ML Major Project Report

Group - 3

Sujay C Sharma(2017A7PS0012G)

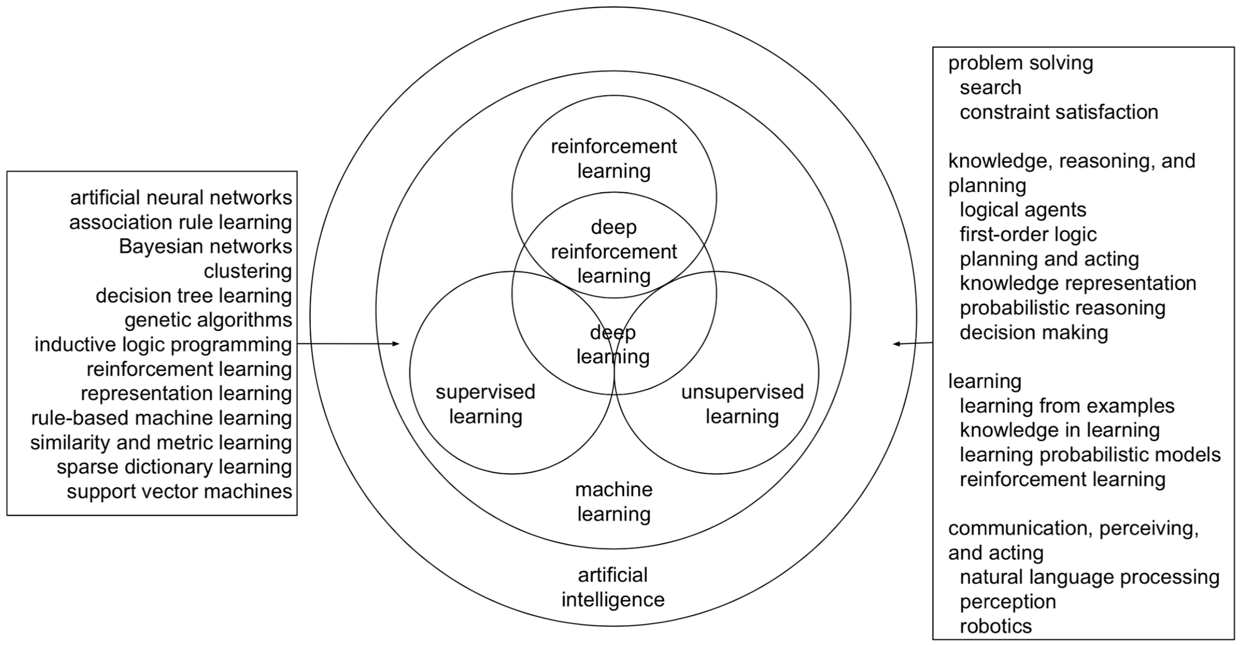
A Sri Sai Ganesh Reddy(2017A7PS0030G)

Rajath Reghunath(2017A7PS0059G)

Introduction:

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. Reinforcement learning differs from supervised learning in not needing labelled input/output pairs to be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The relation and general overview of where reinforcement learning stands in the context of machine learning and artificial intelligence can be understood with the help of the following illustration:



Yuxi Li, Deep Reinforcement Learning, https://arxiv.org/abs/1810.06339, 2018

A more detailed explanation regarding all things related to Reinforcement Learning and its relationship to Machine Learning can be understood from the accompanying paper [1].

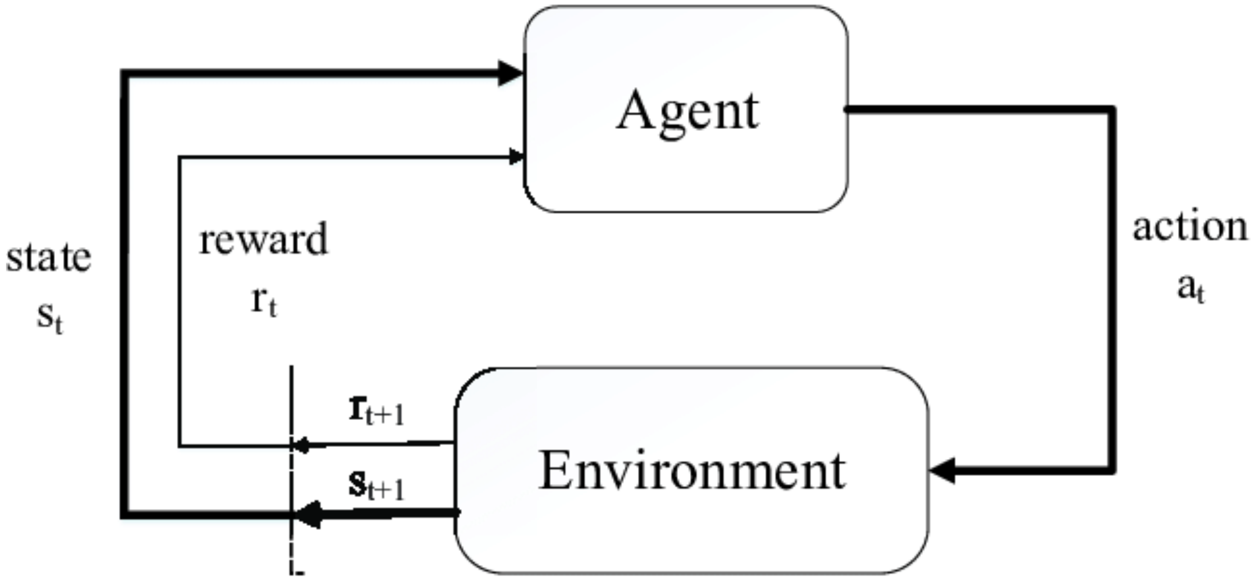
The core idea of reinforcement learning is to state the environment in the form of a Markov Decision Process (MDP). This is important as most reinforcement learning algorithms utilize dynamic programming techniques. Due to this reason, reinforcement learning is sometimes referred to as approximate dynamic programming particularly in the discipline of operations research.

A typical reinforcement learning problem is modelled as:

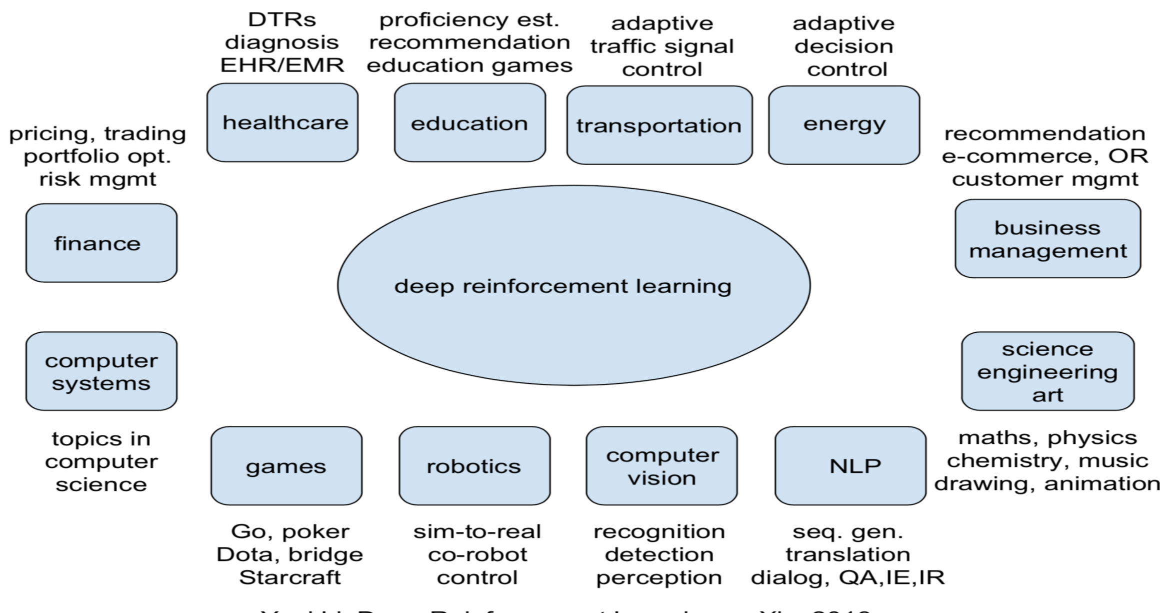
- An environment and a set of agent states
- A set of actions which can be taken by the agent
- Probability of a transition from State A to State B due an Action C taken by the agent.
- Reward associated with the transition from State A to State B due to an Action C taken by the agent.

- Rules describing what the agent observes.

A reinforcement learning agent interacts with its environment in discrete time steps. At each time *t*, the agent receives an observation o_t and a reward r_t . It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the *transition* (s_t , a_t , s_{t+1}) is determined. The goal of a reinforcement learning agent is to collect as much reward as possible.
The basic reinforcement learning framework can be understood by the following illustration:



Reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off. This makes reinforcement learning particularly well-suited to a wide range of real world problems as mentioned in the following illustration:



Yuxi Li, Deep Reinforcement Learning, arXiv, 2018

A detailed study regarding the history and applications of reinforcement learning, while beyond the scope of this report can be found in [2].

The Task:

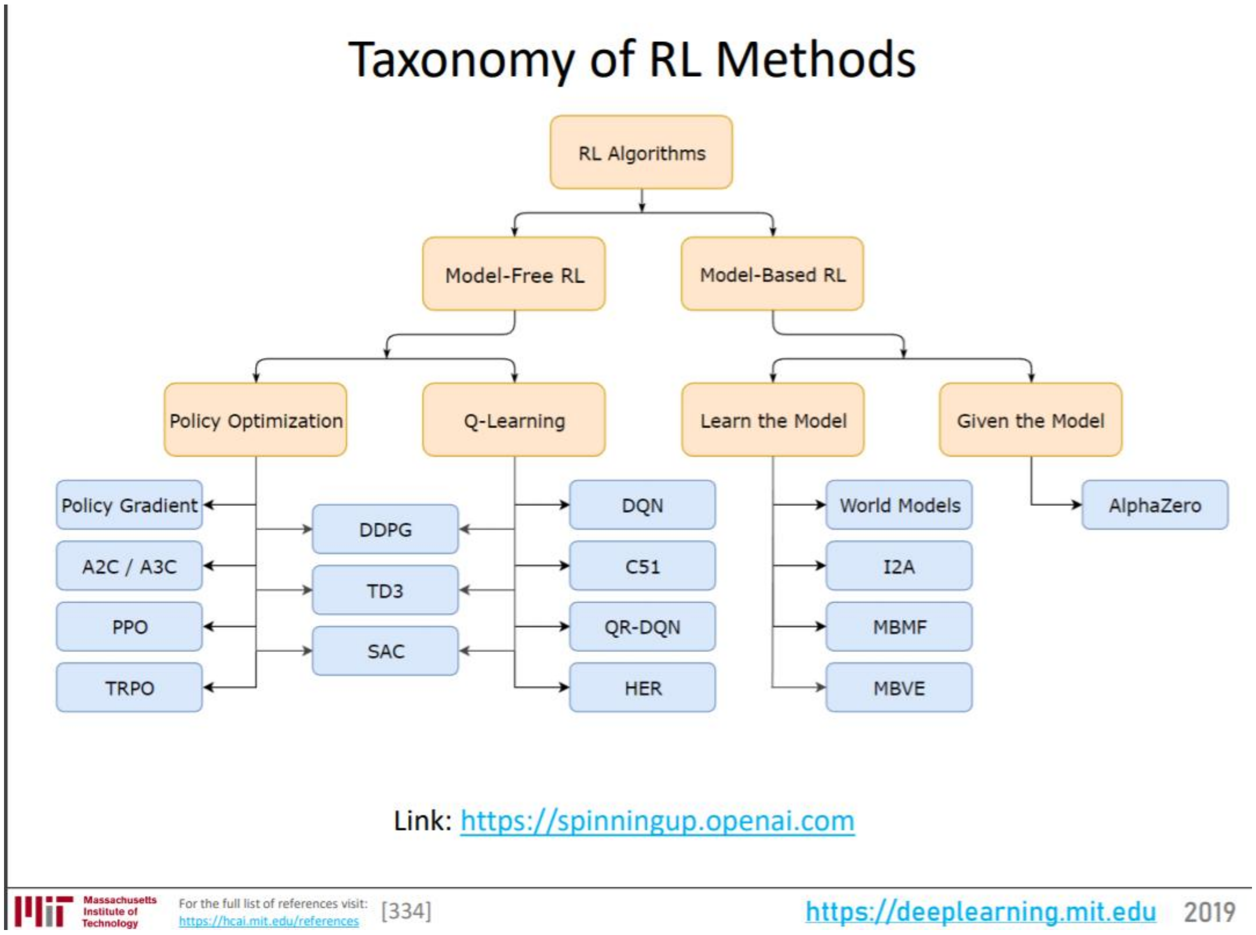
Three variations of a standard CartPole environment defined in OpenAI gym [3] were given as tasks.The standard CartPole environment consists of a pole and cart.The pole is attached by an un-actuated joint to the cart,which moves along a frictionless track.The pendulum starts upright and the goal is to prevent it from falling over by increasing and reducing the cart's velocity.This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson[4].

The first variation adds friction to the cart and pole, and random fluctuations in gravity[5].The second incorporates the first and on top of that introduces noise and noisy controls,which means that the cart's force in the desired direction is less/more than expected[6].The third is an extension of the second with noisy sensors/sensor observation of the pole angle at any moment[7].

An interesting real world simulation of the CartPole problem can be seen here [8].

Approaches:

Reinforcement learning algorithms can be classified as shown in the following illustration:



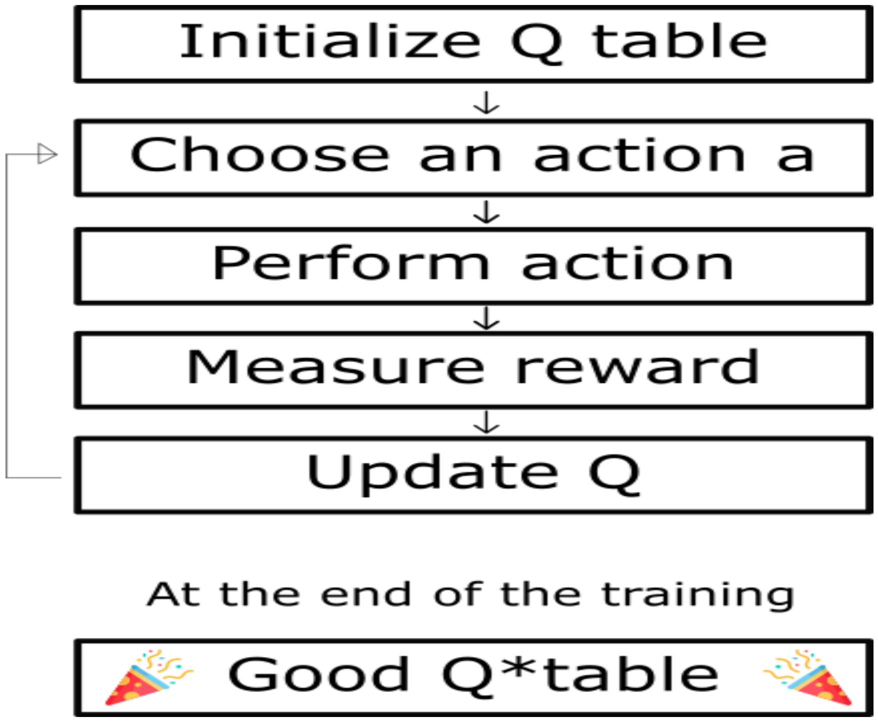
An explanation of the different types of algorithms can be found in [9].

After going through [9], we decided to try the following approaches:

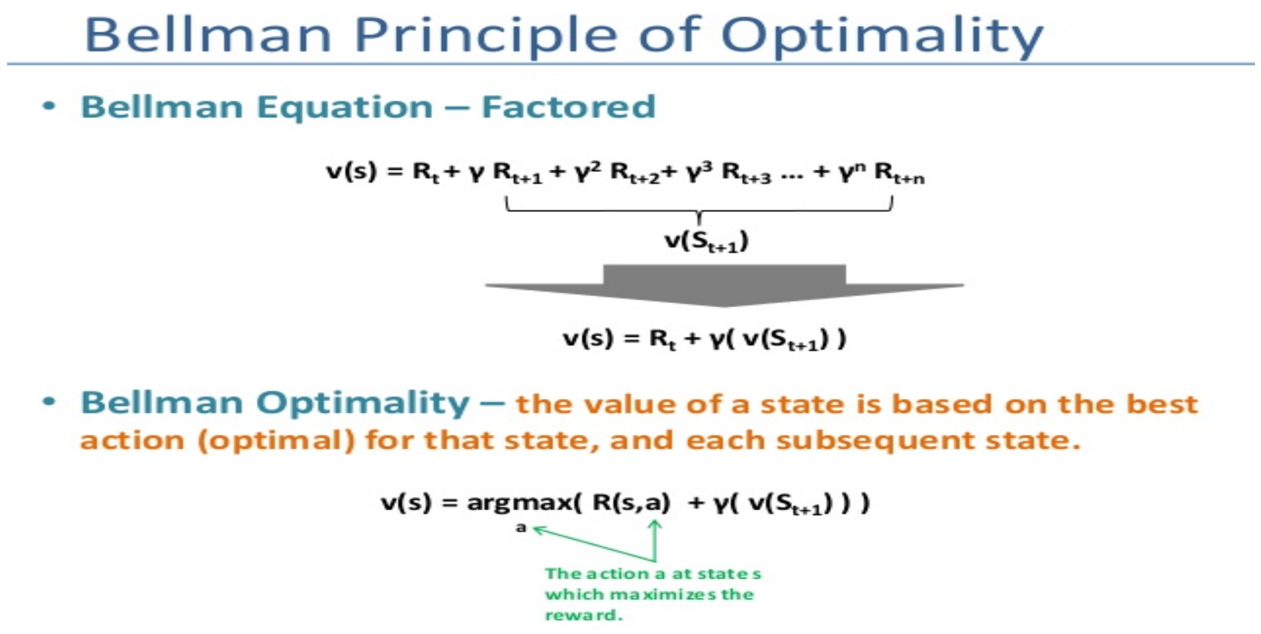
- Naive Q Learning
- Deep Q Learning (DQN)
- Proximal Policy Optimization (PPO)
- Genetic Algorithms

Naive Q Learning:

Q Learning is a model free reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. "Q" is the name given to the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state. A Q table is initialised with a finite number of states in it. The entries of the Q table correspond to the action that must be taken at that state. These entries are initialised to random values and are updated after every step using the Bellman equation. The Q learning process is described in following illustration:



The Bellman Equation is shown in the following illustration:



Bellman Principle of Optimality

- **Bellman Equation – Factored**

$$v(s) = R_t + \gamma \underbrace{R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} \ldots + \gamma^n R_{t+n}}_{v(S_{t+1})}$$

$$v(s) = R_t + \gamma(\, v(S_{t+1})\,)$$

- **Bellman Optimality – the value of a state is based on the best action (optimal) for that state, and each subsequent state.**

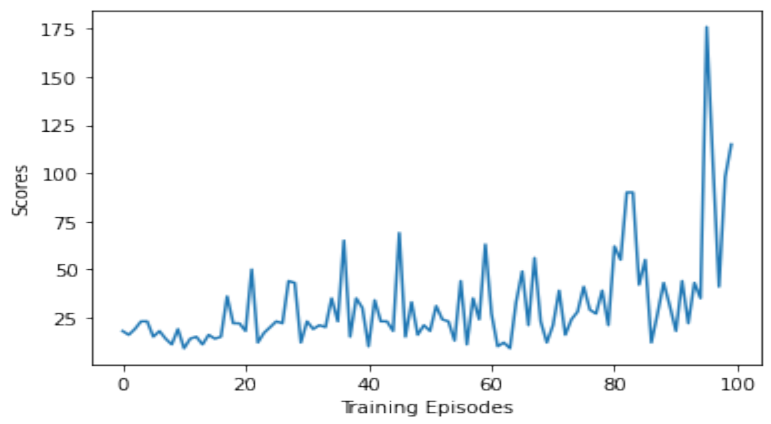
$$v(s) = \underset{a}{\mathrm{argmax}}(\, R(s,a) \;+ \gamma(\, v(S_{t+1})\,)\,)$$

The action a at state s which maximizes the reward.

A thorough description of the Bellman Equation and the proof of its convergence can be found in [10].

Applying the Q learning algorithm to the CartPole environment has one small caveat. The state returned by the Cartpole environment is a 4 Tuple (Cart Position, Cart Velocity, Pole Angle, Pole Velocity At Tip) of continuous values. Therefore, before applying the Q learning algorithm, these continuous values must be turned into discrete ones in order to construct the Q table.

After discretizing these state values, we trained our model over a 100 episodes and obtained the following results:



Hyperparameters used are:

- Discrete space = (1,1,6,12)
- Alpha = 0.1 (Learning Rate)
- Epsilon = 0.1 (Exploration Rate)
- Gamma = 0.9 (Discount Factor)

The 4 continuous values were discretized into a discrete space of dimensions (1,1,6,12) based on the observation that the CartPole rarely went off the screen and typically lost due to the angle exceeding the threshold upon rendering the episodes. Therefore, the parameters of Cart Position and Cart Velocity were not of much importance. This also helped reduce the complexity of the Q table.

A learning rate of 0.1 was used which is typically one of the standard values used along with 0.01, 0.001.

An adaptive exploration rate was used whose minimum value was 0.1 to ensure that the model took random values initially to explore the environment and then reduced the number of random actions gradually.

A high discount factor of 0.9 was used as the goal of the game was to survive for 500 steps and therefore, a high weightage was given to future rewards as well. Note that discount factors >= 1 are usually not advised due to the possibility of the Bellman equation not converging as shown in [10].

Although highly successful in the original CartPole game with an average score of 500 over 100 test runs, the model was erratic and could only obtain average scores ranging from 300 - 450 over 100 test runs for each of the 3 tasks. This led us to consider a more powerful model.

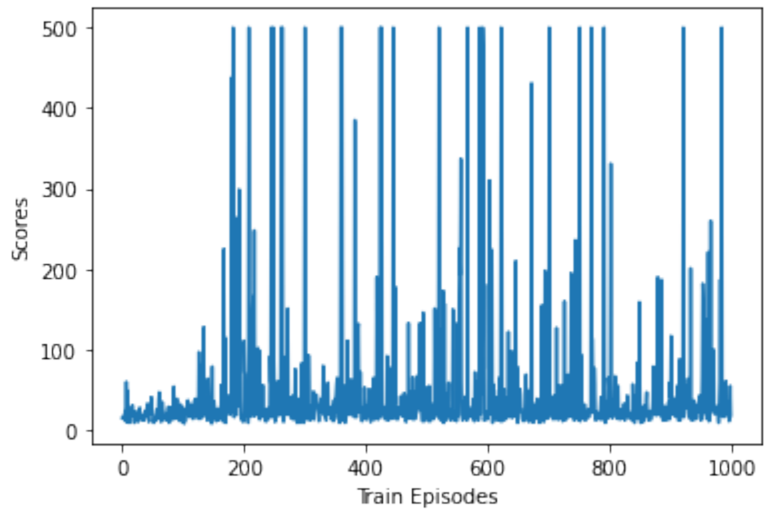
Deep Q Learning (DQN):

Deep Q Learning (Deep Q Neural Networks) is a variant of Q Learning in which deep learning architectures like neural networks are combined with the Q Learning algorithm to solve problems. The addition of neural networks makes the algorithm capable of scaling to previously unsolvable problems. As a neural network is capable of learning from raw continuous data from the environment, the issue of discretizing the state is no longer prevalent in Deep Q learning. This opens up a wide array of applications that were previously thought to be impossible to solve with reinforcement learning. Deep Q Learning is a very popular algorithm particularly for solving the games defined in OpenAI Gym such as Atari games as shown in [11].

Training of the neural network is done using a biologically inspired technique called Experience Replay where a random sample of prior actions are used for training rather than the most recent action.

PyTorch was used to implement the neural network architecture owing to the fact that it is beginner-friendly and has a vast amount of resources and tutorials about how to use it.

We trained our model over a 1000 episodes and obtained the following results:



Hyperparameters used are:

- Batch Size = 64
- Alpha = 0.01 (Learning Rate)

- Epsilon = 0.9 (Exploration Rate)
- Epsilon Decay = 0.995 (Decay in Exploration Rate)
- Gamma = 0.95 (Discount Factor)
- Experience Replay Buffer Capacity = 2000

Neural network architecture used is:

- Input Layer = 4 units (Observation Space for CartPole-v1 is 4)
- Hidden Layer = 50 units
- Output Layer = 2 units (Action Space for CartPole-v1 is 2)
- Activation Function = ReLU
- Optimizer = Adam
- Loss Function = Mean Squared Error

While most hyperparameters were set based on the same logic as the Q Learning algorithm, there were some minor tweaks here and there.

An adaptive exploration rate was used whose initial value was 0.9 and whose minimum value was 0.1 to ensure that the model took random values initially to explore the environment and then reduced the number of random actions gradually. The exploration rate was multiplied by the decay in exploration rate after every action was taken.

A high discount factor of 0.95 was used as the goal of the game was to survive for 500 steps.

The batch size was set to 64 which is one of the standard values along with 32 and 128.

The experience replay buffer capacity was set to 2000 as we wanted to store a significant number of experiences while still maintaining a small model size.

The neural network architecture used included a single hidden layer of 50 units as the CartPole environment is one of the simpler OpenAI Gym games with a state comprised of just 4 values. Using a single hidden layer was sufficient and kept the size of the model small.

The activation function used was ReLU (Rectified Linear Unit) as it is easier to compute, reduces training time and resolves an issue known as the vanishing gradient problem which usually affects large neural networks much bigger than the model used here. The advantages of using ReLU are mentioned in [12]. A further study of the vanishing gradient problem and methods to resolve it can be found in [13].

Adam is a popular optimizer used which was designed specifically for training neural networks. Information about Adam can be found in [14].

The loss function used is mean squared error.

The Deep Q Learning model was highly successful and managed to get consistent scores of 500 over 100 test runs in the original CartPole game and the 3 tasks as well.

Proximal Policy Optimization:

Proximal Policy Optimization is one of many algorithms based on the policy gradient theorem. More about policy gradient theorem and other algorithms based on it can be found in [15].The two main components of a policy gradient algorithm are the policy model and the value function.A value function is learnt in addition to a policy model because knowing a value function can assist in policy updates.This way of doing things is called the Actor-Critic Method.

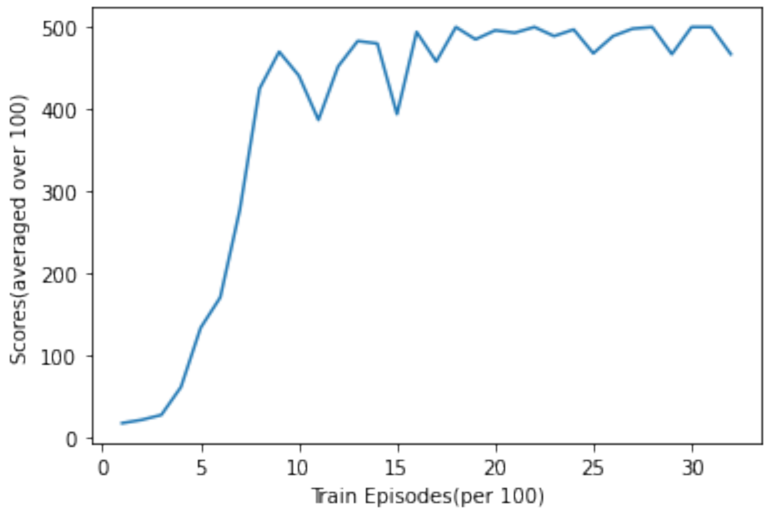
Actor-Critic consists of two models,which may optimally share parameters:

- Critic updates the value function parameters and depending on the algorithm it could be action-value or state-value.

- Actor updates the policy parameters in the direction suggested by the critic.

Two neural networks were used to model the actor and critic,since neural networks are very powerful function approximators.PPO follows the same ideology as TRPO[15],(don't change the policy too much at one step) but implements this differently by using clipped surrogate objective.Without a limitation on the distance between parameters of old and new policy, to maximize objective function would lead to instability with extremely large parameter updates and big policy ratios.PPO imposes the constraint by forcing the probability ratio between old and new policies to stay within a small interval around 1,precisely [1-eps_clip,1+eps_clip],where eps_clip is a hyperparametr.In addition to this, a form of memory was used to do a Monte Carlo estimate of state rewards.

After a few attempts, the training graph of the best model obtained is as follows:



The hyperparameters that gave the best results are:

- max_episodes = 3500 (max training episodes)
- n_latent_var = 64 (number of variables in hidden layer)
- update_timestep = 700 (update policy every * timesteps)
- lr = 0.002 (learning rate for both)
- betas = (0.9,0.999) (for adam optimizer)

- gamma = 0.99 (discount factor)
- K_epochs = 8 (update the policy for K epochs)
- eps_clip = 0.2 (clip parameter for PPO)

The neural network architecture used for actor model:

- Input Layer = 4 units (Observation Space for CartPole-v1 is 4)
- Activation Function = Tanh
- Hidden Layer = 64 units
- Activation Function = Tanh
- Hidden Layer = 64 unit
- Output Layer = 2 units (Action Space for CartPole-v1 is 2)
- Softmax
- Optimizer = Adam
- Loss Function = Surrogate

The neural network architecture used for critic model:

- Input Layer = 4 units (Observation Space for CartPole-v1 is 4)
- Activation Function = Tanh
- Hidden Layer = 64 units
- Activation Function = Tanh
- Hidden Layer = 64 unit
- Output Layer = 1 units (output of state-value function)
- Optimizer = Adam
- Loss Function = Surrogate

Most of the hyperparameters used were obtained by trial and error and other sources.It might seem that the update_timestep is too high,since judging by the max_episodes, the update only happens five times throughout it's training. After several tweakings of this parameter,we notice that if this parameter is too high the model doesn't learn much because the policy update happens very rarely and if this is too low the model forgets what it has learnt frequently,so an optimum value was necessary to achieve good performance which came out to be around 700.The learning rate is another hyperparameter that has a similar issue,where it shouldn't be too high or too low.The gamma hyperparameter is usually kept near 1.

The other hyperparameters such as number of hidden layers and number of units in each layers can also be experimented with.After a few trails, the best model obtained through this method gave 500 score consistently for task-1 and task-2.Apart from a few hiccups of 490's score here and there it was giving a 500 score for task-3 as well.More about proximal policy optimization algorithms can be found in the original paper published by John Schulman,Filip Wolski,Prafulla Dhairwal,Alec Radford and Oleg Kilmov [16].

Genetic Algorithm:

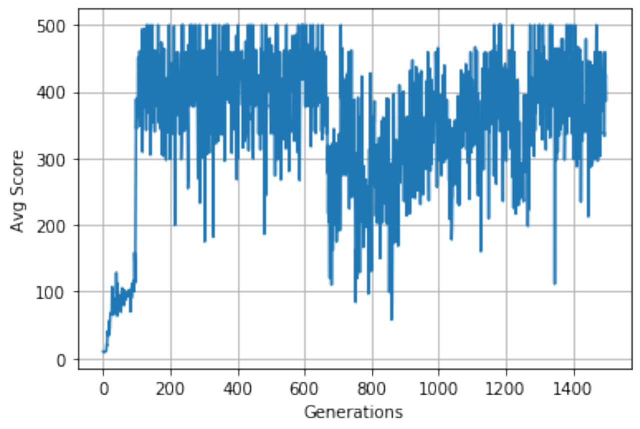
Genetic Algorithms (GA) are based on the theory of natural evolution. It is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). A GA uses a highly abstract version of evolutionary processes to evolve solutions to given problems. Each GA operates on a population of artificial *chromosomes*. These are strings in a finite alphabet (usually binary). Each chromosome represents a solution to a problem and has a *fitness*, a real number which is a measure of how good a solution it is to the particular problem [17].

Genetic Algorithms are valuable alternatives to gradient-based approaches. Gradient based RL approaches can be inefficient because often the agent does not always receive relevant information from the actions it took. Hence, it can remain stuck with a bad policy for a long time [18]. GA aims to prevent that through an artificial system of evolution. This involves certain phases like *selection*, *crossover* and *mutation,* operating on a *population*. Population consists of a set of agents or individuals (*chromosomes*) , and each chromosome is made of a set of genes that represent the state (parameters) of that agent.

In the selection phase, the fittest chromosomes are chosen from the population, based on a fitness function (that assigns each a *fitness score)*. Then a pair (parents) is chosen from this set, and in the crossover phase, new offspring are generated by exchanging the genes of these parents till a randomised crossover point. In the mutation phase, some of these offspring are subjected to mutation (to add random noise). This helps with diversity and premature convergence[19].

The genetic algorithm implemented uses a very simple neural network and survival of the fittest policy. Initially, it generates a random population to generate diversity in training data. Then the algorithm is run for a certain number of generations. Each generation is allowed to reproduce. This stage involves choosing two individuals with the best fitness from the previous generation. Then they are subjected to crossover (which generates new chromosomes) and mutation (adds random noise). The new offsprings are checked for their fitness (here, we've set fitness policy as getting score 500) and only the crossed over set with the fittest chromosomes are retained in the model. Future generations will replace this 'best generation' as the model matures according to the fitness policy.

We ran the model for 1500 generations on task2 and the generated model was able to support the other two tasks as well.



Hyperparameters used:

- test_runs = 10 (test runs to initialize the population randomly)
- num_generations = 1500
- in_node_num = 4
- hid_node_num = 2
- mutation_power = 10
- crossover_generations = 10

Neural network architecture:

● Input Layer	:- 4 units (reLu)
● Hidden Layer 1	:- 4 units (reLu)
● Hidden Layer 2	:- 2 units (reLu)
● Output	:- 1 unit (softmax)

The model seems to rely a lot on the randomness involved in the algorithm. Though it generates a good model that works across all three tasks most of the time, occasionally, we have noticed that the generations might fail to converge to a solution, becoming worse and staying so after the first few generations. However, those cases are very rare and running the training again fixes it.

Inferences and Conclusions from our Approaches:

- **Training Methodology:** All of our models except the genetic algorithm one performed better when training was done on the original cartpole environment followed by using the frozen model for 100 runs on each of the 3 modified environments as opposed to training the models on each environment separately and then evaluating them. We suspect that this is due to the random variations in gravity, friction and sensor output making it difficult for the model to learn the optimal policy while training.
- **Adaptive Exploration Rate:** Using an adaptive exploration rate with a decay factor led to far better results than using a constant exploration rate throughout the training process. The adaptive learning rate helped the model to make more random actions initially and explore the environment while limiting the random actions taken as the model began to learn and improve.
- **Reward Function:** The default reward returned by the CartPole environment is a reward of 1 for every step taken by the cart. This constant reward function makes it harder for the model to realise what actions taken were good ones and what actions taken were bad ones. To resolve this issue, a custom reward function was used based on the 4 state parameters returned by the environment. This led to improvements in performance particularly for the Q Learning and Deep Q Learning approaches.
- **Best Performing Model:** The best performing model for all 3 tasks of the 4 approaches attempted ended up being the Deep Q Learning Model which was trained on the original CartPole environment for a 1000 episodes.

Future Scope:

- While a Deep Q Learning model was sufficient to solve the 3 tasks given, a lot of improvements have been made in the field of deep reinforcement learning beyond Deep Q Learning such as Double DQNs, Multi-step DQNs and prioritised experience replay where the experiences from which the model learns are chosen specifically to enhance learning. A detailed study of advances in deep reinforcement learning can be read about in [20].
- The models used have been trained on the original CartPole environment as traditional reinforcement learning models are vulnerable to noisy scenarios. However, studies have been ongoing to build reinforcement learning models that are noise resistant and can be trained in noisy settings as this often mirrors real world scenarios. An example of training models in a noisy variant of the CartPole, Pendulum and Atari environments can be found in [21].
- Ensemble methods can be used that combine multiple different reinforcement learning algorithms in a single agent. This can enhance learning speed and final performance by combining the chosen actions or action probabilities of different algorithms. An implementation of ensemble methods in reinforcement learning can be seen in [22].

Contributions:

Naive Q Learning and Deep Q Learning:	**Sujay C Sharma**
Proximal Policy Optimization:	**A Sri Sai Ganesh Reddy**
Genetic Algorithm:	**Rajath Reghunath**

Note: Although these were the approaches assigned to each member of the group, we used the platform of Google Colab and helped each other to finish all the approaches.

References:

[1] : Li, Y. (2018, October 15). Deep Reinforcement Learning. Retrieved from https://arxiv.org/abs/1810.06339

[2] : Sutton, R. S., & Barto, A. (2018). *Reinforcement learning: an introduction*. Cambridge, MA: The MIT Press.

[3] : Openai. (n.d.). openai/gym. Retrieved from https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

[4] : A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834-846, Sept.-Oct. 1983.

[5] : task1.py. (n.d.). Retrieved from https://github.com/ganeshalamuru/ML_main_project/blob/master/task1.py

[6] : task2.py. (n.d.). Retrieved from https://github.com/ganeshalamuru/ML_main_project/blob/master/task2.py

[7] : task3.py. (n.d.). Retrieved from https://github.com/ganeshalamuru/ML_main_project/blob/master/task3.py

[8] :Liu, Cheng-Yueh. [Cheng-Yueh Liu]. (2017, October 15). *Cart-Pole Reinforcement Learning* [Video file]. Retrieved from https://www.youtube.com/watch?v=5Q14EjnOJZc

[9] : Part 2: Kinds of RL Algorithms¶. (n.d.). Retrieved from https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[10] : A.g., Barto, Bradtke, Barto, Kushner, Clark, … P.j. (1991, January 1). Q-learning. Retrieved from https://link.springer.com/article/10.1007/BF00992698

[11] : Volodymyr, Koray, David, Alex, Ioannis, Daan, … Martin. (2013, December 19). Playing Atari with Deep Reinforcement Learning. Retrieved from https://arxiv.org/abs/1312.5602

[12] : What are the advantages of ReLU over sigmoid function in deep neural networks? (1964, September 1). Retrieved from https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks

[13] : Huber, Adrian, Jithendar, Liu, & Shih-Chii. (2019, July 5). Overcoming the vanishing gradient problem in plain recurrent networks. Retrieved from https://arxiv.org/abs/1801.06105

[14] : Kingma, P., D., Jimmy, & Ba. (2017, January 30). Adam: A Method for Stochastic Optimization. Retrieved from https://arxiv.org/abs/1412.6980

[15] : Weng, L. (2018, April 8). Policy Gradient Algorithms. Retrieved from https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[16] : Schulman, John, Wolski, Filip, Prafulla, Radford, … Oleg. (2017, August 28). Proximal Policy Optimization Algorithms. Retrieved from https://arxiv.org/abs/1707.06347

[17] : McCall, John (April 2005). Genetic algorithms for modelling and optimisation. Retrieved from https://doi.org/10.1016/j.cam.2004.07.034

[18] : Chopra,Para (Jan 2019). Reinforcement learning without gradients: evolving agents using Genetic Algorithms. Retrived from https://towardsdatascience.com/reinforcement-learning-without-gradients-evolving-agents-using-genetic-algorithms-8685817d84f

[19] : Mallawarachi, Vijini (Jul 2017). Introduction to Genetic Algorithms. Retrieved from https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[20] : Hessel, Joseph, Hasselt, van, Hado, Schaul, Tom, … David. (2017, October 6). Rainbow: Combining Improvements in Deep Reinforcement Learning. Retrieved from https://arxiv.org/abs/1710.02298

[21] : Wang, Liu, Yang, Li, & Bo. (2020, February 1). Reinforcement Learning with Perturbed Rewards. Retrieved from https://arxiv.org/abs/1810.01032

[22] : Wiering, M. A., & Hasselt, H. van. (n.d.). Ensemble Algorithms in Reinforcement Learning. Retrieved from https://ieeexplore.ieee.org/document/4509588