ENPM667: Project - 1

A Technical Report on

Playing Atari with Deep Reinforcement Learning

(Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, DeepMind Technologies)

Shyamsundar Prabhakar Indra
UID: 119360842

Sai Surya Sriramoju
UID: 119224113

Date of Submission: 23rd November, 2022

A. James Clark School of Engineering (Robotics)
University of Maryland, College Park, MD - 20742

Table of Contents

List of Figures

Abstract

The purpose of this technical report is to introduce, explain and review the classic paper in Reinforcement Learning research - 'Playing Atari with Deep Reinforcement Learning' which was the first of its kind to combine a Deep Learning model with a Reinforcement Learning (RL) algorithm called the 'Q Learning' to learn control policies directly from the high dimensional sensory inputs. The paper goes on to explain how a simple two-layer Convolutional Neural Network (CNN) with a variant of Q-Learning can be trained with raw pixels as the input to get an output which is a value function estimating the rewards of the future. This model was applied to seven Atari 2600 games utilizing the Arcade Learning Environment which is a famous RL testing bed. The model performed better than any prior methods during that time in six games and even outperformed an expert human player in three of the games.

Chapter 1

Introduction

In the last few years, Reinforcement Learning (RL) has been on the rise and has been emerging as a powerful tool for solving complex decision-making problems in control theory. It is possible to train agents to directly respond to high-dimensional sensory inputs, such as speech and vision, which are some of the persistent challenges with RL. The majority of the most effective RL applications in these fields have conventionally depended on hand-crafted features and linear value functions or representations of policies. It is obvious that the performance of such systems significantly depends on the quality of feature representation.

Rapid developments in Deep Learning (DL) over the past 20 years, which can be attributed to the availability of an abundance of data, and great computational resources, have made it possible to extract high-level features from raw sensory data, opening up new possibilities in speech and computer vision. These techniques take advantage of both supervised and unsupervised learning techniques and use a variety of neural network architectures, including Convolutional Neural Networks (CNNs), Multilayer Perceptrons, Restricted Boltzmann Machines, and Recurrent Neural Networks (RNNs).

However, if one thinks about using DL to implement RL, they are bound to experience a number of challenges. First of all, a lot of hand-labeled training data has conventionally been necessary for the majority of current successful DL applications. On the other side, RL algorithms need to be able to pick up new information from a reward that is usually sparse, noisy and delayed. When contrasted with the direct correlation between inputs and targets observed in supervised learning, the latency between actions and the consequent rewards, which can be thousands of timesteps long, appears especially daunting. Another problem is that, in contrast to reinforcement learning, which often involves sequences of highly correlated states, most deep learning algorithms presume that data samples are independent.

The paper, at the time of its publication, came up with a novel approach of using a CNN to implement a Q function, which was optimally used to learn control policies directly from raw pixel data. The CNNs were trained with a variant of the Q-Learning algorithm with stochastic gradient descent(SGD) to update the weights. The paper employs a novel experience replay[13] technique that randomly samples earlier transitions and thereby smooths the training distribution over numerous previous behaviors to address the issues of correlated data and non-stationary distributions.

The paper uses the Arcade Learning Environment[3] (ALEselection)'s of Atari 2600 games to test the algorithm. The Atari 2600 is a demanding RL testbed that offers agents a high dimensional visual input (210x160 RGB video at 60Hz) as well as a varied and engaging collection of challenges that are intended to be challenging even for human players. The paper successfully creates a single neural network architecture which works across all the games without making changes in the neural network architectures and changing the parameters for each game. The network has not been provided with any information about the game and visual features from the game but only video input. The network outperformed the previous RL implementations during that time in six games and even outperformed humans in three games. This report intends to explain all the concepts and algorithms which made this groundbreaking paper a reality and also replicate the results of the same.

Chapter 2

Background

The paper was written with quite a lot of jargon, which would be rather difficult to comprehend for a person without any prior knowledge of RL, DL, or Computer Vision. Hence, in this chapter, we will be going through all the important underlying concepts and algorithms which form the backbone of the algorithm proposed in the paper, to enable the reader to understand the same in better detail.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is usually described as a form of a Machine Learning algorithm, or a highly non-linear control algorithm. We would instead like to describe it as the science of higher-level decision-making. Given an agent in an unknown environment, it can be summarized as the algorithm through which the agent learns the optimal behavior in an environment to obtain 'maximum reward'. This optimal behavior is learned through interactions with the environment and observations of how the environment responds in turn.

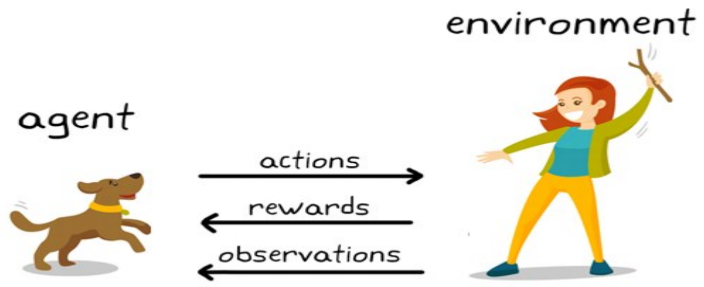


Figure 2.1: Basic flow in a RL algorithm[29]

This can be visualized as analogous to a dog being taken to a park for the first time, where the owner can be thought of as the 'environment', and the dog is the 'agent'. The owner plays fetch with the dog, which the dog is initially clueless about. Slowly it learns to perform the initial action of fetching the stick, & eventually learns to actually return it to the owner. During this phase, the owner keeps 'rewarding' the dog with a treat, which 'reinforces' this behavior. A simple flow of the same is shown in Fig 2.1

In the absence of a supervisor, the learner must independently discover the sequence of actions that maximize the reward. This discovery process is akin to a trial-and-error search. The quality of actions is measured by not just the immediate reward they return, but also the delayed reward they might fetch. As it can learn the actions that result in eventual success in an unseen environment without the help of a supervisor. This can be better illustrated with the example of a robot trying to navigate a completely unknown maze as shown in Fig 2.2. The robot initially starts by taking random actions and observing the rewards (moving towards the objective), and the penalties (moving into the red blocks which are walls). After some period of such arbitrary decisions, it learns more about the environment, after which it takes more informed decisions and observes higher rewards, eventually being able to successfully navigate the maze with the least number of steps (and highest possible rewards).

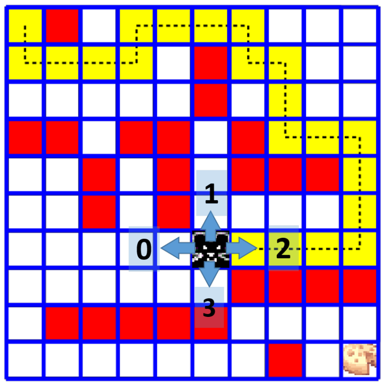


Figure 2.2: Maze navigating robot in an unknown maze [30]

There are multiple algorithms available to solve an RL problem which can be majorly classified into model-based ones like Imagination-Augmented Agents (I2A), and non-model-based ones like Q learning. We would not dive deeper into either in this section, however, we will revisit the same in Chapter 4 to illustrate how a form of Q-learning is cleverly leveraged to formulate the novel Deep RL algorithm in the paper.

2.2 Deep Learning

The next important component/tool that one should know to understand Deep RL is Deep Learning itself. Deep learning can be thought of as a subset of Machine learning. In simpler terms, it can be thought of as a fitting problem, not just any ordinary fitting problem, but non-linear fitting. It is a field that relies on studying computer algorithms to learn and advance on its own. Deep learning uses artificial neural networks, the underlying idea of which mimics how humans think and learn, whereas machine learning uses simpler principles. Up until recently, the complexity of neural networks was constrained by computational capacity. Larger, more complicated neural networks are now possible, thanks to developments in big data analytics, which enables computers to watch, learn and respond to complex events more quickly than people. Speech recognition, language translation, and image categorization have all benefited from deep learning. Any pattern recognition issue may be resolved with it without the need for human interaction.

2.2.1 Perceptron

A perceptron is smallest unit of a neural network. A perceptron is a linear classifier which takes inputs, along with assigned weights and gives an output. The basic diagram of the perceptron can be seen in Fig 2.3. In the case of the perceptron shown in the figure, the output $y = f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5)$, where f is a non-linear activation function.

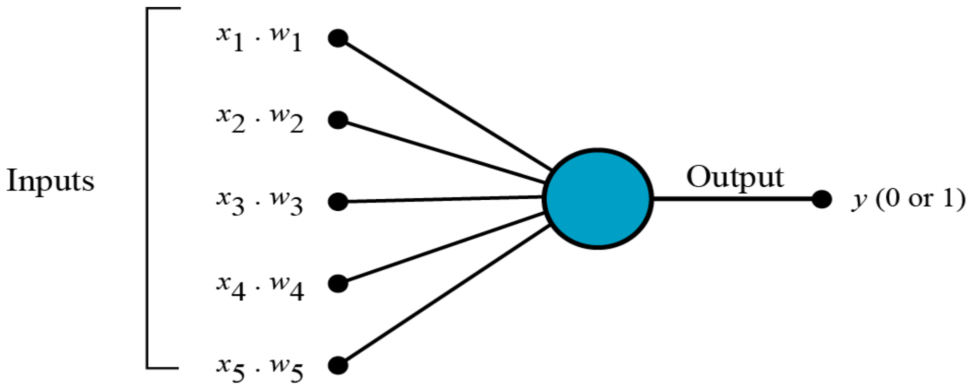


Figure 2.3: Perceptron[31]

2.2.2 Deep Neural Networks

A single perceptron can be considered as a function, whereas a deep neural network is a highly connected network of such functions. A deep neural network consists of more than two layers. Each layer consists of perceptrons and these perceptrons are connected to the perceptrons of the next layer. A DNN (Deep Neural Network) as shown in Fig 2.4 has an input layer, two hidden layers, and an output layer. It takes an input at a particular layer and the output of that particular layer is passed to the next layer as an input, and so on till the output layer. The first layer is called as input layer and the last layer is called as output layer and the layers present in between are addressed as hidden layers. The first Deep Learning Networks were modeled after how neurons in the human central nervous system are connected with one another.

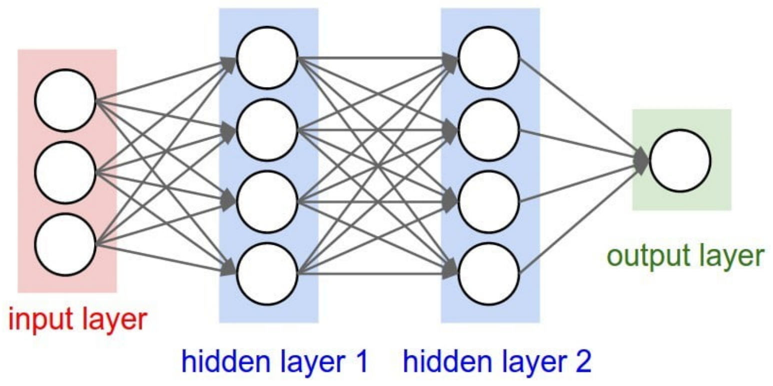


Figure 2.4: Deep Neural Network[32]

In the case of a simple classification problem, the number of neurons in the last layer corresponds to the number of classes we are training in the dataset. Let us take an example of detecting hand-written digits from 0-9 and detecting the number perceived in the image. The number of neurons in the last layer would be 10 as we are detecting digits from 0 to 9. Let us assume that the image of each digit is 28x28. During training, the number of neurons in the input layer corresponds to the number of pixels which is 784, which is a flattened version of an image. The image of the handwritten number is shown in Fig 2.5

In the example, we take a neural network architecture consisting of one input layer of 784 neurons, a hidden layer with 15 neurons, and an output layer corresponding to the number of classes as shown in Fig 2.6. Each

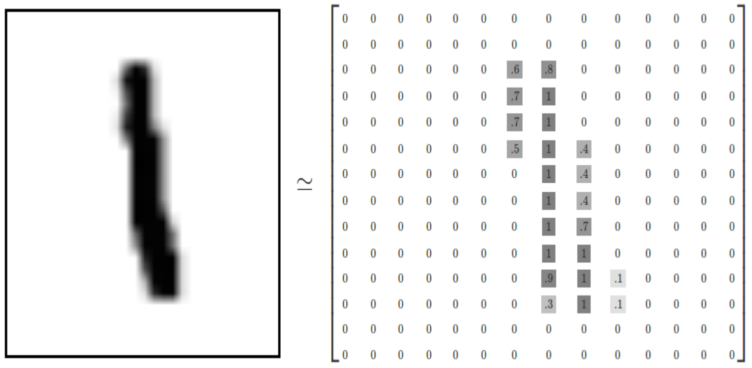


Figure 2.5: A handwritten digit represented using the pixels[36]

neuron in the neural network is initialized with random weights and biases. Each layer does a 'weighted addition' operation, the result of which is then sent into an activation function and passed as input to the next layer.

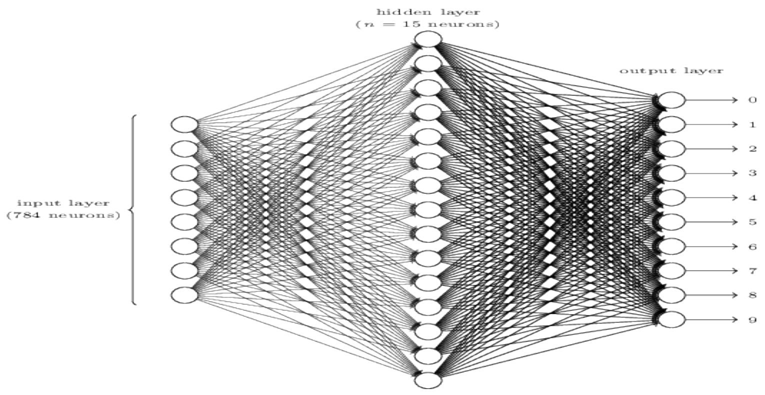


Figure 2.6: Neural Network Architecture for digit recognition[37]

2.2.3 Back Propagation in Neural Networks

As explained in the above section, the last layer is employed with a function that scales the output of each neuron in terms of probability. The function is called the softmax function. In each iteration, the result of these detections will be compared with ground truth and the difference is calculated, which is the error function. The derivative of this error function is found with respect to each and every weight in the network, and the error derivative is backpropagated from the last layer to consecutive previous layers, thereby tuning the weights of the network iteratively. Based on the difference the weights and bias assigned to each neuron are modified to maximize the accuracy using a cost function. This process keeps on repeating until desired accuracy is achieved.

2.2.4 Convolutional Neural Networks

A CNN (convolutional neural network) as shown in Fig 2.7 is a feed-forward neural network that is used to analyze images by processing data using grid-like topology. It is also called as ConvNet.

A CNN consists of a convolutional layer, a pooling layer, and a fully connected layer. The majority of computations happen in the convolutional layer which is the core building block of a CNN. A filter kernel moves across the convolution layer, and after each iteration, a dot product is calculated between the input pixels and the filter. The output is known as a feature map. This allows CNN to extract meaningful features. The pooling layer like the kernel sweeps over the input image, the pooling layer reduces the number of parameters in the input and also results in information loss (which can help in preventing overfitting), reduced complexity, and efficient CNN. The Fully connected layer is where the desired use case happens, here fully connected means all the units from one layer are connected to every node of the next layer.

The advantage of using CNNs when working with images over DNNs is that the operations involved in CNNs are computationally inexpensive, especially with the programming languages being developed around CNN-based architectures. CNN is indispensable for image-related tasks such as image recognition, object classification, and pattern recognition. CNN leverages principles from linear algebra, such as matrix multiplication, vector algebra, etc. The algorithm proposed in this paper makes use of a simple two-hidden-layer CNN to process raw sensory input from the RL environments.

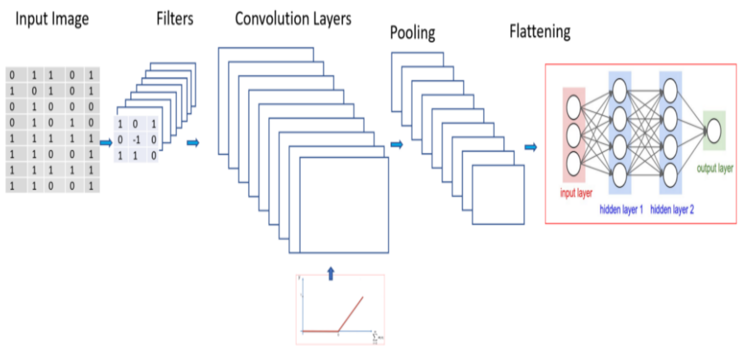


Figure 2.7: Convolutional Neural Network[38]

Chapter 3

Related Work

3.1 TD-Gammon

Gerald Tesauro created the computer backgammon game TD-Gammon in 1992 at IBM's Thomas J. Watson Research Center. Its name is derived from the fact that it is an artificial neural network trained using TD-lambda, a type of temporal-difference learning.

- TD-Gammon developed to a superhuman level of play[24] purely by reinforcement learning and self-play.
- Similar to Q-learning, TD-gammon uses a model-free reinforcement learning method with a multi-layer perceptron with a single hidden layer to approximate the value function[21].
- Applying the TD-Gammon on chess, Go and checkers were less successful and it led to the widespread belief that it only worked with backgammon.
- It was also demonstrated that the Q-network might diverge when model-free reinforcement learning algorithms, like Qlearning[15], were combined with non-linear function approximators[25] or even off-policy learning[1]. Consequently, linear function approximator [14] with improved convergence guarantees[25] were the main focus of most research in reinforcement learning.

3.2 Gradient Temporal Difference

An algorithm introduced in 2009 by Sutton et al. converges in the off-policy scenario.

- Restricted Boltzmann machines have been used to estimate the value function or the policy[9], whereas deep neural networks have been used to estimate the environment E.

- Gradient temporal-difference techniques have partially addressed the divergence problems with Q-learning.
- When evaluating a fixed policy with a nonlinear function approximator or learning a control policy with a linear function approximation using a restricted variant of Q-learning, these methods have been shown to converge.
- These methods have not yet been extended to nonlinear control.

3.3 Neural Fitted Q-learning (NFQ)

With observation (s) and action (a) as its inputs and Q(s, a) as its output, Neural Fitted Q-Iteration uses a deep neural network to represent a Q-network. The batch offline updates by gathering experience during the episode and updating with that batch, as opposed to online Q-learning.

- NFQ[20] optimizes the sequence of loss functions, using the RPROP algorithm to update the parameters of the Q-network.
- In contrast to stochastic gradient updates, which have a low constant cost per iteration and scale to large data sets, it employs batch updates, which have computational costs per iteration that are proportional to the size of the data collection.
- NFQ has also been successfully applied to simple real-world control tasks using purely visual input, by first using deep autoencoders to learn a low dimensional representation of the task, and then applying NFQ to the representation[12].
- The current approach applies reinforcement learning end-to-end, directly from the visual inputs; as a result, it may learn features that are directly relevant to discriminating action-values representation of the parameters of the Q-network.

Chapter 4

Atari Games Environment

It is important that we can visualize and get to know about the RL test bed environments on which the paper makes the RL agent play and learn, to actually understand and learn how the paper's algorithm achieves what it does. This chapter intends to give a brief overview of the seven games played by the RL agent in the paper:

4.1 Beamrider

Beam Rider is a 3D arcade action game. The spaceship is controlled to clear hostile aliens in space. The spaceship is located at the bottom of the screen and it can stop only on one of five beams. The spaceship is armed with lasers and torpedoes. The screenshot of the Beamrider gameplay can be seen in Fig 4.1

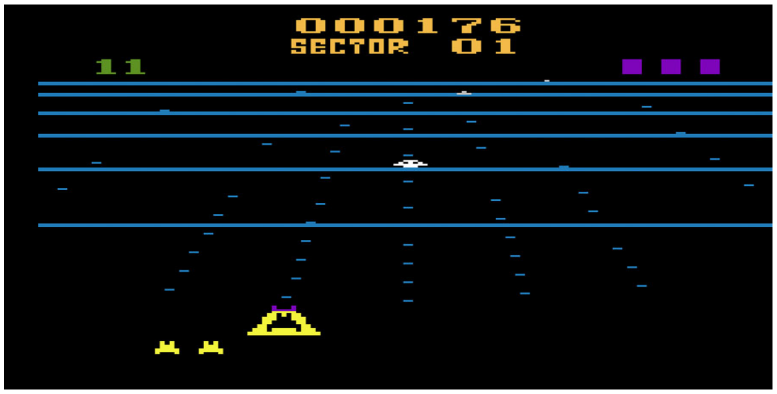


Figure 4.1: Beamrider Game[39]

The player's objective is to clear the Shield's 99 sectors of alien craft while piloting the Beamrider ship. The Beamrider is equipped with a short-range laser lariat and a limited supply of torpedoes. The player is given three at the start of each sector. To clear a sector, fifteen enemy ships must be destroyed.

4.2 Breakout

The breakout game consists of 8 layers of blocks, a ball, and a paddle. The paddle is considered as the agent and is used to bat the ball at the blocks. The moment the ball touches the block, the block disappears and a score is given. When the ball misses the paddle, a life is lost. There are five lives in total. The screenshot of the gameplay can be seen in Fig 4.2

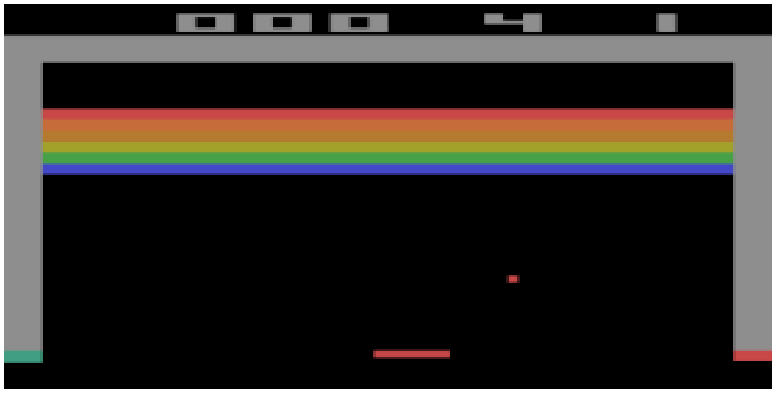


Figure 4.2: Breakout Game[40]

The main objective of the game is to bat the ball at the colored blocks until there are none left while making sure the ball does not pass your paddle.

4.3 Enduro

Enduro is a racing video game. The objective of the game is to complete an endurance race. The game consists of a lane and the car needs to be driven in the lane. The car is considered the agent in the game. The car has to avoid the other cars from crashing. The screenshot of the gameplay can be seen in Fig 4.3.

4.4 Pong

Pong is a game that simulates table tennis. The player controls the paddle by moving the joystick vertically across the left and right sides of the screen. Two paddles are present and two players can play the game. The players use the paddle to hit the ball back and forth. The goal of the game is to reach eleven points before the opponent. Points are earned when the opponent

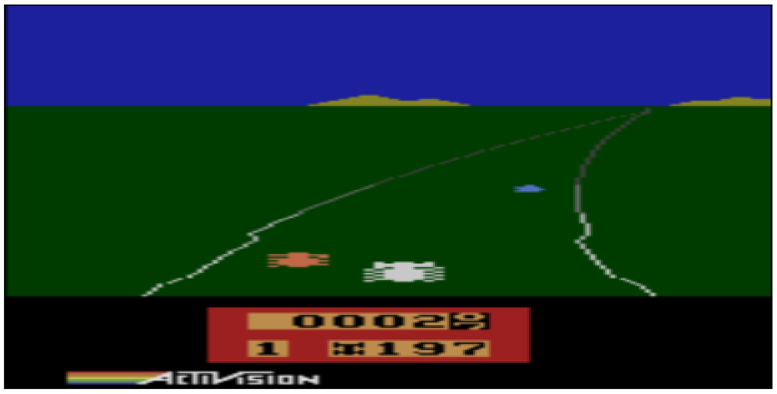


Figure 4.3: Enduro Game[41]

fails to return the ball. The screenshot of the gameplay can be seen in Fig 4.4.

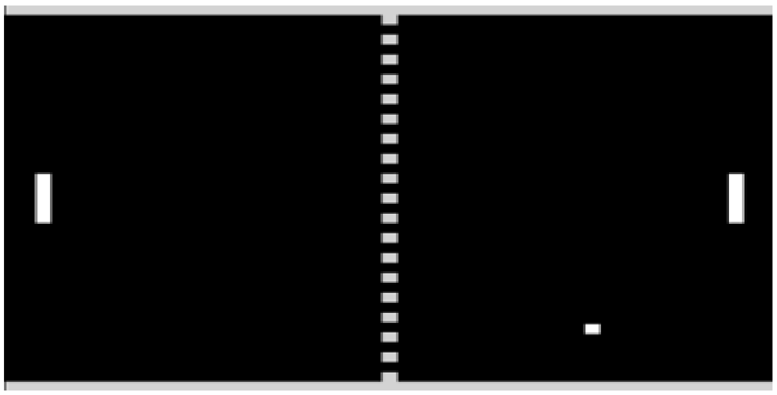


Figure 4.4: Pong Game[42]

4.5 Q*Bert

Q*Bert is an arcade game with puzzle elements played from a third-person perspective to convey a third-dimensional look. The character in the game is called a Q*Bert and the character starts at the top of the cube initially, having a total of 28 cubes. Q*bert has to jump on each cube and the color of the cube changes. In the later stages after advancing, the cubes need to be hit twice. The screenshot of the gameplay can be seen in Fig 4.5.

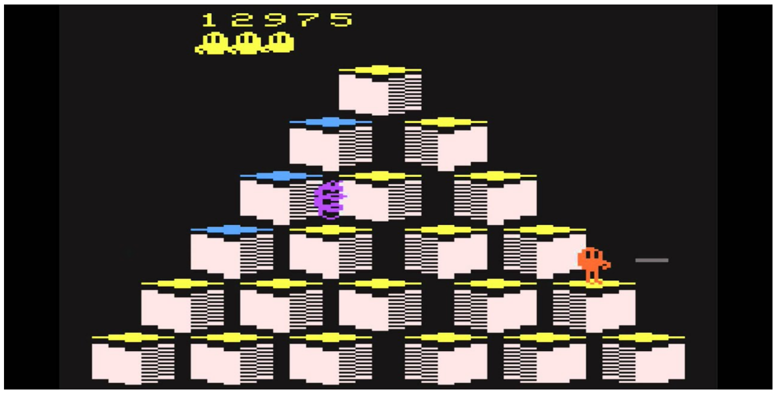


Figure 4.5: Q*Bert Game[35]

4.6 Seaquest

Seaquest is an underwater shooter game, the player controls the submarine which is also an agent. The player uses the submarine to shoot the enemies and rescue the divers. The enemies are sharks and submarines which shoot missiles at the player. The player must shoot the enemies with an unlimited supply of missiles and save the divers. The screenshot of the gameplay can be seen in Fig 4.6.

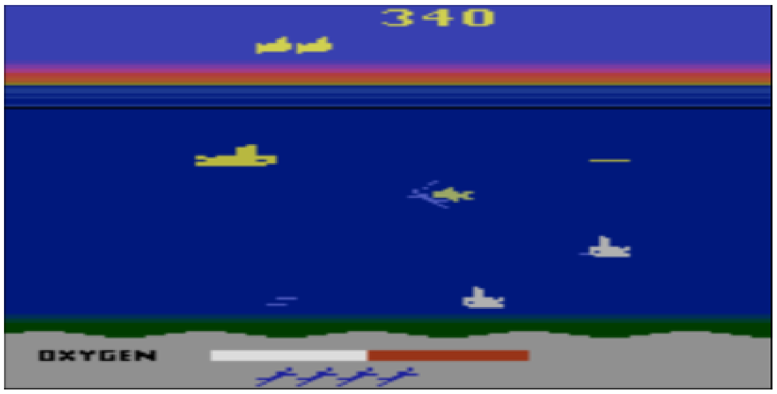


Figure 4.6: Seaquest Game[34]

4.7 Space Invaders

Space invaders is a fixed shooter game in which the player moves a laser cannon horizontally at the bottom of the screen and fires at the aliens on the top. The aliens are in five rows and they move right and left as a group, moving down each time they reach the screen edge. The main objective of the game is to shoot down the aliens and the player loses once the aliens reach the bottom of the screen. The screenshot of the gameplay can be seen in Fig 4.7.

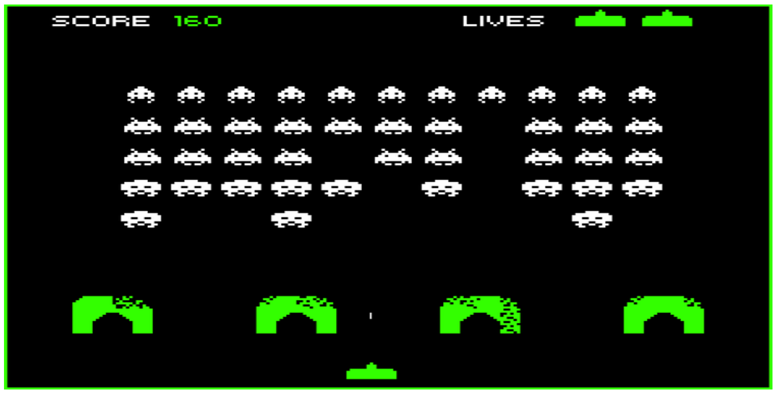


Figure 4.7: Space Invaders Game[33]

Chapter 5

Deep Reinforcement Learning

We move on to explain how the actual DRL algorithm was Mathematically and logically formulated, and how a DL network architecture was built around it. To understand it in better detail, we are dividing it into two further subtopics, the first deals with only the Mathematical aspect of the algorithm, and the second deals with how DL networks directly correspond to the Mathematical formulation, rather than thinking of it as a black box.

5.1 Q-Learning

Q-Learning is a model-free Reinforcement learning policy that will find the next best action, given the current state of the agent in the environment. It chooses this action at random and aims to maximize the reward. Before we dive deep into the formulation itself, we would like to introduce the reader to specific terms corresponding to the paper's implementation itself:

- **Environment:** It is the fundamental unit of an RL testing bed. For an action 'a' at given a state s_t, the environment returns a reward 'r' and the next state s_{t+1} upon taking the action. Here it is one of the ALE Selection's Atari 2600 games
- **Action:** It is the input that an RL environment takes at state s_t to give s_{t+1}. In this case, it is one of the joystick inputs that the game can take, which can vary from game to game. For eg., moving the striker in Breakout using a 'left' button is an action.
- **State:** At any given time step 't', state refers to the setup of the environment at that time. For example, in the case of Breakout, state refers to where the striker is present, and the location of all the remaining bricks to be broken at a specific time instant. In the case of Atari games, the state is extracted as a 210x160 RGB image showing the environment, at every time step.

- **Goal:** The ideal terminal state s_n that an agent achieves. In the case of Atari games, some have a goal, and some don't. For eg., In the case of Breakout, the goal is breaking all the bricks on the screen, whereas in a game like Seaquest, there's no terminal goal per se, but it's all about getting the maximum reward.

- **Episode:** One iteration of playing the game completely in the given environment. The termination of the game could be either due to achieving the goal or due to taking an action that heavily penalizes and ends the game, for eg., The striker missing the ball in the Breakout game.

- **Policy:** Given a state s_t, policy π refers to the set of actions $[a_t, a_{t+1}, ..., a_n]$ that the agent plans to take. In the case of Atari games, it is the series of button inputs that will be the input to the environment starting from state s_t

- **Value function:** mapping from states to real numbers, where the value of a state represents the long-term reward achieved starting from that state and executing a particular policy.

- **Markov decision process (MDP):** A probabilistic model of a sequential decision problem, where states can be perceived exactly, and the current state and action selected determine a probability distribution on future states. Essentially, the outcome of applying an action to a state depends only on the current action and state (and not on preceding actions or states).

- **Reward:** Given a state s_t, and an action 'a' to the state, the reward is the output from the environment which is either a positive score that encourages the action or a negative score that penalizes the action. For eg., in the case of seaquest, the agent is positively rewarded if it successfully shoots down an enemy and penalized if it runs out of oxygen or collides with an enemy.

The paper formulates an RL agent that interacts with the emulated Atari environment $\mathcal{E}$, in a set of actions, states, and rewards. At every time step, the agent chooses one out of the allowed action set corresponding to the specific game, A = 1, . . . , K. The action is then taken as an input by the emulator which modifies the state of E from s_t to s_{t+1} and returns a reward R. As explained previously, the states that we receive as output upon taking an action from the emulator are in the form of RGB image frames.

In general, the reward depends on the entire sequence of prior actions and states, hence the actual feedback about an action may only be received after many thousand states. For example, in the case of the Pong game, any action taken by the RL agent, will not get an immediate reward or penalty, but a set of such actions will either result in a positive reward (winning the point), or a negative penalty (losing the point) only thousands of time steps later.

Since, it can be clearly seen that if the agent is only able to observe the state s_t and not any of the previous states which led to the current state, the task is only partially observed, i.e., in other words, multiple emulator states are perceptually aliased, and it is impossible for the agent to fully understand the current situation only from its current state. The algorithm deals with this setback by considering a sequence of actions and states, $S_t = s_1, a_1, s_2, ..., a_{t1}, s_t$, and learning game strategies depending on these sequences. It is also assumed that all such sequences terminate in a finite number of time steps. This assumption gives rise to a large but finite Markov Decision Process, in which each sequence is distinct, and now these sequences are considered to be the 'states', i.e., S_t as defined above, becomes the new state, instead of individual observations of the environment. The goal of the agent is to interact with the environment by selecting actions in a way that maximizes future rewards. The paper makes the standard assumption that future rewards are discounted by a factor of γ per time step and defines the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. The usage of γ is an intuitive RL practice since it tells the agent that it is important to give a higher priority to maximizing the rewards in the near future over the rewards in the distant future. From a mathematical point of view, this is done to ensure that it is possible to compute the rewards summation even if the sequence goes on to infinity since the rewards would then form a 'convergent' geometric series.

Now we move on to how the actual Q learning algorithm has been formulated in the paper. Q function is an action-value function that gives cumulative future rewards upon following a policy π after observing a state (sequence) s and taking an action 'a' at that state. In a better representative manner:

$$Q(s,a) = E_\pi[R_t | s_t = s, a_t = a, \pi]$$

Where E is the expected value of the cumulative reward R_t starting from time t till the end of the episode T when a policy π is followed. Now we define an 'optimal Q function' - Q^* as the Q function which gives the maximum possible reward by following the optimal policy π^*, after taking an action a_i

where $i \in A$, the action space of a specific game, after s_t.

$$Q^*(s,a) = max_{\pi^*} E[R_t | s_t = s, a_t = a_i, \pi^*]$$

The optimum Q function obeys an important identity called the Bellman equation, the details of which aren't in the scope of this paper. It enables to formulate the optimum Q function in the form of a recursive equation. This is based on the intuition that the optimal Q function at the next state $s_t + 1$ is given by $Q^*(s_{t+1}, a_{t+1})$. With this idea Q^*, after a simple manipulation can be expressed as an iterative series:

$$Q^*(s_t, a_t) = E_{s_{t+1}}[r + \gamma max_{a_t+1} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t]$$

In simpler words it can be demonstrated as the optimum Q function output at a state s_t upon taking an action a_t can be written as a sum of the reward r as a result of action a_t and the optimum Q value at the state s_{t+1}. Such value iteration algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^*$ as i $\rightarrow \infty$. This approach is however very basic and is totally impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is a common practice to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. The paper uses a non-linear neural network function approximator to achieve the same. Such a non-linear function approximator neural network with weights θ is referred to as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i:

$$L_i(\theta_i) = E_{s_t, a_t}[y_i - Q(s_t, a_t, \theta_i)^2]$$

where $y_i = E_{s'}[r + \gamma max_{a'} Q(s', a'; \theta_{i-1})]$. In an ideal case where Q is a perfect action-value estimation function both these values would be equal and the loss would be zero. However, that's not the case since we learn the Q network, and we make use of this loss function to do the same. The parameters from the previous iteration θ_{i-1} are kept fixed when optimizing the loss function $L_i(\theta_i)$. One should note that in contrast to the targets used for supervised learning which are fixed before the start of learning, here the targets depend on the network weights themselves. If the reader is not very clear with the mathematical formulation of Q learning, do not worry, things will become much clearer when looked at from a DL perspective. (Yes, ironically DL which is supposed to be complicated in nature can actually simplify things). An important thing that one should make note of is that the Q learning algorithm is **model-free**, i.e., it directly solves the RL task directly by using the samples from the environment rather than trying to explicitly model the environment itself. Yet another important property is that it is **off-policy**, which means that it tries to learn the greedy strategy $a = max_a Q(s, a; \theta)$ while following an ϵ behavior distribution (this will be discussed in detail in the ϵ greedy strategy explained in the next section) which also ensures proper exploration of the environment.

5.2 Deep Q learning

The previously discussed TD-Gammon's architecture proved to be the starting point for the approach discussed in the paper. The architecture updates the parameters of a network that estimates the value function, directly from the on-policy samples of experience drawn from the agent's interactions with the environment. However in contrast to TD-Gammon and other similar online approaches, the paper utilizes a technique called the experience replay, where the agent's previous experiences at each time-step get stored, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D = e_1, e_2, e_3..., e_n$, across different episodes into a replay memory.

The CNN architecture that will be discussed in detail is essentially the Q network that takes in a state as an input and outputs the Q value for every action possible at that state. It tries to learn and eventually become the optimal Q function Q^*. After the experience replay data set is filled with initial arbitrary actions, states, and rewards, the agent selects and runs a state s_i from D according to ϵ-greedy policy. **ϵ-greedy policy:** The agent generates a random number n between 0 and 1 and decides whether to infer over the state-action pair chosen from the experience replay or take a random action a_r in the given state, according to whether n is greater or lesser than ϵ respectively. ϵ can be a fixed number between 0 and 1 or it can be a variable number (a linearly decaying ϵ value used in the paper's algorithm). In the latter case, the Q network takes the state s_i from the experience replay as input and outputs the Q value for every possible action. It picks the Q value for the action corresponding to state s_i from the experience replay and executes the action to receive a reward r. Let's call the Q value for this action q_1. The Q network also runs with s_{i+1} as the input, to find all the Q values for the action space, and chooses the best Q value in this action space, and let's call this q_2. Ideally, a perfect Q network would have $q_1 = q_2 + r$, but this won't be the case since we are trying to make the Q network learn and approximate to the closes possible Q function. Hence, the difference between these two is the error $e = q_1 - (q_2 + r)$. The network uses this error function to backpropagate the errors iteratively, and optimally update the weights of the network.

This approach has several advantages over standard online Q-learning[23]. Firstly the Q network gets a varied range of data which is always important for a DL network not to overfit. Secondly, learning directly from consecutive samples is very inefficient due to the strong correlations between the samples. Randomizing these samples in the form of an experience replay breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy, the current parameters determine the next data sample on which the parameters are trained. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. Experience replay also ensures that the behavior distribution is averaged over many of its previous states, smoothing out the learning. In practice, the paper's algorithm stores only the last N experience tuples in the replay memory and samples at random from dataset D when performing updates. One of the possible limitations of this approach is that the algorithm overwrites the previous experience with the recent ones, and doesn't differentiate between important transitions. A more prioritized sampling strategy might be to emphasize transitions from which the most learning is possible, which is not in the scope of this paper.

Chapter 6

Preprocessing and Model Architecture

The paper applies a preprocessing step on the Atari frames which are 210 $\times$ 160 pixel images with a 128 color palette, can be computationally demanding, instead of working with them directly, every single raw frame is preprocessed by first converting to grey-scale format and down-sampling it to a 110$\times$84 image. The final input representation is obtained by cropping an 84 $\times$ 84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions, which expects square inputs. For the experiments in this paper, the function ϕ from Algorithm 1 as shown in Fig.6.1 applies this preprocessing to the last 4 frames of history and stacks them to produce the input to the Q-network.

There are several possible ways of parameterizing the Q function using a neural network. Since Q maps history action pairs to scalar estimates of their Q value, the history, and the action have been used as inputs to the neural network by some previous approaches. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. The paper instead uses an architecture that takes only a state as the input and has a separate output unit for each possible action. The outputs correspond to the predicted Q-values of the individual action for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. A gist of the algorithm of the DRL model is shown in figure 6.1

The exact architecture used for all seven Atari games is as follows: The input to the neural network is 84 $\times$ 84 $\times$ 4 images produced by preprocessing. The first hidden layer convolves 16 8 $\times$ 8 filters with stride 4 with the input image and applies a rectifier non-linearity[10, 18]. The second hidden layer convolves 32 4 $\times$ 4 filters with stride 2, again followed by a rectifier non-linearity. The final hidden layer is fully-connected and consists of 256

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 6.1: Deep Q Learning Algorithm in a nutshell

rectifier units. The output layer is a fully connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the 7 games played by the paper's model. The paper refers to a CNN model trained to be a Q function as a Deep Q-Network (DQN)

Chapter 7

Experiments

As mentioned before, the paper managed to test the DQN on seven of the popular ATARI games – Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders. The same network architecture, learning algorithm, and hyperparameters settings were used across all seven games, which shows its robustness to adapt to different environments without any tuning. The evaluation of the agents was done on real and unmodified games. However, during the training phase, the rewards were clipped to [-1,0,1]. i.e., all positive rewards were put to 1, 0 rewards unchanged, and negative rewards were put to -1. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. However, at the same time, it could affect the performance of the DQN agent since it cannot differentiate between rewards of different magnitudes. RMSProp algorithm (Backpropagation using Root Mean Squared Error for optimization) with mini-batches of size 32 was used for training the DQN using backpropagation. The behavior policy during training was ϵ-greedy with ϵ decreasing linearly from 1 to 0.1 over the first million frames and fixed at 0.1 thereafter. The paper trained the agent for a total of 10 million frames and used a replay memory of one million most recent frames. Following previous approaches to playing Atari games, the paper also used a simple frame-skipping technique[3]. More precisely, the agent sees and selects actions on every kth frame instead of every frame, and its last action is repeated on skipped frames. Since running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly k times more games without significantly increasing the runtime. The paper used k = 4 for all games except Space Invaders since using k = 4 made the lasers invisible because of the period at which they blink. Hence k = 3 was used to make the lasers visible and this change was the only difference in hyperparameter values between any of the games.

We tried to replicate the same DQN algorithm with the same architecture and were successful to a good extent. However, owing to time and computational constraints, we were able to train the model for only 3 of the games, namely Breakout, Seaquest, and Space Invaders. We trained for 10 million time steps, the same as the paper's model, however, we were able to train the models with an experience replay memory of only 50000 frames as opposed to the paper's million frames, since it posed to be a severe bottleneck on the GPU. The models were trained on an Nvidia 1060 GPU and the training for each model almost took 1.5 days.

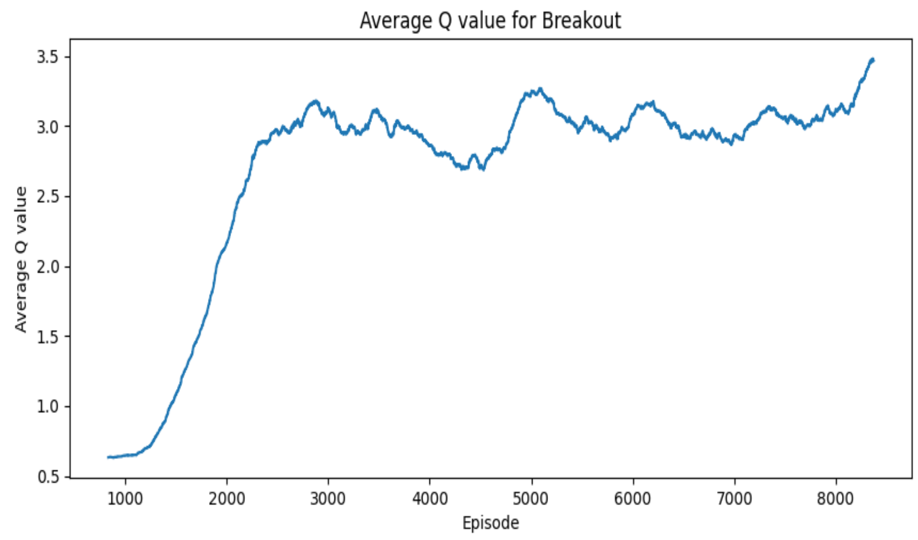
Chapter 8

Training and Stability

By comparing it to the training and validation sets, one may quickly monitor a model's progress during training in supervised learning. However, it might be difficult to appropriately assess an agent's training progress in DQN. The paper periodically computes the average reward throughout the training. Since even tiny changes in the network weights can result in significant changes in the distribution of the states, the average total reward metric frequently exhibits high levels of noise.

The bottom graphs of Fig. 8.1, 8.2, and 8.3 represent the average rewards per episode during training. Because of the noise in the averaged reward graphs, it can make one think that the learning algorithm is not progressing steadily. The policy's estimated action-value Q function, which gives an estimate of the amount of discounted reward the agent can receive by adhering to its policy from any given state, is another, more reliable statistic. Prior to training, a random strategy was executed to generate a fixed set of states, and then the average of the maximum predicted Q for these states was plotted. As one can see in the plots in Fig.8.1, 8.2, and 8.3, the average maximum Q value (as shown in the upper graphs) climbs much more smoothly than the average total reward received by the agent, which is very similar to the paper's graphs.

However one can also see that the average rewards are quite different from the graphs in the paper. This is because the paper plots the unclipped average rewards obtained whereas our implementation plots the clipped rewards, which is why the average rewards plotted during the training phase are lower. It can be seen in the next chapter, how close our results actually are in comparison to the paper's implementation when talking about the average and maximum rewards during inference. One more difference is that the x-axis in graphs with respect to the paper's implementation is the number of epochs, whereas in our implementation it is the number of episodes (with varying numbers of steps per episode). But either way, the total number of steps taken in both cases = 10 million.



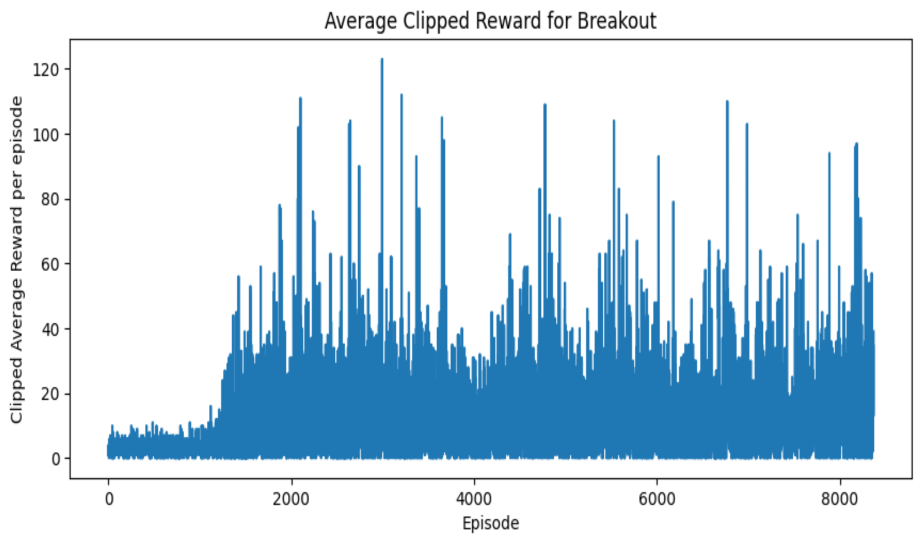
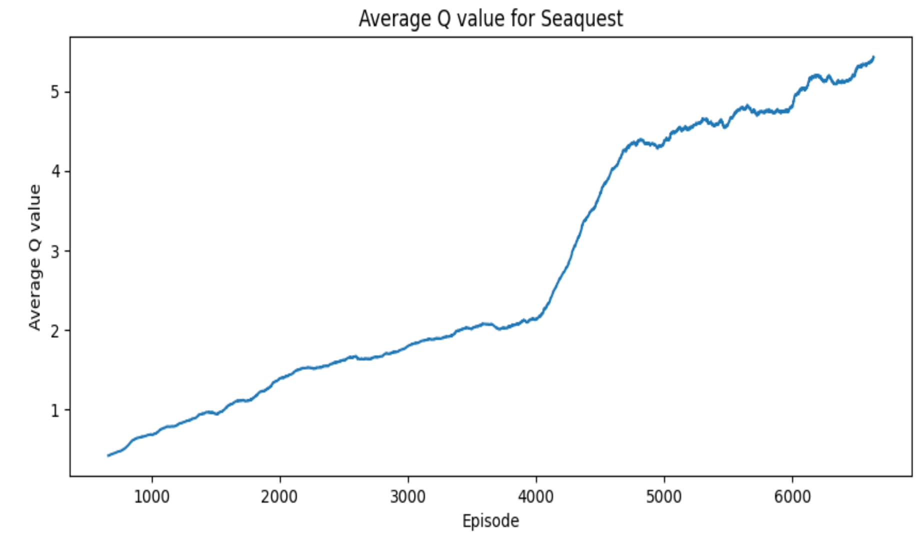


Figure 8.1: Breakout training statistics



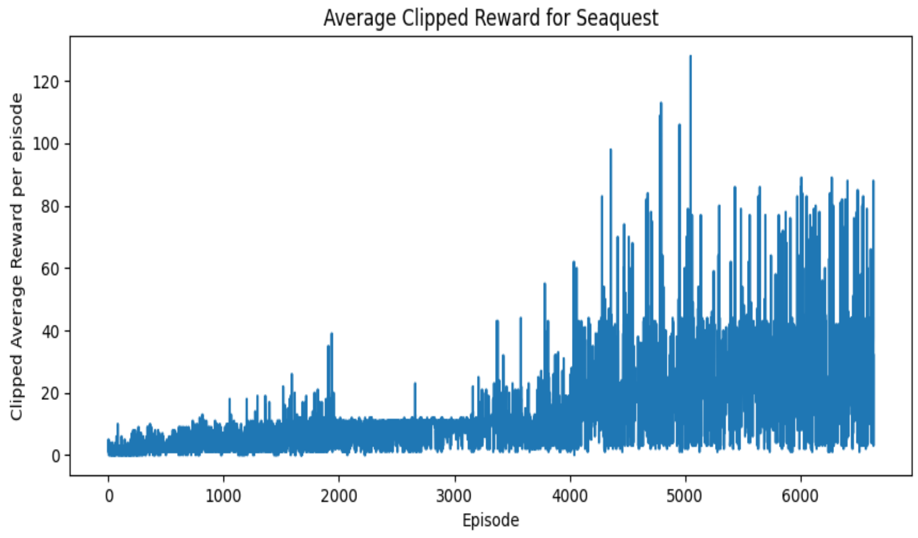
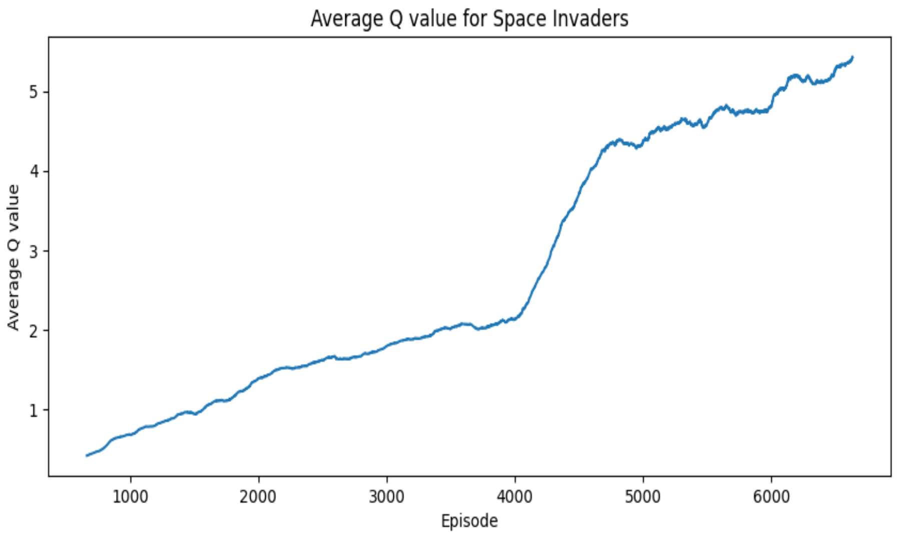


Figure 8.2: Seaquest training statistics



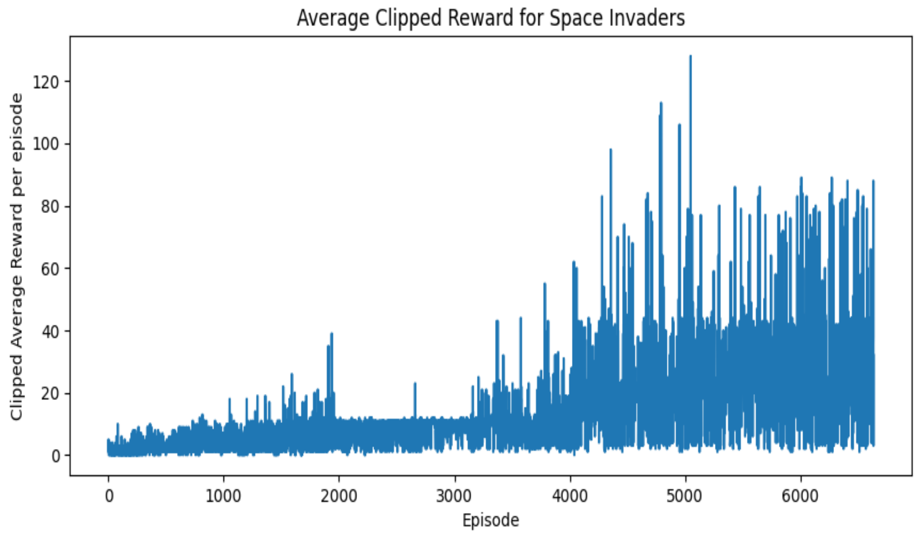
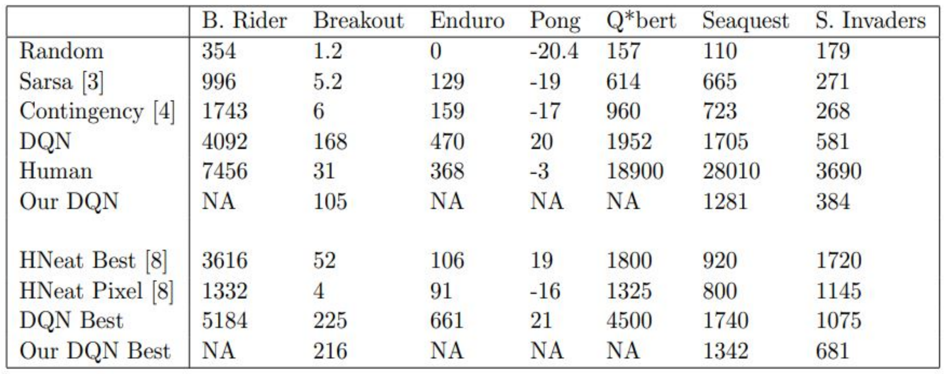


Figure 8.3: Space Invader training statistics

Chapter 9

Evaluation and Inferences



	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
Sarsa [3]	996	5.2	129	-19	614	665	271
Contingency [4]	1743	6	159	-17	960	723	268
DQN	4092	168	470	20	1952	1705	581
Human	7456	31	368	-3	18900	28010	3690
Our DQN	NA	105	NA	NA	NA	1281	384
HNeat Best [8]	3616	52	106	19	1800	920	1720
HNeat Pixel [8]	1332	4	91	-16	1325	800	1145
DQN Best	5184	225	661	21	4500	1740	1075
Our DQN Best	NA	216	NA	NA	NA	1342	681

Figure 9.1: The upper table compares the average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports the results of the single best-performing episode for HNeat and DQN, and our own implementation of the DQN algorithm with an experience replay size of 50000 frames

The results of DQN were compared with the best-performing methods from the RL literature [3, 4] during that time. The method labeled Sarsa used the Sarsa algorithm to learn linear policies on several different feature sets hand engineered for the Atari task and the paper reports the best score from the Sarsa algorithm[3]. Contingency used the same basic approach as Sarsa but augmented the feature sets with a learned representation of the parts of the screen that are under the agent's control[4]. Both of these methods incorporate significant prior knowledge about the visual problem by using background subtraction and treating each of the 128 colors as a separate channel. Since many of the Atari games use one distinct color for each type of object, treating each color as a separate channel can be similar to producing a separate binary map encoding the presence of each object type. In contrast, DQN only receives the raw RGB screenshots as input and learns to detect objects on its own. In addition to the learned agents, scores for an expert human game player were also reported. Human performance is the median reward achieved after around two hours of playing each game. Note that the reported human scores are much higher than the ones in [3]. For the learned methods, the paper follows the evaluation strategy used in [3, 5] and report the average score obtained by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps.

The first six rows of table 9.1 show the per-game average scores on all games. The paper's approach (labeled DQN) outperforms the other learning methods by a substantial margin on all seven games despite incorporating almost no prior knowledge about the inputs. A comparison was made to the evolutionary policy search approach from [8] in the last three rows of table 1. The HNeat Best score reflects the results obtained by using a hand-engineered object detection algorithm that outputs the locations and types of objects on the Atari screen. The HNeat Pixel score is obtained by using the special 8-color channel representation of the Atari emulator that represents an object label map at each channel. This method relies heavily on finding a deterministic sequence of states that represents a successful exploit. Such learned strategies pertaining to certain cases will not be able to generalize to random perturbations; therefore the algorithm was only evaluated on the highest-scoring single episode. In contrast, DQN was evaluated on ϵ-greedy control sequences, and can therefore generalize across a wide variety of possible situations. The paper goes on to show that in spite of not using any specialized feature inputs in any of the games, except Space Invaders, the max evaluation results (row 9), as well as the average results (row 4) achieve better performance. Finally, it is remarkable that DQN achieves better performance than an expert human player on Breakout, Enduro, and Pong and it achieves close to human performance on Beam Rider. The games Q*bert, Seaquest, and Space Invaders, on which the DQN implementation is far behind from human performance, are more challenging because they require the network to find a strategy that extends over long time scales.

Rows 6 and 10 show the average and maximum rewards respectively, from our implementation of the DQN algorithm across three games - Breakout, Seaquest, and Space Invaders. In spite of putting up good scores when compared to the best RL algorithms of that time, it isn't as good as the paper's implementation of DQN. The only difference between our implementation and the paper's implementation is the much smaller experience replay size, which is 1 million frames vs 50000 frames. This speaks volumes about how the experience replay formulation was the backbone of the algorithm's excellent results when compared to other best RL implementations of its time, in spite of using a very shallow network of just two hidden layers, when compared to 50+ layered networks in 2022. This also supports the reason behind the agent not performing in games like Q*bert, Seaquest, and Space Invaders, as well as a human player, since it was bottlenecked by experience replay size since the network had to find strategies spanning long time scales. It is also quite fascinating to observe how in spite of not being able to consistently perform as well as the paper's DQN implementation (lower average reward), our trained DQN agent was able to explore and find the most optimum policy to play Breakout, by making a tunnel in the bricks as shown in Fig.9.2, which helps it in achieving a max reward per episode closer to the paper's DQN implementation. This can actually be attributed to the ϵ-greedy policy, which allowed the exploration of the agent and enabled it to find quirky strategies like tunneling.

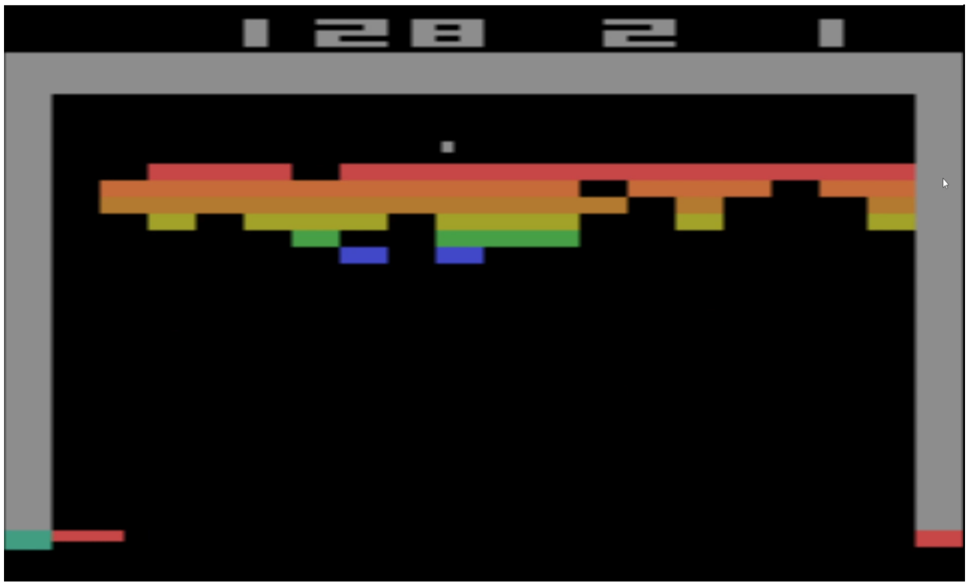


Figure 9.2: Our DQN agent manages to find the most optimal policy of making a tunnel in the bricks to earn the most rewards

One more interesting observation that we would bring to light to the reader is how the DQN agent learned in the Seaquest game to actually shoot down fish to earn rewards and come to the surface to replenish oxygen as shown in Fig 9.3 so that it can extend the duration of the episode and potentially earn more rewards. However owing to the smaller experience replay memory, it ends up being close to clueless after replenishing oxygen and ends up crashing into an enemy.

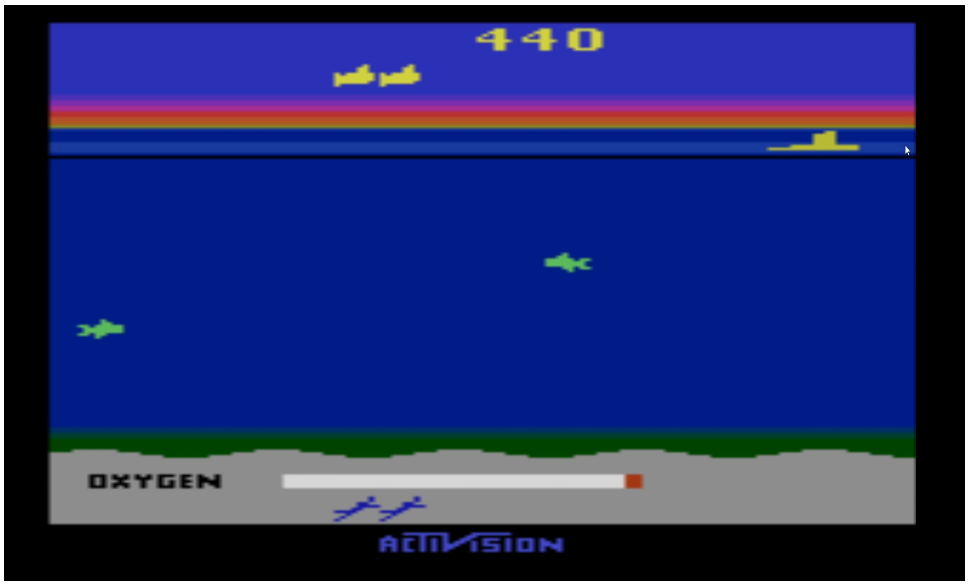


Figure 9.3: Our DQN agent manages to learn to go to the surface to replenish oxygen, extending the episode and potentially earning more rewards.

Chapter 10

Discussion and Future Work

The paper was a groundbreaking research material during its time of publication in 2013, which unanimously took over the other RL implementations of its time which needed much more involved handcrafted features to solve RL problems. It came up with the novel idea of combining a simple 2-hidden-layer CNN with a Q-learning algorithm to solve complicated RL problems, along with using an experience replay buffer which proved to be the backbone of its success. Now we are here in 2022, and yet the field of RL and its problems, unlike many Supervised Learning problems are far from being solved. After DeepMind's paper in 2013, which set the pace for RL research in the coming years, there have been several implementations of Deep RL, building up the concepts introduced in the 2013 paper.

In 2015 DeepMind [27] came up with a follow-up paper that leverages a parallel DQN to overcome over-estimations that weren't a known issue in 2013. More recent RL research has been in the direction of Policy Learning algorithms, which, unlike Q learning, without learning a function that could output Q values for every action available given a state s, directly learns a network that outputs the optimal action to be taken directly instead of its corresponding Q value as shown in [28]. Along with these prominent approaches, there are several other methods coming up like Meta RL, Unsupervised RL, etc.

Reinforcement Learning problems, as mentioned before, are far from being solved and the plethora of sophisticated formulations which come up these days are quite overwhelming to put it simply. We believe that RL has a huge role to play in the future, starting from simple household robots which can autonomously navigate a house to a complicated autonomous driving vehicle that will revolutionize the way the world works. RL according to us, is the truest form of Artificial Intelligence, which can provide end-to-end autonomy, and the prospects of the same in the future make us very excited to look forward to it. We are very glad that we got the opportunity to explain, review and implement the classic DeepMind RL paper from 2013 which is at the forefront of all future research in RL.

Bibliography

[1] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In Proceedings of the 12th International Conference on Machine Learning (ICML 1995), pages 30–37. Morgan Kaufmann, 1995.

[2] Marc Bellemare, Joel Veness, and Michael Bowling. Sketch-based linear value function approximation. In Advances in Neural Information Processing Systems 25, pages 2222–2230, 2012.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47:253–279, 2013.

[4] Marc G Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In AAAI, 2012.

[5] Marc G. Bellemare, Joel Veness, and Michael Bowling. Bayesian learning of recursively factored environments. In Proceedings of the Thirtieth International Conference on Machine Learning (ICML 2013), pages 1211–1219, 2013.

[6] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. Audio, Speech, and Language Processing, IEEE Transactions on, 20(1):30 –42, January 2012.

[7] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In Proc. ICASSP, 2013.

[8] Matthew Hausknecht, Risto Miikkulainen, and Peter Stone. A neuro-evolution approach to general atari game playing. 2013.

[9] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In European Workshop on Reinforcement Learning, page 43, 2012.

[10] Kevin Jarrett, Koray Kavukcuoglu, MarcAurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2009), pages 2146–2153. IEEE, 2009.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pages 1106–1114, 2012.

[12] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In Neural Networks (IJCNN), The 2010 International Joint Conference on, pages 1–8. IEEE, 2010.

[13] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[14] Hamid Maei, Csaba Szepesvari, Shalabh Bhatnagar, Doina Precup, David Silver, and Rich Sutton. Convergent Temporal-Difference Learning with Arbitrary Smooth Function Approximation. In Advances in Neural Information Processing Systems 22, pages 1204–1212, 2009.

[15] Hamid Maei, Csaba Szepesvari, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy ´ learning control with function approximation. In Proceedings of the 27th International Conference on Machine Learning (ICML 2010), pages 719–726, 2010.

[16] Volodymyr Mnih. Machine Learning for Aerial Image Labeling. PhD thesis, University of Toronto, 2013.

[17] Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. Machine Learning, 13:103–130, 1993.

[18] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning (ICML 2010), pages 807–814, 2010.

[19] Jordan B. Pollack and Alan D. Blair. Why did td-gammon work. In Advances in Neural Information Processing Systems 9, pages 10–16, 1996.

[20] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In Machine Learning: ECML 2005, pages 317–328. Springer, 2005.

[21] Brian Sallans and Geoffrey E. Hinton. Reinforcement learning with factored states and actions. Journal of Machine Learning Research, 5:1063–1088, 2004.

[22] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2013). IEEE, 2013.

[23] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1998.

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

[25] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Automatic Control, IEEE Transactions on, 42(5):674–690, 1997.

[26] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[27] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N. (2015). Dueling Network Architectures for Deep Reinforcement Learning. arXiv. https://doi.org/10.48550/arXiv.1511.06581

[28] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv. https://doi.org/10.48550/arXiv.1707.06347

[29] `https://www.analyticsvidhya.com/blog/2021/02/introduction-to-reinforcement-learning-for-beginners`

[30] `https://www.samyzaf.com/ML/rl/qmaze.html`

[31] `https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53`

[32] `https://s7280.pcdn.co/wp-content/uploads/2020/07/Two-or-more-hidden-layers-comprise-a-Deep-Neural-Network.png`

[33] `https://www.smithsonianmag.com/science-nature/original-space-invaders-icon-1970s-America-180969393/`

[34] https://en.wikipedia.org/wiki/Seaquest_(video_game)#/media/File:Seaquest_Screenshot.png

[35] https://i.ytimg.com/vi/vkZhWsiHCqM/maxresdefault.jpg

[36] https://www.tensorflow.org/images/MNIST-Matrix.png

[37] http://neuralnetworksanddeeplearning.com/chap1.html

[38] https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filte

[39] https://www.8-bitcentral.com/reviews/2600beamrider.html

[40] https://www.retrogames.cz/play_222-Atari2600.php

[41] https://en.wikipedia.org/wiki/Beamrider#/media/File:Beamrider_Atari_8-bit_PAL_screenshot.png

[42] https://gist.github.com/straker/81b59eecf70da93af396f963596dfdc5

Appendix A

Deep Q Learning codes

A.1 Main executable file - main.py

```
import argparse
from test import test
from environment import Environment
import warnings
warnings.filterwarnings("ignore")


def parse():
    parser = argparse.ArgumentParser()
    parser.add_argument('--folder_name',default=None)
    parser.add_argument('--model_path',default=None)
    parser.add_argument('--env_name', default=None,
        help='environment name')

    parser.add_argument('--train_dqn', action='store_true',
        help='whether train DQN')
    parser.add_argument('--continue_train', action='store_true',
        help='whether continue to train')
    parser.add_argument('--test_dqn', action='store_true',
        help='whether test DQN')

    parser.add_argument('--video_dir', default=None, help="output
        video directory")
    parser.add_argument('--do_render', action='store_true',
        help='whether render environment')

    args = parser.parse_args()
    return args


def run(args):
    "******Deep Q Learning******"
    if args.train_dqn:
        env_name = args.env_name or 'AssaultNoFrameskip-v0'
        env = Environment(env_name, args, atari_wrapper=True)
        from agent_dir.agent_dqn import AgentDQN
        agent = AgentDQN(env, args)
        agent.train()

    if args.test_dqn:
        env_name = args.env_name or 'AssaultNoFrameskip-v0'
        env = Environment(env_name, args, atari_wrapper=True,
            test=True)
        from agent_dir.agent_dqn import AgentDQN
        agent = AgentDQN(env, args)
        test(agent, env, total_episodes=100)

if __name__ == '__main__':
    args = parse()
    run(args)
```

A.2 Environment definition - environment.py

```
import gym
import numpy as np
from atari_wrapper import make_wrap_atari

class Environment(object):
    def __init__(self, env_name, args, atari_wrapper=False,
        test=False):

        if atari_wrapper:
            rewards = not test
            self.env = make_wrap_atari(env_name, rewards)
        else:
            self.env = gym.make(env_name)

        self.action_space = self.env.action_space
        self.observation_space = self.env.observation_space

        self.do_render = args.do_render
        if args.video_dir:
            self.env = gym.wrappers.Monitor(self.env,
                args.video_dir, force=True)

    def seed(self, seed):
        '''
        Control the randomness of the environment
        '''
        self.env.seed(seed)

    def reset(self):
        '''
        When running dqn:
            observation: np.array
                stack 4 last frames, shape: (84, 84, 4)
        '''
        observation = self.env.reset()

        return np.array(observation)


    def step(self,action):
        '''
        When running dqn:
            observation: np.array
                stack 4 last preprocessed frames, shape: (84, 84, 4)
            reward: int
                wrapper clips the reward to {-1, 0, 1} by its sign
                we don't clip the reward when testing
            done: bool
                whether reach the end of the episode?
        '''
        if not self.env.action_space.contains(action):
            raise ValueError('Ivalid action!!')

        if self.do_render:
            self.env.render()

        observation, reward, complete, data = self.env.step(action)

        return np.array(observation), reward, complete, data


    def get_action_space(self):
        return self.action_space
    def get_observation_space(self):
        return self.observation_space


    def get_random_action(self):
        return self.action_space.sample()
```

A.3 Agent definition along with CNN Architecture and Experience Replay

```
import random
import math
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import torch.nn as nn
import json
import os

from agent_dir.agent import Agent
from environment import Environment

from collections import deque, namedtuple

use_cuda = torch.cuda.is_available()


class DQN(nn.Module):
    '''
    This architecture is the one from OpenAI Baseline, with small
        modification.
    '''
    def __init__(self, channels, num_actions, duel_net=False):
        super(DQN, self).__init__()
        self.duel_net = duel_net
        self.conv1 = nn.Conv2d(channels, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc = nn.Linear(7*7*64, 512)
        self.head = nn.Linear(512, num_actions)
        self.relu = nn.ReLU()
        self.lrelu = nn.LeakyReLU(0.01)

        if self.duel_net:
            self.fc_value = nn.Linear(512, 1)
            self.fc_advantage = nn.Linear(512, num_actions)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.lrelu(self.fc(x.view(x.size(0), -1)))
        if self.duel_net:
            value = self.fc_value(x)
            advantage = self.fc_advantage(x)
            q = value + advantage - advantage.mean()
        else:
            q = self.head(x)

        return q

class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, buffer_size, batch_size):
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.buffer_size = buffer_size
        self.position = 0
        self.experience = namedtuple("Experience",
            field_names=["state", "action", "reward", "next_state",
            "done"])
    def add(self, state, action, reward, next_state, done):
        e = self.experience(state, action, reward, next_state, done)
        if len(self.memory) < self.buffer_size:
            self.memory.append(None)
        self.memory[self.position] = e
        self.position = (self.position + 1) % self.buffer_size
    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)
        states = torch.cat([e.state for e in experiences if e is not
            None]).float().cuda()
        next_states = torch.cat([e.next_state for e in experiences
            if e is not None]).float().cuda()
        actions = torch.from_numpy(np.vstack([e.action for e in
            experiences if e is not None])).long().cuda()
        rewards = torch.from_numpy(np.vstack([e.reward for e in
            experiences if e is not None])).float().cuda()
        dones = torch.from_numpy(np.vstack([e.done for e in
            experiences if e is not
            None]).astype(np.uint8)).float().cuda()
        return (states, actions, rewards, next_states, dones)
    def __len__(self):
        return len(self.memory)


class AgentDQN(Agent):
    def __init__(self, env, args):
        self.env = env
        self.input_channels = 4
        self.num_actions = self.env.action_space.n

        if args.test_dqn:
            if args.model_path == None:
                raise Exception('give --model_path')
        else:
            if args.folder_name == None:
                raise Exception('give --folder_name')
            self.model_dir = os.path.join('./model',args.folder_name)
            print(self.model_dir)
            if not os.path.exists(self.model_dir):
                os.mkdir(self.model_dir)

        # build target, online network

        self.online_net = DQN(self.input_channels, self.num_actions)
        self.online_net = self.online_net.cuda() if use_cuda else
            self.online_net
        self.target_net = DQN(self.input_channels, self.num_actions)
        self.target_net = self.target_net.cuda() if use_cuda else
            self.target_net
        self.target_net.load_state_dict(self.online_net.state_dict())

        if args.test_dqn:
            self.load(args.model_path)

        if args.continue_train:
            self.load(args.model_path)

        # discounted reward
        self.GAMMA = 0.99

        # training hyperparameters
        self.train_freq = 4 # frequency to train the online network
        self.num_timesteps = 3000000 # total training steps
        self.learning_start = 10000 # before we start to update our
            network, we wait a few steps first to fill the replay.
        self.batch_size = 32
        self.display_freq = 100 # frequency to display training
            progress
        self.target_update_freq = 1000 # frequency to update target
            network

        # optimizer
        self.optimizer = optim.RMSprop(self.online_net.parameters(),
            lr=1e-4)
        self.steps = 0 # num. of passed steps. this may be useful in
            controlling exploration
        self.eps_min = 0.1
        self.eps_max = 1.0
        self.eps_step = 200000
        if args.continue_train:
            self.eps_step = self.learning_start

        self.plot = {'steps':[], 'reward':[]}

        # TODO:
        # Initialize your replay buffer
        self.memory = ReplayBuffer(10000, self.batch_size)

    def save(self, save_path):
        print('save model to', save_path)
        model = {'online': self.online_net.state_dict(), 'target':
            self.target_net.state_dict()}
        torch.save(model, save_path)

    def load(self, load_path):
        print('Load model from', load_path)
        if use_cuda:
            self.online_net.load_state_dict(torch.load(load_path)['online'])
        else:
            self.online_net.load_state_dict(torch.load(load_path,
                map_location=lambda storage, loc: storage)['online'])

    def init_game_setting(self):
        # we don't need init_game_setting in DQN
        pass

    def epsilon(self, step):
        if step > self.eps_step:
            return 0
        else:
            return self.eps_min + (self.eps_max - self.eps_min) *
                ((self.eps_step - step) / self.eps_step)

    def make_action(self, state, test=False):
        if test:
            state =
                torch.from_numpy(state).permute(2,0,1).unsqueeze(0)
            state = state.cuda() if use_cuda else state
            with torch.no_grad():
                action = self.online_net(state).max(1)[1].item()
        else:
            if random.random() > self.epsilon(self.steps):
                with torch.no_grad():
                    action = self.online_net(state).max(1)[1].item()
            else:
                action = random.randrange(self.num_actions)

        return action

    def update(self):
        if len(self.memory) < self.batch_size:
            return

        experiences = self.memory.sample()
        batch_state, batch_action, batch_reward, batch_next,
            batch_done, = experiences

        if self.dqn_type=='DoubleDQN' or self.dqn_type == 'DDDQN':
            next_q_actions =
                self.online_net(batch_next).detach().max(1)[1].unsqueeze(1)
            next_q_values =
                self.target_net(batch_next).gather(1,next_q_actions)
        else:
            next_q_values = self.target_net(batch_next).detach()
            next_q_values = next_q_values.max(1)[0].unsqueeze(1)

        batch_reward = batch_reward.clamp(-1.1)
        current_q = self.online_net(batch_state).gather(1,
            batch_action)
        next_q = batch_reward + (1 - batch_done) * self.GAMMA *
            next_q_values
        loss = F.mse_loss(current_q, next_q)
        self.optimizer.zero_grad()
        loss.backward()

        self.optimizer.step()
        return loss.item()

    def train(self):
        best_reward = 0
        episodes_done_num = 1 # passed episodes
        total_reward = [] # compute average reward
        total_loss = []
        while(True):
            state = self.env.reset()
            # State: (80,80,4) --> (1,4,80,80)
            state =
                torch.from_numpy(state).permute(2,0,1).unsqueeze(0)
            state = state.cuda() if use_cuda else state
            done = False
            episodes_reward = []
            episodes_loss = []
            while(not done):
                # select and perform action
                action = self.make_action(state)
                next_state, reward, done, _ = self.env.step(action)
                episodes_reward.append(reward)
                total_reward.append(reward)
                # process new state
                next_state =
                    torch.from_numpy(next_state).permute(2,0,1).unsqueeze(0)
                next_state = next_state.cuda() if use_cuda else
                    next_state
                # TODO:
                # store the transition in memory
                self.memory.add(state, action, reward, next_state,
                    done)
                # move to the next state
                state = next_state
                # Perform one step of the optimization
                if self.steps > self.learning_start and self.steps %
                    self.train_freq == 0:
                    loss = self.update()
                    episodes_loss.append(loss)
                    total_loss.append(loss)
                # update target network
                if self.steps > self.learning_start and self.steps %
                    self.target_update_freq == 0:
                    self.target_net.load_state_dict(self.online_net.state_dict())
                # save the model
                self.steps += 1

            avg_ep_loss = sum(episodes_loss)/len(episodes_loss) if
                len(episodes_loss) > 0 else 0

            print('Episode: %d | Steps: %d/%d | Avg reward: %f |
                Loss: %f'%
                 (episodes_done_num, self.steps, self.num_timesteps,
                  sum(episodes_reward), avg_ep_loss),end='\r')

            self.plot['steps'].append(episodes_done_num)
            self.plot['reward'].append(sum(episodes_reward))

            if episodes_done_num % self.display_freq == 0:

                avg_reward = sum(total_reward) / self.display_freq
                avg_loss = sum(total_loss) / len(total_loss) if
                    len(total_loss) > 0 else 0

                if self.steps < self.eps_step:
                    phase = "Exploring phase"
                else:
                    phase = "Learning phase"

                print('%s | Episode: %d | Steps: %d/%d | epsilon: %f
                    | Avg reward: %f | Loss: %f'%
                        (phase, episodes_done_num, self.steps,
                           self.num_timesteps,
                           self.epsilon(self.steps), avg_reward,
                               avg_loss))

                if avg_reward > best_reward and self.steps >
                    self.eps_step:
                    best_reward = avg_reward
```

```
                  self.save(os.path.join(self.model_dir,
                      'e{}_r{:.2f}_model.cpt'.format(episodes_done_num,
                      avg_reward)))
                  with open(os.path.join(self.model_dir,
                      'plot.json'), 'w') as f:
                      json.dump(self.plot,f)

              total_reward = []
              total_loss = []

          episodes_done_num += 1

          if self.steps > self.num_timesteps:
              break

      self.save(os.path.join(self.model_dir,'e{}_model.cpt'.format(episodes_done_num)))
      with open(os.path.join(self.model_dir, 'plot.json'), 'w') as
          f:
          json.dump(self.plot,f)
```

A.4 Script to test the model - test.py

```
import argparse
import numpy as np
from environment import Environment
import time
from tqdm import tqdm

seed = 1000000

def test(agent, env, total_episodes=30):
    rewards = []
    env.seed(seed)
    for i in range(total_episodes):
        state = env.reset()
        agent.init_game_setting()
        done = False
        episode_reward = 0.0

        #playing one game
        while(not done):
            action = agent.make_action(state, test=True)
            state, reward, done, info = env.step(action)
            episode_reward += reward
            time.sleep(1/60)
        print('Episode:{}/{} | reward:
            {:.2f}'.format(i+1,total_episodes,episode_reward),end='\r')
        rewards.append(episode_reward)
    print('\nRun %d episodes'%(total_episodes))
    print('Mean:', np.mean(rewards))

def run(args):

    if args.test_dqn:
        env = Environment('AssaultNoFrameskip-v0', args,
            atari_wrapper=True, test=True)
        from agent_dir.agent_dqn import AgentDQN
        agent = AgentDQN(env, args)
        test(agent, env, total_episodes=100)

if __name__ == '__main__':
    args = argparse()
    run(args)
```

A.5 OpenAI GYM wrapper for environment processing - atariwrapper.py

```
"""

original code:
github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py

"""
import numpy as np
from collections import deque
import gym
from gym import spaces
import cv2

class NoopResetEnv(gym.Wrapper):
    def __init__(self, env, noop_max=30):
       """Sample initial states by taking random number of no-ops
           on reset.
       No-op is assumed to be action 0.
       """
       gym.Wrapper.__init__(self, env)
       self.noop_max = noop_max
       self.override_num_noops = None
       if isinstance(env.action_space, gym.spaces.MultiBinary):
           self.noop_action = np.zeros(self.env.action_space.n,
               dtype=np.int64)
       else:
           # used for atari environments
           self.noop_action = 0
           assert env.unwrapped.get_action_meanings()[0] == 'NOOP'

   def _reset(self, **kwargs):
       """ Do no-op action for a number of steps in [1,
           noop_max]."""
       self.env.reset(**kwargs)
       if self.override_num_noops is not None:
           noops = self.override_num_noops
       else:
           noops = self.unwrapped.np_random.randint(1,
               self.noop_max + 1) #pylint: disable=E1101
       assert noops > 0
       obs = None
       for _ in range(noops):
           obs, _, done, _ = self.env.step(self.noop_action)
           if done:
               obs = self.env.reset(**kwargs)
       return obs

class FireResetEnv(gym.Wrapper):
   def __init__(self, env):
       """Take action on reset for environments that are fixed
           until firing."""
       gym.Wrapper.__init__(self, env)
       assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
       assert len(env.unwrapped.get_action_meanings()) >= 3

   def _reset(self, **kwargs):
       self.env.reset(**kwargs)
       obs, _, done, _ = self.env.step(1)
       if done:
           self.env.reset(**kwargs)
       obs, _, done, _ = self.env.step(2)
       if done:
           self.env.reset(**kwargs)
       return obs

class EpisodicLifeEnv(gym.Wrapper):
   def __init__(self, env):
       """Make end-of-life == end-of-episode, but only reset on
           true game over.
       Done by DeepMind for the DQN and co. since it helps value
           estimation.
       """
       gym.Wrapper.__init__(self, env)
       self.lives = 0
       self.was_real_done = True

   def _step(self, action):
       obs, reward, done, info = self.env.step(action)
       self.was_real_done = done
       # check current lives, make loss of life terminal,
       # then update lives to handle bonus lives
       lives = self.env.unwrapped.ale.lives()
       if lives < self.lives and lives > 0:
           # for Qbert somtimes we stay in lives == 0 condtion for
               a few frames
           # so its important to keep lives > 0, so that we only
               reset once
           # the environment advertises done.
           done = True
       self.lives = lives
       return obs, reward, done, info

   def _reset(self, **kwargs):
       """Reset only when lives are exhausted.
       This way all states are still reachable even though lives
           are episodic,
       and the learner need not know about any of this
           behind-the-scenes.
       """
       if self.was_real_done:
           obs = self.env.reset(**kwargs)
       else:
           # no-op step to advance from terminal/lost life state
           obs, _, _, _ = self.env.step(0)
       self.lives = self.env.unwrapped.ale.lives()
       return obs

class MaxAndSkipEnv(gym.Wrapper):
   def __init__(self, env, skip=4):
       """Return only every ‘skip‘-th frame"""
       gym.Wrapper.__init__(self, env)
       # most recent raw observations (for max pooling across time
           steps)
       self._obs_buffer =
           np.zeros((2,)+env.observation_space.shape, dtype=’uint8’)
       self._skip      = skip

   def _step(self, action):
       """Repeat action, sum reward, and max over last
           observations."""
       total_reward = 0.0
       done = None
       for i in range(self._skip):
           obs, reward, done, info = self.env.step(action)
           if i == self._skip - 2: self._obs_buffer[0] = obs
           if i == self._skip - 1: self._obs_buffer[1] = obs
           total_reward += reward
           if done:
               break
       # Note that the observation on the done=True frame
       # doesn’t matter
       max_frame = self._obs_buffer.max(axis=0)

       return max_frame, total_reward, done, info

class ClipRewardEnv(gym.RewardWrapper):
   def _reward(self, reward):
       """Bin reward to {+1, 0, -1} by its sign."""
       return np.sign(reward)

class WarpFrame(gym.ObservationWrapper):
   def __init__(self, env):
       """Warp frames to 84x84 as done in the Nature paper and
           later work."""
       gym.ObservationWrapper.__init__(self, env)
       self.width = 84
       self.height = 84
       self.observation_space = spaces.Box(low=0, high=255,
           shape=(self.height, self.width, 1))
   def _observation(self, frame):
       frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
       frame = cv2.resize(frame, (self.width, self.height),
           interpolation=cv2.INTER_AREA)
       return frame[:, :, None]

class FrameStack(gym.Wrapper):
   def __init__(self, env, k):
       """Stack k last frames.
       Returns lazy array, which is much more memory efficient.
       See Also
       --------
       baselines.common.atari_wrappers.LazyFrames
       """
       gym.Wrapper.__init__(self, env)
       self.k = k
       self.frames = deque([], maxlen=k)
       shp = env.observation_space.shape
       self.observation_space = spaces.Box(low=0, high=255,
           shape=(shp[0], shp[1], shp[2] * k))

   def _reset(self):
       ob = self.env.reset()
       for _ in range(self.k):
           self.frames.append(ob)
       return self._get_ob()

   def _step(self, action):
       ob, reward, done, info = self.env.step(action)
       self.frames.append(ob)
       return self._get_ob(), reward, done, info

   def _get_ob(self):
       assert len(self.frames) == self.k
       return LazyFrames(list(self.frames))

class ScaledFloatFrame(gym.ObservationWrapper):
   def _observation(self, observation):
       # careful! This undoes the memory optimization, use
       # with smaller replay buffers only.
       return np.array(observation).astype(np.float32) / 255.0

class LazyFrames(object):
   def __init__(self, frames):
       """This object ensures that common frames between the
           observations are only stored once.
       It exists purely to optimize memory usage which can be huge
           for DQN's 1M frames replay
       buffers.
       This object should only be converted to numpy array before
           being passed to the model.
       You'd not belive how complex the previous solution was."""
       self._frames = frames

   def __array__(self, dtype=None):
       out = np.concatenate(self._frames, axis=2)
       if dtype is not None:
           out = out.astype(dtype)
       return out

def make_atari(env_id):
   env = gym.make(env_id)
   assert 'NoFrameskip' in env.spec.id
   env = NoopResetEnv(env, noop_max=30)
   env = MaxAndSkipEnv(env, skip=4)
   return env

def wrap_deepmind(env, episode_life=True, clip_rewards=True,
   frame_stack=False, scale=False):
   if episode_life:
       env = EpisodicLifeEnv(env)
   if 'FIRE' in env.unwrapped.get_action_meanings():
       env = FireResetEnv(env)
   env = WarpFrame(env)
   if scale:
       env = ScaledFloatFrame(env)
   if clip_rewards:
       env = ClipRewardEnv(env)
   if frame_stack:
       env = FrameStack(env, 4)
   return env

def make_wrap_atari(env_id='Breakout-v0', clip_rewards=True):
   #env = gym.make(env_id)
   env = make_atari(env_id)
   return wrap_deepmind(env, clip_rewards=clip_rewards,
       frame_stack=True, scale=True)
```