International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 11, No.1 (Jan-2022)

https://dx.doi.org/10.12785/ijcds/110144

# A Brief Study of Deep Reinforcement Learning with Epsilon-Greedy Exploration

**Hariharan N[1] and Paavai Anand G[1]**

[1]*Department of Computer Science and Engineering, SRM Institute of Science and Technology, Chennai, TN, India*

*Received 16 Apr. 2021, Revised 15 Nov. 2021, Accepted 4 Jan. 2022, Published 20 Jan. 2022*

**Abstract:** This paper analyses a simple epsilon-greedy exploration approach to train models with Deep Q-Learning algorithm to involve randomness that helps prevail the agent over conforming to a single solution. This allows the agent to explore different solutions for a problem even after finding a solution. This helps the agent find the global optimum solution without being stuck in a local optimum. A simple block environment is built and used to assess the agent's ability to reach the destination, block A to reach block B. The model is trained repeatedly by feeding the game image and rewarding it based on the decisions made. The weights of the Neural Network of the Reinforcement Learning model are then adjusted by training the model after every iteration to improve the result. Furthermore, two different environments from the Gym library in Python is used to corroborate the results obtained. Here we have used TensorFlow to build and implement the model on the GPU for better and accelerated computation.

**Keywords:** Reinforcement Learning, Deep Q-Network, Epsilon-Greedy, Exploration, Randomness, TensorFlow, Gym

## 1. Introduction

Neural Networks are the basis of Deep Learning that allows computers to learn without being programmed to perform a specific task [1]. This allows machines to display somewhat similar to what is considered as intelligent behavior. With this technique computers need not be programmed for every specific task but can rather be allowed to learn and perform them in a way that is not limited to the input [2]. This is similar to our biological brain that functions with the interconnection of various neural networks each designed for different tasks like hearing or seeing. A model like this, but limited, can be achieved through weighted nodes that corresponds to a function and yields an output when given an input [3]. These weights are constantly updated as the computer learns, or in simpler terms can change the output to match that of the given training dataset. Artificial neural networks are systems motivated by the distributed, massively parallel computation in the brain that enables it to be so successful at complex control and recognition or classification tasks [4]. This dynamic capacity of the networks to adapt and change to provide the desired output for any input, related to the ones it has been trained on, has revolutionized the world by automating almost everything from manufacturing to marketing.

Though this seems too simple for something that is said to almost replicate the human brain, the most complicated thing known, it does not entirely mimic the functionalities of a biological brain. The human mind is a cluster of billions of neurons each transmitting information or signals at the speed of 80 to 120 meters per second. In addition to this the brain processes millions of signals in the form of information simultaneously computing a huge amount of data in a fraction of a second [5].

While it may seem nearly impossible to have so much computational power, at least for the next few years, it is possible for computers to learn and probably master a specific task. This is done through various types of learning.

1) Supervised Learning: where the computer is provided with both the inputs and the outputs for the training dataset and can tweak the weights until the output of the networks resembles the expected output. This is usually achieved through a loss or cost function which essentially dictates how different the output of the network is from the expected output. The weights are then adjusted to reduces this value.

2) Un-supervised Learning: where the computer is provided with the training data alone with no outputs. The computer then extracts features from each data and groups them together based on learnt similarities. This closely resembles real world scenarios where most data are unlabeled, and we usually match them with things that look similar to it. For example, all apples were initially grouped together as they look and taste similar and were then labelled as apples.

*E-mail address: hg8463@srmist.edu.in, paavaiag@srmist.edu.in*

3) Reinforcement Learning: where the computer learns based on the outcome of its action. This is probably the most human-like learning technique as humans usually learn better from having known their outcome. When a person performs an action, he/she is punished or rewarded. Reinforcement learning uses this method to make the computer learn. The computer is provided with a dataset and is then asked to provide an output. This is then compared with the desired output, and if it is similar the computer is rewarded, meaning the weights are not changed much, and if not, it is punished, meaning the weights are changed to better learn and predict correctly for the next data [6].

Although Supervised and Un-supervised learning are the most common algorithms, Reinforcement learning has unexplored potential to be the most applicable solution for most real-world problems. This learning is entirely based on the computer's ability to retry the given scenario repeatedly and correcting it each time until an optimal solution is obtained. This resembles closely to the way humans learn, the trial-and-error method. We explore various ways to solve a given problem, each time correcting ourselves from our mistakes, and try to find the best solution. This is exactly how reinforcement learning works, except since the computer does not understand the concept of mistake, we add the concept of reward [7]. Rewards are used to correct and adjust the weights of the network so as to refrain from making the same mistake.

In this paper, we will look at how reinforcement learning with epsilon-greedy exploration can be used to help the model learn more effectively for 3 distinctive environments. The block environment allows the agent, block A, to move around in its environment to find and reach the position of block B. In doing so, the agent tends to make mistakes like hitting the walls or not making it to the goal by being stuck in a perpetual loop. Here the reward system helps the agent learn and correct its mistakes and help achieve the goal. To corroborate this result, we have implemented the same approach for Mountain-Car environment and Acrobot environment from the Gym library.

## 2. Related Work

This section discusses important work in the literature that is related to our work.

### A. Q-Learning

Q-Learning is a form of model free algorithm which does not use transition probability distribution to solve the problem and instead works based on trial-and-error method in a reward-based system. It is a type of Reinforcement learning that helps correct the actions of the model after each mistake. This is done by accessing how different the output produced by the model is to the expected output. This is then used to compute Equation 2-A and the Q-values of each action in the model is then changed accordingly based on its value [8].

$$Q(s,a) = r(s,a) + y * (max)_a Q(s^{'}, a) \tag{1}$$

Here, the algorithm maintains a Q-Table whose values are constantly updated after every iteration. This table is used to predict the next move in the environment based on adjusting its Q-value as time proceeds. Initially this table will be filled with random values for each parameter which dictates the nature of the environment, say position or distance from goal, which will then be updated based upon the result of the action, or the reward. The reward will determine how these values will be changed, as if it yields a low value, the value in turn will be low and vice-versa. These values are updated after every episode and the discount help determine how impactful each Q-value is for the future Q-value. If the future action is not fruitful, then the previous action will be changed at the rate of the value of discount and so on. This loops back to the first step thus changing the Q-values of the actions of all the steps taken before, using the formula, that has led to this result.

As the agent moves in the environment, the Q-values are constantly updated after every set of steps throughout the learning stage. When the model succeeds for the first time, the Q-values of the actions of the path that lead to this success will be updated. In this case it is increased since the action with the highest Q-value is always taken for every step. These values are then optimized in the coming iterations so that the agent is more inclined to choose the path that yields success. These values are perfected as the training sessions progress over time.

The Q-values in the Q-table are updated using the Bellman Equation that falls under Markov Decision Process that incorporates randomness in the decision making [9]. This usually means the agent's decisions are not wholly responsible for success, and there exists some randomness that governs the environment. This paper represents a stochastic model which incorporates the presence of random actions to allow exploration. This means the present decisions are responsible for the future results along with the presence of a certain level of randomness [10].

### B. Markov Decision Process

Markov Decision Process (MDP) provides a framework for the decision-making situations where the outcome is determined by both the agent's decision and randomness. It is a discrete-time stochastic control process for sequential decision making when the outcomes are uncertain. In an MDP, there exists a set of environment 'E', agent 'X', a set of states 'S', a set of actions 'A', and a set of rewards 'R'. For each step, the agent can perform 'A' number of actions with each action returning a reward 'r' from a set of rewards 'R'. This makes the process of receiving reward an arbitrary function 'f' that maps state-action pairs with Equation 2-B.

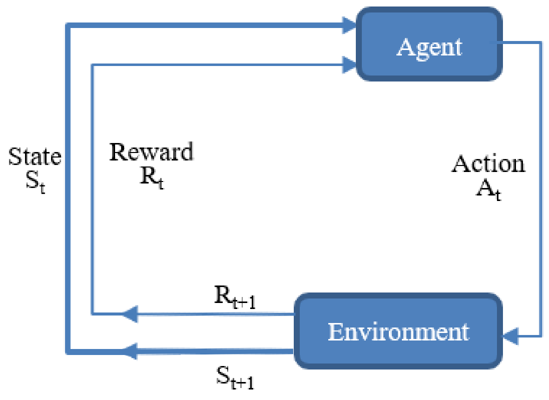


Figure 1. SARS Process

$$f(S_t, A_t) = R_{t+1} \tag{2}$$

This forms a trajectory of a sequential process where the agent in state 's0' performs an action 'a0', resulting in a new state 's1' for which it is presented with reward 'r1'.

As shown in Figure 1, choosing an action in a state generates a reward and determines the state at the next decision epoch through a transition probability function. These strategies are prescriptions of which action to choose under any eventuality at every future decision [11].

## C. BELLMAN EQUATION

While having a reward system to help the agent learn seems preliminary, the bellman equation provides a dynamic programming method to determine the value of each action 'a' in a state 's' as the maximum possible reward of the current state plus the value of the next state. This value, thus considering the impact of the action in the current state to that of the future state, determines the value of each action, or a node in deep neural networks, thus changing the transition probability.

$$V(s) = (max)_a(R(s,a) + \gamma \sum_{s'} P(s,a,s') * V(s')) \tag{3}$$

In Equation 2-C, 'V(s)' denotes the value of the action, 'R(s,a)' is the reward for the action a taken in state 's', 'V(s')' is the value of the future state and 'P(s,a,s')' is the probability that the action a taken in state 's' leads to the state 's''.

This equation determines the value of the state based on the max value of all the actions, the sum of the reward for taking the action, and the product of the discount value and the future state weighted by the probability of the agent choosing the state. The Bellman equation connects the value at any time-step to the expected value at the subsequent time-steps, thus valuing the element of exploration. It connects the uncertainty at any time-step to the expected uncertainties at subsequent time-steps, thereby extending the potential exploratory benefit of a policy beyond individual time-steps [9].

**Algorithm 1** Implementation of Bellman Equation

```
Ensure: $D_t = random\_batch\_from\_memory$
Ensure: $S_c = present\_states$
Ensure: $S_n = new\_states$
  $Q_c \leftarrow Prediction(S_c)$
  $Q_n \leftarrow Prediction(S_n)$
  $X \leftarrow x$
  $Y \leftarrow y$
  for $i \in D_t$ do
      $q_n \leftarrow reward + discount * max(Q_n, i)$
      $Q_c, i, action \leftarrow q_n$
      $X_i \leftarrow S_c, i$
      $Y_i \leftarrow Q_c, i$
  end for
  return X, Y
```

In this paper, we have implemented the bellman equation for a deep Q-network to process the Q-values to determine the action to be taken by the agent. Algorithm 1 shows the pseudo code of the implementation of Bellman equation to correct the weights using reward obtained for the current state and the future actions. With this the data is generated for the model to train on after each episode.

## D. Deep Q-Learning

Deep Q-Learning uses a Neural Network instead of a Q-table to help calculate the Q-values for taking an action in the environment. Here, we train a Neural Network and use weights to estimate the Q-values of each action in every step. These weights are then updated depending on the reward for each action. Notice that here we do not update the Q-values themselves, but instead we correct the weights that are used to calculate them. This means that they are versatile and do not use a Q-table to store all the values, thus requiring very less memory. Once learnt, these weights can then be used to calculate the Q-values by predicting using the parameters and can thus help the model to perform the most appropriate action [3]. It need not be trained for every possible dataset but can instead be scaled to be applied for different situations.

## E. Epsilon- Greedy exploration

In this paper, the agent can explore random actions before conforming to a single solution through epsilon-greedy method. This invokes a certain level of randomness to help the agent learn better and faster and refrain from being stuck in a local minimum. Without randomness, one cannot explore all the available options as the agent naturally conforms to a single solution after attaining it once [9]. If one is to be stuck to the regular regime, or in this case one single solution, the agent may not explore other solutions, one of which may be the most optimal solution for the problem. This is the same case with the agent in any environment. When the agent reaches the goal state for the first time, it is rewarded. This then results to the correction of the weights which will then result in the agent reaching the goal through the same path. This means the agent will eventually settle to this path without having explored other options or paths. This solution may be the local optimum, thus preventing the model from yielding the global optimum result. But, with the introduction of certain level of randomness, the agent, even after having found a solution, will continue to look for other solutions. In epsilon-greedy method, the agent will perform random actions if it satisfies a certain condition [12].

The value of epsilon determines the probability with which the agent performs a random action. As shown in Algorithm 2, if a random number generated between 0 and 1 is lower than the value of epsilon, the agent can take random actions and if not, the action taken by the agent is determined by the model.

**Algorithm 2** Epsilon-Greedy function

```
Ensure: $x = random\_number$
  if $x < epsilon$ then
      $action \leftarrow Random$
  else
      $action \leftarrow Prediction$
  end if
```

In this paper, the value of epsilon starts at 1 and slowly decreases as the number of training episode increases. This, however, is also not a constant decrease, but exponential. The value of epsilon decreases with the help of a logarithmic function. The logarithmic value of part of the episodes for which the agent can explore, is used to determine a probability with which the agent can explore. The value of epsilon determines this probability, thus not allowing the agent to take random actions regularly, but only when the probability allows in certain steps and in a controlled manner.

$$\prod_{1}^{n} \epsilon = 0.01 \tag{4}$$

**Algorithm 3** Exponential decrease function

```
Ensure: $n = number\_of\_Episodes$
  $t \leftarrow 0.8$
  $E_d(n,t) \leftarrow 10 * (log(0.01)/(n * t))$
  $\epsilon \leftarrow \epsilon * E_d(n,t)$
```

Algorithm 3 implements the exponential function proposed in Equation 2-E. Here, we allow the agent to try random actions with a probability of 0.01 or greater for 80% of the number of episodes. If the random number generated is less than the value of $\epsilon$ (epsilon), the agent then takes a random of the 4 available actions. If not, the action resulted from the prediction of the Neural Network determines move to be taken by the agent.

## 3. Literature Survey

There are several works like the survey done by K. Arulkumaran; et al., [13] or Li, Yuxi [14] that describe and analyse the working of deep reinforcement learning. The work of Zhou; et al., [15] implements and analyses strategies for a sensor node optimised by Reinforcement Leaning in a model of Markov Decision Process. The work of Fan; et al., [16] shows the theoretical analysis from both algorithmic and statistical perspectives of Deep Q-Learning. It shows the agent's performance in a simple deep reinforcement learning model. This idea was later developed and applied for Atari games in the work of Mnih, Volodymyr; et al., [17]. It shows the agent's potential to explore to some extent with the use of Bellman equation. With this the agent is allowed for minimal exploration until its first solution.

The Epsilon-Greed Action Selection was then introduced to allow the agent to continue exploration despite having found the solution in the work of Michael Wunder; et al., [12]. It shows how the epsilon-greedy exploration yields higher-than-Nash outcomes. The work of Bell-Thomas; et al., [18] shows the application of variation Bayesian Inference in a Double Deep Q-Network. It shows the value of epsilon decrease over time and uses that to train the agent with exploration. In the work of Azizzadenesheli; et al., [19], performance comparison is made by employing epsilon-greedy, Boltzmann Exploration and Thomson Sampling for exploration/exploitation.

In this paper, the agent can explore random actions with epsilon-greedy action selection with an addition of level of decrease in the probability of selecting randomness over training epochs. Like the work of Hester; et al., [20], we have used a Deep Q-Network model with a certain level of randomness to allow the agent to actively explore. With the help of discussions on Deep Reinforcement Learning and epsilon-greedy by Harrison [21], we have incorporated the use of a logarithmic function and allowed the value of epsilon to decrease exponential until a certain fraction of the training session. This determines how the value of epsilon deteriorates over each step and thus alters the probability with which the agent can explore. The agent's ability to actively explore is thus not pre-determined or fixed but controlled by a randomly generated number with a variable probability. This provides a simple solution that incorporate chaotic randomness while still maintaining the rigidity of the prediction made by the model. The model is fed directly with the game image, similar to the work of Tai; et al., [22], where the raw sensor information is used to build the exploration strategy.

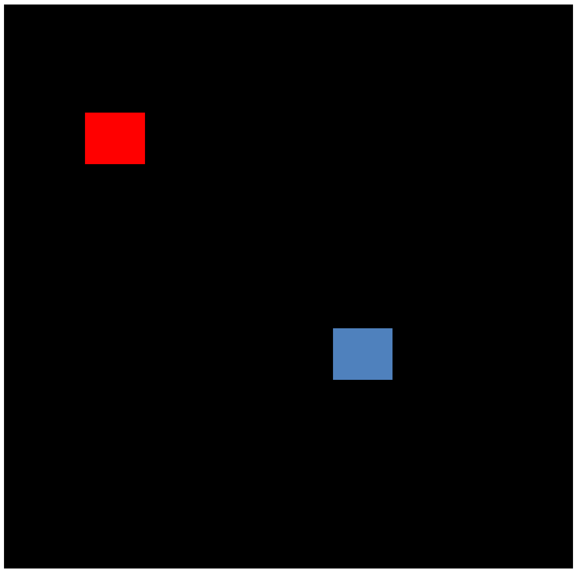


Figure 2. Block Environment

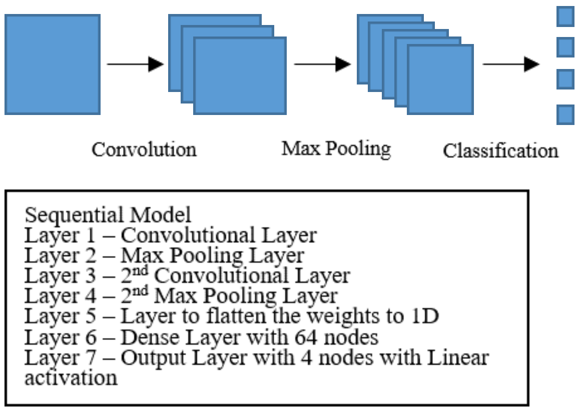
## 4. Methodology

Here, we show the implementation of our approach in Deep Reinforcement Learning in 3 distinct models.

### A. *Block Environment*

In this environment, TensorFlow is used to implement the Deep Q-learning model to train our agent, in this case the red block, to reach the goal, blue block. As shown in Figure 2, we have implemented a 10 by 10 sized window using cv2 library for rendering the environment. Each actor, block A and B, will occupy a single pixel in the window. The model is then trained to help block A move towards and reach block B. This setup is an example of a multi-agent model where both the agents, are free to move. But, unlike an ensemble model, only block A here is free to take actions, the position of block B is randomized after every game [23]. This further increases the complexity, demanding more exploration from the agent before settling for a single solution.

Figure 3 shows the architecture of the simple Convolution Neural Network (CNN) model used in this project, consisting of several layers stacked up sequentially. The first layer is the Convolutional layer that takes as input the NumPy array that consists the window of size 10 by 10 by 3, where the 3rd layer of 3 values is the RGB value of each pixel. The next layer is the 'MaxPooling' layer to extract only the useful information from the array thus improving the efficiency and reducing the complexity of the model. This is then Flattened in the next layer from 3 dimensions to a single flat array for making the calculations a lot easier. This is followed by a Dense layer consisting of 64 nodes and then the output layer consisting of 4 nodes or outputs. A few test runs were made with different number of nodes and found 64 to be a balance between time and performance.



Sequential Model
Layer 1 – Convolutional Layer
Layer 2 – Max Pooling Layer
Layer 3 – 2nd Convolutional Layer
Layer 4 – 2nd Max Pooling Layer
Layer 5 – Layer to flatten the weights to 1D
Layer 6 – Dense Layer with 64 nodes
Layer 7 – Output Layer with 4 nodes with Linear activation

Figure 3. Model Layers

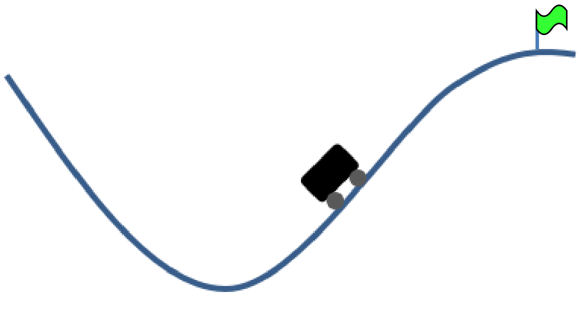


Figure 4. Mountain-car Game Environment

The 4 output nodes are the possible actions that the agent can perform, move up, down, left, or right. The index of the node that holds the highest value will then be considered as the suitable action required to be performed by the agent to reach its goal. A CNN is used here as the image of the game environment as whole is fed as the input.

The model iterates through the networks many times and updates the weights through Bellman equation for updating the Q-values with the help of the reward or punishment obtained from the moves made by the agent. This is done with the help of Algorithm 1. The training of the model is determined by the Episodes for which the agent will be trained. Each episode contains a specific number of steps for which the agent will be active. Here, the agent is allowed to take as many numbers of steps as it takes to reach the goal for each episode, and then the game will be reset.

The dataset is divided into batches, each of size 64, to help accelerate training. The number of episodes is fixed to 25000, and each time the game resets, the Q-values, actions, rewards, and states, are updated and the model is trained by setting the updated Q-values as outputs. The model is trained for 5 epochs for each episode.

### B. *Mountain-Car Environment*

In this environment, a car starts from downhill and must reach the top of the mountain, as shown in Figure 4. The car uses momentum by moving front and back, to launch

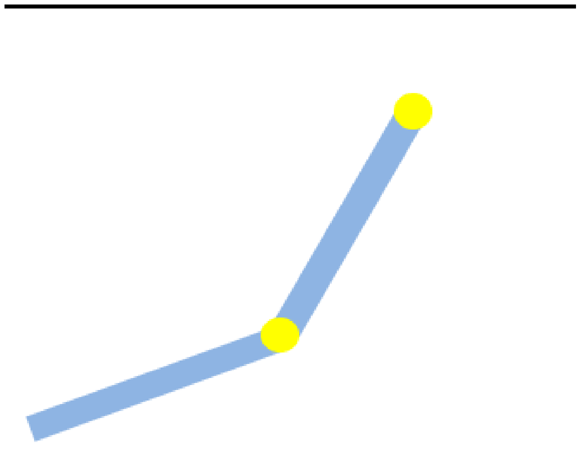


Figure 5. Acrobot Game Environment

off to the goal

This is exported from an existing open-sourced library called Gym by OpenAI in Python. OpenAI provides several environments and toolkit for reinforcement learning research [24]. A mountain-car environment is chosen here to better explain the advantages of allowing the agent to explore. A clear contrast between the local minimum and global minimum can be visualized in this environment.

The model is run for 5 epochs an episode for 2500 number of episodes with the same level of exploration. The same model is also trained without allowing the agent to actively explore to show the contrast.

### *C. Acrobot Environment*

Here, a two-link pendulum swings freely, as shown in Figure 5. The goal is to wing the end-effector at a height at least the length of one link above the base [25]. This is exported from Gym in Python and is the work of R Sutton; et al., [25] and A Geramifard; et al., [26].

The model is run for 5 epochs an episode for 1000 episodes with the same level of controlled exploration. The same model is also trained without allowing the agent to actively explore to show the difference in performance.

In all three environments, a memory table is used to help collect data for training the model with. The model is trained using randomly selected batch of data of size 64 from the memory table. The initial few steps are randomized to generate values for the memory table to initialize the weights. Then the training begins, where each parameter of the bellman equation is calculated and used to update the Qvalues. This is then used to train and correct the weights of the model.

During each step, a random value is calculated. This value is weighed against an epsilon value that deteriorates over every iteration. As shown in Algorithm 2, epsilon-greedy exploration is implemented which sets the probability with which the agent is allowed to explore different paths that allows it to eventually find the most optimal solution, instead of being stuck in a single solution. This exploration decreases eventually as the epsilon value deteriorates which helps the agent settle for the global optimum solution at the end of the training session. This is done with the help of an exponential decrease function as implemented in Algorithm 3.

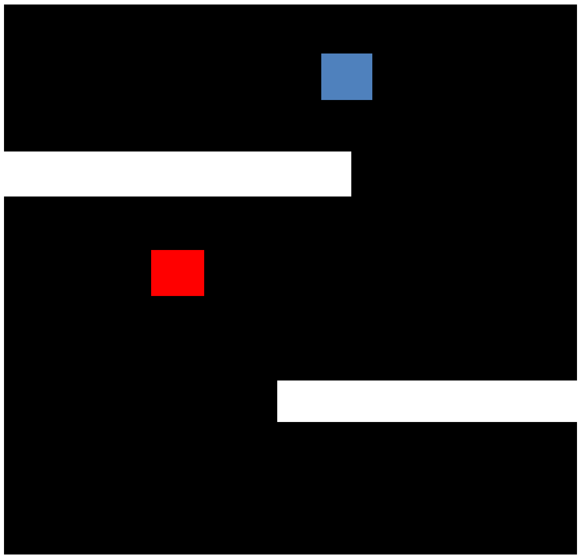


Figure 6. Block Environment with Obstacles

This model represents a stochastic model having a random probability for every move and thus having improbable prediction which in turn allows room for exploration. Stochastic models take account for random chaos in the environment that participates along with the decisions of the agent to affect the final outcomes.

## 5. Result

### *A. Result of Block Environment*

The model was executed on Google cloud using a single 12GB Nvidia Tesla T4 GPU. The model was run repeatedly for 25000 episodes. The results show the agent reaching the goal taking anywhere between 5-20 seconds for each episode. Two models were run, one allowing the agent to make random choices occasionally and the other with no randomness. Another environment with obstacles, to hinder the agent from reaching its goal, was also added for comparison. Then 10 test data were presented to compare how well the model performed. Figure 7 shows the results and demonstrates how quickly the agent reached the goal on an average for all the test data. As metrics for comparison, we have plotted the results against time.

With identical training and testing, the results from Table I provide an acute representation of how a certain level of randomness can allow the agent to explore for finding different solutions for a problem, thus when provided with

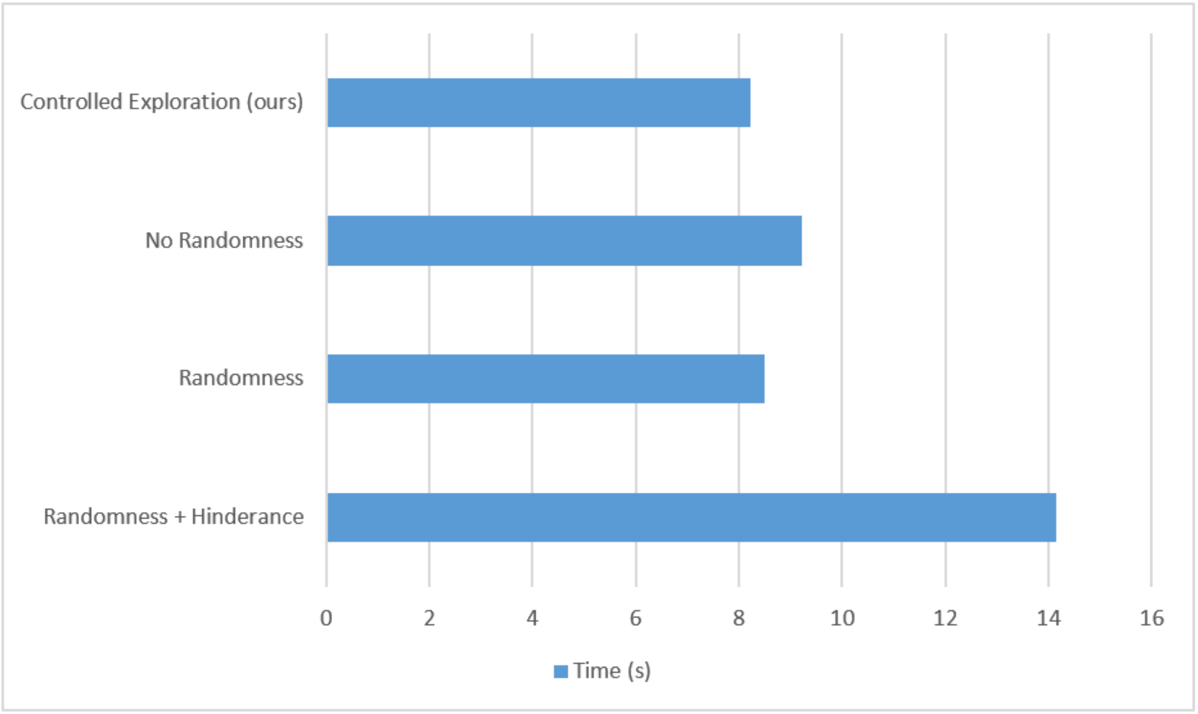


Figure 7. Total time taken to reach the goal during test run in Block Environment

| Method | Time Taken in seconds | Performance Ratio |
|---|---|---|
| No Randomness | 9.21 | 1.0 |
| Randomness | 8.5 | 1.08 |
| Controlled Exploration (ours) | 8.23 | 1.12 |
| Randomness + Hinderance (our method) | 14.16 | 0.65 |

TABLE I. Block Environment Results

an unknown test data, it can perform slightly better than when trained without randomness.

The element of randomness to some extent helped refrain the agent from conforming to a single solution and yielded almost 12% improvement in performance. Moreover, the controlled form of exploration improves the performance by over 3%. This is because when the agent performs a random action, it traverses through a path it has never been before. This eventually leads to multiple ways of solving the given problem. The agent can now solve a single problem in multiple ways and can thus find the most optimal solution in each scenario by accessing all the available options. Thus, the agent will not be stuck without options when one path is blocked. This also allows the agent to gain more knowledge about the environment, thus increasing awareness when any change is incorporated in it. This is the reason why the agent trained with the allowances of random action in a controlled way was able to react and reach the goal quickly for the given test data. However, there is still more research required to help resolve certain constraints that would further improve the performance of the agent.

Moreover, it was also found that when the border was removed for the agent to better explore different options without having to hit the end of the screen, it found different ways to approach the goal. Though not having explored the feature of having no boundaries, the agent soon figured it out when performing random actions. It used the border to its benefit by going over one side and emerging from the other to reach the goal faster. This demonstrates how the agent explores various paths of solution before settling for the most optimal one.

Another set of training was done with the model having random obstacles (white blocks) to block the path of the agent from reaching its goal, as shown in Figure 6. Here, the agent cannot stick to its original solution as the path may be blocked. The agent soon realized the obstacle must be evaded and managed to find different solutions. Though it took more steps, the agent managed to approach its goal through different paths. Having run the model through various obstacles, it was found that the agent has learnt various unprecedented tricks to finish the game efficiently.

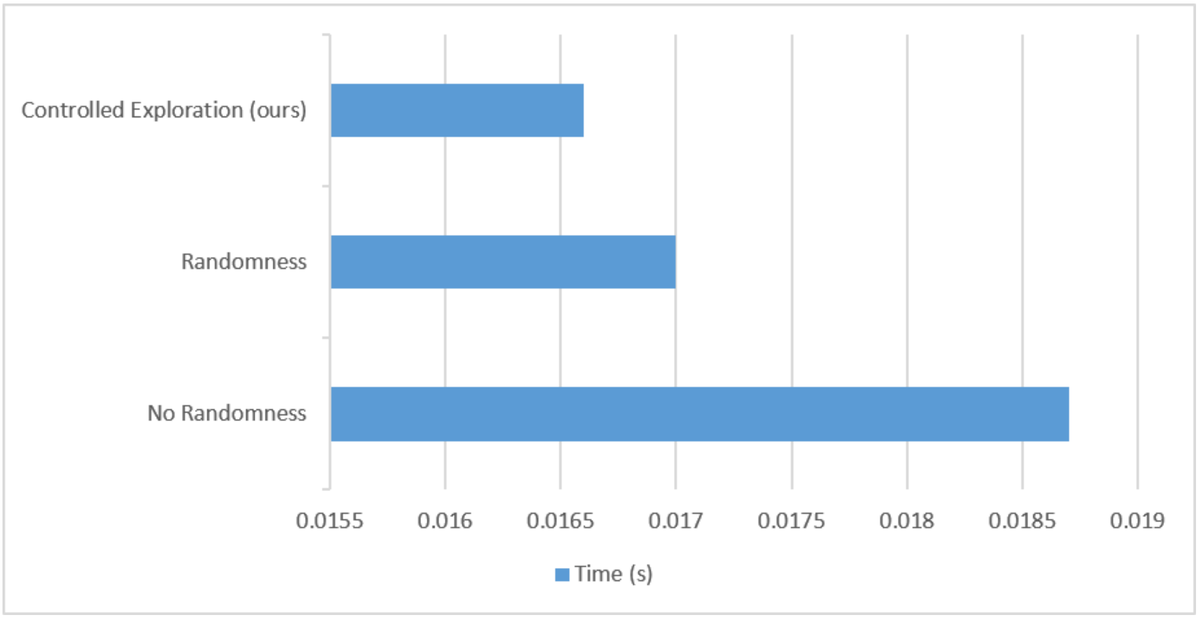


Figure 8. Total time taken to reach the goal during test run in Mountain-Car Environment

| Method | Time Taken in seconds | Performance Ratio |
|---|---|---|
| No Randomness | 0.0187 | 1.0 |
| Randomness | 0.017 | 1.1 |
| Controlled Exploration (ours) | 0.0166 | 1.13 |

TABLE II. Mountain-Car Game Results

For example, when presented with an environment that refrained the agent from moving to the other half of the screen, the agent moved to the end of the screen to appear from the other side and arrive at the goal much faster than anticipated.

Another observation made during the implementation of this project was the setting of right value as reward given to the agent for each action. The right reward value seemed to play a significant role in how quickly the model reaches an optimal solution. It also plays a role in how intense the calculations are by determining how large the difference between the obtained output and expected output is. This reduces the weights used to calculate the outputs, which in turn reduces the complexity of the calculations involved. If the rewards were too low, the changes in the Qvalues become insignificant which reduces the value of any move made by the agent. On the other hand, if the rewards were too high, the difference seems too great, and the calculations intensify. Thus, having the right value for rewards is an excellent way to help the agent learn quickly and efficiently.

One finding during the training session is that, when the agent can explore, it performed random actions despite being close to the goal. Though random actions have benefits when it comes to exploration, this in certain instances delayed the agent from finishing the game.

## B. *Result of Mountain-Car Environment*

The above result is clearly supported by the Mountain-Car environment, where a single agent is required to use the momentum efficiently and reach the goal. The model was trained with and without the inclusion of randomness and the results in Figure 8 were obtained.

As we can see from Table II, the model, when trained with randomness performed better by reaching the goal quicker than when trained without randomness. It can be found that the agent in the model trained with randomness reached the goal 1.13 times faster than the agent in the model trained without randomness. This means there was a performance boost of 13% when the concept of exploration was incorporated into the model. Moreover, controlled exploration yields over 2% increase in performance.

This difference seen in the performance of the model is largely due to the model being stuck at a local minimum. With the agent being allowed to explore with an exponentially decreasing probability, the model will have enough room to make random moves even after a solution, thus increasing the chances of finding the global minimum. This shows how the model trained with randomness was able to perform better, by not being stuck at the local minimum and reaching the global minimum.

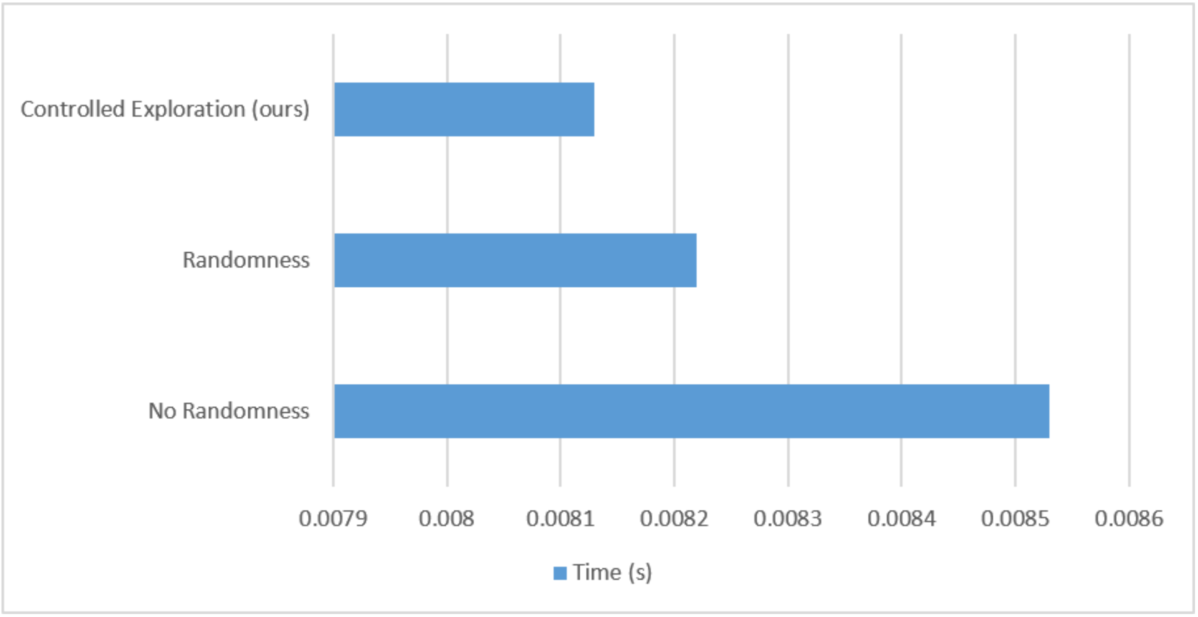


Figure 9. Total time taken to reach the goal during test run in Acrobot Environment

| Method | Time Taken in seconds | Performance Ratio |
|---|---|---|
| No Randomness | 0.00853 | 1.0 |
| Randomness | 0.00822 | 1.038 |
| Controlled Exploration (ours) | 0.00813 | 1.044 |

TABLE III. Acrobot Game Results

Also, one thing to keep in mind is that since the agent will explore more in this approach, the initial success rate will be minimum compared to when trained without randomness. This is because the agent will not settle for the initial path and will continue to learn until the epsilon value deteriorates completely. Thus, this is an expensive method and requires a greater number of epochs to find the solution.

## *C. Result of Acrobot Environment*

The result of this environment also supports the claim that the model indeed performs better when the agent can actively explore.

From the results obtained in Figure 9 and Table III, the model, when trained with randomness, performs 1.044 times better than when trained without randomness. Simply allowing the agent to explore yields at least 4-5% better performance. Our controlled form of exploration yields an improvement of over 0.5

It can be noted that the final few epochs did not yet yield a perfect result. This is because the number of training epochs were not sufficient, and the agent required more training to explore the environment. This same trend can be observed with the Mountain-Car environment as even in the last few episodes, the model still showed some spikes in time. Despite this, the model performed better than when trained without exploration. As the exploration increases, the number of training episodes required to run for optimizing the model will also increase. When allowing to explore, the agent will try and make random moves to better understand the environment. So, the model will not be optimized correctly with less training, thus affecting the performance. Future work is to be conducted to understand the correlation between the exploration rate and number of training episodes to find the right balance that is also efficient in terms of computation.

While many existing Reinforcement Learning models fail to incorporate randomness while training the agent, with the obtained results, it is confident to say that allowing the agent to explore could be advantageous for most environment. This increases the agent's learning range, thus allowing the agent to have a complete understanding of the entire surrounding. If the whole environment is explored by the agent, any small change to the environment will not require the model to be completely retrained again. As observed, the model will itself adapt to the changes depending on how well it has explored and how significant the changes are to the environment.

Though exploration is necessary for the agent to find multiple solutions, this could sometime hinder the learning process. While the bellman equation ensures the learning of the model, it is important for the agent to perform these actions so as to improve the model. If the agent can perform random actions without a controlling metric, the agent will never conform to any solution, thus invalidating the learning of the entire model. To balance learning from both the model and the random actions in the environment, we have shown the implementation of the exponential decrease function for the value of epsilon. This offered a controlled approach for training the model while incorporating a certain level of randomness.

## 6. Conclusion and Future Work

Deep reinforcement learning is a method of learning that resembles closest to that of a human being. It is a model-free form of learning that does not require a collection of data consisting of all possible states but instead allows the computer to form its own data to learn from by making mistakes and adjusting the weights to correct them. This reward-based learning offers an independent approach to training a model without having to rely on data.

With Reinforcement learning's wide range of applications such as industry automation or autonomous vehicle, with a controlled allowance of exploration, the agent is sure to reduce the error. This method, to be more effective, requires a form of randomness to help the agent explore different paths for a single problem. Finding different solutions not only helps the agent to not be stuck in the local minimum, but also to adapt to different scenarios quickly. With the result obtained from the tests it is evident that controlled randomness to some extent will allow the agent to explore better and in turn perform better with the unseen data.

Although this can help the agent learn more about the environment, it is an expensive method of learning. Since it will not conform to a single solution, it requires many iterations to learn effectively. We managed to provide a controlled way of exploration which allowed the agent to perform random action only till a certain point. However, this again restricts the agent if the training session is less. Therefore, more research on this topic is required to produce greater results in a smaller number of training episodes.

One other point to note is that while allowing the agent to explore with a certain probability, one must also try to base it on the position of the agent. For example, if the agent is right next to the goal, it becomes pointless to perform a random action. This posed a great problem in the initial stages of training when the probability was high. Allowing the agent to explore in a controlled and sensible way will further help the agent learn effectively. Future work in this aspect will investigate the performance drop due to this factor and will provide an improved approach.

## References

[1] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[2] M. Van Gerven and S. Bohte, "Artificial neural networks as models of neural information processing," *Frontiers in Computational Neuroscience*, vol. 11, p. 114, 2017.

[3] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International conference on machine learning*. PMLR, 2016, pp. 2829–2838.

[4] R. R. Nadikattu, "The supremacy of artificial intelligence and neural networks," *International Journal of Creative Research Thoughts*, vol. 5, no. 1, 2017.

[5] O. Sporns, G. Tononi, and R. Kötter, "The human connectome: a structural description of the human brain," *PLoS computational biology*, vol. 1, no. 4, p. e42, 2005.

[6] B. Kiumarsi, K. G. Vamvoudakis, H. Modares, and F. L. Lewis, "Optimal and autonomous control using reinforcement learning: A survey," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 6, pp. 2042–2062, 2017.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[8] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for Dynamics and Control*. PMLR, 2020, pp. 486–489.

[9] B. O'Donoghue, I. Osband, R. Munos, and V. Mnih, "The uncertainty bellman equation and exploration," in *International Conference on Machine Learning*, 2018, pp. 3836–3845.

[10] X. Lin, S. C. Adams, and P. A. Beling, "Multi-agent inverse reinforcement learning for certain general-sum stochastic games," *Journal of Artificial Intelligence Research*, vol. 66, pp. 473–502, 2019.

[11] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[12] N. L. Kuang and C. H. Leung, "Performance effectiveness of multimedia information search using the epsilon-greedy algorithm," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 2019, pp. 929–936.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[14] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[15] L. Zhou, H. Tang, H. Li, and Q. Jiang, "Dynamic power management strategies for a sensor node optimised by reinforcement learning," *International Journal of Computational Science and Engineering*, vol. 13, no. 1, pp. 24–37, 2016.

[16] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for Dynamics and Control*. PMLR, 2020, pp. 486–489.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[18] A. Bell-Thomas, "Exploring variational deep q networks," *arXiv preprint arXiv:2008.01641*, 2020.

[19] K. Azizzadenesheli, E. Brunskill, and A. Anandkumar, “Efficient exploration through bayesian deep q-networks,” in *2018 Information Theory and Applications Workshop (ITA)*. IEEE, 2018, pp. 1–9.

[20] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband *et al.*, “Deep q-learning from demonstrations,” in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[21] H. (Sentdex), “Python Programming Tutorials,” https://www.pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/, 2019, [Online; accessed 10-September-2020].

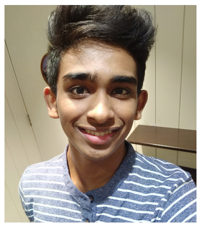
[22] L. Tai and M. Liu, “A robot exploration strategy based on q-learning network,” in *2016 IEEE international conference on real-time computing and robotics (RCAR)*. IEEE, 2016, pp. 57–62.

[23] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie, “Multi-objective workflow scheduling with deep-q-network-based multi-agent reinforcement learning,” *IEEE access*, vol. 7, pp. 39 974–39 982, 2019.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[25] J. Rafati and D. C. Noelle, “Sparse coding of learned state representations in reinforcement learning,” in *Conference on Cognitive Computational Neuroscience, New York City, NY, USA*, 2017.

[26] A. Geramifard, C. Dann, R. H. Klein, W. Dabney, and J. P. How, “Rlpy: a value-function-based reinforcement learning framework for education and research.” *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 1573–1578, 2015.



**Hariharan N** Hariharan is a Computer Science graduate from SRM Institute of Science and Technology, India. He is a passionate learner and an Artificial Intelligence enthusiast. He has worked on several projects with models implementing Supervised Learning and Reinforcement Learning in TensorFlow and PyTorch. He is a Computer Science aspirant and a talented programmer. He is currently fascinated by the idea of robot or agent performing random exploration in a system and increasing intelligence. His expertise includes Python, TensorFlow, PyTorch, Machine and Deep Learning and other relevant fields.

**Paavai Anand G** Paavai Anand is an Assistant Professor in the Department of Computer Science and Engineering, SRM Institute of Science and Technology, India. She has obtained her B.E degree from Periyar University, India. She has obtained her M. E and Ph. D degree from CEG campus, Anna University, Chennai. She has more than 10 years of teaching experience. She has worked on a set of Machine Learning algorithms like Probabilistic, Bootstrapping and Genetic Algorithms for improving the performance of search engines that comprises of both the surface and the deep web. She has published papers in various International conferences and international journals including the most prestigious ACM Computing Surveys journal. She has also received best paper awards for some of her conference publications. Currently she is working on Automated Machine Translation and Information Extraction using Machine Learning, and classification of web pages. She is an enthusiastic learner who always aspires to learn new things.