Implementing Uniform and Prioritized Experience Replay to DQNs

Implementation Track, 4 Team Members

Abstract

Experience replay is a technique that helps online reinforcement learning agents use their memory to remember experiences from the past and enhance their decision-making process from this memory. In this paper, we have first applied uniform experience replay using a deep Q-network, in which transitions are uniformly sampled from the agents' replay memory. In the second part, we apply proportional prioritization to replay the transitions that we categorize as more important, at a higher frequency. Prioritization gives a more robust and effective learning system. We apply experience replay and prioritized experience replay to deep Q-networks. DQN is a reinforcement learning algorithm, which has achieved state-of-the-art performance on Atari games and Robotics Environment from the OpenAi Gym.

Keywords: Prioritized experience replay, Q Learning, DQN, uniform experience replay

1 Motivation

The first step in understanding the amalgamation of neural networks with reinforcement learning is the methodologies used to develop the Markov Decision Processes (MDP). MDP governs the mathematical modeling for the decision policy of the agent, subject to a given environment. The "Deep Reinforcement Learning" in (1) lists the outline of the algorithm which can be used to achieve the skeleton model for a basic RL task. This model is applied on 2600 Atari Game environments from the OpenAi gym framework and compares the performance of the agent within these different environments. This sets a benchmark for the basic application of CNN in RL.

As the complexity of the model increases, and the agent is subjected to performance constraints, like achieving a score of more than 100 in 60 seconds, or completing the task in minimum computations, it is essential to deploy a better methodology to the algorithm. One of the recently developed methods is the use of prioritization of the replay memory of the agent. Here, the agent prioritizes it's experiences instead of sampling from them equally. The way in which it decides which segment of the memory to prioritize is quite interesting. Schaul, Quan, Antonoglou, and Silver (2) lists a step-by-step methodology for achieving this on different levels. Furthermore, it also compares the performance of the agent deployed under different methods in achieving a variety of tasks.

Most of these methods are tested on the Atari Game environment, where the agent is under state and action constraints (that is it is restricted under a confined space with limited actions). In the scope of these cited papers, these methods are extended to different data sets like MNIST where the objective is label prediction from RL. Thus, it allows performing the learned techniques in real-world model predictive problems.

2 Related work

Experience replay has a lot of research put into it as a way to stabilize DQN networks, as well as make them more efficient. It accomplishes this by saving a fixed number of previous samples for training and then randomly sampling from them.

2.1 Revisiting Fundamentals of Experience Replay

(3) released a paper explaining some of the mechanics of experience replay's effect on reinforcement learning. Similar to our experiments, they conducted their analysis on the Atari openAI game selection. They fixed the total number of

gradient updates and the batch size per gradient update to the model's parameters in these tests, ensuring that all agents train on the same number of total transitions. The results of their experiments were that when using the same number of the transition's to train their model, the bigger the experience replay buffer, the better the dopamine rainbow model performance. However, this result did not hold for the DQN model which differs by a double update while training.

This motivated the problem, how to improve DQL learning in both efficiencies of training and performance.

2.2 Prioritized Experience Replay

In 2016, the idea of Prioritized Experience Replay was introduced by (2). They examined how selecting which transitions are repeated may make experience replay more economical and effective than if all transitions are replayed evenly. An RL agent may learn more successfully from certain transitions than from others, depending on the circumstances. Transitions might be more or less unexpected, redundant, or task-relevant depending on the situation. As part of their experiments, they used a similar Q-learning model, as the previous work, but expanded experience replay to prioritized stochastic gradient descent. Their results showed an increase in overall performance compared to experience replay, as well as better performance in the same number of training steps compared to the uniform sampling of replays.

3 Problem Statement

DQL is a technique under Q Learning where a deep neural network is made to learn the Q values. The experiences stored in the memory are used to learn the neural network using backpropagation and a probabilistic approach. Most complex environments have 2 or 3-dimensional input (observation space), usually in the form of images.

For example, the Atari game set, available from OpenAi Gym, lists more than 2600 games. As the complexity of the game increases, the observation space is defined from the snapshot of the image at a particular instance. The Atari game environment takes an action as input and executes this action in the game, resulting a colored image as an output of the next state.

To incorporate such a 3-dimensional input, a convolutional neural network is to be used. The motivation for using deep neural networks is the fact that it allows a large number of input nodes to be passed into the convolution during the forward pass. The architecture can be defined in such a way that as the information feeds into the network, the dimensions are reduced to a specific number, the number of actions (output). These values denote the probability values, ranging from 0 to 1. Each value indicates the probability of the action (Q values) which can lead to maximum reward. The action with maximum probability value is chosen by the agent and a reward is computed as the state changes.

4 Introduction

Reinforcement Learning is a field under machine learning where an agent interacts with a given environment and is assigned to achieve a particular goal in it. For instance, a 10 x 10 grid can be considered as a given environment, where an agent is placed at one location (A) and the goal is to travel to another location (B). The agent can execute actions from a defined action set. In the given example, these actions can be left, right, up, and down. A general constraint imposed (objective) in such examples is to have the minimum number of steps to achieve the goal.

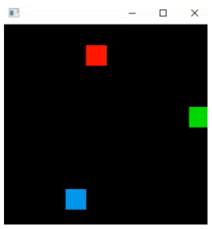
Figure 1: 10x10 grid environment, where the Agent (in blue) is tasked to reach the goal (in green), by avoiding the Enemy (in red)

Say the complexity of the environment is now increased, by introducing an obstacle. The task is to reach the goal location (B) without walking into the enemy cell. This can be further extended to have multiple enemies, which move in a defined fashion. It can be observed that as the complexity increases, the number of ways in which the agent can achieve this task reduces. At first, the agent had the liberty of using a large set of actions that would fulfill its goal of reaching the destination. But with increased complexity, there would only be a handful of sets of actions that would lead to success.

The intuition and consciousness embedded in decision-making enables humans to easily adapt to changing environments. Part of the reason is experienced learning. This important concept is extended to the field of reinforcement learning, enabling the agent to initially explore as many methods as possible, generally in a random fashion, and understand which specific steps lead to fruitful results. A reward is assigned in such situations, and a penalty when not. The idea is to maximize the reward, enabling the agent to "learn" the method of achieving a task, as the environment changes. In the new setup, a heavy penalty (negative reward) is imposed if the agent walks into an enemy cell. In this way, the agent will avoid taking those actions which reduce the total reward.

5 Methodology

5.1 Introduction to Q Learning

Q learning is an off-policy learner, suggesting that the agent learns the optimal policy independent of what action it takes. The value of Q refers to the expected value of a certain action in a certain state. In this technique, a Q table is introduced which lists the Q values of all the states and all the actions. The Q value is stored in a matrix in which the rows of the matrix represent the states and the columns represent the actions. During the training phase, an agent will have some percent chance of taking a random action, in order to let it explore the full action space. Otherwise it will choose the action which has a maximum Q value at the current space. The Q value for different actions is calculated using the Bellman equation (equation 1). This lookup table computes the rewards achieved in the expected future and is maximized for the defined task. This acts as a guide for the agent in taking the best set of actions to maximize the given task. The values in this Q table are set randomly at the first instance. In many instances, also uniformly set to zero, to reduce computational complexity. As the agent explores, the values in this lookup table follow the Bellman equation based on the current state and action. The bellman equation uses the cumulative expected reward (weighted by the discount factor) and is defined as follows

$$Q_{i+1}(s,a) = E[r + \gamma max_{a'} Q_i(s', a')|s, a] \tag{1}$$

γ denotes the discount factor, which suggests the amount of weight-age given to the reward achieved at this instance. As observed from the given bellman equation, the values in the Q table are computed based on the expectations from the current state and action.

As the agent explores, the values in the Q table are updated to maximize this reward. The iterative formula to update the Q value in the Q table is as follows:

$$Q^{new}(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a)) \tag{2}$$

Here, α corresponds to the learning rate hyperparameter which can be used to finetune the model.

The estimate of optimal future value is calculated as $\max_a Q(s_{t+1}, a)$ (obtained from the output of the nerual network). Thus, the term $Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a)$ denotes the learned Q value from deep learning technique.

5.2 Experienced Replay and Q Learning

The methodology implemented which enables the agent to learn (from its exploration) and store the actions, state, and the rewards, which led to either success or failure in achieving the set task, is called experienced replay. As the name suggests, the agent is bound to replay its actions from its past experiences. These experiences are stored in a memory block, named the replay memory, in the form of actions and rewards.

As the agent explores the environment by a set of random actions, the replay memory keeps on getting filled with these values. After a certain threshold of memory, also denoted as the "batch size", the algorithm feeds this information into its Q table and performs the previously mentioned iterative computation in updating the Q values. The agent now takes input from these Q values to perform the actions suggested by them. The process of accumulation of memory is repeated until it reaches the batch size. And so on.

This algorithm can be thought of as analogous to stochastic gradient descent, where the iterative process doesn't wait for all computations of gradient descent, but merely a batch size of it, and move on using the descent values from this batch size.

It is important to understand how this “replay memory” is defined and what classes are used to write the entire algorithm. Initially, a simple “Cartpole” environment is imported from OpenAi Gym to perform and test these methods. In this environment, the agent is a cart pole hanging on a horizontal string. The first step in building a reinforcement learning script is to understand the observation space of the environment and the action space of the agent. In the latter half of the project, an implementation of the training model with a better network architecture is executed and applied to the previously mentioned Atari game-set.

The values of states, actions and the future states after performing said action, and the reward achieved, are stored in these predefined memory blocks.

The value iteration algorithm (equation 1) provides the necessary convergence for the required action value function given by $Q_i \rightarrow Q^\star$ as $i \rightarrow \inf$. The better way than this is to use an approximation function to estimate this value, i.e. $Q(s, a; \theta) \approx Q^\star(s, a)$. This is generally a linear approximation but we may use a non-linear approximation too sometimes, such as neural networks. We take this neural network function as Q-network with weights, θ. A Q-network can be typically trained by minimising $L_i(\theta_i)$, which is a sequence of loss functions, which change during every iteration i:

$$L_i(\theta) = E_{s,a}[(y_i - Q(s, a; \theta_i))^2]$$

where,

$$y_i = E_{s'}[r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \nabla_{\theta_i} Q(s, a; \theta_i)]$$

5.3 Deep Q Learning (DQL)

Recent breakthroughs in speech recognition and computer vision have been based on effectively training the deep neural networks on large training data sets. By feeding an apt amount of data into the deep neural networks, it is mostly possible to get better representations than the old handcrafted features. The success in this has been the motivation for the chosen approach in reinforcement learning. The aim is connecting the RL algorithm to deep neural nets, which can operate on the RGB images directly and produce an action-value function through the stochastic gradient descent updates. In all, this gives a function DQN, which has the program for a deep q-network; the technique experience replay is then applied to it, and finally, the efficiency is improved by applying stochastic gradient descent to the program.

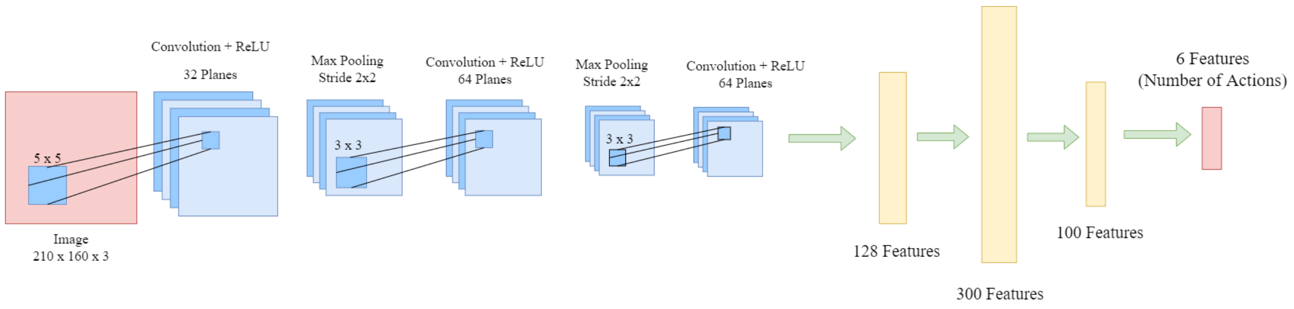

Figure 2: The architectural overview of using CNN for Reinforcement Learning

Inspired from the neural architecture in (2), the team has developed a novel network suiting the images from the Atari Gamse set, as seen in figure 2. In contrast to the general online approaches, we have utilized an experience replay (1), in which the experiences of an agent are stored at every time step in a dataset, which is pooled over a large number of episodes into the replay memory. We apply updates in Q-learning (also called minibatch updates) to experience samples taken out at random from the stored sample pool, as seen in Algorithm 1. The agent then selects and produces an action, which is in accordance with an ϵ greedy policy.
Giving numerous lengths to neural networks as inputs have always been a challenge, the Q-network works on fixing this issue and acts on a fixed representation of length that is produced by another function.

Experience replay has many advantages over online Q-learning (2). For instance, every experience step is used in various weights updates, which increases efficiency. Another advantage is that when the samples are randomly selected, correlations break and hence variance of updates decrease, which can help increase the stability of training.

The Q values of the given batch size are calculated using the Bellman equation (equation 1). Then that batch of images is passed through the neural network and the Q values are obtained as output from the neural network. The Q values are subtracted from the Q values computed from the formula (Which is the actual Q value). This difference is to be minimized such that the output from the neural network gives true Q values. When the network (or agent) is trained enough then these actions are only decided by the Q values that we get after passing the present image to our neural network and action is decided based on the given Q values. The action which has the highest Q value is taken and passed to the environment (Atari game).

Algorithm 1 Deep Q-Learning with Prioritized Experience Replay

```
Initialize some replay memory M to a capacity N
Initialize Q, i.e. action-value function with random weights
for episode = 1, numgames do
    Initialise the sequence $s_1 = x1$ and the sequence pre-processed $\phi_1 = \phi(s1)$
    for t = 1, T do
        With uniform (ER) or scaled(PER) probability $\epsilon$, either select a random action $a_t$
        or select action, $a_t = max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and find the reward $r_t$
        Set sequence $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process sequence $\phi_{t+1} = \phi(s_{t+1})$
        Store this transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in the replay memory M
        start sampling random minibatches of transitions $((\phi_j, a_j, r_j, \phi_{j+1}))$ from M
        Set $y_j = r_j$, for terminal sequence $\phi_{j+1}$
        Or use Bellman's equation to set $y_j = r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta)$, for non-terminal sequence $\phi_{j+1}$
        Perform a probabilistic step for prioritization on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

5.4 Prioritized Experience Replay

To improve upon our Experience Replay algorithm we introduce the concept of prioritized experience replay. Instead of the uniform sampling of pooled episodes worth of training data, we assign different sampling probabilities to the experiences. This will prioritize which replays the agent can "learn the most from". The TD error is used to assign different probabilities to different data points. TD error is a way of judging "how surprising" a training point is, and the more surprising data points will be replayed and used for training more frequently, to train our model quicker. More technically, TD error is the difference between the model's expected output for a certain state and the observed output. The greater the error the, more incorrect the model was for responding to a situation.

$$\delta_i = r_t + \gamma * Q(s_{t+1}, \mathit{argmax}[Q(s_{t+1}, a; \theta_t^-) - Q(s_t, a_t; \theta_t)]) \tag{3}$$

In order to apply the TD error to prioritize the replay memory, the following two equations have been used to assign sampling probabilities to each transition (4).

$$p_i = |\delta_i| + \epsilon \tag{4}$$

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5}$$

As seen in equation 4, a scalar ϵ is added to the TD error δ in order to insure each transition has some probability of being sampled. This will help prevent over fitting to the data that has the highest prioritization. The value α will determine the extent to which we prioritize data, with α = 1 representing a uniform sampling rate and a higher α meaning replay points with a higher TD value having a higher rate of being sampled. The values ϵ and α will be considered hyper-parameters that will will experiment with to try and achieve the best performance of our model.

5.5 Experiments and Datasets

Experiments are performed on various OpenAi Gym game sets, shown in figure 3. These are trained using experience replay initially to test if the training was working properly and then applied prioritization to improve the performance of the trained model. We checked our learning algorithm, hyper parameters, and network architecture setting on different atari games, to show that our approach is robust enough to work well on a variety of data with no incorporation of any specific information related to the data. We implemented reinforcement learning to the unmodified data and the reward function was learnt from the given data during training. We then evaluated the performance measures w.r.t average rewards and reward per episode for both uniform experience replay and prioritized experience replay (5).

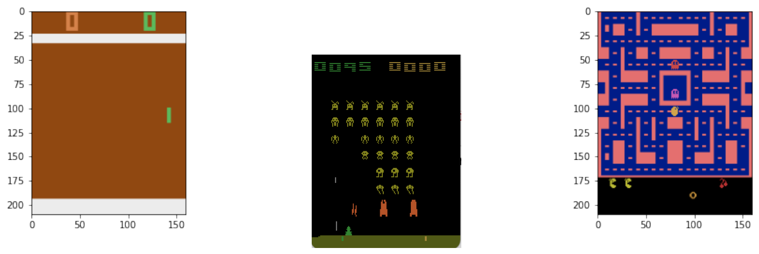
Figure 3: Environments for Pong-v0, SpaceInvaders-v0, MsPacman-v0

6 Evaluation

6.1 Training Setup

In our project, we used a 6 layered feed-forward neural network. This is a 3 layered convolutional neural network followed by 3 fully connected layers. Each convolutional layer comprises of a pure convolution followed by ReLU and max pooling operation. The 3 fully-connected layers in the network are also segregated by a rectifier non-linearity. The final layer is a soft-max which helps in obtaining a normalized distribution over all possible labels. The complete architecture is shown in 2. We used the MSE loss criterion for our model.

Layers	**Input Channels**	**Output Channels**
Convolution Layer 1	3	32
Convolution Layer 2	32	64
Convolution Layer 3	64	64
Layers	**Input Features**	**Output Features**
Fully Connected Layer 1	128	300
Fully Connected Layer 2	300	100
Fully Connected Layer 3	100	6

Table 1: Model Network Summary

The table 2 shows the hyper parameters we chose for our learning agent.

Hyper Parameter	**Value**
Alpha	0.4
Beta	0.4
Gamma	0.99
Learning Rate	0.001
Epsilon	0.1
Memory Size	1000

Table 2: Model Hyper Parameters

6.2 Results of Uniform and Prioritized Experience Replay

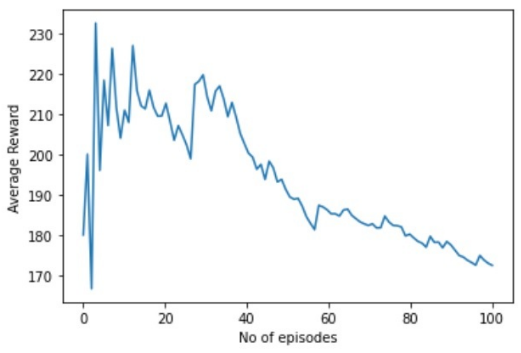

Figure 4: Average Reward vs Episode Number for Uniform Experience Replay

Figure 4 show the graphs for avg reward vs episode number for 100 iterations (or games), in case of Uniform Experience Replay.

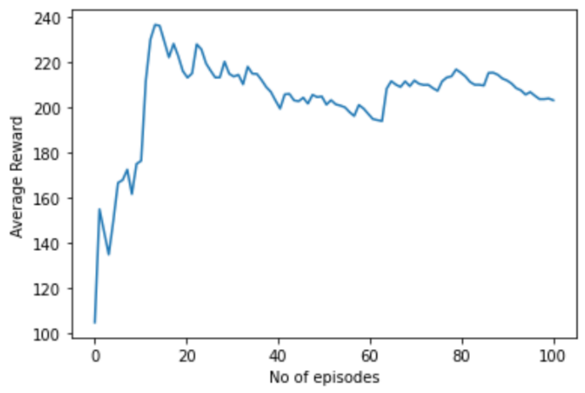

Figure 5: Average Reward vs Episode Number for Prioritized Experience Replay

Figure 5 show the graphs for avg reward vs episode number for 100 iterations (or games), in case of Prioritized Experience Replay.

The average reward was not as robust for Uniform ER than the Prioritized ER as the graph was varying between 170 and and 200 average value in the former while in PER, the average value of rewards ranged between 200 and 240, when we tested it for 500 games.

6.3 Comparison between ER and PER

We used a better suited network architecture for the DQN than what was used in (1), which improved our performance characteristics as mentioned in the section 6.2.
As seen from the graphs 4 and 5, proportional prioritization shows a significant improvement in the performance of the model. Prioritized Experience Replay reduces the time of computation and increases the robustness of the system.

7 Conclusions

This paper explores and compares experience replay and prioritized replay. We implemented our training on the Atari gameset and deduced that proportional prioritization of the replay cache increases the robustness and performance of the trained model as compared to uniform experience replay. The results from these experiments provide a very promising improve upon the foundation of Q-learning and reinforcement learning as a whole. Future work for this project includes applying prioritized learning to the class imbalanced MNIST data set and experimenting with how we can to change our prioritization scheme to account for the imbalance.

Acknowledgment and Contributions

The project heavily depends on the mathematical and computational concepts from deep neural networks. Attending Prof. Scott's and Prof. Hero's lecture on CNN and SVM was the foundation of implementing these concepts. Furthermore, the mathematical concepts taught in the initial phase of the Machine Learning course (Homework 01 and 02), helped a lot in grasping new concepts of reinforcement learning. The probabilistic approach became easier to understand because of our previous probability knowledge. Furthermore, the GSIs were extremely helpful in clearing all the doubts we had. With our understanding of machine learning techniques from class, we were able to comprehend a completely new, recent and a difficult topic with ease.

The division of workload for the project can be see below.
AUTHOR 1:
1.Researched about study material for deep q-networks and RL basics.
2.Implemented experience replay and its proportional prioritization on Atari environment.
3.Wrote the Abstract, Methodology(Deep Q-Learning), Evaluation, and Conclusion.

AUTHOR 2:
1.Researched about study material of uniform and prioritized experience replay.
2.Implemented experience replay on Atari environment.
3.Wrote the Introduction, Motivation, and Methodology(Experience Replay).

AUTHOR 3:
1.Researched about the papers published on prioritized experience replay.
2.Implemented experience replay on Atari environment.
3.Wrote the Methodology(Prioritized Experience Replay, and Experiments and Datasets), and Related Works.

AUTHOR 4:
1.Researched about the papers been published on experience replay and the datasets they used.
2.Implemented experience replay on Atari environment.
3.Wrote the Problem Statement, and Methodology(Introduction to Q-Learning), and References.

References

[1] D. S. A. G. I. A. D. W. Volodymyr Mnih, Koray Kavukeuoglu and M. Riedmiller, "Playing atari with deep reinforcement learning," 2014. [Online]. Available: https://arxiv.org/abs/1312.5602

[2] I. A. Tom Schaul, John Quan and D. Silver, "Prioritized experience replay," *International Conference on Learning Representations, ICLR*, 2016. [Online]. Available: https://arxiv.org/abs/1511.05952

[3] R. A. Y. B. H. L. M. R. W. D. William Fedus, Prajit Ramachandran, "Revisiting fundamentals of experience replay," *International Conference on Machine Learning, ICML*, 2020. [Online]. Available: https://arxiv.org/pdf/2007.06700.pdf

[4] "Prioritised experience replay in deep q learning," 2020. [Online]. Available: https://adventuresinmachinelearning.com/prioritised-experience-replay/

[5] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *MIT Press*, 1998.

[6] L.-J. Lin, "Reinforcement learning for robots using neural networks," *Technical report, DTIC Document*, 1992. [Online]. Available: https://www.proquest.com/docview/303995826?pq-origsite=gscholar&fromopenview=true

[7] D. B. G. B.-M. M. H. H. v. H. D. S. Dan Horgan, John Quan, "Distributed prioritized experience replay," *International Conference on Learning Representations, ICLR*, 2018. [Online]. Available: https://arxiv.org/pdf/1803.00933.pdf