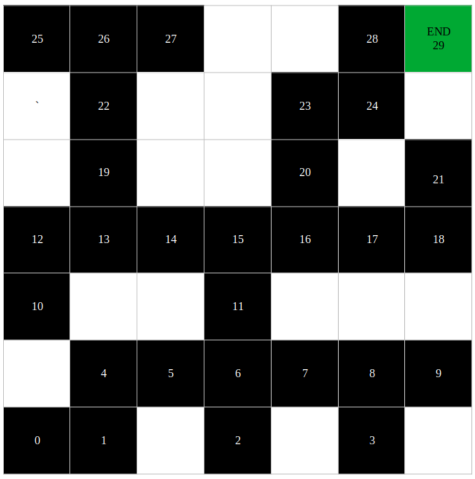
1. BASIC TASK

1.1. Domain and Task



The Internet is the greatest invention of the 20th Century because it changed the course of humanity. It literally has impacted us all in very beneficial ways. In business, it helps to connect sellers and buyers from all over the world in just a few clicks. Many companies were able to capture the opportunity to expand their business and in some rare case becoming a giant corporations like Amazon, Alibaba etc. In case of Amazon, today the company on average has to process around 1.6 million orders per day. Hence, logistic has become a big concern to such company since this number are more likely to continue increasing in the future. Solely depending on human labour forces is not a optimal option moving into the future and automation seems to be the most optimal choice. Using this as a context, the first part of this coursework we will use reinforcement learning to build a robot for handling the pick up packaging task in the warehouse. This will help the company to cut costs and increase efficiency because a robot can work continuously 24 hours without being tired.

The warehouse environment is represented by 2-dimensional 7x7 grid world with a single agent or robot with the task of gathering individual items from various locations in the warehouse in order to fulfil customer orders. After picking the items from the shelves, the robot must bring the items to a specific location within the warehouse where the items can be packaged for shipping. In order to ensure the maximum efficiency and productivity, the robot will need to learn the shortest path to travel to the end destination. This is similar problem with automated car parking system where the car needs to learn where to park We will use Q-learning to accomplish this task. Below is the diagram of the warehouse.

1.2. State Transition Function and Reward Function

State Transition Function

The state transition function says how the state machine changes state in response to an action. If the machine is in state s, and it receives an action a, then the new state of the machine will be s' = δ(s,a). The grid world consists of 30 states which are coloured by black. The path makes up by all of these black cells is the route that the machine can travel along. The green cell is the item packaging area that the robot needs to reach. The green cell is the terminal state. At any white state, a robot can either move up, down, right, and left. Let’s say the robot randomly starts at state 0 (row 7, col 1). The possible transition function can be broken down into below table:

Current State S	Action	Next State S’
S$_{71}$ (row 7, col 1)	Right	S’$_{72}$ (row 7, col 2)
S’$_{72}$ (row 7, col 2)	Left	S’$_{71}$ (row 7, col 1)
S’$_{72}$ (row 7, col 2)	Up	S’$_{62}$ (row 6, col 2)
S’$_{62}$ (row 6, col 2)	Down	S’$_{72}$ (row 7, col 2)
S’$_{62}$ (row 6, col 2)	Right	S’$_{63}$ (row 6, col 3)
S’$_{63}$ (row 6, col 3)	Left	S’$_{62}$ (row 6, col 2)
S’$_{63}$ (row 6, col 3)	Right	S’$_{64}$ (row 6, col 4)
S’$_{64}$ (row 6, col 4)	Left	S’$_{63}$ (row 6, col 3)
S’$_{64}$ (row 6, col 4)	Right	S’$_{65}$(row 6, col 5)
S’$_{64}$ (row 6, col 4)	Down	S’$_{74}$ (row 7, col 4)
S’$_{64}$ (row 6, col 4)	Up	S’$_{54}$ (row 5, col 4)

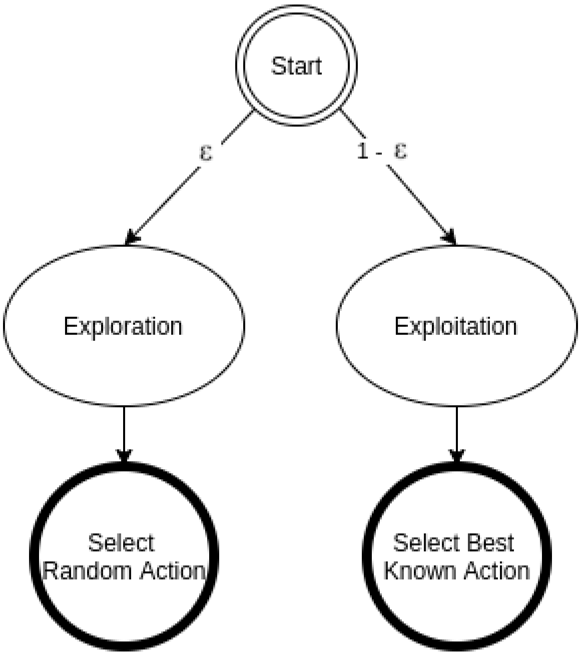
Reward Function

The Reward Function is an incentive mechanism that tells the robot what is correct and what is wrong using reward and punishment. When a robot moves from its current state to a new state by taking an action a, it will receive a value from the reward function: r’ = r(s,a) where r is the reward received for taking action a in state s. The robot may begin at any black cell, but its goal is always to maximize its total rewards. For each black cell the reward value is -1. The green terminal state has the value of 100. If the robot reaches the end state, the run will stop. The reason why we give -1 to each black cell is because we want to avoid the situation when the robot just runs around and accumulate points over and over again. It also gives the robot the incentive to find the shortest path to the End state to maximise the reward.

1.3. **Learning Policy**

Optimal Policy

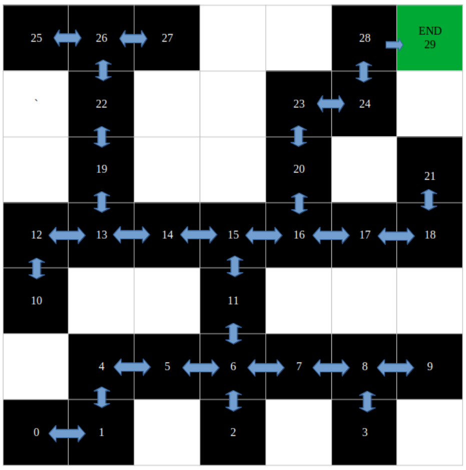
Q-learning is a model-free algorithm. In Q-learning, we select an action based on its reward. The agent always choose the optimal action. Hence, it generates the maximum reward possible for the given state. For this basic task of the coursework, we use epsilon-greedy policy. In epsilon-greedy policy, the agent use both exploitation to take advantage of prior knowledge and exploration to look for new knowledge (see figure 1). The epsilon-greedy approach selects the action with the highest estimated reward most of the time. The goal is to have a balance between exploration and exploitation. Exploration allows us to have some room for trying new things, sometimes contradicting what we have already learned. With a small probability of ε, we choose to explore, not to exploit what we have learned so far. It means that the action is selected randomly, independent of the action-value estimates. If we run a large enough number of trials where each action is taken large number of times then the epsilon-greedy action selection policy will discover the optimal actions policy for sure.

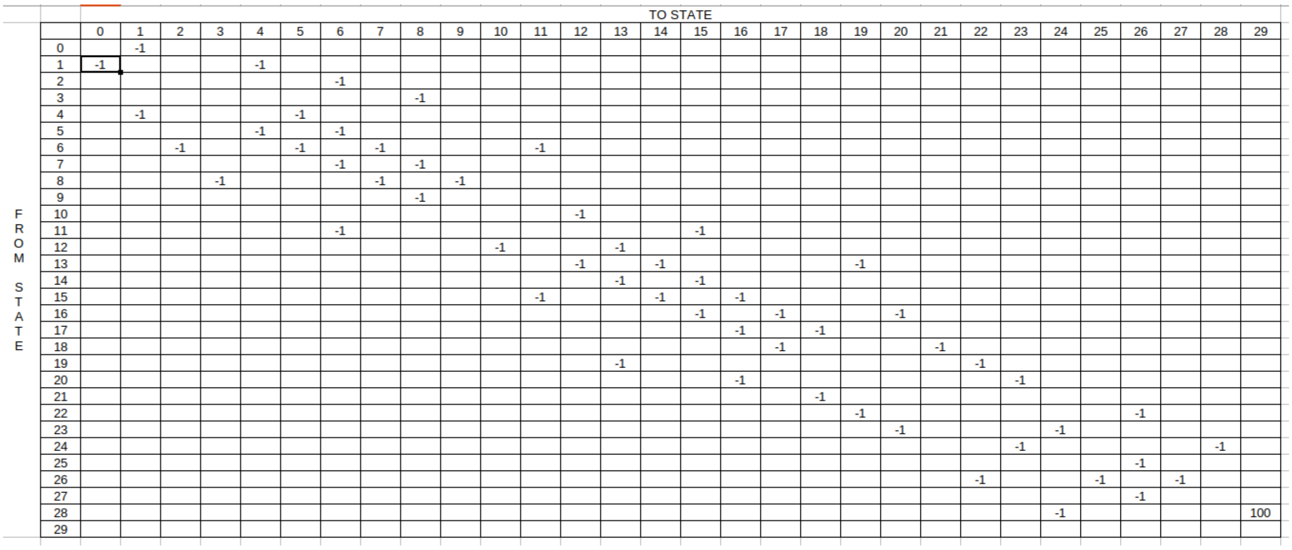


1.4. **Graphical Presentation of The Problem and R Matrix**

There are 30 states in the warehouse grid. The robot can move between states given there are possible actions between states. This can be represented as a graph using 2-way arrow in Figure 3 below, with rewards, punishments, and end states, and allowed travel states. This two way direction arrow indicates which direction the robot can travel.

The R-matrix represents the actions which are possible from any given state, along with the numeric reward for every possible action in each state. The tabulate table of R matrix is constructed using excel file. See the below picture. There are total of 57 cells with values.





TO STATE

FROM STATE

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0		-1																												
1	-1				-1																									
2							-1																							
3									-1																					
4		-1				-1																								
5					-1		-1																							
6			-1			-1		-1				-1																		
7							-1		-1																					
8				-1				-1		-1																				
9									-1																					
10													-1																	
11							-1									-1														
12											-1			-1																
13													-1		-1					-1										
14														-1		-1														
15												-1			-1		-1													
16																-1		-1			-1									
17																	-1		-1											
18																		-1				-1								
19														-1									-1							
20																	-1							-1						
21																			-1											
22																				-1							-1			
23																					-1				-1					
24																								-1					-1	
25																											-1			
26																							-1			-1		-1		
27																											-1			
28																									-1					100
29																														

Figure 4: The Original R-matrix

1.5. The Parameter Values For Q-Learning

In Q-learning, there are 3 key parameters: learning rate (α), discount factor (γ), and epsilon-greedy factor (ε).

Learning rate: can simply be defined as how much you accept the new value vs the old value. the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly.

Discount factor: It's used to balance immediate and future reward. This value can range anywhere from 0 to 1. A low discount rate means that we want the agent to focus on maximising the immediate return by taking a suitable action. On the other hand, a high discount factor means that we want the agent to focus on the long term rewards. At 0 and 1, the effect of discount factor is highest.

Epsilon-Greedy factor: is a simple factor to balance exploration and exploitation by choosing between exploration and exploitation randomly.

Parameter	Value
learning rate (α)	0.9
discount factor (γ)	0.9
epsilon-greedy factor (ε)	0.9

Figure 5: Default Q-learning Parameters

1.6. Updating Q Matrix

If R-matrix is the right hand of the Q-learning algorithm, then Q-matrix is the left hand. When Q-learning algorithm is implemented, a Q-table or Q-matrix is created that mimics the shape of R-matrix or the shape of [state, action]. The initial values of all cells in the matrix are zero. And its Q-values then is updated and stored after each episode. The agent will use this Q-values table as a reference to choose the best action based on the values on the Q-table. In order to update Q-values, Q-learning uses Bellman equation. The formula for the equation is described as below.

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot maxQ(s_{t+1}, a_t) - Q(s_t, a_t))$$

The below table shows how this Q^{new} is updated. Lets assume the starting state of the robot is 0. The optimal route for it is 0-1-4-5-6-11-15-16-20-23-24-28-END.

Time Step	$Q(s_t,a_t)$	Q Update
t_1	Q(0,1)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_2	Q(1,4)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_3	Q(4,5)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_4	Q(5,6)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9

t_5	Q(6,11)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_6	Q(11,15)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_7	Q(15,16)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_8	Q(16,20)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_9	Q(20,23)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_{10}	Q(23,24)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_{11}	Q(24,28)	0.0 + 0.9*(-1.0+ 0.9*0.0 – 0.0) = -0.9
t_{12}	Q(28,29)	0.0 + 0.9*(100+ 0.9*0.0 – 0.0) = 90

1.7. Represent Performance

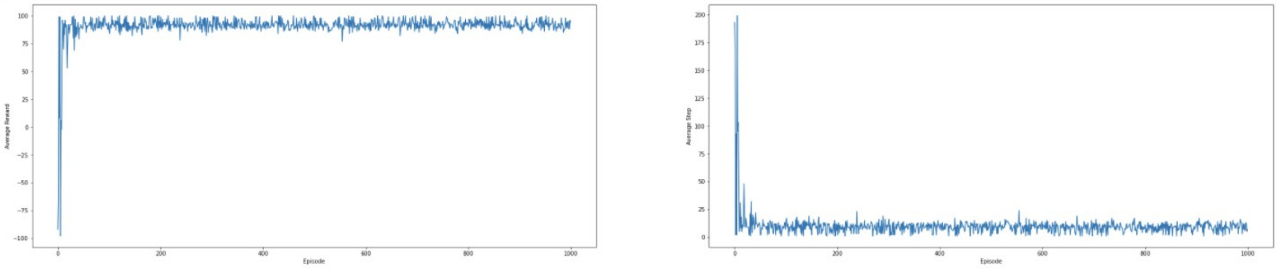


Figure 6: Alpha=0.9, Gamma=0.9, Epsilon=0.9

The two key metrics that we use to measure the performance of our robot are: average reward vs. episode, and average steps vs. episode. This is the performance of the robot with the default value of alpha, gamma and epsilon. We will analyse the results in the later part and see what it means.

1.8. Run The Experience With Different Parameters Values and Policies

Different Learning Rate (Alpha = 0.1, 0.3, 0.6, 0.9)

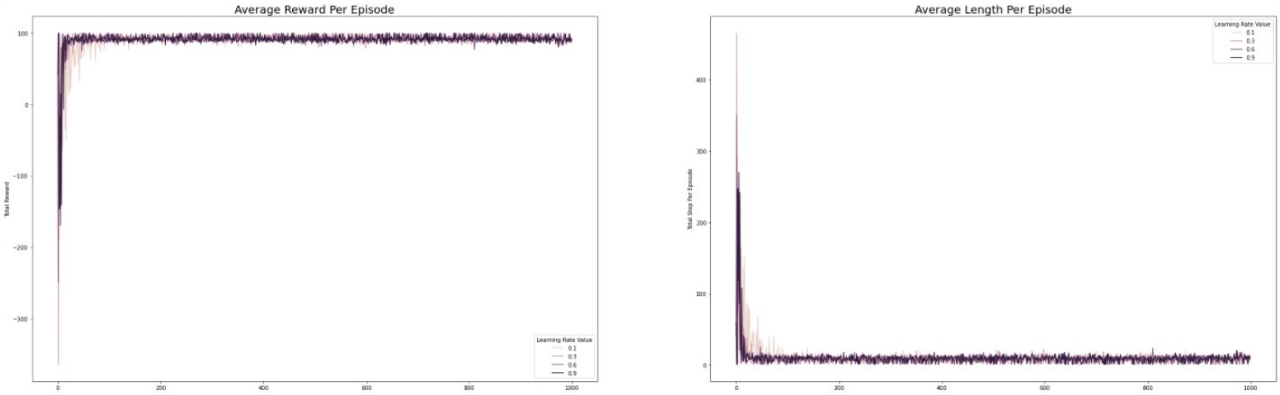


Figure 7: Alpha=(0.9, 0.6, 0.3, 0.1), Gamma=0.9, Epsilon=0.9

Different Discount Factor (Alpha = 0.1, 0.3, 0.6, 0.9)

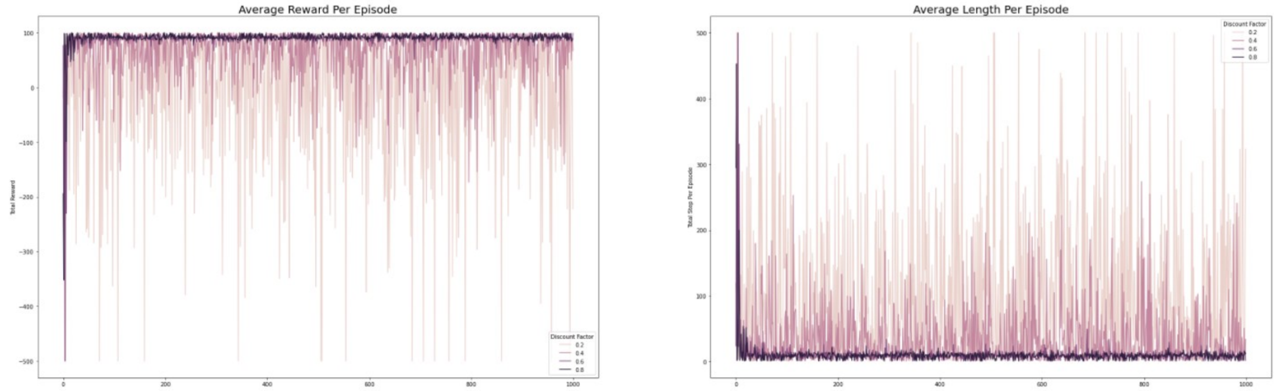
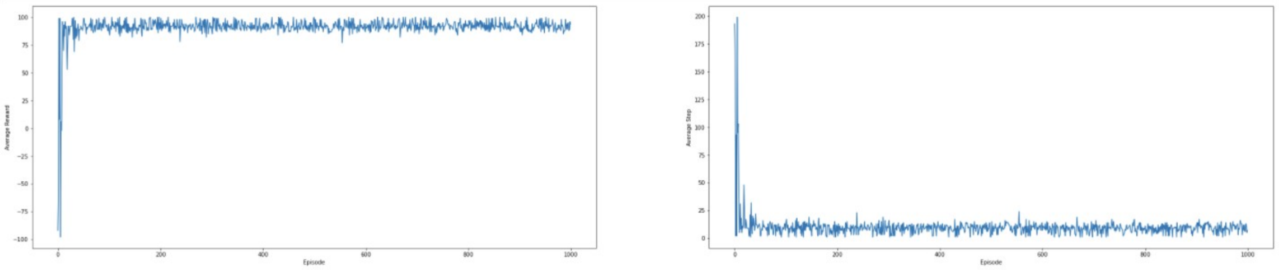
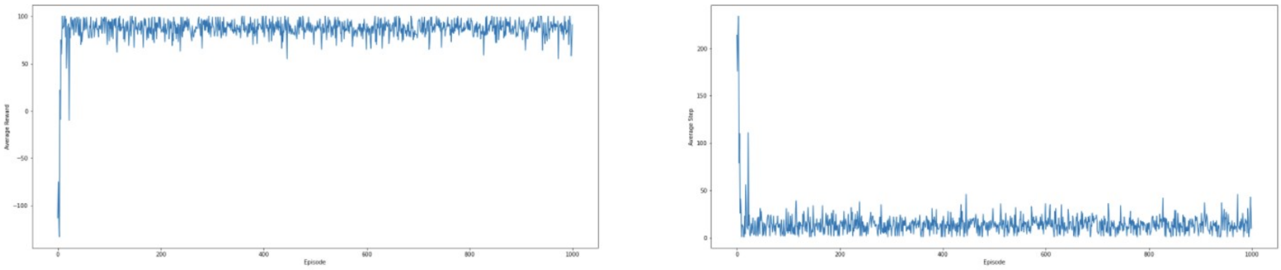


Figure 8: Alpha=0.9, Gamma=(0.9, 0.6, 0.3, 0.1), Epsilon=0.9

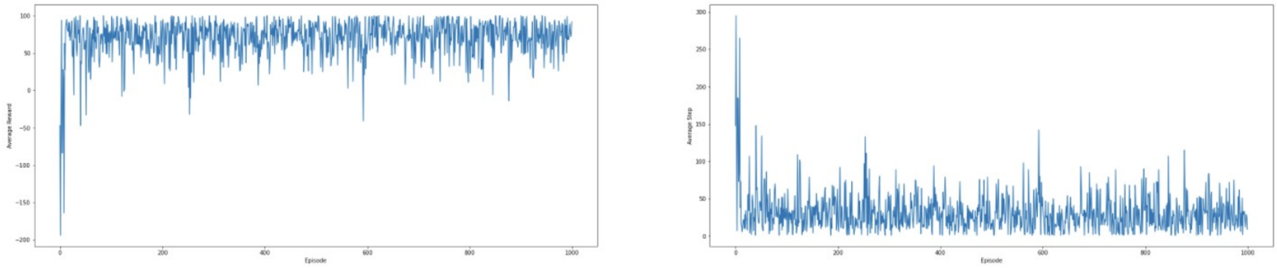
Different Epsilon (Epsilon = 0.9)



Different Epsilon (Epsilon = 0.6)



Different Epsilon (Epsilon = 0.3)



Different Epsilon (Epsilon = 0.1)

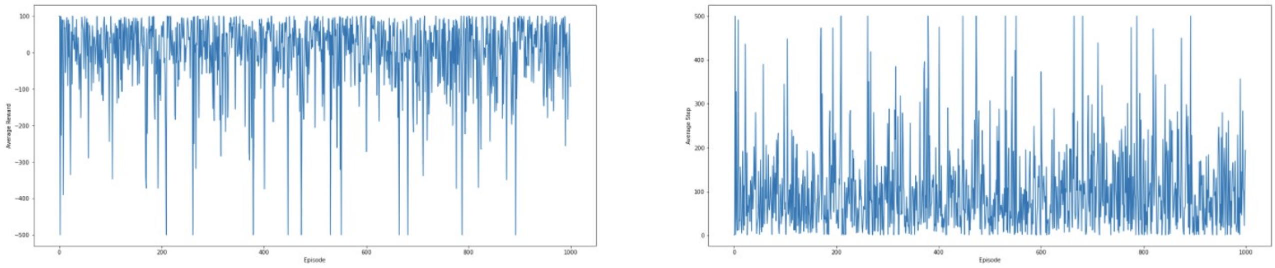


Figure 8: Alpha=0.9, Gamma=0.9, Epsilon=(0.9, 0.6, 0.3, 0.1)

1.9. **Analyse The Results Quantitatively and Qualitatively**

As we have mentioned about the two key metrics to measure the performance of our robot which are average reward vs episode, and average step vs. episode. If Q-learning algorithm is really effective, we expect this two metrics will convergence (upward convergence for average reward and downward convergence for average step) at some point after a period of learning. For average reward. It is very easy why we expect these. If the robot effectively learns more about the environment, it will quickly learn the fastest way to the end destination and earn more points after each episode, and taking less steps to reach the final destination.

When we run the first trial with default values of alpha=0.9, gamma=0.9, epsilon=0.9, we can see that the performance (Figure 6) is really good when running with 1000 episode. Both graphs have converged quickly within two figure episodes. It means that the robot can quickly learn its way to the destination. Next we will try to test our robot with different parameters values.

By looking at Figure 7, when we train our robot with different learning rate of (0.9, 0.6, 0.3, 0.1), we can see that the performance of our robot is still pretty good since it still learns its way to the end destination very quick. Of course, there are differences between different learning rate value. With the learning rate of 0.1, it took the robot longer to converge. The velocity of convergence is increasing with the value of learning rate. This shows how important the learning rate in reinforcement learning. One of the reason why our robot still can performance well with the learning rate of 0.1 is because our warehouse environment is deterministic and small. In a complex and sophisticated environment, this would not be the case, especially continuous ones.

In figure 8, we look at the robot performance with different gamma values of 0.9, 0.6, 0.3, 0.1). There is a significant different in the performance of our robot with different gamma value. The two metrics almost does not converge with gamma=03 and gamma=0.1. With gamma=0.6, the performance is better but far away from the performance of gamma=0.9. It shows that the value of gamma has significant impact on the performance of the robot. We don't know between gamma and learning rate which one has more influence on in the reinforcement learning but in our small environment, gamma seems to be significantly more influential.

Next, we train our robot with different epsilon values of *(0.9, 0.6, 0.3, 0.1).* Out of 4 different values of epsilon, the two metrics converged as expected at epsilon=0.9, 0.6, 0.3. The only value of epsilon which does not converge is 0.1. Of course, the speed of convergence is different with 3 values. The higher the value of epsilon, the faster the convergence. In this environment, epsilon even has more impact to our robot than the value of gamma. It totally makes sense since epsilon value or policy instruct our robot to take an exploration action or exploitation action. With epsilon=0.1, it instructs the robot to do very little exploration and high exploitation thus leads to non-convergence.

Some even suggests to vary the reward policy but in my opinion gaining from this experiment. It does not matter at all if you set different reward function in the end.

The only drawback or suggestion for future work is that we can train our robot in a more complicated environment. And see how much impact the parameters has on its performance.

2. ADVANCED TASK

2.1. DQN & Double DQN & DQN PER

The chosen environment for this task is LunarLander-v2. The goal is navigate a lander to its landing pad. The initial coordinate is always at (0,0). The agent earns reward from 100 to 140 points. The agent will lose mark if the lander move away from the landing pad. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. The game solved is 200 points. Landing outside of the pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. There are 4 discrete actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. The code for this task is inspired from an pre existing code on GitHub (https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/02_dqn_pong.py) and also from class tutorial 6**.**

In the task one of this coursework, we applied Q-learning to solve a warehousing problem by using state and action pair to create R matrix and Q values matrix and update q value after a taking a certain action and it proved to work really well. However, a major drawback of Q-learning is that it can only work in the environments with discrete and finite state and action spaces. It is not feasible in a more complex and sophisticated environments with continuous state and infinite action spaces which requires high computational power. To overcome this problem, Deep Q-learning and its improvements such as Double DQN, PER, Dueling DQN etc. was created. Our chosen environment is much more complex and sophisticated then the warehouse problem. Therefore, it needs a better algorithms to perform the task. We chose to implement DQN, Double DQN, and DQN PER.

DQN

Rather than using value iteration to directly compute Q values and find the optimal Q function, we instead use a function approximator to estimate the optimal Q function. Neural network is good at this. Neural network makes use of a deep neural network to estimate the Q values for each state action pair in a given environment. And in turn the network will approximate the optimal Q function. The approach of combining Q learning and deep neural network is called Deep Q-learning and a deep neural network that approximates the Q function is call a deep Q network or DQN.

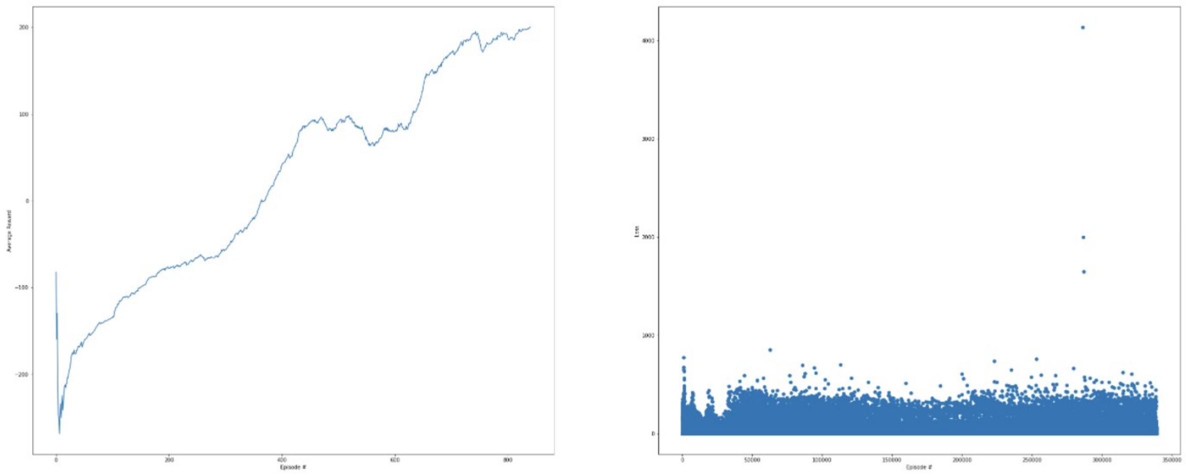


Figure 9: DQN

Double DQN

The purpose of double DQN is to solve the problem of over-estimating the true return because of the max operator. The idea is to train two separate value networks independently. One network is used to find the greedy action with maximal Q-value an the other is used to estimate the Q-value itself. When the first network choose an over-estimated action, the second network may provide a less overestimated value for it.

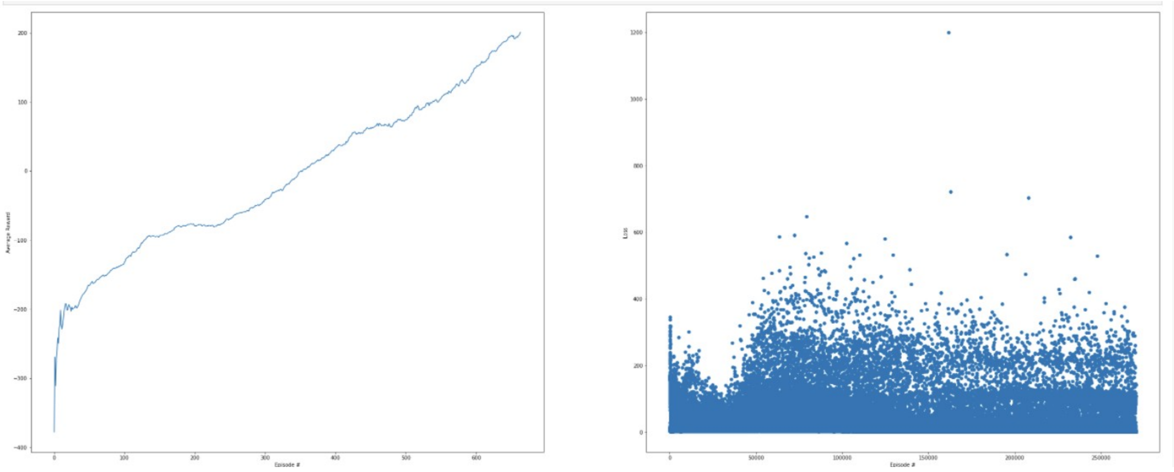


Figure 10: Double DQN

DQN PER

In DQN and Double DQN, the experience replay lets agents remember and reuse experiences from the history or the past. The drawback point of this method is that the experiences transitions were uniformly sample from a replay memory at the same frequency that they were experienced regardless of their significance. Prioritized Experience Replay allows the agent to replay the important transitions more frequently and therefore it can learn more effectively and quicker.

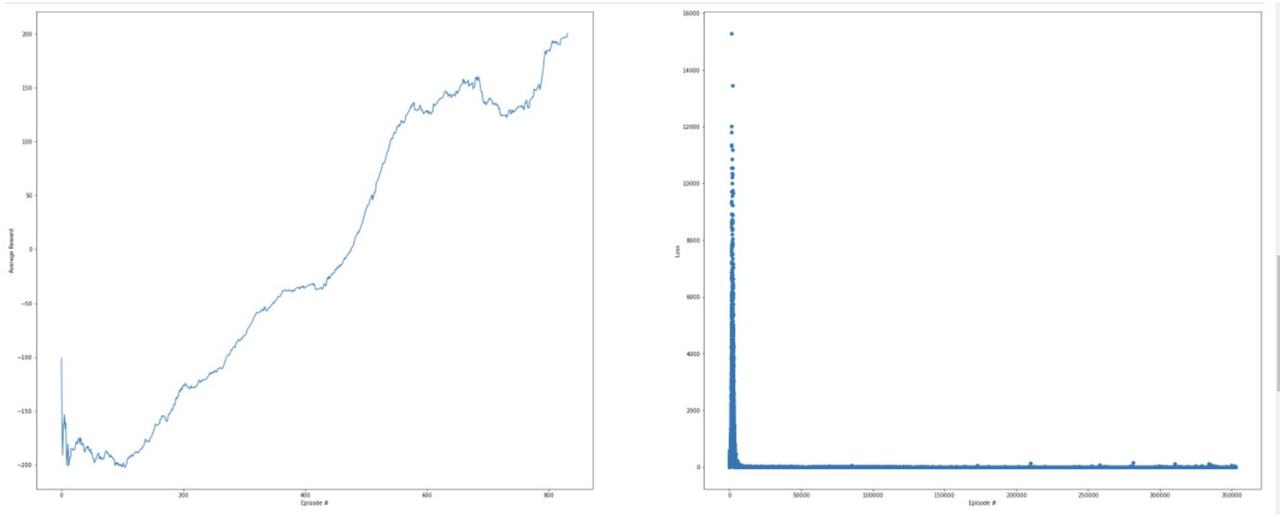


Figure 11: DQN+PER

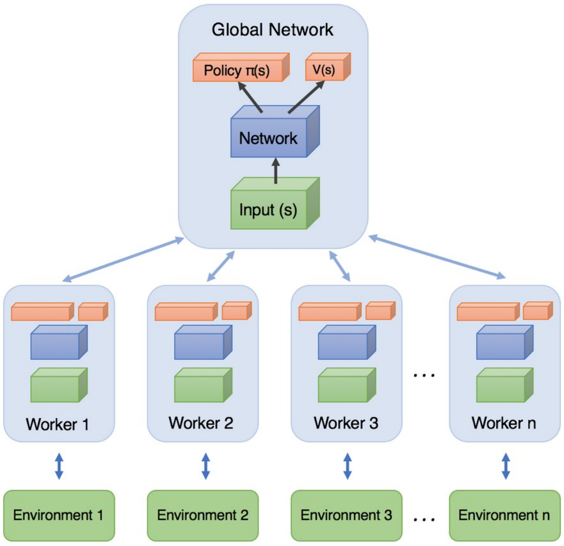
We use the same parameter configuration for all three models such as: gamma, batch size, learning rate. Max epsilon, min epsilon, epsilon decay. As we mentioned above, the game will completed when the lander reaches 200 points. There are four key metrics which we will use to measure the performance of each model against each other: average reward per episode, loss per episode, game completion, total steps or frame to completion. From the graphs, we can see that the lander managed to complete the game at 200 points reward in all different model DQN, Double DQN, DQN+PER. Since Double DQN and DQN+PER are the two improvement of DQN. We expect their results on the other three metrics are going to be better than the results of DQN. However, based on our researches, there are no comparison between Double DQN and DQN+PER. Checking through the graphs, our expectation is right. The performance of Double DQN and DQN+PER is better and more robust then DQN. For DQN, it took more than 900 episode for the game to be solved. While it only took more than 600 episode for Double DQN, and more than 800 episode for DQN+PER. The performance curve of average reward of Double DQN is much better than DQN and even DQN+PER. And the performance curve of average reward of DQN+PER is better than DQN. In this case Double DQN performed better than DQN+PER in tern of average reward per episode.

In term of total steps or frames to completion, the similar results is also found in this metric. When Double DQN took around 270k steps or frames to solve the game, while DQN+PER took more than 350k steps or frames and DQN took more than 330k steps or frames to solve the game. This performance is also easy to explain when Double DQN took much less episode to complete the game. When we look at the graphs for the last metric which is the loss per episode, we can see that the similar results is not presented. DQN+PER performed better then Double DQN and Double DQN performed better than DQN. It can be explained that DQN+PER used prioritize experience replay so that it can minimized the loss better than Double DQN.

In conclusion, the two improvements of DQN are both performed much better than DQN. Double DQN is better than DQN+PER in term of average reward per episode and total steps or frames, while DQN+PER is better than Double DQN in term of loss. We run three models many times. And the results of each time is not much of a difference.

2.2. **Atari Learning Environment – Task 9, 10 & 11**

The scope of this task is to apply one of the reinforcement learning algorithms from RLLIB to one of the Atari environment. And finally, we need to analyse the results quantitatively and qualitatively. The chosen reinforcement learning algorithm is Asynchronous Advantage Actor Critic (A3C) and the Atari environment is the Breakout-v0. It includes a bat, a moving ball and a wall of blocks. The reward policy is that when a ball hits a block, the player will get 1 point and the block is removed. The player has to control the bat at the bottom of the screen to avoid the ball going out of play, which lead to losing one of 5 lives. The game ends when the player loses all 5 lives. This is the game of maximising the score so there is no threshold for the game to finish. However, in this task we will train our agent to achieve the average mean reward of 50 points. We will use stopping criteria in the configuration section of the code to set this value. This will be discussed more later.

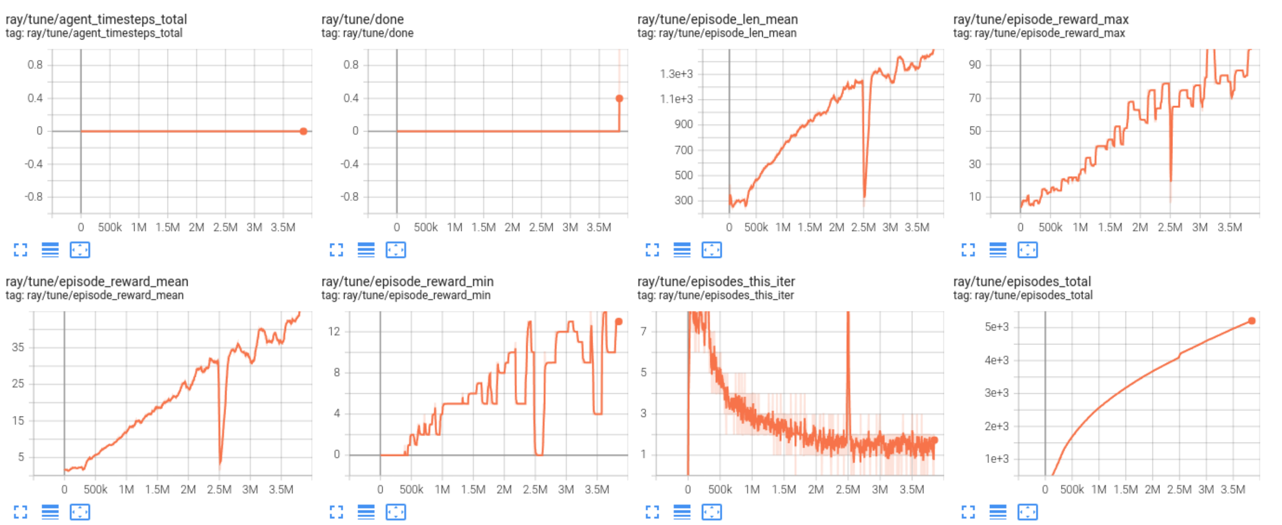


A brief introduction of the chosen A3C algorithms. Together will Proximal Policy Optimization, A3C is one of the newest algorithms bring created in the field of reinforcement learning. This is developed by Google's Deep-mind. The different between this new algorithms and the popular but old ones lies in the word "Asynchronous". The algorithm will employ multiple independent agents with each one of them having its own set of the network parameters and a different copy of the environment. These agents will interact with their version of environments asynchronously and gain knowledge about the environment in each interaction. This is a pretty smart algorithm since it can explore a bigger part of the state and action space in much less time but this comes with the cost of expensive hardware. There is a global network which controls each of these agents. Those knowledge gains by each agent will be contributed to the total knowledge of the global network. Agents are trained in parallel and update periodically a global network but it does not happen simultaneously and explain where the asynchronous comes from. After each update, the agents reset their parameters to those of the global network and continue their independent exploration and training until it is time for them to updating again. The key take-way point here is that there is multiply interactions within the algorithm between network vs. agent, and agent vs agent. The flow of information does not only from network to agents but also from agent to agent because when each agent resets and update its parameter from the network, which has the information of all other agents as well. A3C algorithm combines the best part of simpler techniques which use either policy gradient method or value iteration method. It predicts both the policy function π(s) and value function V(s). The agent uses the value of the V(s) to update the optimal π(s).

Selecting the right algorithms to an environment is important. There are different attribute to each of the algorithms such as framework, discrete action ,continuous action, multi-agent and model support that suits certain environment. Beside this, there are 2 more factors which we need to consider before choosing the algorithm are sample efficiency and stability. Sample efficiency means how much data needed for learning. This factor can be measured by how many environment steps the agent took to reach a fixed level of performance. Taking DQN vs. PPO in CartPole environment for example, DQN algorithm reaches the target score of 500 in much lesser steps when comparing to PPO. Sample efficiency matters when simulation is not available and we have to use real systems during training like a physical robot because stepping through the environment is slow and may take many months to accumulate a reasonable amount of steps. We can increase the speed by parallelize but it is expensive to purchase another robot. If we have an environment where environment step is slow or limited, we need to choose sample efficient algorithm, such as DQN first that extracts the most learning out of each step. Stability means how wildly the performance swing over time. Come back to our example, even DQN is much more sample efficiency than PPO, it is also much less stability than PPO. When PPO algorithm reaches max score of 500, its performance is more or less stable if we continue the training for many more steps. In the contrast, DQN performance swings up and down with a high volatility.

There are environments which sample efficiency does not matter, for example all gym environment which are model based computer simulations and our choosing environment is Breakout-v0 falls that category. So PPO seems to be the algorithm to use. However, the context of our task is different when we want to train a model to reach a particular target of 50 episode reward mean and than stop the training. It means that our agent need to achieve that level as soon as possible. Therefore, we are not going to use PPO and instead of using A3C algorithm which is some kind of a mixture performance between PPO and DQN. Just to confirm our choice is right, we also carry out a training model using PPO for comparing. The only reason, we set the target of 50 points is because the lack of computational power and energy cost since it takes too long to train a model, especially with PPO. The Breakout-V0 environment is a discrete actions space which A3C supports. A3C also support multi-agent. That makes our choice. The results of our training are visualized using Tune Analysis. We will try to run the training model with different number of worker, the more workers the faster we train, to comparing the results just like in Ray website when they use 32, 64 ND 128 workers. However, due to the lack of computer capability we train the model using 2, 16 and 32 workers. The results of those training are in the graphs.

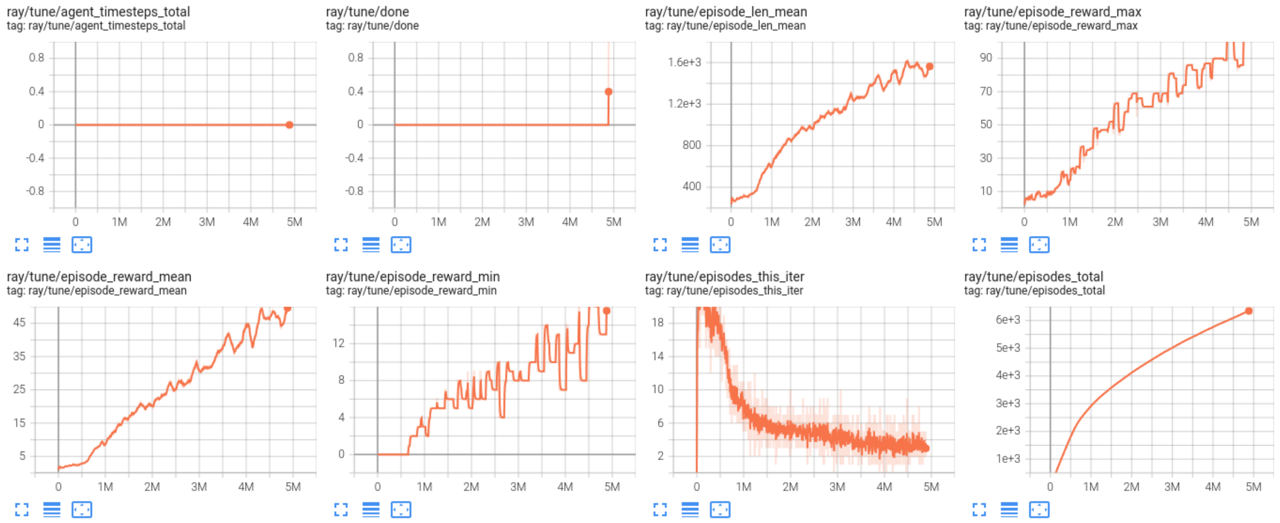
For A3C with 2 workers



We will analyse model performance using 5 key metrics: episode reward mean (first graph, second row), episode length mean (third graph, first row), average reward max (fourth graph, first row), average reward min (second graph, second row), and episode this iteration (third graph, second row). Inspecting through the graphs, we can say that out model is trained pretty well. The training is completed at around 4 millions steps.

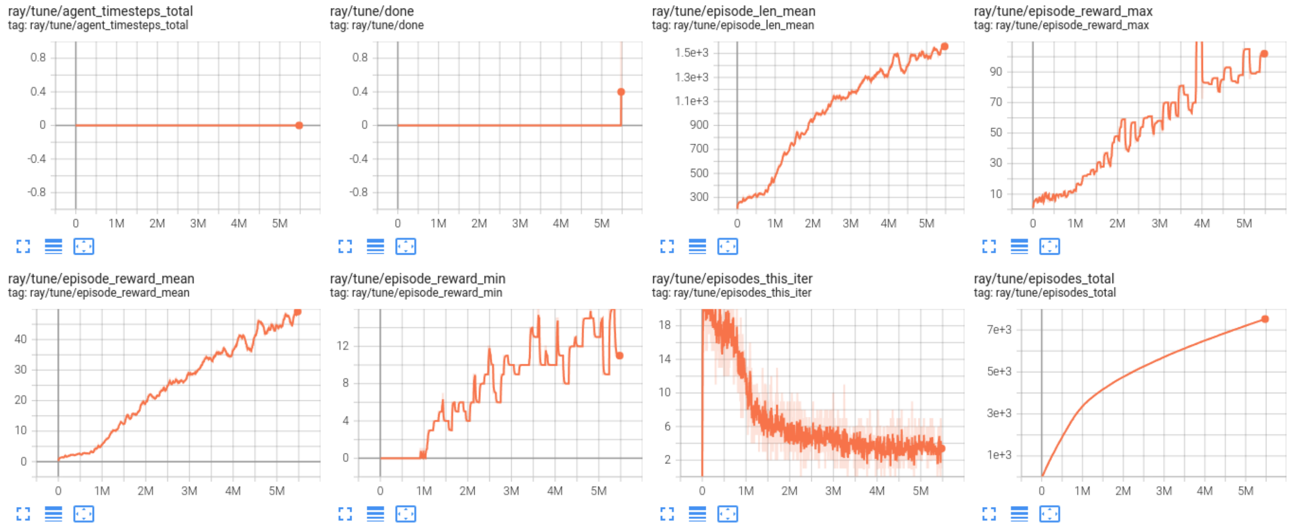
The episode reward mean is the most importance metric among all and it is also our target. It measures how many point the agents earn in each episode. We expected this number to increase the more time the model being train. We also expected the same learning performance of the rest of the metrics except the episode this iteration. We want this number to decrease over time since the more time the agent being trained the less episode it needs over time. The episode reward mean is steadily increase from the beginning until the complete. The similar performance pattern is also presented on the graph of episode length mean, average reward max, average reward mean. And the result of episode this iteration is as what we expected. However, we can notice that at the 2.5 million steps. There are a significant drop in the performance of the first 4 metrics and a huge jumps in the performance of the last metric. This behaviour reminds us about the US stock market in March 2020. I could not think of any answer for this kind of behaviour but reinforcement leaning is know to be prone to performance collapse. Solving this issue is the main motivation behind algorithms like PPO or TRPO. The performance curve of all the metrics seem to me smoother during the first half of the training. It becomes a little more volatile during the second half. This does not seem to be any problem and it can be explained by laws of nature, the higher you drop the more pain it is. For example when comparing the average reward mean drops from 100 to 80 – a 20% drop to when it drops from 1 to 0.8 – also a 20% drop but when you plot them on the chart the drop will look different in two case. If we use DQN here, the volatile will be much higher and much less for PPO. Another small point we can notice is that the max reward actually hit above 100 points which is quite a significant threshold of performance for this game. The total training time is 13387 seconds.

For A3C with 16 workers



Inspecting through all graphs of the key metrics, the performance of our model is what we expected. All first 4 metrics are increasing steadily as the agents gain more knowledge about the environment. If we look close enough, we can see that the steepness of increase of the episode average mean is higher than with 2 workers. Another noticeable point is that the large drop in the performance of all the metrics in the first trial does not happen here. We can see that the learning curve of all the metrics is smoother and less volatile then the first trial. There is quite a different in average mean min curve comparing to the first one. And the curve of the last metric is convergence flatter than the first one. The total training hour is 4490 seconds, much lower than the first. The total time step is 5 million steps, a 30% increasing of first trail but this is for 16 agents when comparing to only 2 agents which means that on average each agent on the second trial took much less steps than the first trial. We can come up with an feasible explanation for the big drop in performance of trial 1. Since we use more workers, the total knowledge is much more than with 2. These knowledge will average down by 16 workers so it reduced the variance of the model. If one agent performed bad, it will still be fine because it will be average by 16. For the 2 worker trial, if one agent performed bad, it will have bigger impact on the model since we only have 2. This is the beauty of A3C.

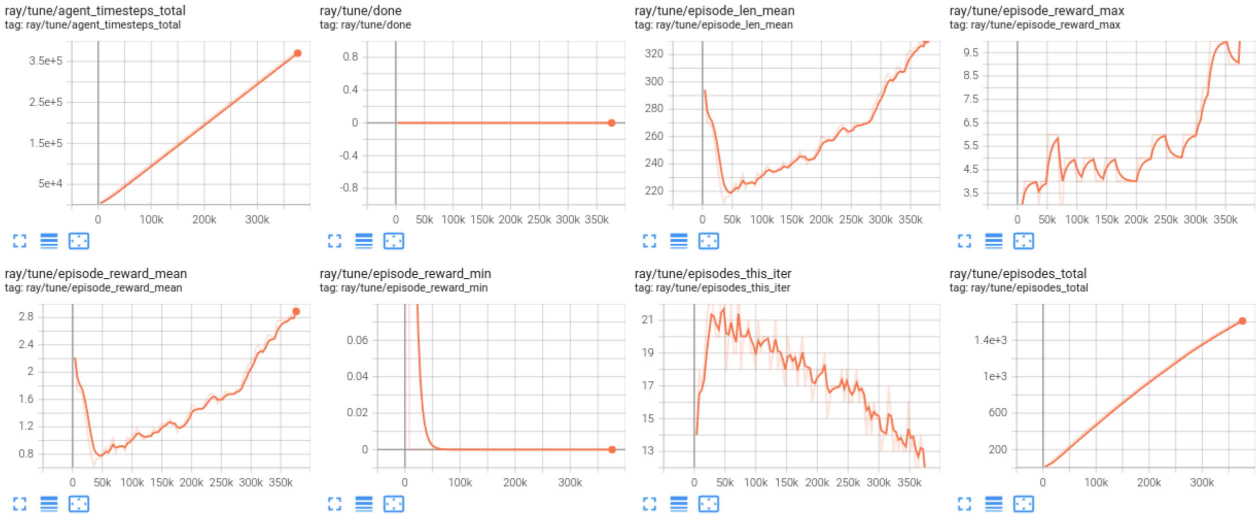
For A3C with 32 workers



The different in performance of 5 key metrics when comparing to 16-workers are better but not a lot. The curve looks smoother than 16-worker curves. The total training time is 2876 seconds. Total time step is a bit higher but on average each agent took much less steps than 16-workers. This is because the architecture design of A3C when all knowledge gain by agents will be pooled in to the global network and then update to each one of them.

Implemtation of PPO

For PPO with 2 workers



This is the performance results for 5 key metrics of PPO algorithm. This trial is running for 18 hours without GPU and 2 workers. As we have mentioned above. We can see that PPO is very stable in term of performance. There is no volatile in the curve of episode reward mean but the speed of reward increasing is very slow, sometime it feels like it is almost impossible for it to get to the target of 50 points. We late managed to use GPU to train the model but the result is far from being compared to A3C algorithms. Even our chosen environment falls into stability type but PPO has hard time training it.

References

Hado van Hasselt, Arthur Guez, David Silver (2015). Deep Reinforcement Learning with Double Q-learning.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning.

Adam Paszke. REINFORCEMENT LEARNING (DQN) TUTORIAL

Tom Schaul, John Quan, Ioannis Antonoglou, David Silver. Prioritized Experience Replay