Cooperation and Competition based Mechanisms for Multi-Agent Self Play and Transfer Learning

Shiv Kumar, Department of Information Technology,
Netaji Subhas University of Technology

12 October 2019

1 Introduction

Ever since the first programmable computers were conceived, researchers have wondered whether such machines would one day become intelligent. Machine Learning has dynamically changed the fundamental way in which general artificial intelligence based problems are being solved today. These problems range from being able to make automated decisions for certain scenarios to recognising patterns in data streams that are allowing companies and researchers to develop insights about man kind in a way never seen before. allowing better generalisation on functions which define the behaviour or pattern of a given sample of data. One subfield of Machine Learning is Reinforcement Learning which deals with learning from feedback or experience in order to maximise the reward function (the Reward Hypothesis). This field learns patterns in behaviour and is said to be a way close to how humans learn. This field is seen as a clear pillar on which General Artificial Intelligence can be seen to stand on in the coming decades. This field has infinite applications wherein the autonomous agent or herein the machine can learn without supervision. This would allow for machines to be trained in problems where even humans do not have sufficient knowledge or where domain knowledge is very limited. This kind of learning elevates the need for marking the behavior of each input feature as is the requirement in supervised learning and just requires the definition of positive and negative rewards, and the freedom to try different actions. This field has also gained attraction due to the rise of deep learning which allows for efficient representation learning and coupled with reinforcement learning allows for brilliant applications. An application of which can be seen as the performance of a RL algorithm called DQN, which learned how to play Atari games directly from the video of the gameplay, without any supervision. This field has numerous applications and will be seen playing a major role in elevating the problems faced by major chunk of technologies of the future, like Self Driving Cars, or Autonomous Drones.

2 Motivation

Most of the research done in Reinforcement Learning is in the domain of single agents where the agent just interacts with an environment and has a stationary distribution to sample states, actions and rewards from. Whereas the major real world problems where RL is seen as a prospective break through deal with multi agent environments. Therefore this study deals with the behaviour of agents in the presence of other agents which themselves are learning. This study deals with how these learning agents cooperate or compete in order to maximise selfish or common rewards and goals.

Till now the neural nets employed in Deep RL are known to be not so deep. This is because of the over fitting seen in Deep RL wherein agents over fit over the underlying environment and hence face difficulty in generalising to other similar environments which humans do with ease. This study deals with how this overfitting can be reduced allowing for deeper neural networks to be incorporated into the general paradigm of Deep RL. Furthermore this project deals with how knowledge gained while learning in one environment can be used to interact efficiently in other environments, which can allow for more practical and efficient use of Deep RL.

Furthermore the training of Deep RL networks are known to be very slow. This study also gives a brief overview as to how the training can be made more efficient in general through techniques like quantisation and pruning.

3 Brief Background

3.1 Markov Games

Here we consider multi-agent Markov games(Littman,1994).A Markov game for N agents is a partially observable Markov decision process (MDP) defined by: a set of states S describing the state of the world and the possible joint configuration of all the agents, a set of observations O_1,...,O_N of each agent, a set of actions of each agent A_1,...,A_N, a transition function T : S $\times$ $A_1 \cdots A_N \rightarrow$ S determining distribution over next states,and a reward for each agent i which is a function of the state and the agent's action r_i : S $\times$ $A_i \rightarrow$ R. Agents choose their actions according to a stochastic policy $\pi_{\theta^i} : O_i \times A_i \rightarrow [0, 1]$, where θ_i are the parameters of the policy. For continuous control problems considered here, π_θ is Gaussian where the mean and variance are deep neural networks with parameter θ. Each agent i aims to maximize its own total expected return

$$R^i = \sum_{t=0}^{T} \gamma^t r_t^i$$

where γ is a discount factor and T is the time horizon.

3.2 Value based methods

Value based methods are solutions to the RL Problem wherein the policy of the agent is generated through the estimates of the action value function. State value function defines the total discounted reward that can be earned from the current state, whereas action value function defines the total discounted reward that can be generated from the current state on taking a particular action. So the policy is generated by sampling the epsilon greedy action where the greedy action has a relatively higher probability of being selected but other actions also have some probability of being sampled, allowing us to taking the exploration and exploitation dilemma, which happens to be one of the fundamental issues in Reinforcement Learning. The value function can be estimated using a table mapping between states and actions or as is being done today using a neural network. The neural network allows for a generalisation over the continuous states and therefore happens to be a more robust way of estimating the value function.

3.3 Deep Q-Network

Deep Q-Network or DQN is the neural network alias of estimating the value function for the agent. The parameters of the value function are the weights of the neural network. It consists of a set of two neural networks. One acts as the estimator and the other acts as the target value generator. The target value generator uses older but periodically updated values (updated from the estimator network) to stabilise the process of updating the weights of the estimator network. Depending on the problem at hand, the network can also learn representations from the observations and hence does not require explicit feature engineering. DQN uses a replay buffer which encompasses the process of first collecting some amount of data sampled using the current value function and then randomly picking shuffled batches of experience from this buffer for training of the networks which solves the issue of correlation between different state action pairs.

3.4 Policy Gradient methods

Policy gradient methods are the set of solutions to the RL problem wherein the policy is estimated directly, rather than depending on the value function. Here the objective function is defined in terms of the policy parameters (weights of the network) which allows optimisation of the parameters in a way similar to that of the maximum likelihood function, wherein the weights of the network are tuned in the direction of gradient of objective function with respect to the weights. A process known by the name of gradient ascent. The simplest policy gradient method is REINFORCE, which involves sampling of trajectories from the current policy estimates. Using the trajectories to estimate the gradient of the objective function and using the gradient to update the parameters of the policy. Due to the high variance involved in the reward estimations (especially in the earlier training steps), REINFORCE is known not to perform really well on challenging continuous state problems. The variance problem is handled by involving two key methods i.e. reward to go (through the law of causality) and using a baseline to calculate the advantage of a certain action over the typical action for the current state by reducing the reward from current state on the particular action on subtracting the mean possible reward.

3.5 Actor Critic Methods

Wherein normal policy gradient methods use the average possible rewards for the current state on all possible actions as a baseline, the actor-critic methods are a type of policy gradient methods wherein an estimate of the value function is used as the baseline (referred to as the critic, *which defines how good the current policy or actor is*). This method is observed to be very efficient in reducing the variance problem of the policy gradient problem, allowing for faster and more stable in the learning process.

3.6 Deep Deterministic Policy Gradient(DDPG)

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance. Since DDPG is off-policy and uses a deterministic target policy, this allows for the use of the Deterministic Policy Gradient theorem. DDPG is an actor-critic algorithm as well; it primarily uses two neural networks, one for the actor and one for the critic. These networks compute action predictions for the current state and generate a temporal-difference (TD) error signal each time step. The input of the actor network is the current state, and the output is a single real value representing an action chosen from a continuous action space. The critic's output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal.

3.7 Proximal Policy Optimization(PPO)

Proximal Policy Optimization is an improvement over the sample inefficient REINFORCE, wherein the concept of importance sampling is used to make more efficient use of the sampled trajectories. Policy gradient algorithms are based on the fact that the true value of the objective function is the probabilistic weighted sum of all possible trajectories for a problem. Since all trajectories are computationally impossible to sample for high dimensional state space, therefore we operate over the expectation over many sampled trajectories to compute the true value with an assumption that as the number of sample trajectories approaches infinity, the

true expected value will be equal to the true value. In PPO, the premise is that the same sampled trajectory might have a different weight when the policy improves but doesn't necessary mean that the trajectory can't be sampled at all, and hence allows us to reuse the sampled trajectories again to estimate the objective function. This method mathematically does introduce a bias in the estimation but can be ignored if the policy estimations are relatively close (*proximal*). This method has given state of the art results in many problems in the domain of RL, and is also employed in this project.

3.8 General Advantage Estimate(GAE)

General Advantage Estimation is a concept based on the generalisation of the n-step bootstrapping concept, wherein the actual reward of the trajectory is taking for the next n-steps, and post that the discounted reward for the trajectory is estimated through the estimate of the value function. GAE, goes a step further and rather than agreeing on a single n to cut of the current trajectory, it takes in the weighted average of the different n step parameters, which gives a much better estimation of the expected reward for the trajectory without having the need to explicitly roll out the whole trajectory before modification.

4 Methodology

4.1 Development of game environments in PyGame to support multi agent behavior

The prime objective of the project is to be able to develop environments suitable and appropriate to the study of Multi agent reinforcement learning. These include games like the snakes game, robber-police game, the proximity tracking game, among many others. These games will be implmented in a python framework called Pygame. This framework was selected after doing 2 weeks of development in different frameworks. It was selected due its ease of use, community support and the offered featureset for the framework. These environments are chosen due to their ease of development for multi agent environments. The environments used in literature generally limit the number of agents to two. These are also being developed so as to focus more on the learning algorithms and their efficiency, rather than the complexity of the environment. Since more complex environment lead to more learning. We are going with simpler environment to solely focus on learning based on interaction with the other agents in the environment. This reduces the need for initial dense rewards initialisation which is needed in dexterity based environments where the initial learning is based on the need to learn basic behaviours like walking without falling, kicking, resisting, etc.

4.2 Development of Openai gym wrappers for the developed games to ease interfacing with algorithms

The developed games will then be wrapped in appropriate Openai environments. This brings in a level of abstraction which allows the programmer to focus on one aspect of the process at a time. The Openai environment allows a perfect simulation of an environment for the agent and hence makes implementation and debugging of the codes much easier. It separates the implementation of algorithms and environments which is much needed for effective development of the codes related to the research.

4.3 Baseline Implemenation of Double DQN, DDPG for single agent, DDPG for symmetrical multi agents, PPO with GAE, and MADDPG for unsymmetrical multi agent environments

It is imperative to check the correctness of the set of algorithms on standard benchmarks. Once the standard results are met and requisite experiments are done for the environments, they will be ported to our environments to check the behaviours of agents under different circumstances ranging from being alone to having many different agents in the same environment.

The Double DQN is the standard value based function approximation method and is known to give state of the art results for multiple problems in RL.Double DQN is benchmarked on the Navigation environment in Unity.

The DDPG algorithm is the standard policy gradient and actor critic algorithm and is known to give state of the art results for continuous action spaces. It is also known to train faster due to the direct mapping from states to actions. DDPG for single agents is benchmarked on the Continuous Control environment in Unity. DDPG can be applied to symmetrical multi agent case, as the agents can use the same policy and value function networks since every factor stays same irrespective of the behaviour of the other agents. This was benchmarked in the distributed learning version of Continuous Control environment of Unity and in Tennis, two player multi agent game in Unity as well.

Note: The results for the implementation of the above explained algorithms on the respective Unity environments are shown in the experimental results section.

PPO with GAE is the state of the art sample efficient policy gradient method and is used for competitive self play and will be further employed to study the results in more than 2 agent environments. This is the algorithm which primarily operates on self play. This algorithm will be benchmarked on the dexterity environment in MuJoCo.

MADDPG is the multi agent variant of DDPG where symmetrical environments are not a necessity. Here each agent has its own policy network and a centralised critic network which is optimised by each agent. The specific actors are used while running whereas the critic is only employed during the training phase. MADDPG will be benchmarked on the Tennis environment of Unity.

4.4 Training the agents using the implemented algorithms

The agents in different games will be trained using these benchmarked implementations, and the consequent behaviour will be observed, as to how the agents cooperate and compete in different settings.

4.5 Transfer learning on other environments

This is the main focal point of the project and is the main novelty as well. This project aims to study the robustness of generalisation in different environments and how skills learnt in one environment can be carried forward to different environments. We plan to allow the agents to learn on different environments separately and will then observe how well they can generalise on very similar and very different environments and how well the agents learn when the network weights are initialised with the pre trained weights of other environments.

In this project we also plan to play with the concept of transferring knowledge from discrete action spaces to continuous action spaces, and vice versa.

4.6 Quantisation and other efficiency aspects of the training and inference phase

The primary datatype in which neural network arithmetic is performed is FP32, which have a relatively larger access time, and storage overhead as compared to INT32, INT16, INT8 and INT4 datatypes. The process of quantisation relates to the conversion of the floating point values to the required quantisation level values, and evaluating the effects on training accuracy for supervised learning problems. The same premise would be tried on the deep networks employed in Reinforcement learning problems. Quantisation can be done during training and post training. The process of quantisation done during training can be theoretically reasoned to impact the agents' ability to explore, and hence in this study we try the quantisation policy once a certain number of episodes have been used to train the model, which would allow us to use relatively less storage to store the model and also allows for faster inference which will allow the environment to run faster when the training is completed.

Another possible track could to be have a hyper parameter which controls the amount of pruning done in the network. As the agent learns more about the environment, more pruning can be changed accordingly. The related results will be reported in the final project report.

4.7 Environment Randomisation and Ensemble Policies

In order to learn robust policies which generalize better we plan to introduce randomness in the environment, for example the arena radius for the snake environment can be randomized, agents' start positions can be randomized, the number of opponents for a particular agent in an episode can be randomized. However, we found that while randomization is crucial to learn policies which generalize better, it might hinder learning early on as there might be too many things for the agents to explore. Thus, in order to learn policies that generalize well, we introduce a simple curriculum in the randomization where we plan to start with a small amount of randomization which is easier to solve and operate with and then gradually increase the randomization during training.

Another related problem that was observed is over-fitting to the behavior of the opponent when training for very long. This results in policies which are good against particular types of opponents but do not generalize to other opponents (say opponents trained with a different random seed). This over-fitting can also be observed in win-rates against opponent during training, where one would see oscillations as agents try to adapt to their particular opponent and changes in their strategies. To overcome this we plan learning multiple policies simultaneously. Thus, there is a pool of policies and in each rollout for a particular policy one of the other policies is selected at random as the opponent (in symmetric games, the same policy can also be an opponent). In this case, the pool of all policies as opponents – current and throughout the history of training – creates a natural distribution over related tasks for multi-task learning. We found random policy initialization to provide enough diversity between agent policies, however techniques that explicitly encourage diversity can potentially be incorporated in the future.

4.8 ELU vs ReLU + Batch Normalisation

In the literature related to Deep RL, a major factor in performance variability is the decision to use ELU or ReLu along with batch normalisation for more stable and effective learning. The decision was seen to directly seen to impact the training process, where ELU gave better results than the pair of ReLU and batch normalisation for each implementation ranging from Double DQN, DDPG, and DDPG for symmetrical multi agent environments. Further tests will be done to report results on our environment and a logical assertion for the same will be investigated.

5 Application of the work

The problems that this project deals with are known to directly impact the real world problems faced by scaleable adaptation of RL. Some of the applications of Multi agent deep reinforcement learning are as follows:

5.1 Traffic Management System

The multi agent reinforcement learning premise can be used to train traffic lights to coordinate between each other depending on how light or heavy the traffic is on their side. Traffic lights operate in different environments and places, and hence a more robust and generalised setting can help them improve their efficiency. This is studied in this project through the agent learning in randomised environments with randomly sampled agents.

5.2 Swarm of autonomous drones for safer autonomous traversal

The algorithms designed for multi agent reinforcement learning can be used in the safe and efficient operation of a swarm of autonomous drones for faster delivery of products. The different agents can coordinate and cooperate to deliver optimal amounts of goods by increasing or decreasing the per capita drones in an area depending on the amount of shipments in an area. The randomised environment based robustness can be seen to be a factor here as well. Also, a study of how different agents interact in as many different environments is good for the community as a whole.

5.3 Safe movement of Self Driving Cars

The algorithms designed for multi agent reinforcement learning can be used in the safe movement of a set of Self Driving Cars in different sets of neighbourhood. The different agents can coordinate and cooperate when to move and when to halt to avoid collisions and consequent impacts. The environments are different in each country, that too in a specific localities, and by learning to generalise on different forms of the same environment allows for the agents to have better knowledge of the environment.

5.4 To compare the techniques used by the RL agents and real humans for a given environment ranging from sports to dexterity

A basic premise on why RL is being pursued so much is the fact that it allows humans to identify the biases humans might have had while learning in certain environments, and allows us to discover strategies and techniques that we as humans could not come up with. Clear examples of the same are the alphaZero and also the experiments conducted on Atari games which saw the AI agents to come up with strategies that we as humans did not even think about in the past. And that is the most important application of this work.

6 Work till date

As I was new to the field of Reinforcement Learning when I picked up this project, the primary work done till now involves getting accustomed to the field and the related literature along with implementing all the required papers and algorithms from scratch. I dedicated around a month and a half to learn the basic fundamentals of the field, and am still learning more concepts which would further broaden my understanding on how to encounter the problem statement. After getting a basic understanding of the field, the need for a simulated environment customary to the needs of the project was understood. The process of establishment of the environment has started with the snake game complete for a single player support, which should be completed within a week. The environment construction requires additional knowledge of game development frameworks in python like pyGame, which itself took around a week to get accustomed to. The constructed environment then had to be wrapped up in an Openai gym wrapper, so as to ease the process of experimentation and interfacing. Openai gym is a set of environments open sources by Openai to allow researchers to conduct meaningful research in the field of Reinforcement Learning. Openai allows for the addition of custom environments and that is one of the targets of the project. Once completed the environment used in this project will be open sourced to other researchers enabling them to conduct their own studies.

The project also needed some baseline implementations checked on standard environments to benchmark the performance which allows us to know whether the written code is performing as expecting and at-least resolves one key source of problem in the project implementation. Till now baseline implementations of Double DQN, DDPG and DDPG for Symmetrical Multi Agent Environments have been completed from scratch where each took multiple days of training and debugging to complete. PPO should be completed by the end of this month, once all environments are completed. The results for the same are shown in the next section.

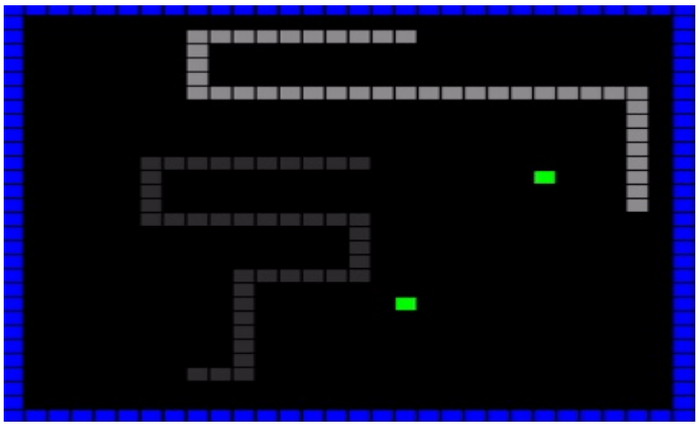
Since the problem of over fitting on a single environment is one of the fundamental problem in Deep RL. Several Regularisation methods specific to neural networks operated in Reinforcement Learning was also experimented with techniques like dropout and batch normalisation giving positive results in some scenario and negative in others (further details in the next section). Other techniques like Cutout, Shake Shake along with different data augmentation techniques for image based observations of the environment will also be studied. These techniques will allow our agent to generalise better on a set of different environments.

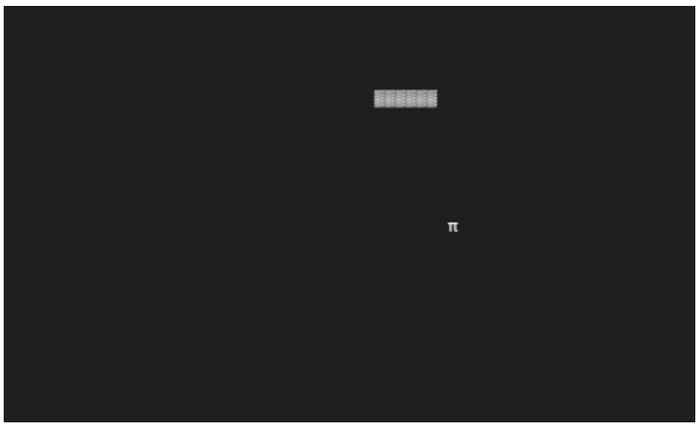
Another key research element of the project was the use of quantisation and pruning to make the neural networks more efficient. This is combined with the study of more efficient neural network models and techniques (read depth wise convolution) to improve the training time of the models which takes anything between a couple of hours to a couple of days depending on the complexity of the environment.

7 Experimental Results

7.1 Implementation of snake environment

The snake game environment is meant to be the key foundational block on which the entire project rests. The implementation of the same has been tried on multiple different frameworks to check better suitability to the Openai Gym Environment. The related screenshots of the environments as attached. The environments support random environment initialisation and need to be supplemented to support ensemble policy opponent sampling.





7.2 Double DQN implementation

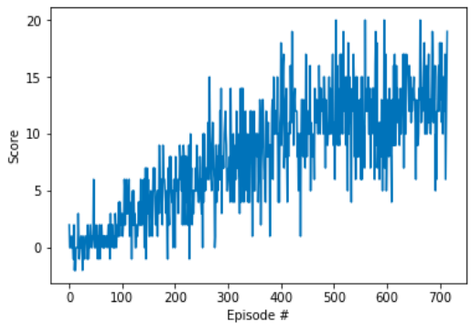
Double DQN algorithm was used to solve the Navigation environment which is a part of the group of environments made available by NVIDIA through Unity. Double DQN algorithm involves the use of two networks namely target and local to map states to action values to improve reward using fixed targets methodology, while operating on a memory buffer to be able to sample different uncorrelated episodes to remove the bias that can get introduced in the agent's behaviour due to the correlation. The neural network takes place of the Q table which was handy in discrete state space in mapping states to action values. Double DQN is an improvement over the normal DQN where the local network was used to select actions for future states, and those actions were used in the target network to estimate rewards. This is done to avoid over estimation. The rewards are high only if both networks of the agent agree on the actions being used. Else the rewards will be less. In the long run, the algorithm prevents the agent from propagating incidental high rewards that may be obtained by chance and do not reflect long term returns (*Overestimation issue).*

On implementation side of things, Double DQN turned out to be more stable than classic DQN. The architecture for the neural networks is that comprising of two hidden layers containing 128 and 64 units each. The configuration was selected after experimenting with many architectures. With 256 units in two layers, the performance got a little slow, whereas with 64 units in two layers, the learning period took longer initially. Then the current architecture was tried and it yielded satisfactory results, and hence it was used henceforth. It may be possible that the performance on other configurations might have been affected by other hyper parameters. So keeping that in mind is very important. The activation function employed on each hidden layer is Exponential Linear Unit or ELU which was observed to give minutely better and more stable performance than ReLU. The output layer of the networks was normal output function to obtain action values corresponding to each possible action in the action space. Adam optimiser is used for optimisation of the parameters of the model parameters. As explained in previous sections Replay buffer has been used to remove correlation between different state and action pairs. The local and target networks are initialised with the same weights. This is done using a hard update function written after the soft update function. The act function is decoupled from the update function, for the obvious reasons of allowing updates to be independent of the experience gain, and also the requirement of being able to modify the update variable in the main notebook and not having to modify the agent code for the same. The act function chooses the random function from numpy to see if the random value is more or less than the current value of epsilon, if it is more than the max action value based action is selected, else a random action is sampled.

For the learn function of the agent, the local network is used to find the best possible actions for the next states and these actions are used to find the td target for the local network to optimise its weights for, hence implementing the double q learning principle to reduce over estimation. The mean square error loss function is used for optimisation purposes. Also as a means to increase stability of the network that gradients have been clipped for each learning step. After which the soft update takes place between target and local network with tau factor. Epsilon greedy based learning is used in the solution where the decay rate is fixed at 0.995 while the minimum epsilon value was fixed at 0.01. The learning rate was kept at 0.0005. The batch size was 64. The replay buffer size was 100000.

After every 100 episodes, the average score of the last 100 episodes, was printed till the average score of the last 100 episodes didn't reach the value of 13, which in our case happened in 715 episodes. The plot for the same is as shown below :

```
Episode 100     Average Score: 0.78
Episode 200     Average Score: 3.73
Episode 300     Average Score: 6.39
Episode 400     Average Score: 8.38
Episode 500     Average Score: 11.18
Episode 600     Average Score: 11.67
Episode 700     Average Score: 12.53
Episode 715     Average Score: 13.04
Environment solved in 715 episodes!     Average Score: 13.04
```



7.3 Deep Deterministic Policy Gradient implementation

DDPG algorithm (suitable for environments with continuous action spaces) which involves the use of four neural networks, two each for both actor and critic was used to solve the Continuous Control Environment, which is a part of the group of environments made available by NVIDIA through Unity. Architecture of Neural Networks :

1. Actor : The actor target and local networks had four layers, out of which two were hidden with 400 and 300 units respectively.
2. Critic : The critic target and local networks also had four layers, again out of which two were hidden with 400 and 300 units respectively.

ELU was used instead of ReLU or Leaky ReLU based on performance, for both actor and critic networks. The function used as activation in output layer of actor network was tanh owing to the range of action space being -1 to 1, whereas the activation function used in output layer of critic network was linear acting as value function values meant to act as critic for the agent.

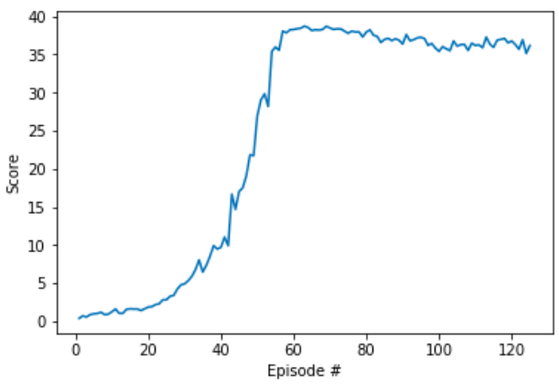
The code for the agent system (generalising for ¿1 agents, distributed learning) comprises of agent class implementation that involves initialisation of the four networks, replay buffer, and OUNoise. The target and local networks for both actor and critic are initialised with same weights. The class also includes act method that gives the array of actions for the different states observed by different agents of the environment by passing the states through the actor network, along with a hint of exploration included using epsilon based chosing (if a random number generated using numpy is greater than than the current value of epsilon that noise is added to the action vector else it isn't. As the value of epsilon decays over time, the less is the probability of exploration, and that is exactly is the basic premise behind exploration and exploitation. This method is used to gain experience on the basis of current policy enacted by the actor network. We also have the step function which stores the current sets of states and actions along with their rewards for all 20 agents one by one in the replay buffer. This increases the amount of exploration done by the agents improving the time taken for the training of the model to happen. This method is decoupled from the update part. To be able to update without needing to explore, another method was defined. Next is the reset function which resets the noise object employed by the agent system per episode. Then we have the learn method which involves, obtaining the next actions on the basis of the next states in the sampled experiences from the replay buffer, from the actor network. Using these actions, Q targets are estimated and are compared with Q expected for the current set of states and actions to calculate the loss function using the mean square error function on which the optimisation of the critic is done. Next the actor network is optimised using the predictions of actions for the current states, and passing them through the critic network. That is followed by soft updates of both target networks. The OUNoise (Added to add exploration in continuous action based environments) class is same as that employed in the ddpg pendlumn with the difference that it is adapted for 20 different agents by passing the num agents argument in the size argument of the class, which allows to sample the required dimensions from the standard normal for noise generation, plus standard normal distribution based noise is added . This is followed by the replay buffer class.

The ipython notebook majorly consists of loading depedencies and initiating the environment and agent. After that we implement the DDPG function with default parameters of 1000 episodes, with maximum of 1000 steps in an episodes, and toggle off (to update in alternate steps if on, implemented to update 10 times in 20 steps). The function also uses the update_num variable which decides how many updates should happen per step given that a lot of experience is being collected per step. One deque of 100 size and one array named scores_deque and scores respectively keep track of mean scores of all agents for the last 100 and all episodes respectively. Once the mean score of the last 100 episodes crosses 30, the function breaks out of the main loop to conclude that the environment is solved and project is completed. After that the plot for score for all episodes is made.

The hyperparameters for the implementation is as follows:

- Buffer size (Replay buffer size) : 100000
- Batch size (Mini batch size): 128
- Gamma (Discount rate) : 0.99
- Tau (Soft update influence rate) : 0.001
- Learning rate for both actor and critic networks : 0.0001
- Starting value of epsilon : 1
- Decay rate of epsilon : 0.0001
- Min value of epsilon : 0.05
- Optimiser used : Adam optimiser
- Loss function : Mean square error function
- 1 learning update per step for both networks

In our case the final process took 125 episodes to solve, and the plot for the same is as follows (also present in the implementation).



7.4 Deep Deterministic Policy Gradient implementation for Multi Agent Environment

The environment was solved using the DDPG algorithm (suitable for environments with continuous action spaces) which involves the use of four neural networks, two each for both actor and critic. DDPG was used here instead of the more reliable MADDPG, as an initial experiment to see the effect of non-stationary environments as detailed in the MADDPG paper. But turns out that the DDPG agent with multiple agents used for experience collection as done in the continous control project performs fairly well. On trying to read about it turns out, that it was not a fluke, where some resources even went on to attribute it to the symmetry of the environment which seemed like a logical assertion.

1. Actor : The actor target and local networks had four layers, out of which two were hidden with 64 and 64 units respectively.
2. Critic : The critic target and local networks also had four layers, again out of which two were hidden with 64 and 64 units respectively.

ELU was used instead of ReLU or Leaky ReLU based on the both personal experience for both actor and critic networks. In this project, due to the good results given by ELU in the previous projects, ELU was the activation function which I started with and it gave good results from the very first go. Hence, for this specific project ReLU and leaky ReLU were not played with. The function used as activation in output layer of actor network was tanh owing to the range of action space being -1 to 1, whereas the activation function used in output layer of critic network was linear acting as value function values meant to act as critic for the agent. The code for the agent system (generalising for ¿1 agents, distributed learning) comprises of agent class implementation that involves initialisation of the four networks, replay buffer, and OUNoise.

The target and local networks for both actor and critic are initialised with same weights. The class also includes act method that gives the array of actions for the different states observed by different agents of the environment by passing the states through the actor network, along with a hint of exploration included using epsilon based chosing (if a random number generated using numpy is greater than than the current value of epsilon that noise is added to the action vector else it isn't. As the value of epsilon decays over time, the less is the probability of exploration, and that is exactly is the basic premise behind exploration and exploitation. This method is used to gain experience on the basis of current policy enacted by the actor network. We also have the step function which stores the current sets of states and actions along with their rewards for all 20 agents one by one in the replay buffer. This increases the amount of exploration done by the agents improving the time taken for the training of the model to happen. To be able to update without needing to explore, another method was defined. Next is the reset function which resets the noise object employed by the agent system per episode. Then we have the learn method which involves, obtaining the next actions on the basis of the next states in the sampled experiences from the replay buffer, from the actor network. Using these actions, Q targets are estimated and are compared with Q expected for the current set of states and actions to calculate the loss function using the mean square error function on which the optimisation of the critic is done. Next the actor network is optimised using the predictions of actions for the current states, and passing them through the critic network. That is followed by soft updates of both target networks.

The OUNoise (Added to add exploration in continuous action based environments) class is same as that employed in the ddpg pendlumn with the difference that it is adapted for 2 different agents by passing the num agents argument in the size argument of the class, which allows to sample the required dimensions from the standard normal for noise generation, plus standard normal distribution based noise is added. This is followed by the replay buffer class.

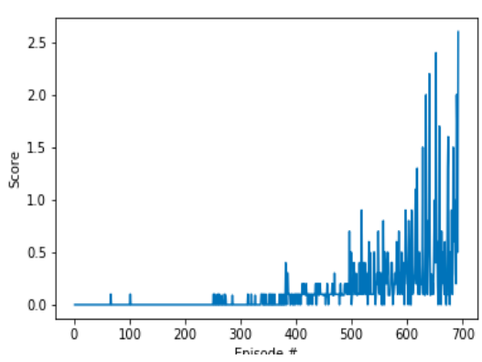
The parameters for the agent system is as follows :

- Buffer size (Replay buffer size) : 100000
- Batch size (Mini batch size): 128
- Gamma (Discount rate) : 0.99
- Tau (Soft update influence rate) : 0.2
- Learning rate for both actor and critic networks : 0.0001
- Starting value of epsilon : 1
- Decay rate of epsilon : 0.0001
- Min value of epsilon : 0.05
- Optimiser used : Adam optimiser
- Loss function : Mean square error function
- 1 learning update per step for both networks

The ipython notebook majorly consists of loading dependencies and initiating the environment and agent. After that we implement the DDPG function with default parameters of 2000 episodes, with maximum of 1000 steps in an episodes, and toggle off (to update in alternate steps if on, implemented to update 10 times in 20 steps). The function also uses the update_num variable which decides how many updates should happen per step given that a lot of experience is being collected per step. One deque of 100 size and one array named scores_deque and scores respectively keep track of mean scores of all agents for the last 100 and all episodes respectively. Once the mean score of the last 100 episodes crosses +0.5, the function breaks out of the main loop to conclude that the environment is solved and project is completed. After that the plot for score for all episodes is made.

In our case the final process took 692 episodes to solve, and the plot for the same is as follows (also present in the implementation).

```
Episode 692     Average Score: 0.51
Environment solved in 692 episodes!     Average Score: 0.51
```



7.5 Quantisation

16bit and 8bit level quantisation and quantaware quantisation was was implemented on pytorch from scratch and tested on CIFAR100, to check accuracy levels on standard datasets before moving to RL related models, and will further be tested in further experiments conducted during the project.

7.6 Efficient Models Experiments

Models like Efficientnet, Mobilenet and Wideresnet were implemented and run on standard datasets like CIFAR100 to achieve start of the art marked accuracy. These will also be further tested on future experiments which are part of the project.

8 References

1. Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. arXiv preprint arXiv:1707.01495, 2017.

2. Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008, 2008.

3. Rich Caruana. Multitask learning. In Learning to learn, pp. 95–133. Springer, 1998.

4. Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In International Conference on Machine Learning, pp. 1329–1338, 2016.

5. Jakob Foerster, Richard Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. arXiv preprint arXiv:1709.04326, 2017a.

6. Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. arXiv preprint arXiv:1705.08926, 2017b.

7. Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural infor- mation processing systems, pp. 2672–2680, 2014.

8. He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daume III. Opponent modeling in deep reinforcement learning. In International Conference on Machine Learning, pp. 1804–1813, 2016.

9. Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. arXiv preprint arXiv:1707.02286, 2017.

10. Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect- information games. arXiv preprint arXiv:1603.01121, 2016.

11. Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

12. Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

13. Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In Proceedings of the eleventh international conference on machine learning, volume 157, pp. 157– 163, 1994.

14. Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm. In Advances In Neural Information Processing Systems, pp. 2378–2386, 2016.

15. Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. arXiv preprint arXiv:1706.02275, 2017.

16. Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems. The Knowledge Engineering Review, 27(1):1–31, 2012.

17. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Belle- mare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

18. Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim

Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning, pp. 1928–1937, 2016.

19. OpenAI. OpenAI Dota 2 1v1 bot, 2017. URL https://openai.com/ the-international/. Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. Autonomous agents and multi-agent systems, 11(3):387–434, 2005.

20. Lerrel Pinto, James Davidson, and Abhinav Gupta. Supervision via competition: Robot adversaries for learning tasks. In Robotics and Automation (ICRA), 2017 IEEE International Conference on, pp. 1601–1608. IEEE, 2017.

21. John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Proceedings of the 32nd International Conference on Machine Learning (ICML-15), pp. 1889–1897, 2015a.

22. John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High- dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438, 2015b.

23. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

24. David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. Nature, 529(7587):484–489, 2016.

25. Karl Sims. Evolving virtual creatures. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques, pp. 15–22. ACM, 1994.

26. Kenneth O Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary com- plexification. Journal of Artificial Intelligence Research, 21:63–100, 2004.

27. Sainbayar Sukhbaatar, Ilya Kostrikov, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. arXiv preprint arXiv:1703.05407, 2017.

28. Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. PloS one, 12(4):e0172395, 2017.

29. Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Proceedings of the tenth international conference on machine learning, pp. 330–337, 1993.

30. Gerald Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3): 58–68, 1995.

31. Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pp. 5026– 5033. IEEE, 2012.

32. Kevin Wampler, Erik Andersen, Evan Herbst, Yongjoon Lee, and Zoran Popovic . Character animation in two-player adversarial games. ACM Transactions on Graphics (TOG), 29(3):26, 2010.

33. Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3-4):229–256, 1992.