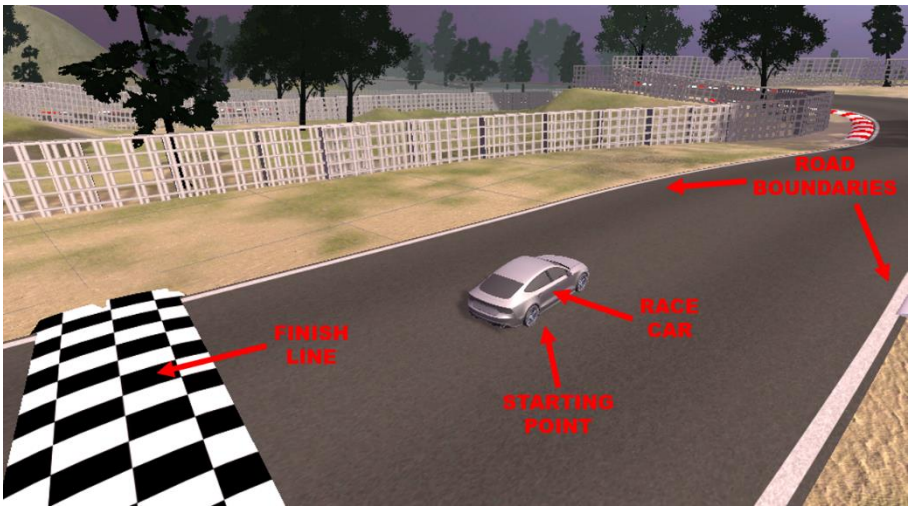
The car in the Unity environment provided 8 sensor values to our Python models. They are:

1. Velocity in the x direction
2. Velocity in the y direction
3. Velocity in the z direction
4. Ray cast distance — Front
5. Ray cast distance — Left
6. Ray cast distance — Right
7. Ray cast distance — Front Left
8. Ray cast distance — Front Right

Out of these 8 sensors values that we received from the car, we sometimes selected a subset of these sensor values for training. We did this to observe its effect on the speed and quality of our training. Exact observations and results from training with a subset of sensor values are explained in more detail in the below sections.

System Design

We built our custom racetrack environment from scratch using Unity, a popular and free game engine. Although we built the environment by ourselves, we used a few readymade game assets for the race car, trees, and the road. This helped us speed up the trivial environment creation and focus more on our more important AI study.



The main components of our Unity environment are:

- Race car
- Road boundaries
- Starting point
- Finish line

Tools Used

1. Unity

   We used Unity to build our car racing game environment.

2. CUDA

   To speed up our deep network training, we used CUDA on our Nvidia GPUs

Languages Used

1. C#

   All Unity code was written is C# using Unity's C# APIs.

2. Python

   We trained all our four reinforcement learning approaches in Python.

Libraries Used

1. Unity ml-agents

   To turn the game environment into a reinforcement learning environment, we used the Unity *ml-agents toolkit*. This helped us define the reward function to give the appropriate rewards to the Python program.

2. Python mlagents

   The Python program communicates with the Unity environment using the Python *mlagents* package. This allows us to retrieve the state and reward information from the Unity environment and send appropriate action commands back to the Unity environment.

3. PyTorch

   We used PyTorch to build our deep networks for the DQN, DDPG, and PPO algorithms.

4. NEAT Python

   The NEAT model used the NEAT Python library.

5. Stable-Baselines3

   Our PPO implementation used the baseline model from Stable Baselines, a fork of the original OpenAI network baselines.

Model Training

So far, we have covered the PEAS problem definition, technologies and libraries used, and Unity race car game environment setup.

We will now cover the exact details and results of implementing and training our four reinforcement learning approaches.

1. Proximal Policy Optimization (PPO)

Introduction

Proximal Policy Optimization is a modern evolution of the trust region method, where the safe area is determined by a function bound below the true policy, which is calculated by the difference between the expected advantage and the approximated KL-divergence. By taking steps to maximize this lower bound, PPO can maximize the true reward value in turn.

Within each safe area, gradient ascent is performed across several micro-batches to distribute the intense computation of recalculating weights for many observations.

Hyperparameters:
The PPO model uses these hyperparameters:

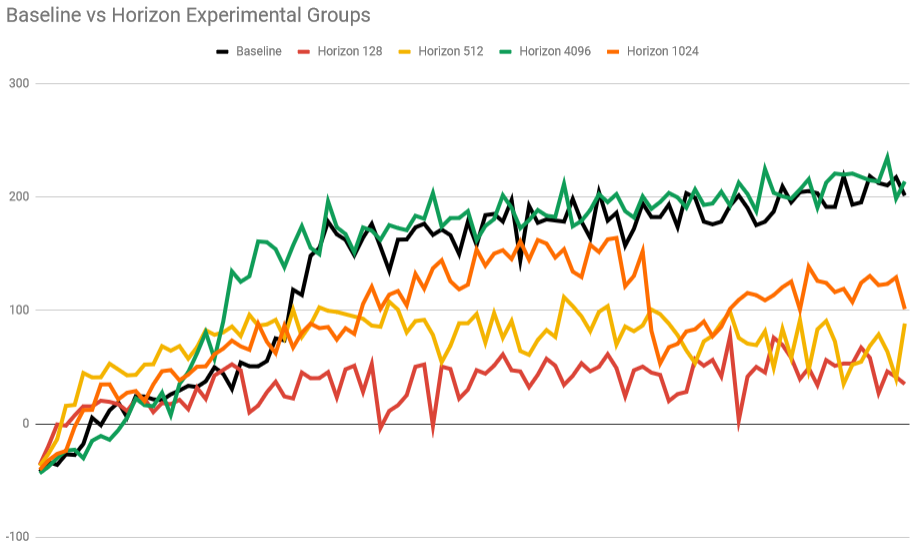
- Horizon, the degree of future-sight
- Batches, the number of training mini batches
- Epochs, the length of minibatch training episodes
- Gamma, the discount factor for future rewards
- Clip, the size of the trust region relative to divergence
- Learning Rate, the gradient ascent learning rate in minibatch training

After testing one million training steps in twenty-five different hyperparameter sets, we found the horizon most vital to determining model success. Learning Rate had the most noticeable effect on the speed of convergence, but its effect on terminal effectiveness is unclear within our experimental training frame.

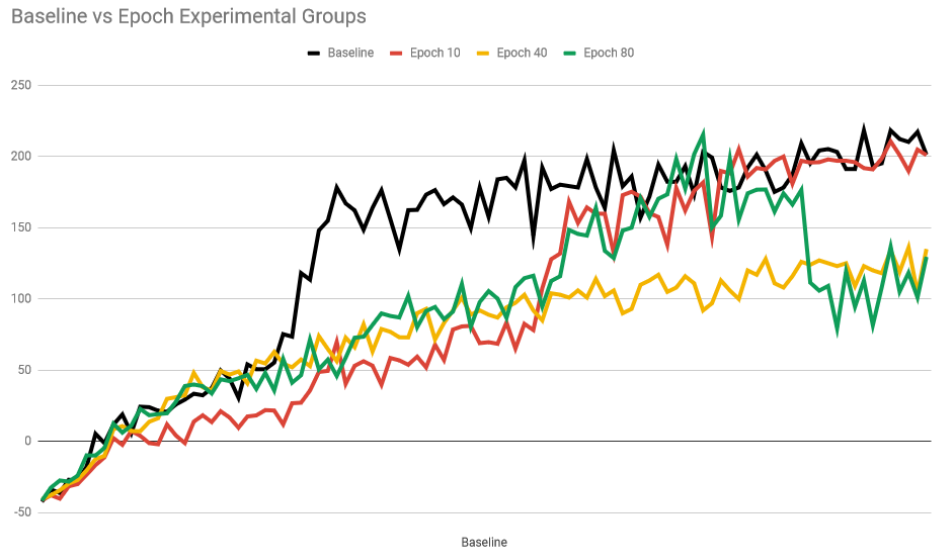
Comparison

Our baseline PPO model converges in around 400,000 steps. We measure in steps rather than epochs, as the nature of PPO makes the relation between epochs and training time unstable. For this model, we use a horizon of 2048 steps, a set of sixty-four minibatches at a time, run for twenty epochs, with a clipping range of 0.2, a gamma discount factor of 0.9, and a learning rate of 3e-4.

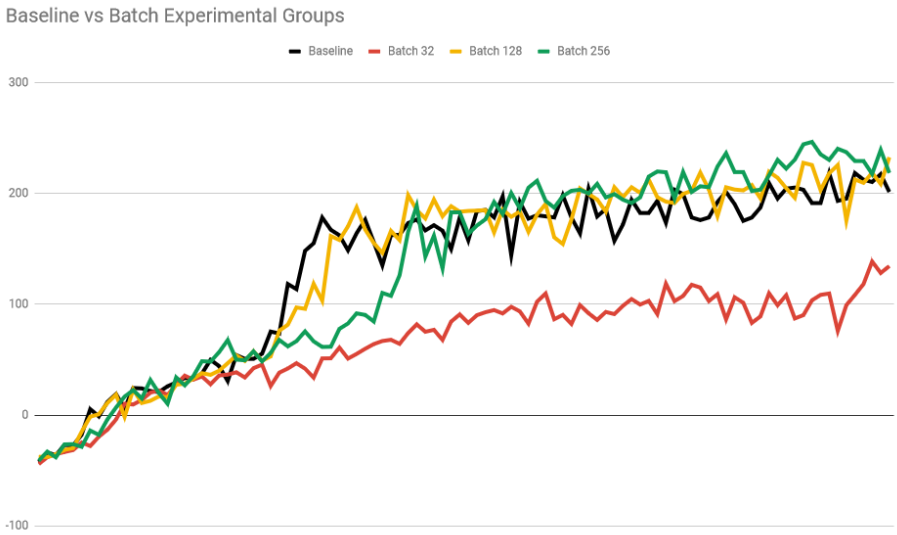
The following graphs use this baseline model as a point of comparison for our experimental models. Each model varies from the baseline by a single hyperparameter.



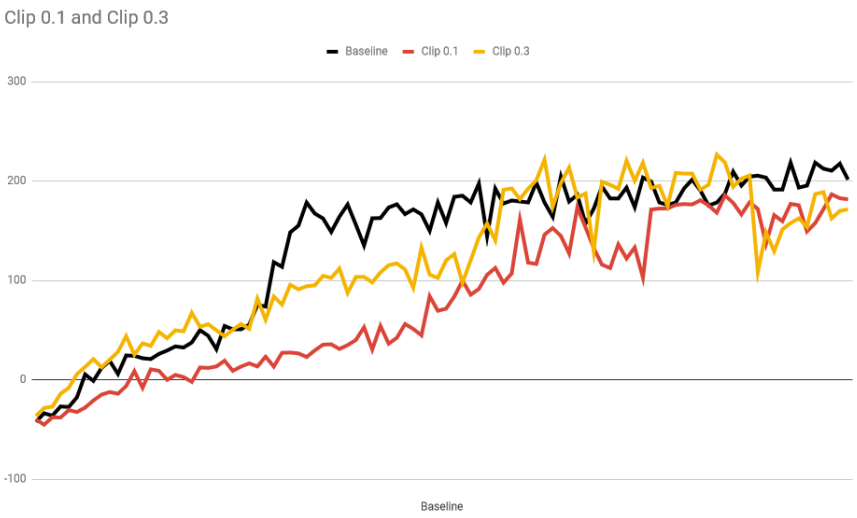
Horizon proves to be the best performance prediction parameter, increasing steadily until around 2048 steps, after which improvement slows down. The large performance drops for the 1024-step horizon model shows the ‘tunnel vision’ that the model learns without a sufficiently distant horizon.



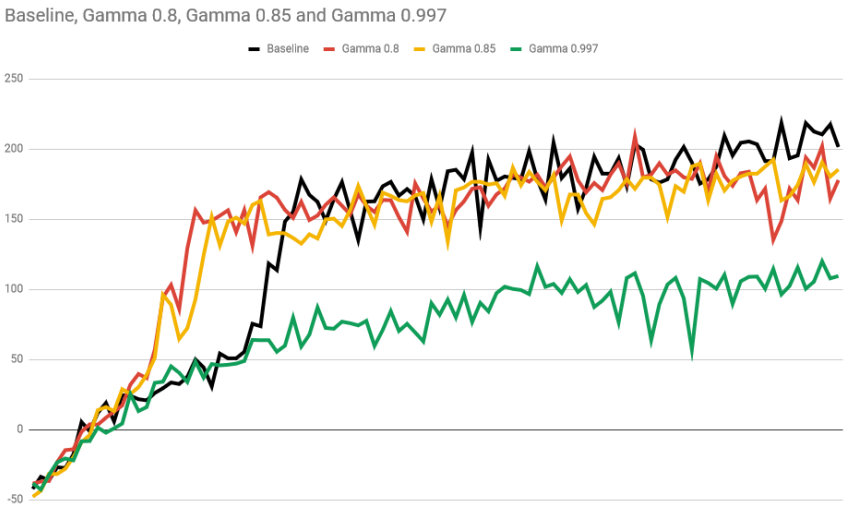
In comparing performance measure to micro-batch epochs, we can clearly see the volatility that comes with longer batch-training sessions. Even once the epoch eighty model has reached the baseline (20-Epoch) convergence point, it quickly collapses back down, as it struggles to generalize and can easily hyperfocus on singular scenarios.



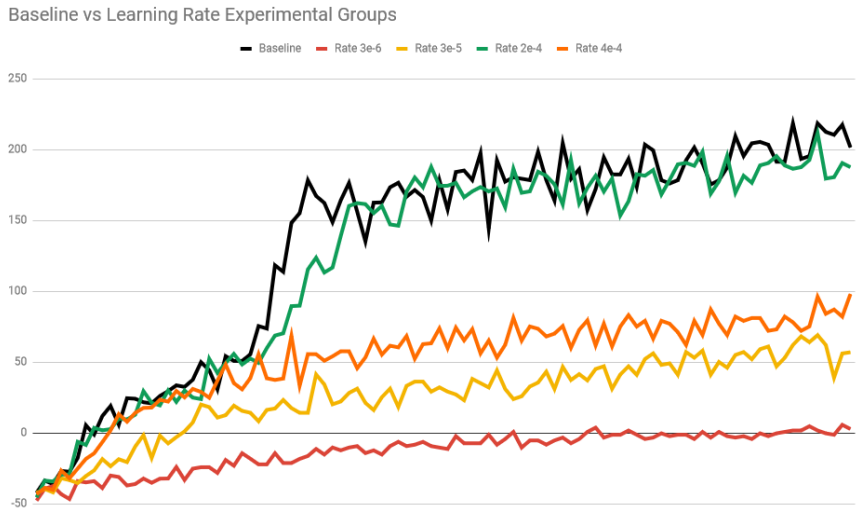
Overall, training across more batches improves quality, with some diminishing returns. Since adding more parallel batches can be computationally expensive, our baseline model (Batches = 64) walks a fine balance between the two ends, even converging slightly faster than the more distributed models, a byproduct of increased generality.



As expected, our experimental model with higher clipping range was able to achieve some advantage that our baseline (clip = 0.2) model missed, but these gains were always quickly crushed by the risks that the model took from its wider trust region. Our model with a shrunken clipping range converged slower, but it did so with fewer large performance drops.



For our environment, lower gamma values, thus higher ‘greed’ were able to converge significantly faster than the baseline ($\gamma = 0.99$). This aligns with the start of our simulation, and it explains why the highest gamma model performed so poorly. In order to learn the first vital steps, our AI needs to concentrate on immediate rewards, especially since any failed turn will immediately render future *potential* rewards meaningless.



As expected, increasing our model's learning rate (Baseline 3e-4 to 4e-4) led to a significant performance decrease, and decreased learning rates slowed training down proportional to the change in learning rate. It is unclear if our slower-learning models would fully converge if given a much longer time to train, but that is the result we would predict.

2. NeuroEvolution of Augmenting Topologies (NEAT)

Introduction

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm is a genetic algorithm for neural networks. The NEAT algorithm starts out with an initial population of networks, picks the fittest and then allows them to reproduce (asexually or sexually) a new generation with a chance for a random mutation.

The variables that control the characteristics of the population and their evolution are set in a configuration file. The configuration file is used to create the population that should evolve to maximize fitness. The NEAT documentation has recommended configuration file values that were used for the project with two outputs and five inputs.

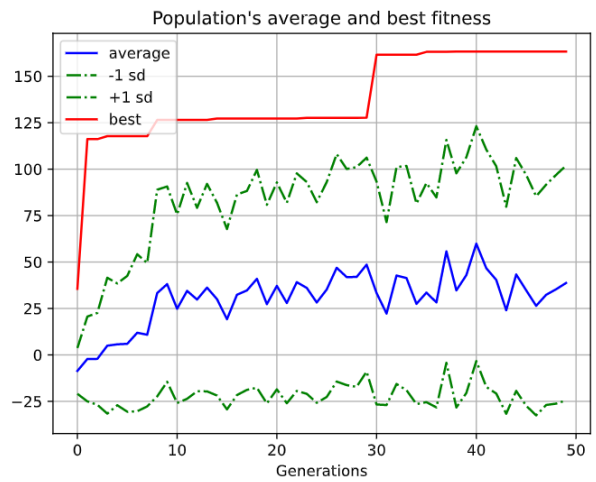
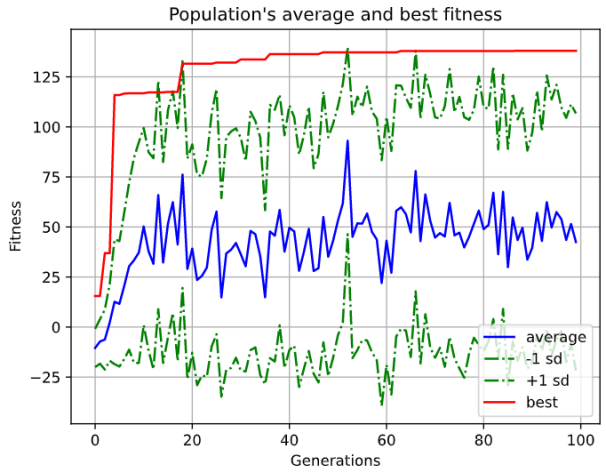
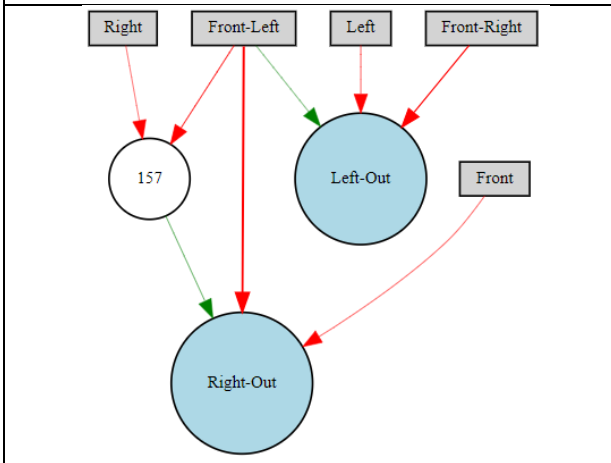
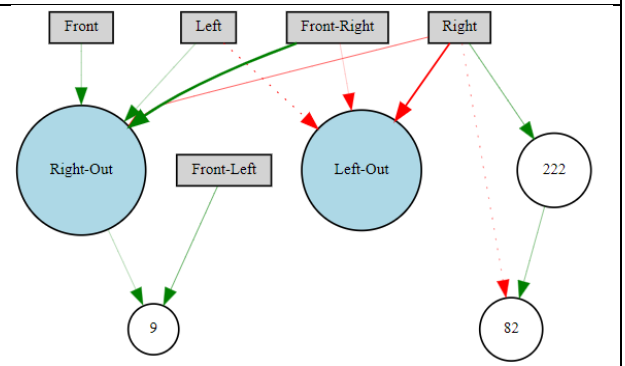
The initial population starts unconnected without hidden nodes and there is a random chance any genome in a population will add a node or change connections and weights. The genomes with the highest fitness reproduce to form a new generation of networks. This process is repeated for a certain number of generations or if a population reaches a certain threshold.

Setup

The inputs were the five ray casts normalized from 0 to 1. There were two outputs with the sigmoid activation function: one to indicate a left turn and the other to indicate a right turn. In order to give an action, the left turn output was subtracted from the right turn output to give an angle between –1 and 1. For this model, there were multiple tests: one with 40 genomes and ran for 50 generations and one with 20 genomes and ran for 100 generations. Each test ran the model 2000 times.

Experiment Results

Graphs of the population average and best fitness per generation is shown here for both model tests and their directed graphs of the networks:

Forty Genome/Fifty Generation	Twenty Genome/One Hundred Generation
	
	

The directed graph's inputs are in the square nodes and the outputs are the light blue nodes. Green arrows mean positive weight and red arrows mean negative weighted connections.

The test with the smaller genome population, but more generations didn't find a solution in the same number of runs. The 40-genome population made it to the finish line, but the 20-genome population did not. The 20-genome population might not have allowed enough diversity per generation in order to find a solution in the same number of tests.

3. Deep Deterministic Policy Gradient (DDPG)

Introduction

The Deep Deterministic Policy Gradient algorithm is an actor-critic algorithm that learns a policy and a Q function at the same time. The algorithm learns the Q function with off policy data and uses that Q function to learn the policy. It is typically compared to a Deep Q Network for continuous action spaces, with the difference being that the policy network is updated with every update of the actor network.

As with all Q learning algorithms a replay buffer is used that contains past experiences of the network and a random sample of those experiences is used to learn/approximate the Q function. At every timestep after an action was generated, noise was added for exploration of the state space. The noise was generated through the default Ornstein-Uhlenbeck process. This action was then clipped, giving us an action space of [-2,2].

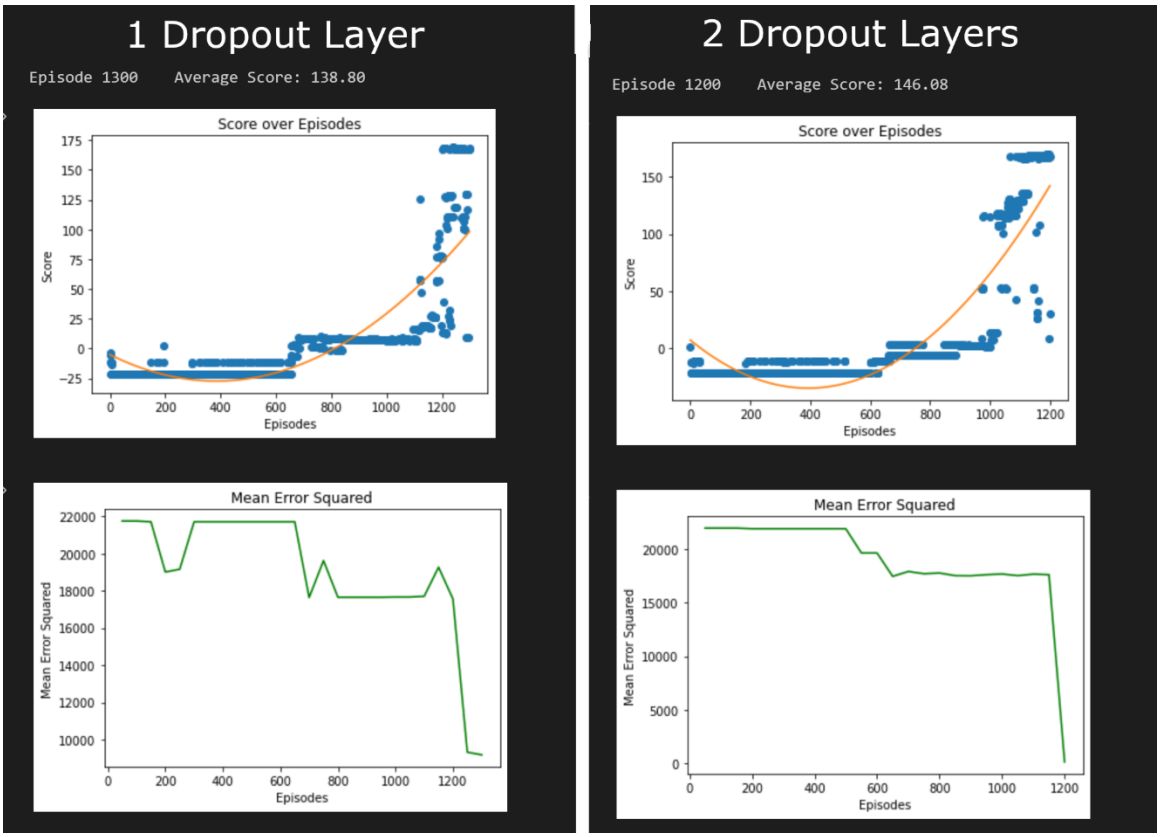
Experiment

During the Unity environment creation, we experimented with multiple movement types for the car object. We experimented with a force-based car script where force was applied to the wheels making the car move, with adding velocity to the entire car object, and translation of the car in world space. We settled on the velocity-based movement script for the car because of the complications with force-based motion in unity and the amount of time that tweaking it would require. This left the x, y, and z velocity being somewhat superfluous. This lead the first experimentation on the learning to be removing the velocity sensor inputs and seeing if this would result in faster learning.

A limitation of the Unity Ml Agents bridge is that only one Unity environment can be open and running at a time, Therefore, in order to test the model on an environment not trained in, the first environment must be closed and a new one open. This created a lot of overhead in continuous opening and closing of environments. To reduce that overhead, we tried just running several episodes on the test track every 50 episodes of training. We would then calculate the mean squared error on the test data before closing that environment again and opening to train one again.

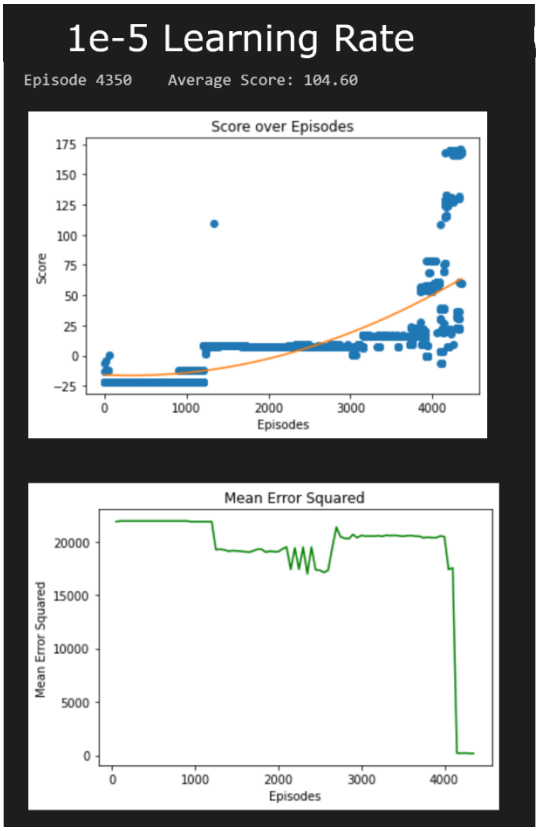
The rate of score over episode for our environments in special in that zeroing inputs and moving straight along a straight piece of road is something that the models learn relatively quickly. From there, the two major obstacles are the left and right turns. Left turns and right turns are distinct enough that the models don't immediately learn one from the other, but inside each category the sharpness or length of the turn can be picked up on relatively quickly. This creates breakpoints in our environments where once the first left and first right turn are successfully learned the model then quickly makes it through the rest of the track. This can be problematic in that the rate the model learns can vary extremely widely between iterations because of the randomness associated with exploring the state space. The randomness also partially explains why there are still a decent amount of low scoring episodes toward the end of training. Because we want to minimize time, the model wants to cut corners as close as possible to the edge of the road. In the Deep Deterministic Policy Gradient model, the noise added to the agents actions when it travels close to the wall can end up in it hitting the wall and ending the episode resulting in a lower score. The simple velocity-based movement of the car just exacerbated this issue and is why in the future a more realistic force-based model would much improve the accuracy.

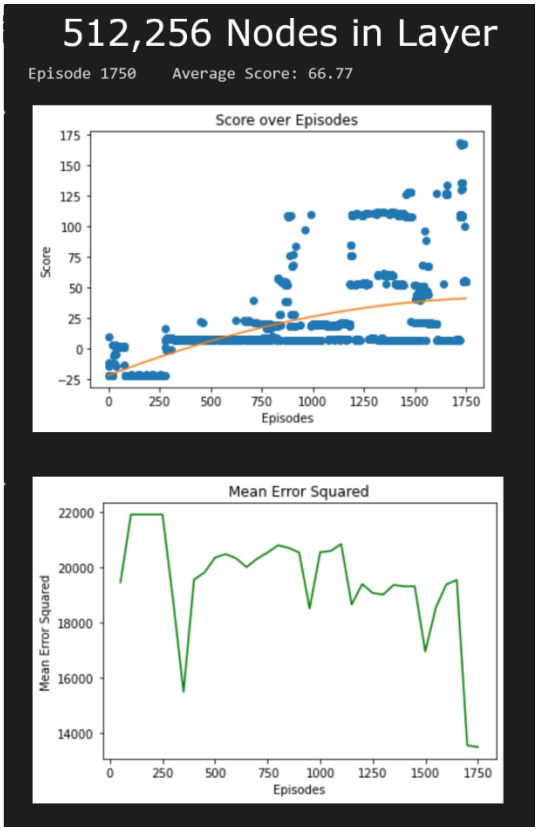
Next, we decided to try adding some dropout to the model to help prevent some overfitting to the training environment.



We experimented with several different dropouts including one layer of 0.2 dropout, two layers of 0.2 dropout, one layer of 0.1 dropout and two layers of 0.1 dropout. The above figures are both using a dropout of 0.1 after each dense/linear layer of the model. The two layers of 0.1 dropout show a definite improvement on the test data showing the two major breakpoints of the environment as well.

We also experimented with the learning rate and number of nodes in layers. In particular, the start DDPG utilizes two liner layers, with one of 256 ones and one of 128 nodes. The test below doubles the number of nodes for each layer resulting in one with 512 nodes and one with 256 nodes.





4. DQN

Introduction

DQN (Deep Q Learning) is a type of Q-learning algorithm that uses a deep network to predict the expected reward from each state, s, on taking action a.

In regular Q learning, we keep a table of states as rows and actions as columns. Then the table is filled with expected total reward for going from that state, s after taking action a, all the way till the end of the episode. Then at each state, we just choose the column (action) with the highest total expected reward.

In deep Q learning, instead of a table, we train a deep network that takes states as inputs and returns the expected total reward we would get in going from that input state, s, after taking all the possible 5 actions.

The expected total reward takes the reward from the entire episode starting from that point into consideration, and not just the reward in going from state, s, to the next state, s'. The dotted line in the below diagram talks about the reward we predict to accumulate till we reach the goal state, without needing to know the exact path.

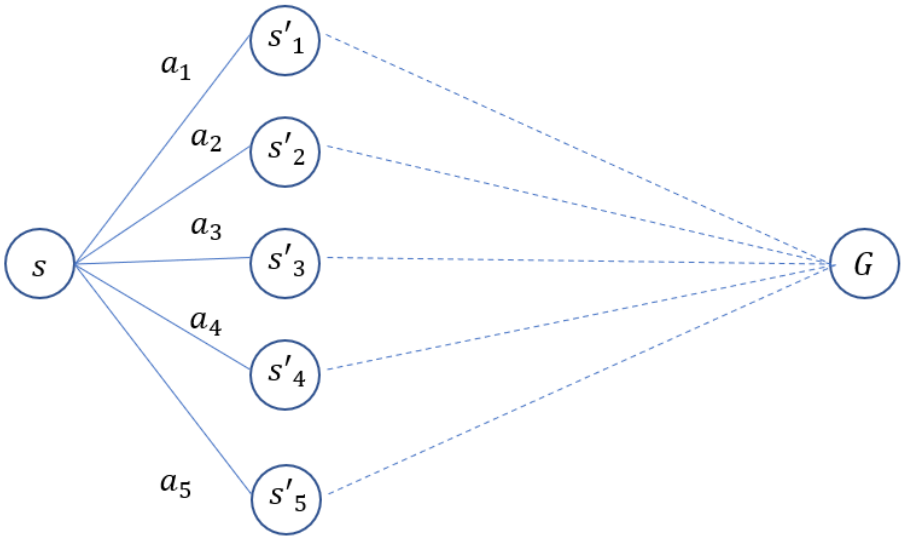


Fig: State graph. Abstraction of only states s, and s', without needing to know the states till the end of the goal. To know the states to the goal state, we simply replace s with s', to get s''. After repeating this sufficiently, at one point, s' will be the goal state G.

At each time frame, we ask the network for the expected total reward in going from state s to states s'_1 to s'_5 by taking actions a_1 to a_5. The network returns a vector of five expected total rewards corresponding to actions a_1 to a_5. Ultimately, we choose the action with the highest expected total reward.

Each node in Fig 1.1 is trained to output the correct expected total reward using the Bellman equation.

$$V(s) = E[\, R_{t+1} + \gamma V(s_{t+1}) \mid S_t = s \,]$$

$$where,$$

$$V(s) = value\ function\ for\ state\ s\ (Total\ Expected\ Reward)$$

$$R_t = instantaneous\ reward\ at\ time\ t$$

$$\gamma = discount\ factor$$

$epsilon$ is the probability of using a random action (explore) over an action from the network (exploit).

In our DQN approach, initially, our model explores the state space taking random actions ($epsilon = 1$). During this, we collect rewards, gain knowledge of the environment, and update our network. Gradually, we start to do $epsilon = epsilon \times epsilon\_decay$ to reduce the randomness of taking an action and start exploiting what we have learned so far. We set a minimum of $epsilon\_min = 0.02$ to always maintain some randomness to always give the opportunity to learn a better solution, in case it is stuck in a local maxima, and not the global maxima.

Setup

We used the same Unity environment as we did with DDPG (explained in the above DDPG section). We ran our DQN algorithm with various hyperparameters. The next section gives more details about these experiments.

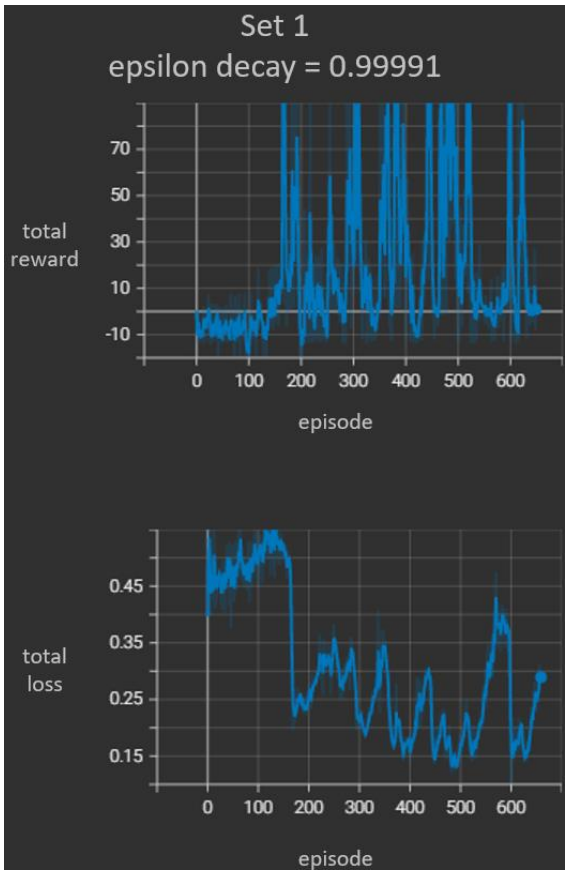
Experiment Results (Answers to critical questions of training choices)

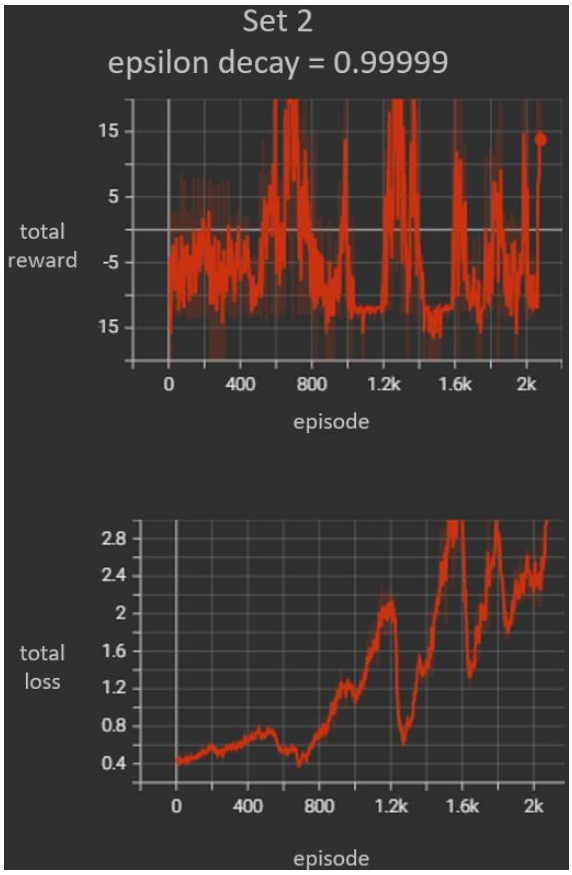
1. Changing the $epsilon\_decay$ for training

$epsilon\_decay$ is related to the rate at which we go from complete exploration (random actions; $epsilon = 1$) to almost complete exploitation (learned actions; $epsilon = 0.02$).

ie. $epsilon = epsilon \times epsilon\_decay$ at every time step.

We tested multiple $epsilon\_decay$ values. The notable values we used for our comparison are $epsilon\_decay = 0.99991$ and $epsilon\_decay = 0.99999$





Set 1: $epsilon\_decay = 0.99991$ **Set 2: $epsilon\_decay = 0.99991$**

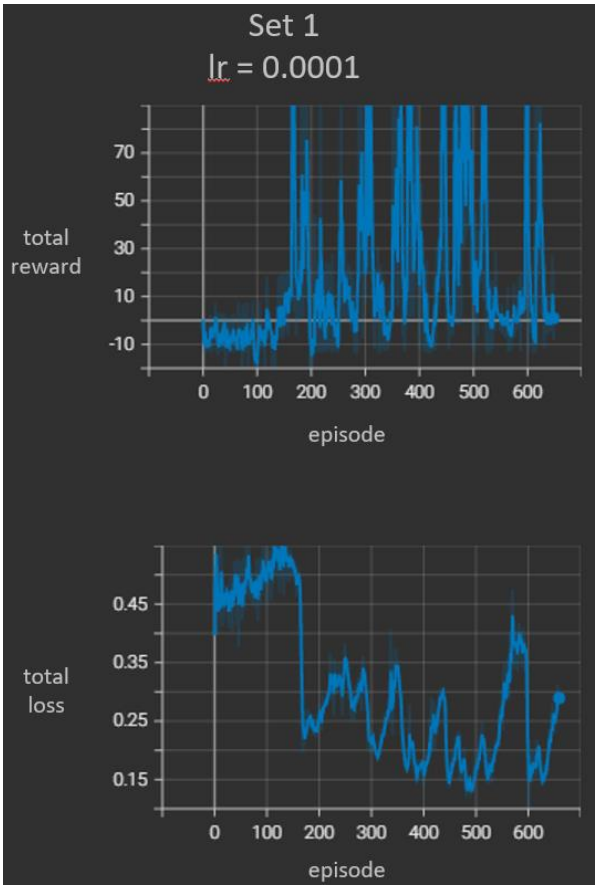
We notice that $epsilon\_decay = 0.99999$ takes more episodes to start learning good results (set 1 first consistent positive reward ~ $epsiode$ #138; set 2 first consistent positive reward ~ $episode$ #537). This is expected since set 2 takes way more random actions; more time spent in exploration since $eqsilon\_decay$ is higher).

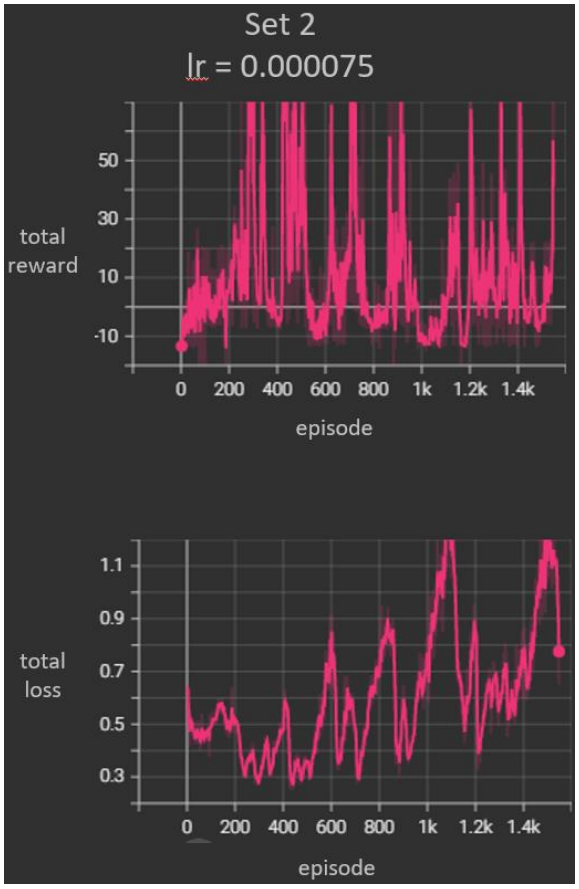
However, even with the larger training time, we do not get better rewards.

Thus, we chose $eqsilon\_decay = 0.99991$ for our final model.

2. Changing the $learning\_rate$ for training

We used the Adam optimizer for our training our neural network. After trying multiple learning rate values, we settled at comparing $learning\_rate = 0.0001$ and $learning\_rate = 0.000075$ for our report.





Set 1: $learning\_rate = 0.0001$ **Set 2: $learning\_rate = 0.000075$**

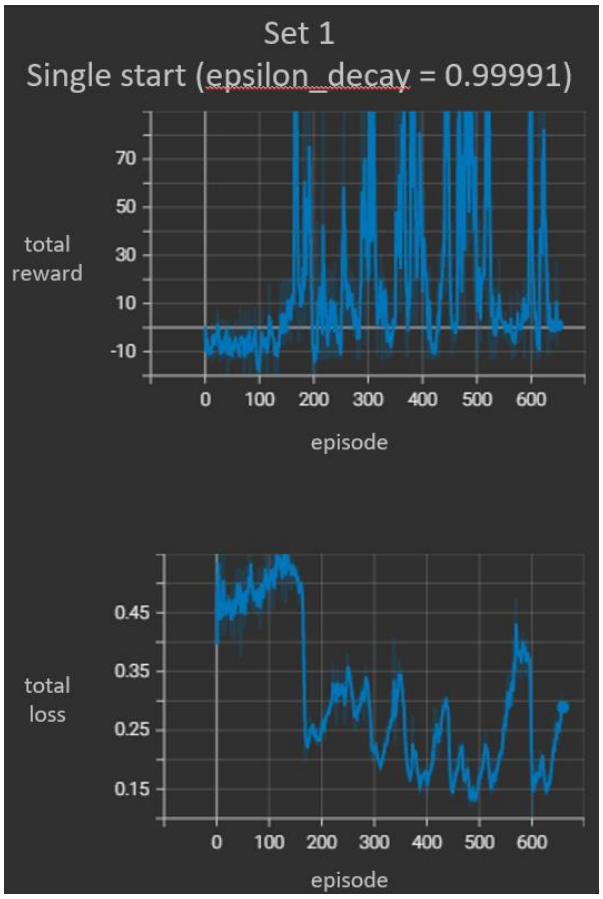
Using $learning\_rate = 0.000075$ started seeing its relatively consistent positive rewards $\sim episode = 191$. However, $learning\_rate = 0.0001$ started seeing its relatively consistent positive rewards $\sim episode = 138$. This is expected since with a higher learning rate we will learn faster.

Set 2's loss function was on overall increasing. Even though set 1's loss function had a few ups and downs (expected in training a DQN model), its loss was on overall decreasing. Thus, we decided to choose 0.0001 as our learning rate.

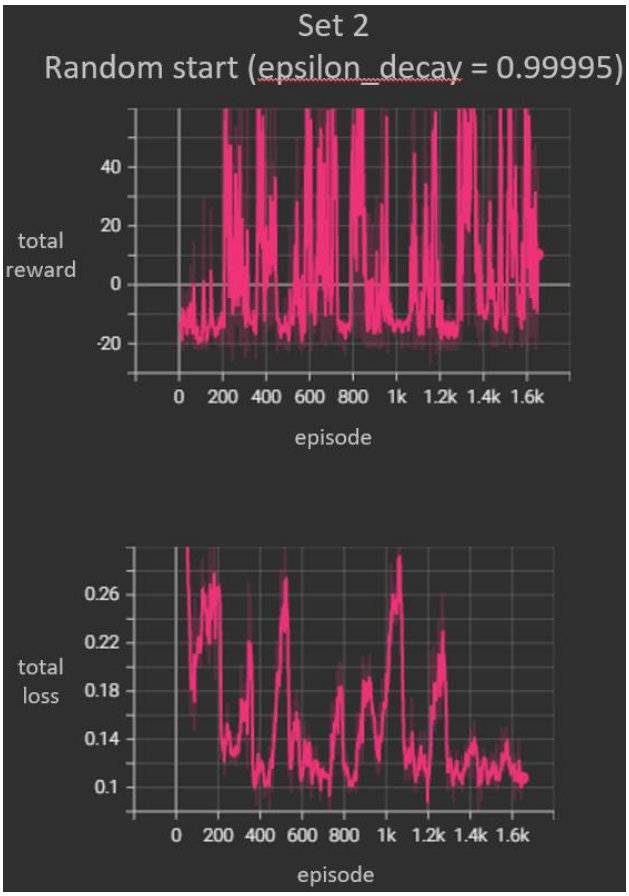
3. Changing the environment for training

To try to improve our training results we implemented a variation in our environment that favored generalization. For doing this, instead of starting each episode from the same starting location, we started each episode from one of the 17 pre-determined spawn locations.

Most of the 17 spawn points are spread evenly across the track. However, some of the points are put in a few difficult states (one with a higher probability of going off-road) to help the car in training for those difficult states.



Set 1: *Single start*
***epsilon_decay* = 0.99991**



Set 2: *Random start*
***epsilon_decay* = 0.99995**

Since the agent would start randomly, we needed to raise the *epsilon_decay* from 0.99991 to 0.99995 to make the results comparable. This is since in the random start case, the agent needed to explore random choices even more since the probabilities of seeing the same state is lower when the starting states are random.

After making both cases comparable, we notice lower rewards for the random start case. Thus, we choose to prefer the single start case.

Discussion/Observations

DQN has many hyperparameters. Some of them are: experience replay set size, neural network structure, GAMMA value (discount factor), learning rate, and epsilon.

Although we tried adjusting these hyperparameters countless times, unfortunately, we could not get our model to be stable:

1. In some rare cases, we could achieve good results (reaching the final goal smoothly and quickly) after around 260 episodes.
2. But in most cases, the agent keeps increasing its total episode rewards for around 50 episodes, and then suddenly it starts getting negative total episode rewards. This cycle of positive and negative keeps alternating without leading to a stable and reliable model even after over 1000 episodes.

Although we could not get our DQN model to become stable, we have studied this model exhaustively. Therefore, we conclude the corresponding reasons and explanations for our model's instability:

1. Experience replay set size

The first is the experience playback set size. The agent puts the perception of the environment into the experience playback as a set of samples of the neural network. Our experience is that if the size of the experience replay set is too small, some experience must be discarded. If the discarded experience is very important, it will bring instability to training. Therefore, the larger the experience replay set, the better. But this is not to say that finding a large piece of memory for experience playback is all right. What experience storage to choose and what experience playback to choose are very important issues.

In our DQN implementation, once the experience playback is full, the oldest experience will be overwritten with the latest experience. If the size of the experience replay set is small, the agent will lose the experience of failing near the spawn point. As a result, soon after the experience replay is full, the chance of the agent failing near the spawn point sharply increases, resulting in unstable training.

2. Neural network structure

According to the knowledge of empirical risk minimization, the complexity of the neural network is related to the number of samples. Therefore, we think that when we construct a neural network, we need to consider the difficulty of the task, and the size of the experience playback. The main note is that if we set epsilon in accordance with the idea of focusing on exploration, the experience in experience playback will be richer, so a more complex neural network structure is required.

But our project does not seem to need to use CNN, as long as 3 layers of linear NN are enough, because it is more difficult to adjust parameters after using CNN. At the same time, it should be noted that the parameters may have to be re-adjusted each time the network is deepened. It is really a thorough understanding of why the neural network is called alchemy through parameter adjustment.

3. GAMMA value

We did not tune this hyperparameter and used the GAMMA value as 0.98. We assumed that the higher the GAMMA value, it means that we want the agent to pay more attention to the future, which is more difficult than paying more attention to the present. Thus, the is slower and it is more difficult to converge.

4. Learning rate

Under the same conditions of other hyperparameters, the learning rates = 0.0001 and 0.000075 were tested respectively.

As discussed above, we noticed that the learning rate that worked the best for us was 0.0001.

However, since the DQN is not a supervised type of problem with the correct answer label, finding the true global minima is hard. This is because the global minima might keep changing as and when the agent makes new observations. Thus, our best learning rate might still not be the best, depending on what observations the agent saw (that is not in our control), that then might change the global minima.

Comparing the Models

Since all our approaches had very different concepts used to train our models, we could not plot a common metric for all models on one graph. ie. DQN and DDPG use epochs, PPO is continuous with sets of minibatches and uses the score as the loss instead, and NEAT has no loss as all, it just has generations. Thus, we will compare the models with 3 metrics: training efforts, learning trend, and model complexity.

1. Training efforts

converge epochs: # epochs it took to converge (training efforts)

a. PPO model:
   The baseline PPO model converges in around 400,000 steps. (Step are a more effective measure than Epochs due to the nature of PPO.)

b. NEAT model:
   It took around 30 generations (which is similar to epochs in scope) to get a finishing solution in one test and it starts to converge around 5 generations.

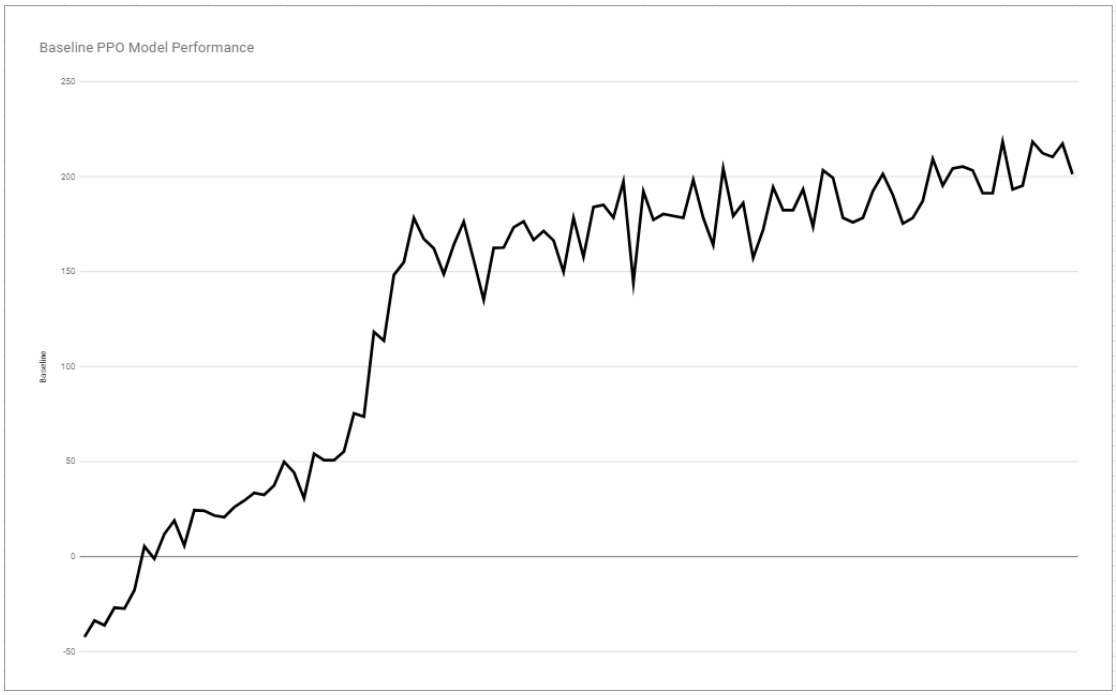
c. DDPG model:
   Starts getting positive rewards in around 800 episodes. In most iterations of the model, in 1000-1500 episodes, the car would start to make it around the entire circuit on a regular basis.

d. DQN model:
   Unstable, good converges in around 260 epochs, bad converges in around 1000 epochs.

2. Learning trend

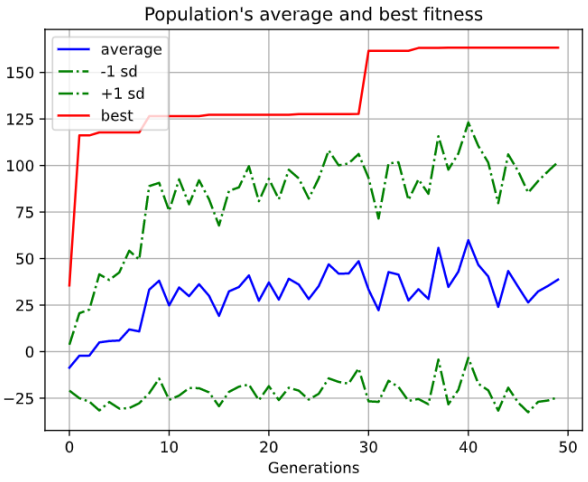
a. PPO model

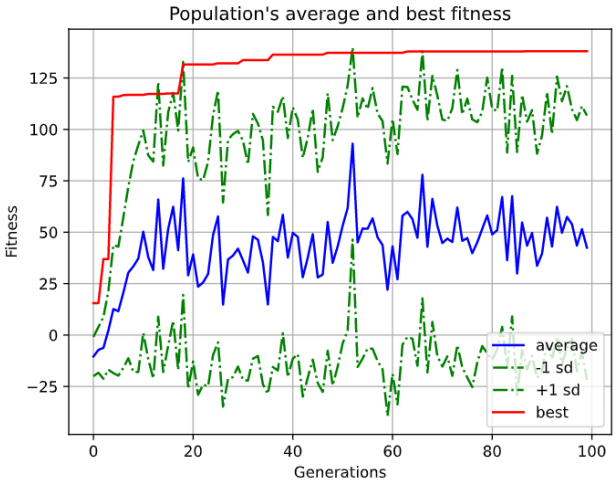
In our PPO model, we can see that learning increases rather sharply before converging and increasing at a slight linear rate.



b. NEAT model

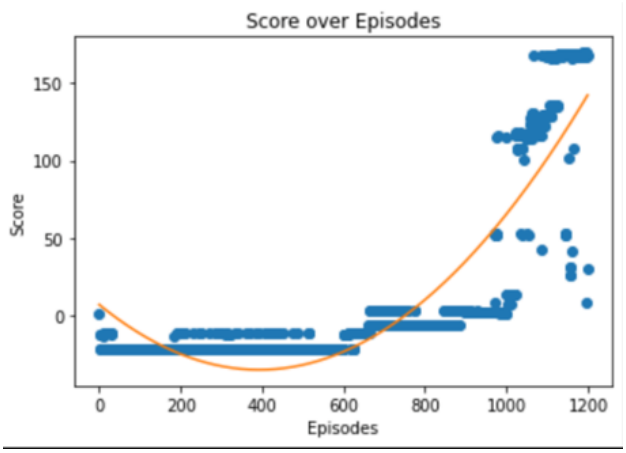
From the following figures shown, we can observe that the overall trend is increasing quickly.





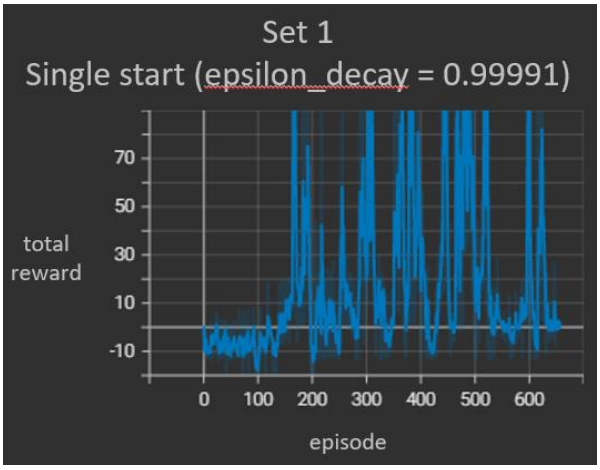
c. DDPG model

From the following figure showed, we can observe that the overall learning trend is increasing. In particular, the learning trend starts increasing after 1000 epochs. This is the result of the rapid learning once the general left curve and general right curve are solved.



d. DQN

From the following figure showed, we can observe that the learning trend keeps oscillating after around 150 epochs, making DQN unstable.



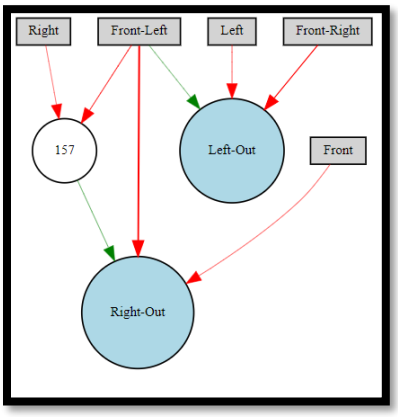
3. Model complexity

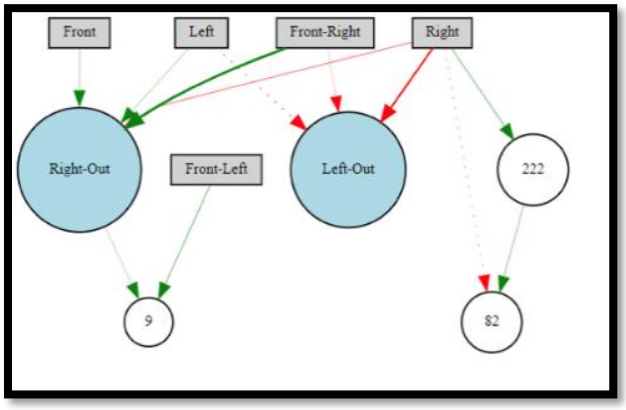
a. PPO model

Our PPO model uses two deep layers, optimized using the advantage function designed in our Unity environment.

b. NEAT model

The model structure is shown in the following figure. The directed graph's inputs are in the square nodes and the outputs are the light blue nodes. Green arrows mean positive weight and red arrows mean negative weighted connections.





c. DDPG model

DDPG models include 3 linear layers and 2 dropout layers, every linear layer used RELU as activation function. Loss function used is MSE.

d. DQN model

DQN model includes 3 linear layers and 1 dropout layer, every linear layer used RELU as activation function. Loss function used is MSE.

Observations: DDPG is an extended algorithm from DQN. They both adopt two methods: Experience Replay and Freezing Target Networks to solve the problem of unstable samples and target values. And the algorithm structure of the two is very similar, and both are the same process, but DDPG has some more Policy series network operations on the basis of DQN. Additionally, the loss functions of the two are essentially the same, except that DDPG joins the Policy network to output continuous action values, so it is necessary to embed the loss function of the Policy network into the original MSE.

Force Model Experimentation

At the start of this project, we hoped to reach the end with a revamped, more realistic physics system. While this force-based model never made it to full maturity, its failings highlight the importance of environment design and model capacity when deploying reinforcement learning models.

The current force-based branch of our environment, in its first working release, readily shows all the problems that come with such an increase of scope. Namely, the new system experiences extremely slow training and very *slippery* progress, prone to large drops in performance as small changes spiral into slipping and crashing. For these reasons, when training in the same timeframe as an equivalent constant velocity model, our force model simply can't keep up.

All our sample force models have been trained using Proximal Policy Optimization, an algorithm which usually converges quickly, but the sheer amount of extra data involved in a more realistic physics system easily overwhelms the network. Encoding these new rules into a model takes exponentially more computational power. With all these problems, we haven't been able to train a force model until convergence. We could improve our odds with better hyperparameter tuning, but, in the end, there are

Conclusions

This project explores the effectiveness and ability of different reinforcement learning models including Proximal Policy Optimization (PPO), NeuroEvolution of Augmenting Topologies (NEAT), Deep Deterministic Policy Gradient (DDPG), and Deep Q Learning (DQN) to complete a Unity-based racing game built for the project. We made thorough comparisons regarding training effort (number of epochs to convergence), learning trend (observing by eye the best learning graph), and model complexity (the network complexity if it is using a deep network) and conclude following:

Algorithm:	Complexity	Max Score	Learning Time	Stability
DDPG	Moderate	160.3	Long	Average
DQN	Moderate	167.3	Long	Extremely Low
NEAT	Low	163.7	Short	Average
PPO	High	165.7	Average	High

Future Direction

1. Fixing the force-based model for the driving agent:
    a. So far, we have been using a velocity-based model, where we specify the velocity of the new position at each time step.
        i. For example, we calculate the new velocity the car should have after taking the action and then set that velocity to the game object.
    b. However, with a force-based model, we hope to include stochasticity on the action.
        i. For example, add a force to the car and let the game engine decide what happens depending on factors that may be out of our control (eg. weather conditions, slippery road, etc.)
2. Adding more actions
    a. Our current model does not have actions for acceleration/break. This is in our custom environment; our car will always have to accelerate to complete the track in the best time.
    b. However, in the future, we want to expand our work to a self-driving car, and not just a race car.
    c. Thus, we plan to add the following actions apart from the existing actions:
        i. acceleration
        ii. brake
        iii. do nothing
3. Train the car with more natural movements
    a. We want to make the car's movements more natural by
        i. dampening the turning
        ii. configure the mass and center of gravity of the car to simulate accurate acceleration and breaking of the car

References

We used some of the below code examples to get ideas and help when implementing our code

Models:

- PPO: https://github.com/nikhilbarhate99/PPO-PyTorch
- NEAT: https://github.com/CodeReclaimers/neat-python
- DDPG: xkiwilabs/DDPG-using-PyTorch-and-ML-Agents: A simple example of how to implement vector based DDPG using PyTorch and a ML-Agents environment. (github.com)
- DQN: https://github.com/transedward/pytorch-dqn

Languages:

- C#: https://docs.microsoft.com/en-us/dotnet/csharp/
- Python: https://docs.python.org/3/

Libraries:

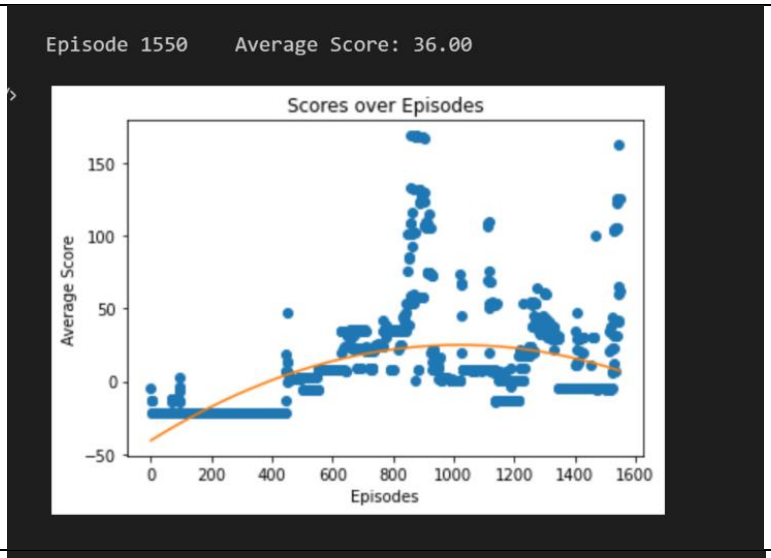
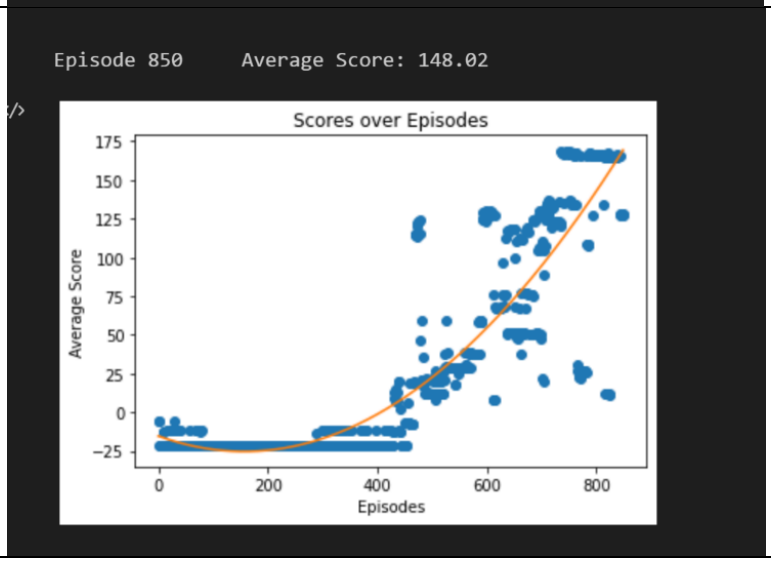
- Unity ml-agents: https://github.com/Unity-Technologies/ml-agents
- Python mlagents: https://pypi.org/project/mlagents/
- PyTorch: https://pytorch.org/docs/stable/index.html
- NEAT Python: https://github.com/CodeReclaimers/neat-python/blob/master/docs/index.rst
- Stable-Baselines3: https://github.com/DLR-RM/stable-baselines3

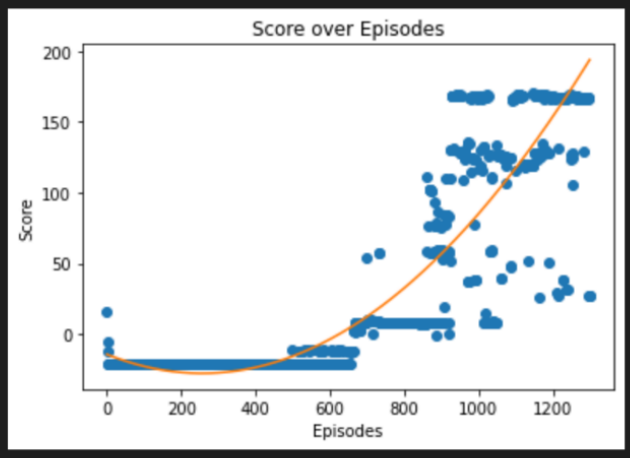
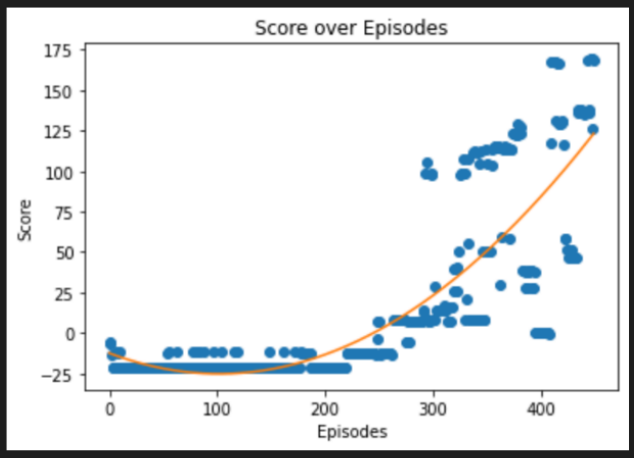
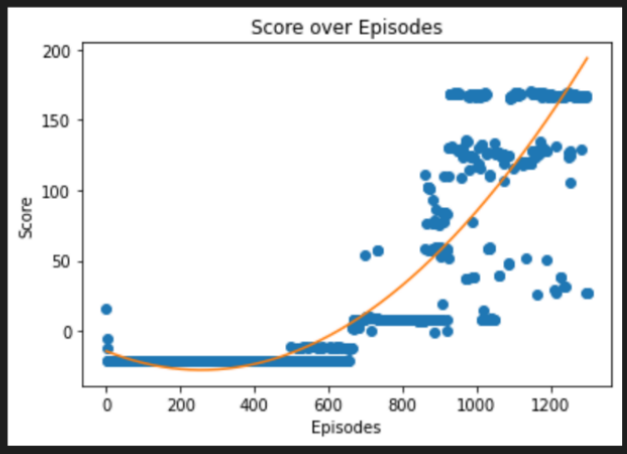
Papers:

- PPO: https://arxiv.org/abs/1707.06347
- NEAT: http://nn.cs.utexas.edu/keyword?stanley:ec02

Appendix

Additional Figures From Testing Prior to Development of Test Data

DDPG 2 Dropout layers at 0.2	
DDPG 1 Dropout layer at 0.2	

DDPG 1e-3 Learning Rate	Episode 1300 Average Score: 154.23 
DDPG layers with 128, 64 nodes	Episode 450 Average Score: 95.95 
DDPG Layers with 512, 256 nodes	Episode 1300 Average Score: 154.23 

DDPG with 0.01 of weight decay