AI GAME BOT USING DEEP REINFORCEMENT LEARNING

A Project work submitted to

SRI VENKATESWARA UNIVERSITY COLLEGE OF ENGINEERING

In partial fulfilment of requirements for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

by

MACHANI AAMANI (11606024)

MOHAN PILLIMITLA (11606031)

SAMMATHA GOWD PAVAN KUMAR (11606038)

Under the supervision of

Dr. M. HUMERA KHANAM
Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING,

SRI VENKATESWARA UNIVERSITY COLLEGE OF ENGINEERING,
TIRUPATI - 517502, 2019-20.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SRI VENKATESWARA UNIVERSITY COLLEGE OF ENGINEERING, TIRUPATI

Declaration

This is to certify that the project entitled "**AI GAME BOT USING DEEP REINFORCEMENT LEARNING"** is a bonafide work performed by us, for the partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** from **Sri Venkateswara University College of Engineering, Tirupati.**

To the best of our knowledge, this project report has not been submitted to any other institution/ university.

MACHANI AAMANI (11606024)
MOHAN PILLIMITLA (11606031)
SAMMATHA GOWD PAVAN KUMAR (11606038)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SRI VENKATESWARA UNIVERSITY COLLEGE OF ENGINEERING, TIRUPATI

CERTIFICATE

This is to certify that the project work incorporated in this entitle "**AI GAME BOT USING DEEP REINFORCEMENT LEARNING"** is genuine and has been carried out under my supervision in the **Department of Computer Science and Engineering**, **Sri Venkateswara University College of Engineering.**

The work is comprehensive, complete and fit for evaluation carried out in partial fulfilment of the requirements for the award of **Bachelor of Technology Degree** in **Computer Science and Engineering** during the academic year 2018-19.

MACHANI AAMANI	**11606024**
MOHAN PILLIMITLA	**11606031**
SAMMATHA GOWD PAVAN KUMAR	**11606038**

To the best of our knowledge matter embodied in the project has not been submitted to any other University/Institution for the award of any Degree or Diploma.

Dr.M.HUMERA KHANA

Project Guide
Professor, Dept of CSE
SVUCE, Tirupati

Dr.P VENKATA SUBBA REDDY

Head of Department
Professor, Dept of CSE
SVUCE, Tirupati

ACKNOWLEDGEMENT

Indeed this page of acknowledgement shall never be able to touch the horizon of generosity of those who rendered their help to us and unfortunately, the list of expressions of thanks no matter how extensive is always incomplete and inadequate but still we would like to express our gratitude towards everyone who helped us in completing this project.

First and foremost, we would like to express our gratitude and indebtedness to our project guide Dr.M. Humera Khanam for her kindness and belief in us, and for her inspiring guidance, constructive criticism and priceless suggestions throughout this project work. We are sincerely thankful to her for her priceless support in improving our understanding of this project. We are also grateful to the head of the department and other faculty members for allowing us to take up this project and their valuable suggestions and critical reviews on the project after project work-1 and helping us in improving.

Thanking you,

MACHANI AAMANI (11606024)

MOHAN PILLIMITLA (11606031)

SAMMATHA GOWD PAVAN KUMAR (11606038)

ABSTRACT

We present a deep learning model for playing games with high level input (image/raw pixel) using reinforcement learning. The games are action limited (like snakes, catcher, air-raider etc.). The model consists of convolution neural networks for processing image inputs and fully connected layers for estimating actions according to the inputs where the idea of taking action is based on Q-learning (model-free reinforcement learning), yet modified for our policy and usage. The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behavior, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Here we use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. It was demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 57 classic Atari video games, using the same algorithm, network architecture, and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks. We apply this method to train our agent as an evaluation set of 57 classic Atari video games using transfer learning.

CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

INTRODUCTION

1.1 INTRODUCTION

Solving games is considered interesting as well as the basis of problem-solving structure because of the hostile, competitive challenges and agents involving the environment. Thus, developing a game AI opens the path of implementing the problem-solving ideas in real life situations. Beside that deep reinforcement learning has achieved several high-profile successes in difficult decision-making problems. But these algorithms typically require a huge amount of data before they reach reasonable performance.

In case of hard-coded bots we do not need these data for making the right move for right situations. Most cases the coded bots use greedy search algorithms for finding the best move possible for a particular state. As an example, in snake's game a bot will try to avoid the walls and find the shortest path for the reward (an apple or a worm). So how can we make a bot understand the same thing by not telling them about the environment, like what the wall is or worm is, just giving them the feedback of what they are doing, which is our way for solving the self-learning feature, using deep Q learning.

Our challenge is to make that huge amount of data before the learning process actually began. To solve that issue we can use many approaches like watching the replay of pro players for that game or we can feed some random data or we can play for real and record the actions taken for each screen. However, all of these are time consuming as well as require previous knowledge which we are trying to avoid because of the feature we are talking about "self-learning".

Another challenge is, we need to make a learning model for games, where every game is different, the environments are different and also the actions. Thus, there is no such fixed model for suiting all the games. There can be found similarities between games, like the racing games can be similar in terms of actions and rewards.

Final challenge is extracting rewards for each game or choosing the right rewards to improve the game play of the bot. The rewards are actually like the feature functions forming the evaluation function, we have seen in previous game solving algorithms, like the Deep Blue, which had over 8,000 feature functions and it was literally unbeatable.

Meanwhile, rewards are different in each game, in most cases rewards are the score in the screen. Although, the modern games include many hidden rewards to make that score, like in a racing game score is the position given after the race is finished (1st , 2nd , 3rd etc.) but the hidden rewards are the consistency in driving, the speed, avoiding obstacles, smooth turns with highest speed possible etc. If we can detect these features and reward the bot performing these with success, we can make it more efficient and faster in case of learning.

With these in mind, we propose a model to overcome those complications. Our model makes data for itself which solves the huge data set problem, then there is a generalized Q function for similar types of games. Depending on the highest Q value and neural network we train the bot. Furthermore, the bot improves itself by playing after each stage with a higher threshold than the previous one.

Our experiment took place in several domains: PyGame Learning Environment, Atari games of OpenAI environment, classic control games like Cartpole, Mountain Car. The Classic control games are comparatively easier ones to learn and perform with a short-term learning phase with least number of epochs, where the other environments took various amounts of time or epochs depending on the number of actions and the game states. What we had successfully done was making the agent realize the environment faster. Unlike the other algorithms the bot was successfully reading the environment faster and acting there, but the moves are not timed perfectly, reflecting the accuracy is not perfect enough from which we can draw a conclusion, there is a trade-off between time and accuracy.

1.1 MOTIVATION

Deep learning agents are still the unresolved ones because what is happening in the network is still not cleared to any scholars. There is research going on in this field and new aspects and propositions are discovered to solve real life problems. Games AI's are somewhat difficult to solve and are challenging also. Solving these difficult tasks with Deep learning can also be a challenge and futuristic work as learning as a human's way. Google's Deepmind can play Atari games but that needs a lot of time to train it. A recent success is achieved by the researchers of OpenAI, who trained their agent to play a 3D game with a more complex scenario and environment which beat top class players of the game Dota2. These kinds of agents can be implemented in real life problem solving like automated management or in medical science or in mining. Moreover, playing with these bots we can enjoy a challenging game, as today's game industry is growing faster than even the media.

1.2 PROBLEM DEFINITION

We are trying to maximize score given the game screen. The screen is RGB, so we need to process it for making the agent understand. Each frame in the screen is considered as state and there are certain actions for each game, depending on the game. We face the challenge to score without manipulating the memory of the game, which is considered as cheating as we are trying to make the agent learn on its own. As we are implementing reinforcement learning, we are also trying to maximize the score while learning how to play.

CHAPTER II

BACKGROUND & LITERATURE SURVEY

BACKGROUND & LITERATURE SURVEY

2.1 BRIEF HISTORY OF GAME BOTS

The existing game bots are mostly hard-coded, like the conditions are given and the bots act according to them, they are mainly called scripted bots. In the case of two player games, the artificial intelligence was starting to develop from 1951 in the game of Nim and Checkers. After that there was AI developed for arcade games like Space Invaders (1978), Galaxian (1979), Pac-Man (1980), but these were human opponents with specific tasks to do, like firing at the human player. To be a strong opponent to win against a human player it took long enough, when a chess game bot by IBM's Deep Blue computer defeated Garry Kasparaov, a grandmaster in 1997. From 2009 onwards, modern gaming introduces more technical bots who can react according to realistic markers, such as sound made by players and footprints they left behind. Moreover, after a year Nintendo, a video game company started a competition called 'Mario Ai Championship', where the task was to make a game bot that can play Mario, a popular Nintendo Console game. There the concept of a new game bot arises when some competitors were able to make one bot that literally knows nothing about the game but completed the game after 34 tries using a neural network. Moreover, that bot was like a human, unable to see beyond the computer screen. In 2014 Google acquired 'DeepMind', an AI company that used deep reinforcement learning to make a bot that is not pre-programmed, can play some arcade games like Space Invaders, Breakout.

2.2 BACKGROUND STUDIES

2.2.1 DEEP LEARNING IN NEURAL NETWORK

Jürgen Schmidhuber talks about the deep learning policies and underlying mechanics in his research. In recent years, deep artificial neural networks (including recurrent ones) have won numerous contests in pattern recognition and machine learning. This historical survey compactly summarizes relevant work, much of it from the previous millennium.

Shallow and Deep Learners are distinguished by the depth of their credit assignment paths, which are chains of possibly learnable, causal links between actions and effects. He reviewed deep supervised learning (also recapitulating the history of backpropagation), unsupervised learning, reinforcement learning & evolutionary computation, and indirect search for short programs encoding deep and large networks.

2.2.2 OPTIMIZATION METHOD FOR DEEP LEARNING

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, Andrew Y. Ng showed that more sophisticated off-the-shelf optimization methods such as Limited memory BFGS (L-BFGS) and Conjugate gradient (CG) with line search can significantly simplify and speed up the process of pretraining deep algorithms. In their experiments, the difference between LBFGS/CG and SGDs are more pronounced if we consider algorithmic extensions (e.g., sparsity regularization) and hardware extensions (e.g., GPUs or computer clusters). Their experiments with distributed optimization support the use of L-BFGS with locally connected networks and convolutional neural networks. Using L-BFGS, their convolutional network model achieves 0.69% on the standard MNIST dataset. This is a state-of-the-art result on MNIST among algorithms that do not use distortions or pretraining.

2.2.3 IMAGENET CLASSIFICATION WITH DEEP CONVOLUTION NEURAL NETWORK

Alex Krizhevsky, Ilya Sutskever, E. Hinton, Geoffrey trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into 1000 different classes. On the test data, they achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way softmax. To make training faster, they used non-saturating neurons and a very efficient GPU implementation of the convolution operation.

To reduce overfitting in the fully-connected layers they employed a recently-developed regularization method called “dropout” that proved to be very effective. They also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

2.3 RELATED WORKS

2.3.1 THE LENET ARCHITECTURE

LeNet was one of the very first convolutional neural networks which helped propel the field of Deep Learning. This pioneering work by Yann LeCun was named LeNet5 after many previous successful iterations since the year 1988. At that time the LeNet architecture was used mainly for character recognition tasks such as reading zip codes, digits, etc.

Below, Figure 2.1 is an intuition of how the LeNet architecture learns to recognize images. There have been several new architectures proposed in the recent years which are improvements over the LeNet, but they all use the main concepts from the LeNet and are relatively easier to understand if you have a clear understanding of the former. So, we are using the concept of LeNet architecture.

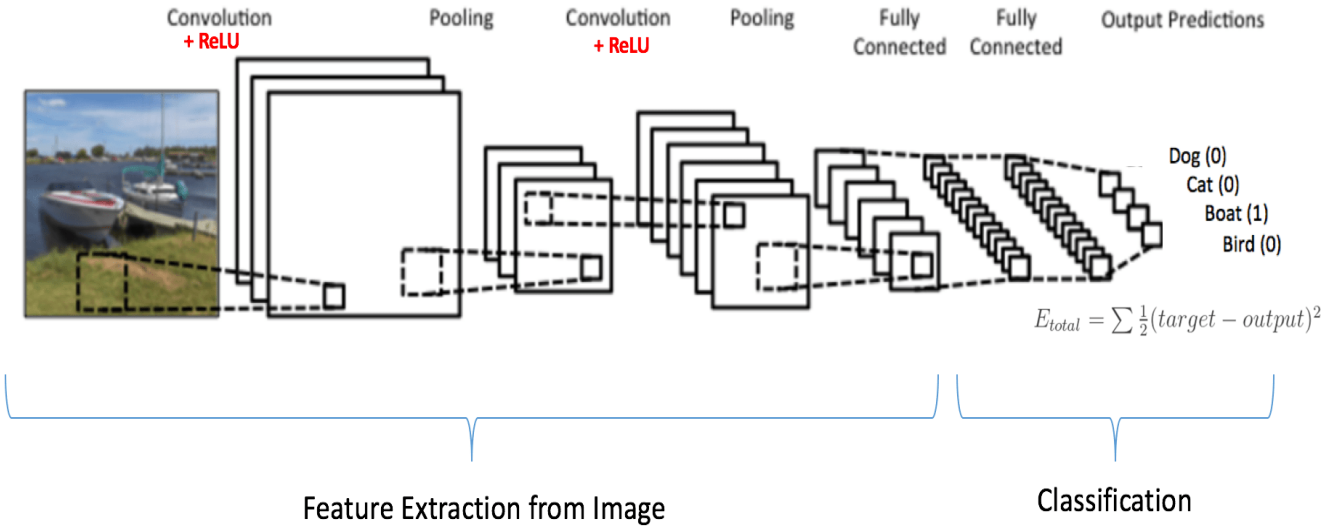

Figure 2.3.1.1: LeNet architecture

2.3.2 ALEXNET MODEL

The overall architecture of the net contains eight layers with weights; the first five are convolutional and the remaining three are fully-connected. The output of the last fully-connected layer is fed to a 1000-way SoftMax which produces a distribution over the 1000 class labels. Their network maximizes the multinomial logistic regression objective, which is equivalent to maximizing the average across training cases of the log-probability of the correct label under the prediction distribution. The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer which reside on the same GPU. The kernels of the third convolutional layer are connected to all kernel maps in the second layer. The neurons in the fully-connected layers are connected to all neurons in the previous layer. Response- normalization layers follow the first and second convolutional layers. Max-pooling layers follow both response-normalization layers as well as the fifth convolutional layer. The ReLU non- linearity is applied to the output of every convolutional and fully-connected layer. The first convolutional layer filters the 224×224×3 input image with 96 kernels of size 11×11×3 with a stride of 4 pixels. The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5 × 5 × 48. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size 3 × 3 × 256 connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size 3 × 3 × 192, and the fifth convolutional layer has 256 kernels of size 3 × 3 × 192. The fully-connected layers have 4096 neurons each.

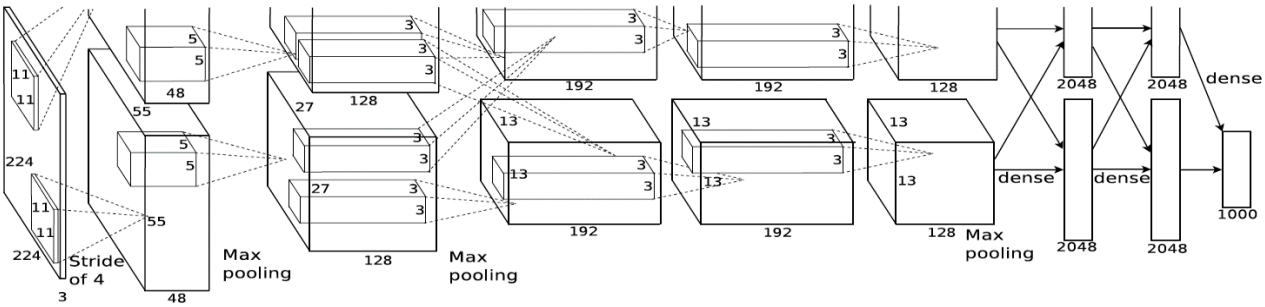

Figure 2.3.2.1: AlexNet architecture

2.3.3 IMAGE INPUT CONVNET AND FULLY-CONNECTED MODEL

This model is also used for the PyGame environment. We featured out the object's position using the conv layers and the fully connected simplified the high-level features of conv layers. We tuned the nodes and layers for better performance.

2.4 GENERALIZATIONS OF DEEP LEARNING

Deep learning is in contrast to "shallow" learning. For many machine learning algorithms, e.g., linear regression, logistic regression, support vector machines (SVMs), decision trees, and boosting, we have input layer and output layer, and the inputs may be transformed with manual feature engineering before training. In deep learning, between input and output layers, we have one or more hidden layers. At each layer except input layer, we compute the input to each unit, as the weighted sum of units from the previous layer; then we usually use nonlinear transformation, or activation function, such as logistic, tanh, or more popular recently, rectified linear unit (ReLU), to apply to the input of a unit, to obtain a new representation of the input from previous layer. We have weights on links between units from layer to layer. After computations flow forward from input to output, at output layer and each hidden layer, we can compute error derivatives backward, and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function. A feedforward deep neural network or multilayer perceptron (MLP) is to map a set of input values to output values with a mathematical function formed by composing many simpler functions at each layer. A convolutional neural network (CNN) is a feedforward deep neural network, with convolutional layers, pooling layers and fully connected layers. CNNs are designed to process data with multiple arrays, e.g., color image, language, audio spectrogram, and video, benefit from the properties of such signals: local connections, shared weights, pooling and the use of many layers, and are inspired by simple cells and complex cells in visual neuroscience (LeCun et al., 2015). A recurrent neural network (RNN) is often used to process sequential inputs like speech and language, element by element, with hidden units to store history of past elements.

A RNN can be seen as a multilayer neural network with all layers sharing the same weights, when being unfolded in time of forward computation. It is hard for RNN to store information for a very long time and the gradient may vanish. Long short-term memory networks (LSTM) and gated recurrent units (GRU) were proposed to address such issues, with gating mechanisms to manipulate information through recurrent cells. Gradient backpropagation or its variants can be used for training all deep neural networks mentioned above. Dropout is a regularization strategy to train an ensemble of sub-networks by removing non-output units randomly from the original network. Batch normalization performs the normalization for each training mini-batch, to accelerate training by reducing internal covariate shift, i.e., the change of parameters of previous layers will change each layer's input distribution. Deep neural networks learn representations automatically from raw inputs to recover the compositional hierarchies in many natural signals, i.e., higher-level features are composed of lower-level ones, e.g., in images, the hierarchy of objects, parts, motifs, and local combinations of edges. Distributed representation is a central idea in deep learning, which implies that many features may represent each input, and each feature may represent many inputs. The exponential advantages of deep, distributed representations combat the exponential challenges of the curse of dimensionality.

CHAPTER III

SYSTEM ANALYSIS

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

DeepMind proposed a model using the paradigm of reinforcement learning (RL), here like a human, the agents construct and learn its own knowledge directly from raw inputs, such as vision, without any hand-engineered features or domain heuristics. This is achieved by deep learning of neural networks. The agents must continually make value judgements so as to select good actions over bad. This knowledge is represented by a Q-network that estimates the total reward that an agent can expect to receive after taking a particular action. The key idea was to use deep neural networks to represent the Q-network, and to train this Q-network to predict total reward. Previous attempts to combine RL with neural networks had largely failed due to unstable learning. To address these instabilities, Deep Q-Networks (DQN) was used, this algorithm stores all of the agent's experiences and then randomly samples and replays these experiences to provide diverse and decorrelated training data. They applied DQN to learn to play games on the Atari 2600 console. At each time-step the agent observes the raw pixels on the screen, a reward signal corresponding to the game score, and selects a joystick direction but it suffered substantial overestimations in some games in the Atari 2600 domain. So, we are using this concept and adding the Transfer learning technique to overcome this problem which would help us to decrease the training time.

3.1.1 DISADVANTAGES

- The Deep Q-Network requires a lot of computational resources in order to train.
- The Deep-Mind Q-Network makes better results if it is trained for a longer period of time.
- It requires a lot of time because of the number of hyperparameters of the convolutional neural network and also the backpropagation.
- So here we make another trick which is transfer learning which is really efficient

3.2 PROPOSED SYSTEM

3.2.1 INTRODUCTION

In short, for the data to train the model at first, we are using random actions for the game, where the domain of the actions must be known. Then there is a learning stage where we introduce a Q function for each action taken and later for the particular state, we are using the action with the highest Q value for that state. In this stage the model has the Q network trained for playing the game, it starts to play the game. This time we record the best moves like before with a higher threshold, and continue it until we have found the satisfied model. As we are trying to make it more human-like, where the humans get information from the screen and play according to, we need to process the screen first. We are calling it pre-processing. For the deep learning model, we have implemented a 1280 neuron model for the classic control games, with 5 hidden layers and fully connected layers. In case of the Atari games we have used convolutional networks for the input image and then 20480 fully connected neurons with 4 hidden layers. We have also used the Alexnet implemented by Alex Krizhevsky which also gave us a good result for defining images.

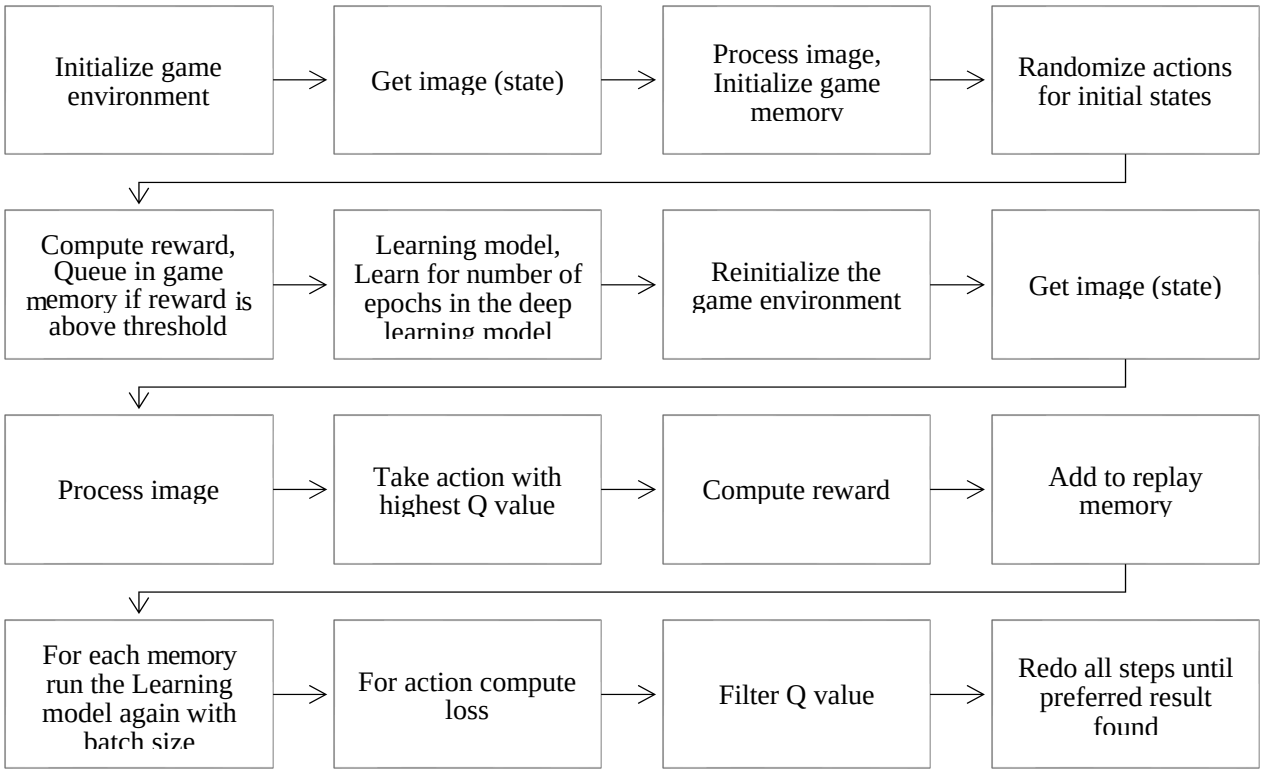

Figure 3.2.1.1: Proposed model

3.2.2 GAME ENVIRONMENT

For setting up the game environment we are mostly using the OpenAi Gym environment. They provide lots of gaming environment with the state, reward and actions. Where the states are basically the array representation of image or in some cases the game memory, for where the agent is and the opponents are. And for other cases the game environments are set up with the help of simulators and PyGame engine. Figure 3.2.1 shows some examples of game environments we have used.

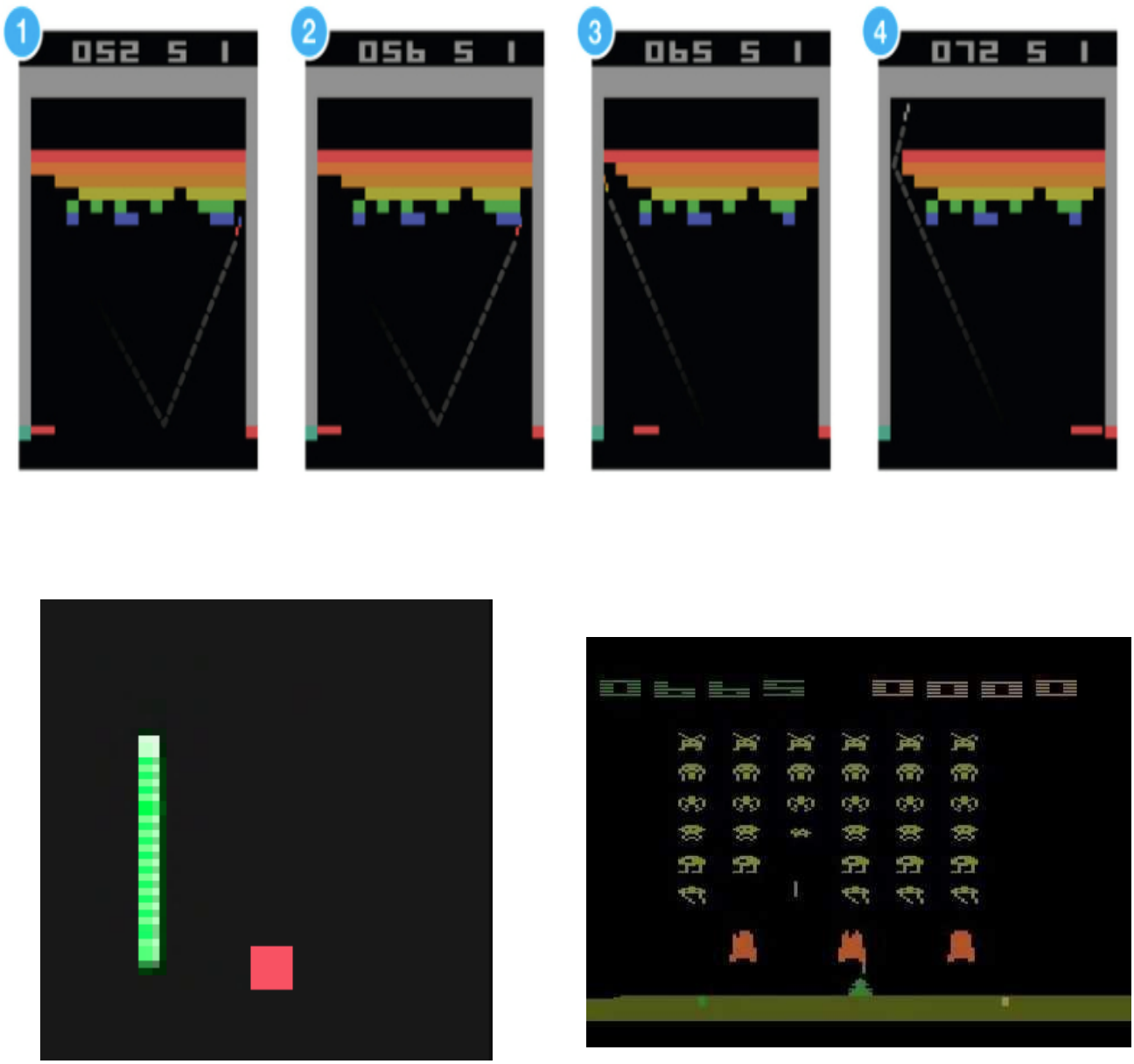

Figure 3.2.2.1: Game Environments (Breakout, Snake, Space invader)

3.2.3 PREPROCESSING (IMAGE PROCESSING)

To get the game screen we are taking continuous screenshots. As we all know a game is continuous screens that come one after another to prove the motion. So, we need a faster way to take screenshots and convert them to array for further processing. We are doing it with the help of python library OpenCV and Scikit Image.

First, we convert the RGB image to a numpy array with the help of OpenCV library. Then we change the dimension with the help of Scikit Image library and also convert it to grayscale for better object detection. Figure 3.3 shows the whole process.

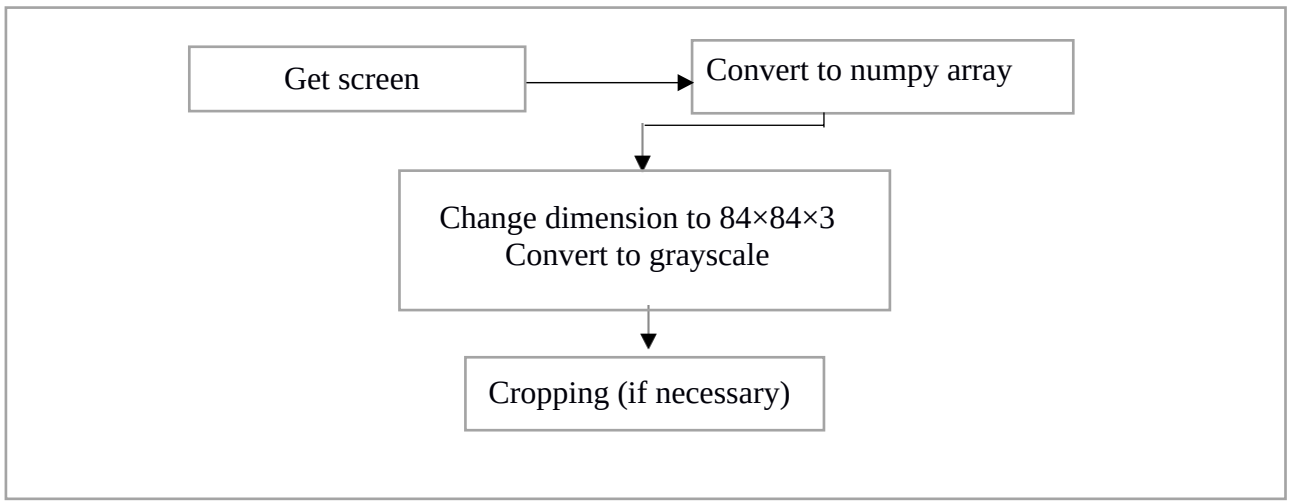

Figure 3.2.3.1: Preprocessing

3.2.3.1 SAMPLE CODE (PREPROCESSING)

```
from gym.core import ObservationWrapper
from gym.spaces import Box

# from scipy.misc import imresize
import cv2

class PreprocessAtari(ObservationWrapper):
    def __init__(self, env):
        """A gym wrapper that crops, scales image into the desired shapes and optionally
            grayscales it."""
        ObservationWrapper.__init__(self,env)
        self.img_size = (84, 84)
        self.observation_space = Box(0.0, 1.0, (self.img_size[0], self.img_size[1], 1))
```

```
def observation(self, img):
        """what happens to each observation"""
        # crop image (top and bottom, top from 34, bottom remove last 16)
        img = img[34:-16, :, :]

        # resize image
        img = cv2.resize(img, self.img_size)
        img = img.mean(-1,keepdims=True)
        img = img.astype('float32') / 255
        return img

from gym.spaces.box import Box
from gym.core import Wrapper
class FrameBuffer(Wrapper):
    def __init__(self, env, n_frames=4, dim_order='tensorflow'):
        """A gym wrapper that reshapes, crops and scales image into the desired shapes"""
        super(FrameBuffer, self).__init__(env)
        self.dim_order = dim_order
        if dim_order == 'tensorflow':
            height, width, n_channels = env.observation_space.shape
            """Multiply channels dimension by number of frames"""
            obs_shape = [height, width, n_channels * n_frames]
        else:
            raise ValueError('dim_order should be "tensorflow" or "pytorch", got
{}'.format(dim_order))
        self.observation_space = Box(0.0, 1.0, obs_shape)
        self.framebuffer = np.zeros(obs_shape, 'float32')

    def reset(self):
        """resets breakout, returns initial frames"""
        self.framebuffer = np.zeros_like(self.framebuffer)
        self.update_buffer(self.env.reset())
        return self.framebuffer

    def step(self, action):
        """plays breakout for 1 step, returns frame buffer"""
        new_img, reward, done, info = self.env.step(action)
        self.update_buffer(new_img)
        return self.framebuffer, reward, done, info

    def update_buffer(self, img):
        if self.dim_order == 'tensorflow':
            offset = self.env.observation_space.shape[-1]
            axis = -1
            cropped_framebuffer = self.framebuffer[:,:,:-offset]
        self.framebuffer = np.concatenate([img, cropped_framebuffer], axis = axis)
```

3.2.4 TRANSFER LEARNING (FOR INITIAL POPULATION)

For our dataset we are creating a random action-based dataset. Since we are making the agent learn on its own, it generates its own dataset rather than the human generated ones. Therefore, the initial population is the random played games. Here we have to make the agent understand the good steps and bad steps. For this reason, we are rewarding the agent for the good steps it takes and punish it for the bad steps. We are defining the good steps which makes the score. Now the score is calculated in many fashions, as an example in the snake's game we are rewarding the agent for eating the apple or the worm, where penalty is given for touching the walls or colliding with its own body. In the case of the OpenAI environment, they give us the reward that is showed on the screen. Now we need to save the state (processed image) with the action taken, where the action is converted to a multi-hot array for distinguishing the prediction of an action, just like the Q matrix is used for the normal Q learning. We set a threshold of score for each episode and we save at least 64 states of the last game memory for the dataset. We are doing this because a scoring state is what we want to train the agent for making it score like that, but the scoring state is reached by some previous action that is taken in previous steps for a successful score. So here is the game memory helping us for the short-term memory. And the threshold is the reinforcement part of our approach because we are forcing the agent to score that much but not telling it how to do that score.

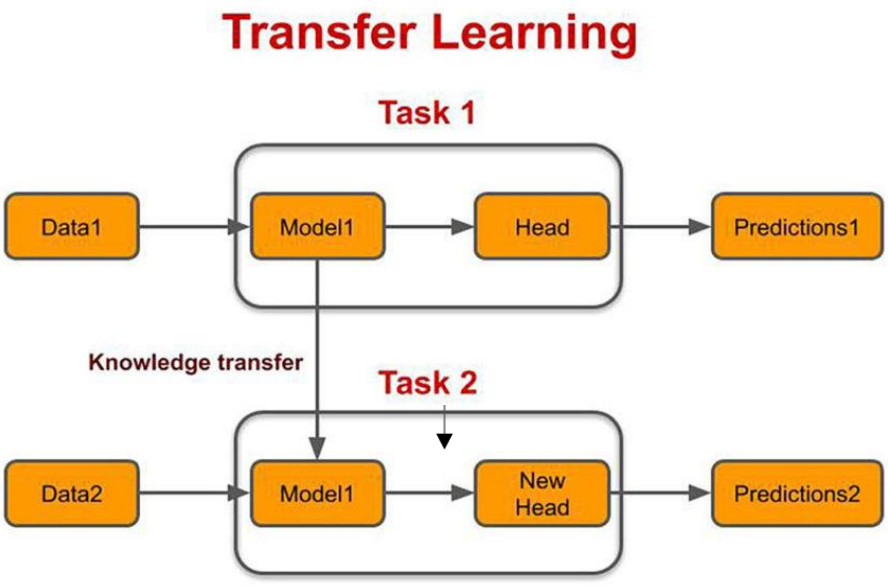

Figure 3.2.4.1: Transfer Learning

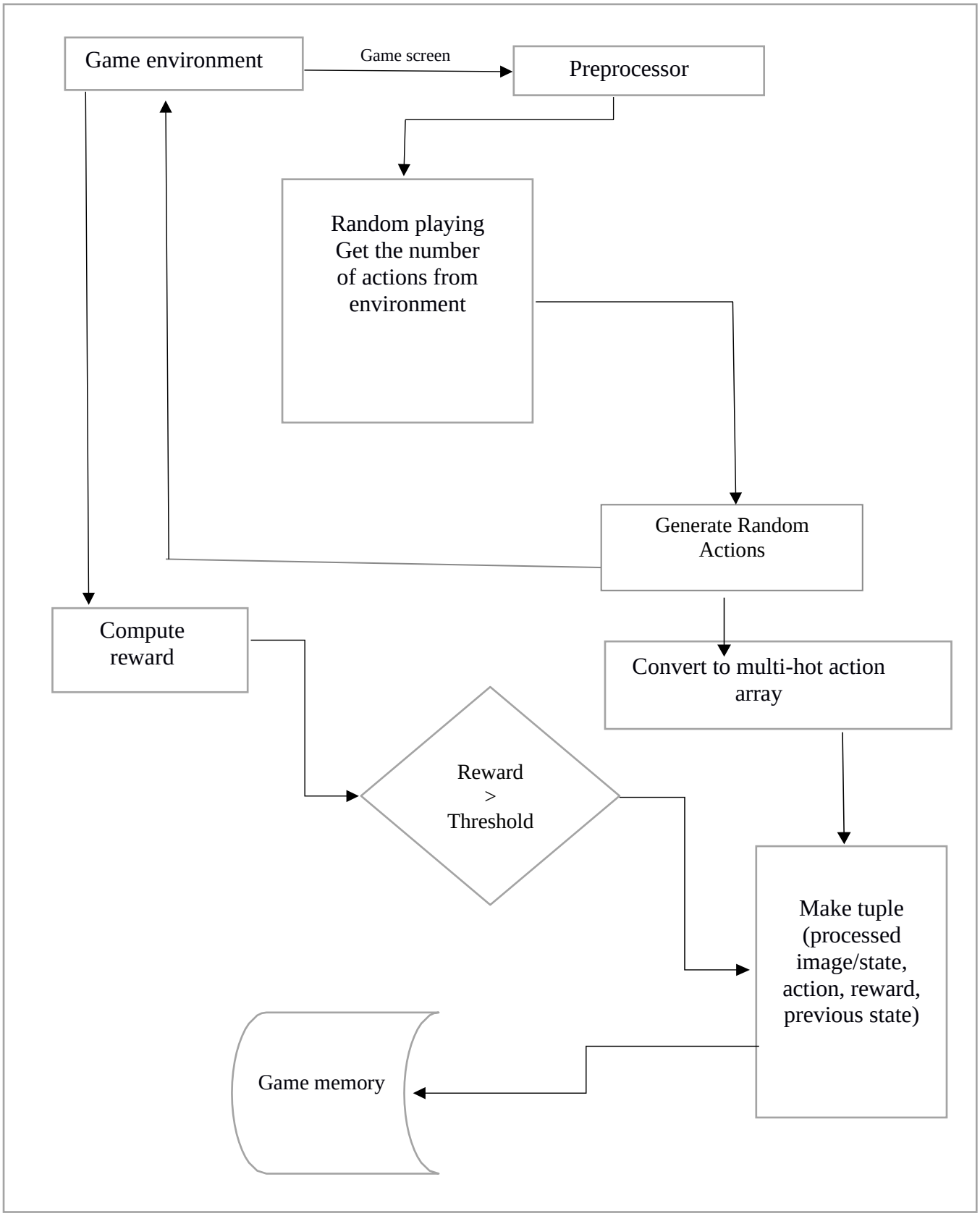

Figure 3.2.4.2: Random playing

3.2.5 TRAINING

Since the Q function obeys the Bellman equation, we trained our neural network to match:

$$Q(s, a) = r + \gamma \max_{a`} Q`(s`, a`) \quad (4)$$

Where Q` is a cached version of the neural network. This cached version updates much more infrequently to prevent instability when training; otherwise, the network would be training on itself, and small initial disturbances could lead to drastically diverging q values.

We have used several models for training. Their comparisons and results are in the results section. The model used for the classic control games like cart pole, mountain cars contains 5 fully connected layers with 128, 256, 512, 256, 128 neurons. These are the hidden layers. The input and the output layers are dependent on the input and action size. As these games are simple the inputs are the position of the cart pole or the mountain car, not even an image data. So, they are super-fast to train and accuracy is much better than the other models.

For the GYM environment we are using the learning model with the deep q network, because the inputs are the images taken as screenshots. We used both the AlexNet and our own simple image classifier. The AlexNet is described above. In the case of our network, the first convolutional layer filters the 84×84×3 input image with 64 kernels of size 3×3×1 with a stride of 4 pixels. The second convolutional layer takes as input the output of the first convolutional layer and filters it with 64 kernels of size 3 × 3 × 16. The third convolutional layer has 128 kernels of size 3 × 3 × 128 connected to the (normalized, pooled) outputs of the second convolutional layer. The fully- connected layers have 128, 256, 512, 256, 128 neurons respectively. This model is efficient because it will store the parameters which will learn by using reinforcement learning. Reinforcement learning helps to learn by interacting and convolutional neural networks are predicting the next action to be performed. We use the concept of replay memory which is a buffer pool which stores the initial action space and transfer learning will be helpful in transferring previously learned parameters or weights.

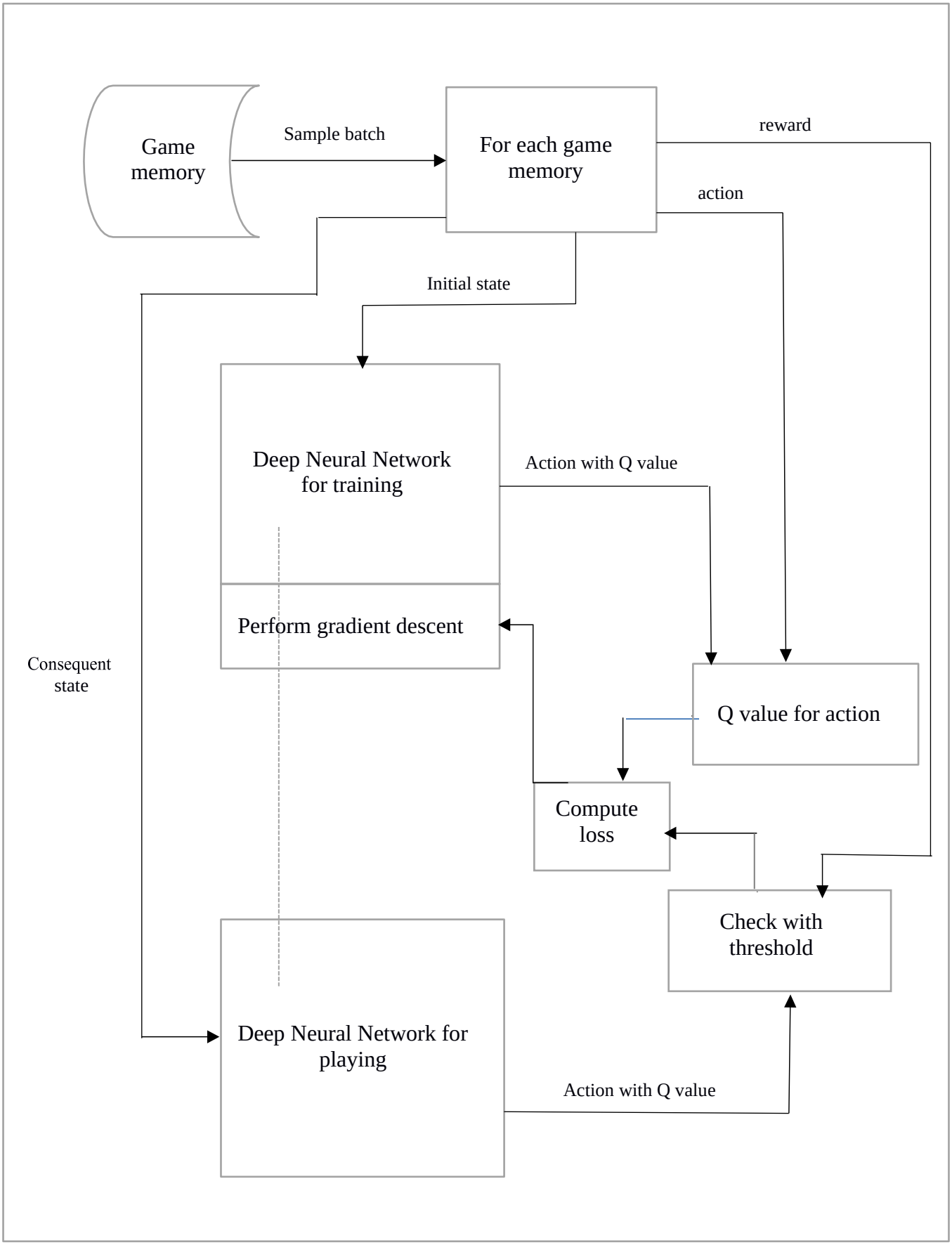

Figure 3.2.5.1: Training and playing process

3.3 SYSTEM REQUIREMENTS AND SPECIFICATION

3.3.1 SOFTWARE REQUIREMENTS

1) Tensor Flow
2) NumPy (Scientific Calculations).
3) Keras (Convolutional Neural Networks).
4) OpenAI Gym (Atari Games and Data Set).
5) Python 3.7 (Programming Language).

3.3.2 HARDWARE REQUIREMENTS

1) High Computational Power CPU or GPU.
2) RAM - 8GB (Min)
3) Hard Disk - 160GB
4) Key Board - Standard Windows Keyboard
5) Mouse - Two or Three Button Mouse
6) Monitor - SVGA

CHAPTER IV

SYSTEM DESIGN

SYSTEM DESIGN

4.1 INITIALIZATION OF DATASET

First, we play random games as described in 3.2.4.

```
env = gym.make(env_name)
env.reset()
goal_steps = 500
score_requirement = threshold
initial_games = 1000
numberOfActions = env.action_space.n
```

For different games we change threshold and goal_steps.
Then we create the initial population.

```
for initial_games
    game_memory = []
    prev_observation = []
    for goal_steps:
        action = randomAction
        observation, reward, done, info = env.step(action)
        observation = processImage
        game_memory.insert([prev_observation, action])
        prev_observation = observation
        if done: break
    if score is greater than score_requirement
        accepted_scores.insert(score)
        for game_memory
        MultiHot the action array
        training_data.insert([screen, action array])
```

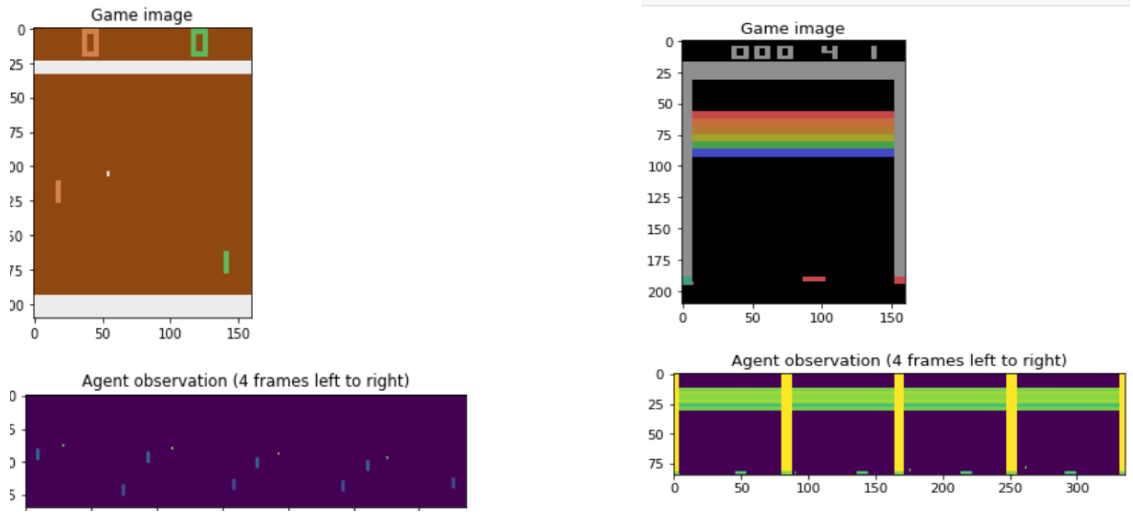

Figure 4.1: Data-set samples of Pong(left) and Breakout(right) environments

4.2 IMPLEMENTATION

4.2.1 CONCEPTS OVERVIEW

Here we use different concepts which are: -

1. Markov Decision Process
2. Convolution Concept
3. Q-Learning
4. Q- Learning Estimation with Bellman Equations

Markov Decision Process

The base idea for the implantation of this model is using Deep Reinforcement Learning. Here the basic procedure is that the agent needs to predict the best and optimal action where it gains a reward for every positive action and loses a reward for every negative action against the environment.

By performing an action, the agent will go to the next state where it again aims to maximize the future discounted reward. It will not be dependent on the past actions performed

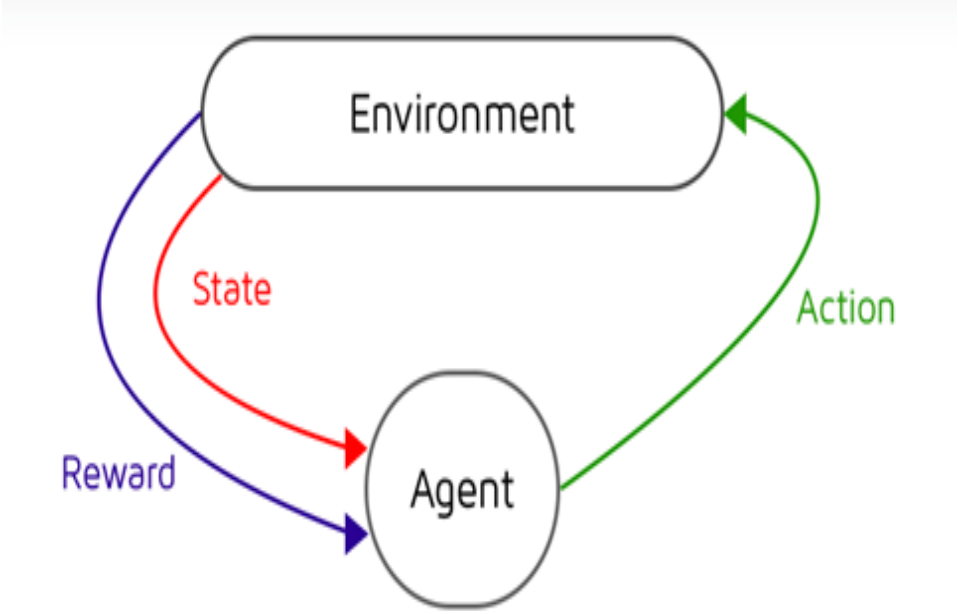

Figure 4.2.1: An illustration of Markov Decision Process

Convolution Layer

Our model consists of 5 layers which has 4 hidden layers and 1 output layer where each has different convolutions and different strides they are: -

- First Hidden layer – 32 X 8 X 8 Filters with Stride 4.
- Second Hidden Layer – 64 X 4 X 4 Filters with Stride 2.
- Third Hidden Layer – 64 X 3 X 3 Filters with Stride 1.
- Fourth Hidden Layer – Fully connected with 512 ReLU units.
- Output Layer- Fully connected with single output which estimates the Q – Function

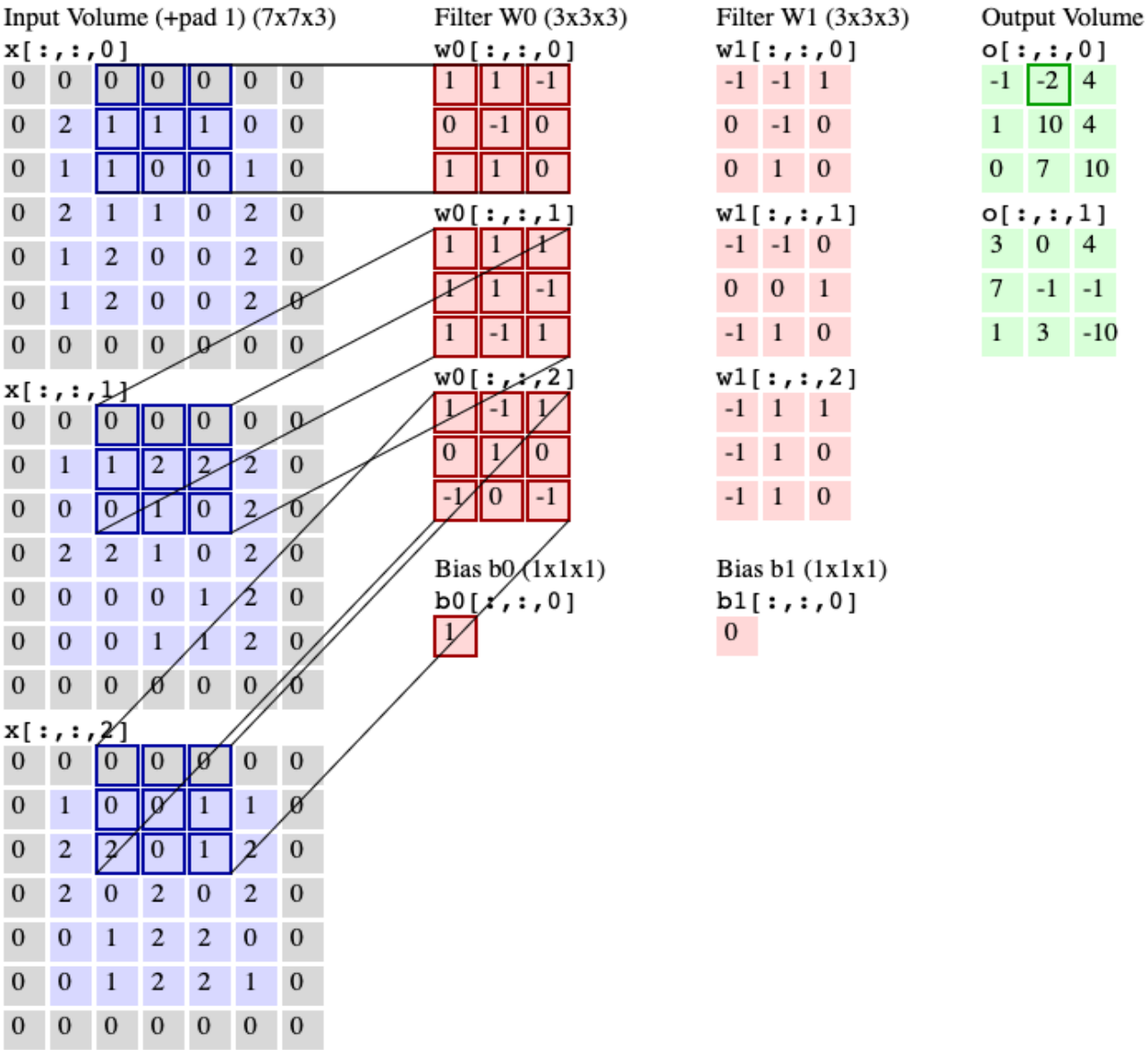

Figure 4.2.2: Convolution Operations

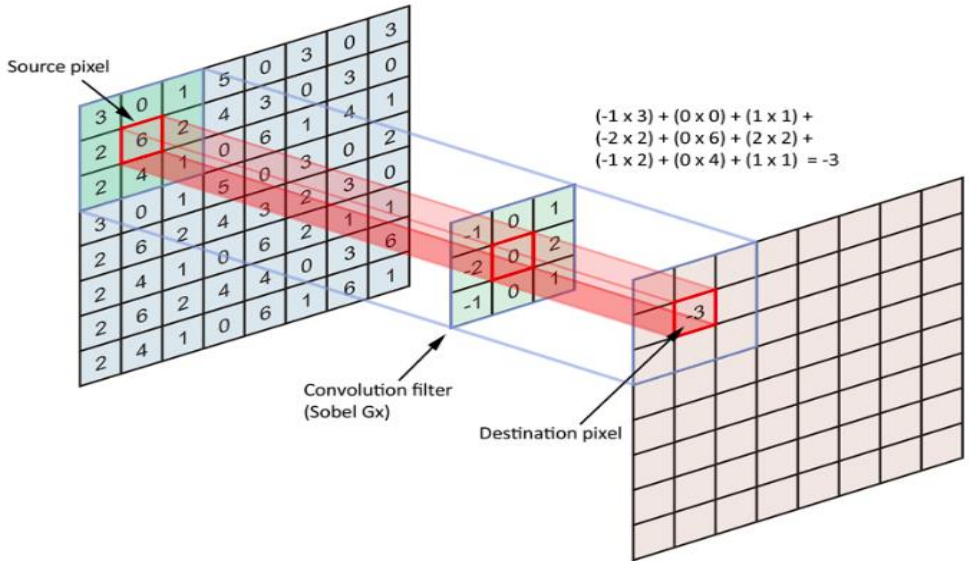

Figure 4.2.3: An illustration of Convolution Operation

- In the above image shown the convolution operation is done on the raw input image using different strides and filters
- Filters are used to detect the features in the image
- Strides are used to share the features among different filters in the network
- Padding is added to avoid the possibility of gaining attention to the features
- The number of valid actions varied between 4 and 18 on the games we considered. Which in turn acts as the output of the Q – Function.

Overview and intuition without brain stuff: Lets first discuss what the CONV layer computes without brain/neuron analogies. The CONV layers parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer, and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

The brain view: If you're a fan of the brain/neuron analogies, every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter). We now discuss the details of the neuron connectivity's, their arrangement in space, and their parameter sharing scheme.

Local Connectivity: When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.

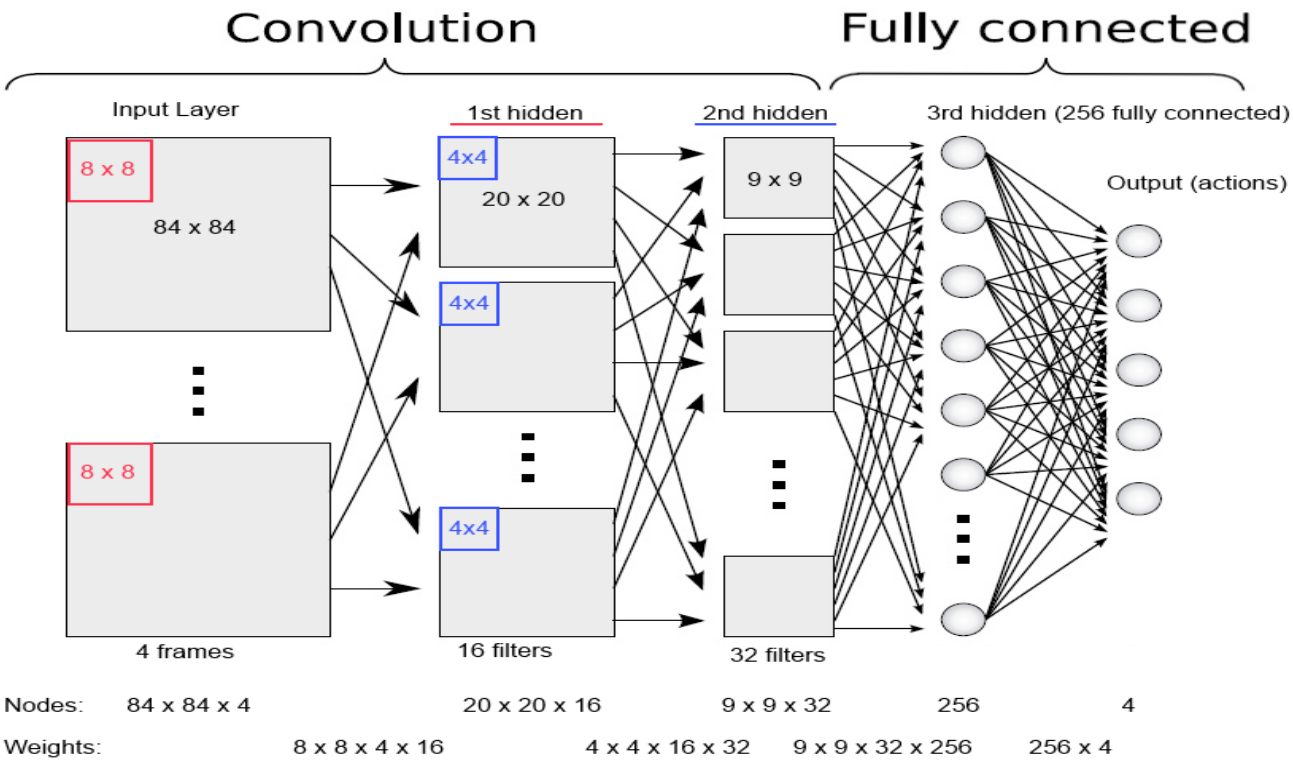

Figure 4.2.4: Deep Q Network

Q – Learning

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy Isn't needed. Q-value function gives expected total reward from state s and action a under policy π with discount factor γ.

$$Q\,\pi\,(s, a) = E\,[rt+1 + \gamma rt+2 + \gamma\,2\,rt+3 + \ldots \mid s, a\,]$$

Let's say we know the expected reward of each action at every step. This would essentially be like a cheat sheet for the agent! Our agent will know exactly which action to perform.

It will perform the sequence of actions that will eventually generate the maximum total reward. This total reward is also called the Q-value and we will formalise our strategy as:

$$Q(s, a) = r(s, a) + \gamma \max_{a} Q(s', a)$$

The above equation states that the Q-value yielded from being at state *s* and performing action *a* is the immediate reward r(s,a) plus the highest Q-value possible from the next state *s'*. Gamma here is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s, a) \rightarrow \gamma Q(s', a) + \gamma^2 Q(s'', a) \ldots\ldots\ldots \gamma^n Q(s''^{\ldots n}, a)$$

Adjusting the value of gamma will diminish or increase the contribution of future rewards.

Since this is a recursive equation, we can start with making arbitrary assumptions for all q-values. With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

Q – Learning estimation with bellman equations:

If you have read anything related to reinforcement learning you must have encountered bellman equation somewhere. Bellman equation is the basic block of solving reinforcement learning and is omnipresent in RL. It helps us to solve MDP. To solve means finding the optimal policy and value functions. The optimal value function V*(S) is one that yields maximum value. The value of a given state is equal to the max action (action which maximizes the value) of the reward of the optimal action in the given state and add a discount factor multiplied by the next state's Value from the Bellman Equation.

$$V(s) = max_a(R(s,a) + \gamma V(s'))$$

Bellman equation for deterministic environment

Let's understand this equation, V(s) is the value for being in a certain state. V(s') is the value for being in the next state that we will end up in after taking action a. R (s, a) is the reward we get after taking action a in state s. As we can take different actions so we use maximum because our agent wants to be in the optimal state. γ is the discount factor as discussed earlier. This is the bellman equation in the deterministic environment. It will be slightly different for a non-deterministic environment or stochastic environment.

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

Bellman equation for stochastic environment

In a stochastic environment when we take an action it is not confirmed that we will end up in a particular next state and there is a probability of ending in a particular state. P (s, a,s') is the probability of ending is state's' from s by taking action a. This is summed up to a total number of future states. For example, if by taking an action we can end up in 3 states s_1,s_2, and s_3 from states with a probability of 0.2, 0.2 and 0.6. The Bellman equation will be

$$V(s) = max_a\ (R\ (s,\ a) + \gamma(0.2*V(s_1) + 0.2*V(s_2) + 0.6*V(s_3))$$

We can solve the Bellman equation using a special technique called dynamic programming.

4.2.2 SAMPLE WORKING CODE

```
from keras.layers import Conv2D, Dense, Flatten
class DQNAgent:
   def __init__(self, name, state_shape, n_actions, epsilon=0, reuse=False):
      """A simple DQN agent"""
       with tf.variable_scope(name, reuse=reuse):
         self.network = keras.models.Sequential()
         # Keras ignores the first dimension in the input_shape, which is the batch size.
         # So just use state_shape for the input shape
         self.network.add(Conv2D(32, (8, 8), strides=4, activation='relu',use_bias=False,
input_shape=state_shape,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
         self.network.add(Conv2D(64, (4, 4), strides=2,
activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
         self.network.add(Conv2D(64, (3, 3), strides=1,
activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
         self.network.add(Conv2D(1024, (7, 7), strides=1,
activation='relu',use_bias=False,kernel_initializer=tf.variance_scaling_initializer(scale=2)))
         self.network.add(Flatten())
         self.network.add(Dense(n_actions,
activation='linear',kernel_initializer=tf.variance_scaling_initializer(scale=2)))

         # prepare a graph for agent step
         self.state_t = tf.placeholder('float32', [None,] + list(state_shape))
         self.qvalues_t = self.get_symbolic_qvalues(self.state_t)
         self.weights = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=name)
         self.epsilon = epsilon

   def get_symbolic_qvalues(self, state_t):
      """takes agent's observation, returns qvalues. Both are tf Tensors"""
      qvalues = self.network(state_t)
      assert tf.is_numeric_tensor(qvalues) and qvalues.shape.ndims == 2, \
         "please return 2d tf tensor of qvalues [you got %s]" % repr(qvalues)
      assert int(qvalues.shape[1]) == n_actions
      return qvalues

    def get_qvalues(self, state_t):
      """Same as symbolic step except it operates on numpy arrays"""
      sess = tf.get_default_session()
      return sess.run(self.qvalues_t, {self.state_t: state_t})

   def sample_actions(self, qvalues):
      """pick actions given qvalues. Uses epsilon-greedy exploration strategy. """
        epsilon = self.epsilon
        batch_size, n_actions = qvalues.shape
        random_actions = np.random.choice(n_actions, size=batch_size)
        best_actions = qvalues.argmax(axis=-1)
        should_explore = np.random.choice([0, 1], batch_size, p = [1-epsilon, epsilon]
```

4.3 NEURAL NETWORK MODELS

4.3.1 DEEP NEURAL NETWORK

Deep neural networks work quite well for inferring the mapping implied by data, giving them the ability to predict an approximated output from an input that they never saw before. No longer do we need to store all state/action pair's Q-values, we can now model these mappings in a more general, less redundant way. These networks also automatically learn a set of internal features which are useful in complex non-linear mapping domains, such as image processing, releasing us from laborious feature handcrafting tasks.

This is perfect. We can now use a deep neural network to approximate the Q-function: the network would accept a state/action combination as input and would output the corresponding Q-value. Training-wise, we would need the network's Q-value output for a given state/action combo (obtained through a forward pass) and the target Q-value, which is calculated through the expression:

$r_{t+1}+\gamma \max_a Q(s_{t+1}, a)$ $r_{t+1}+\gamma \max_a Q(s_{t+1}, a)$ ..... (3)

With these two values, we can perform a gradient step on the squared difference between the target Q-value and the network's output.

This is perfect, but there is still room for improvement. Imagine we have 5 possible actions for any given state: when calculating the target Q-value, to get the optimal future value estimate (consequent state's maximum Q-value) we need to ask (forward pass) our neural network for a Q- value 5 times per learning step.

Another approach would be to feed in the game's screens and have the network output the Q-value for each possible action. This way, a single forward pass would output all the Q-values for a given state, translating into one forward pass per optimal future value estimate.

Q-learning and deep neural networks are the center pieces of a deep Q-network reinforcement learning agent and I think that by understanding them and how they fit together, it can be easier to picture how the algorithm works as a whole. In our network we used the convolution layer for the object detection in image and then fully connected layers for the feature extraction and action prediction.

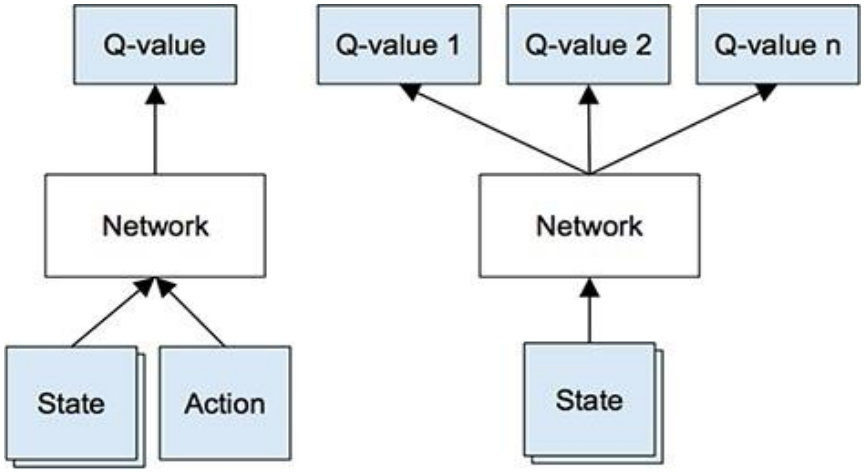

Figure 4.3.1.1: Left: Naive formulation of deep Q-network. Right: More optimized architecture of deep Q-network, used in DeepMind papers. (Image courtesy of Tambet Matiisen's "Demystifying Deep Reinforcement Learning")

4.3.2 DEEP RECURRENT Q-LEARNING FOR PARTIALLY OBSERVABLE MDPs

Matthew Hausknecht and Peter Stone investigates the effects of adding recurrency to a Deep Q-Network (DQN) by replacing the first post-convolutional fully-connected layer with a recurrent LSTM in their research. The resulting Deep Recurrent Q-Network (DRQN), although capable of seeing only a single frame at each timestep, successfully integrates information through time and replicates DQN's performance on standard Atari games and partially observed equivalents featuring flickering game screens. Additionally, when trained with partial observations and evaluated with incrementally more complete observations, DRQN's performance scales as a function of observability. Conversely, when trained with full observations and evaluated with partial observations, DRQN's performance degrades less than DQN's. Thus, given the same length of history, recurrency is a viable alternative to stacking a history of frames in the DQN's input layer and while recurrency confers no systematic advantage when learning to play the game, the recurrent net can better adapt at evaluation time if the quality of observations changes.

4.3.3 PLAYING ATARI WITH DEEP REINFORCEMENT LEARNING AND TRANSFER LEARNING

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martinpresented Riedmiller invented the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. They applied our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. They found that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

The popular Q-learning algorithm is known to overestimate action values under certain conditions. It was not previously known whether, in practice, such overestimations are common, whether they harm performance, and whether they can generally be prevented. Hado van Hasselt, Arthur Guez, David Silver answered all these questions affirmatively in their research. In particular, they first showed that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. They then showed that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. They proposed a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized, but that this also leads to much better performance on several games.

4.3.3.1 TRANSFER LEARNING

It is the process of using the knowledge learned in one process/activity and applying it to a different task. Let us take a small example, a player who is good at carroms can apply that knowledge in learning how to play a game of pool.

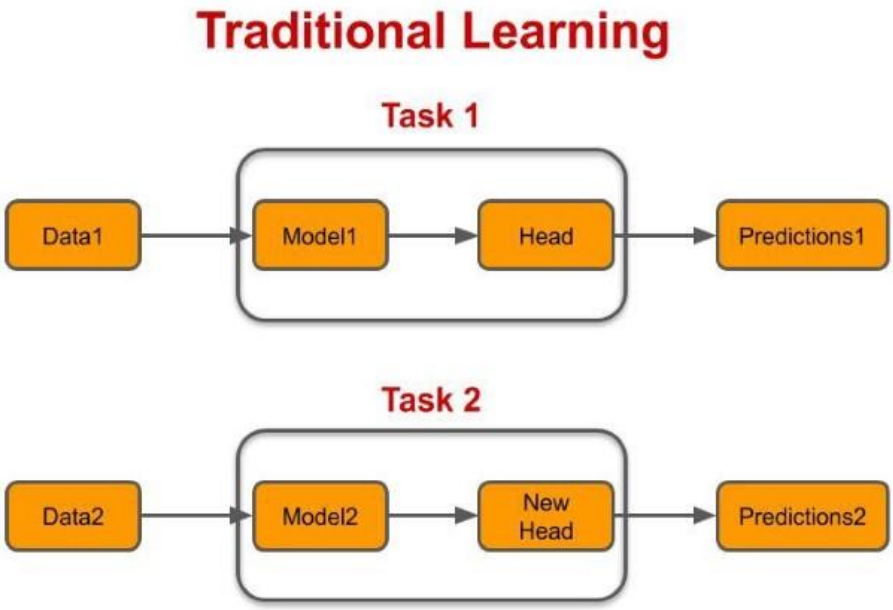

Figure 4.3.3.1: Traditional Learning

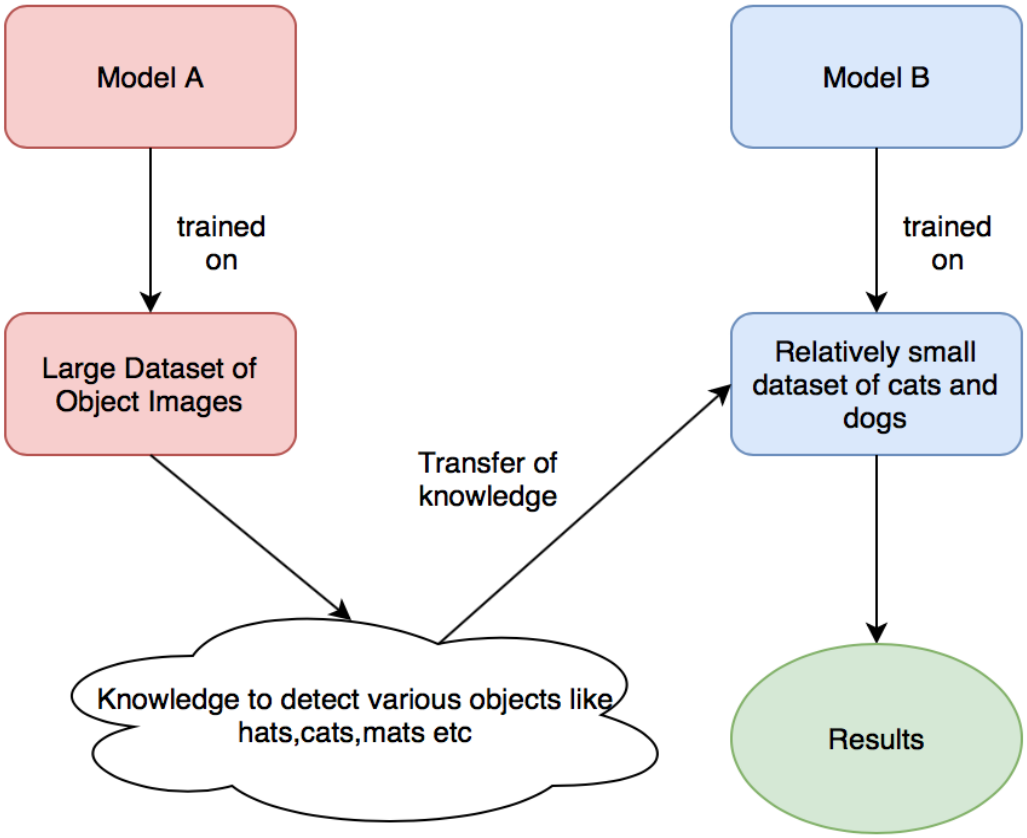

Figure 4.3.3.2: An illustration of Transfer Learning

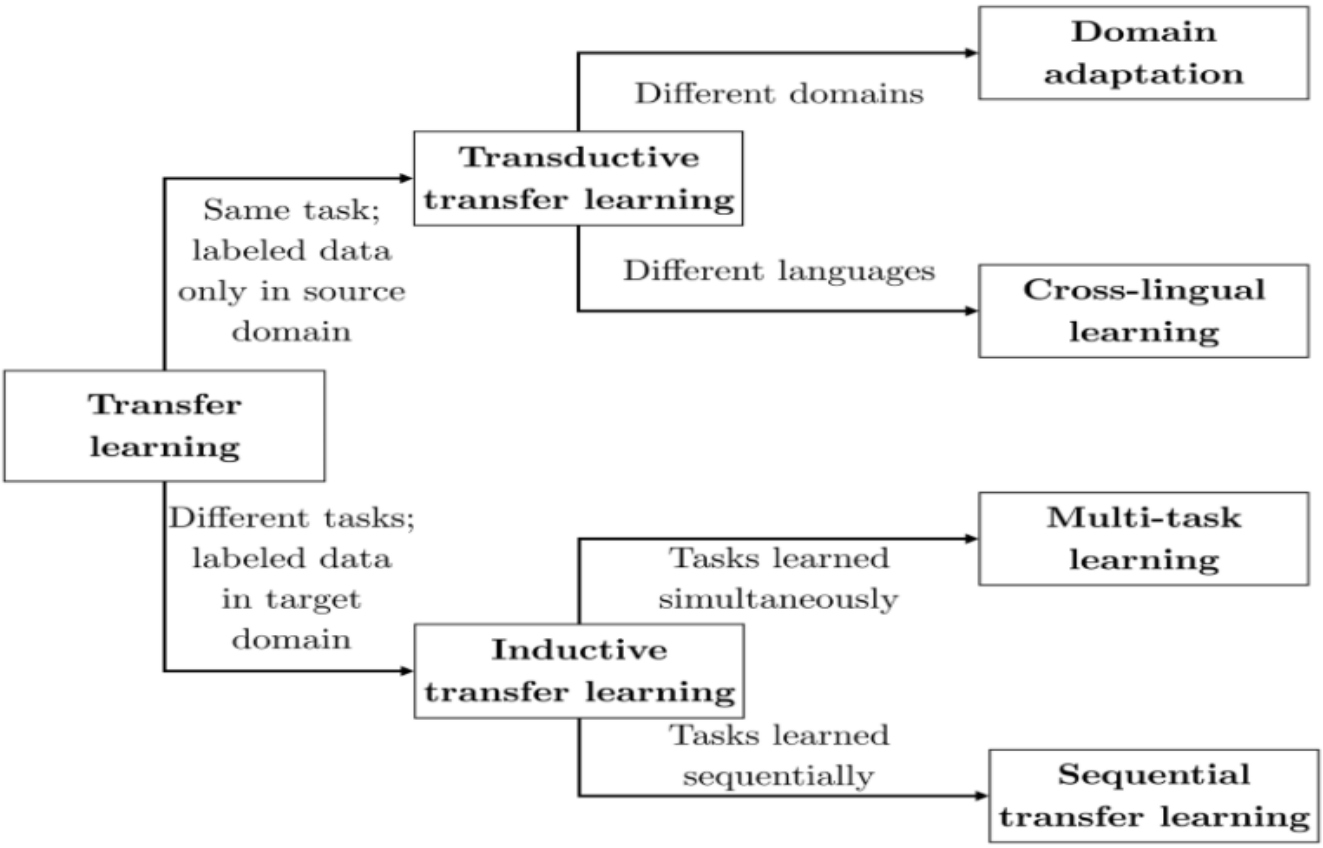

Figure 4.3.3.3: Types of Transfer Learning

4.4 FEASIBILITY ANALYSIS

The feasibility of the project is analyzed in this phase and business proposal is put forth with a very general plan for the project and some cost estimates. During system analysis the feasibility study of the proposed system is to be carried out. This is to ensure that the proposed system is not a burden to the company. For feasibility analysis, some understanding of the major requirements for the system is essential.

Three key considerations involved in the feasibility analysis are

- ECONOMICAL FEASIBILITY
- TECHNICAL FEASIBILITY

4.4.1 ECONOMICAL FEASIBILITY

This study is carried out to check the economic impact that the system will have on the organization. The amount of funds that the company can pour into the research and development of the system is limited. The expenditures must be justified.

Thus, the developed system as well within the budget and this was achieved because most of the technologies used are freely available. Only the customized products had to be purchased.

4.4.2 TECHNICAL FEASIBILITY

This study is carried out to check the technical feasibility, that is, the technical requirements of the system. Any system developed must not have a high demand on the available technical resources. This will lead to high demands on the available technical resources. This will lead to high demands being placed on the client. The developed system must have a modest requirement, as only minimal or null changes are required for implementing this system.

CHAPTER V

PREDICTIONS AND EXPERIMENTAL RESULTS

PREDICTION AND EXPERIMENTAL RESULTS

5.1 PREDICTIONS

All the initial population were created with the random game architecture.

Then we run all the games for 100000 iterations in all the neural nets described above and the results are -

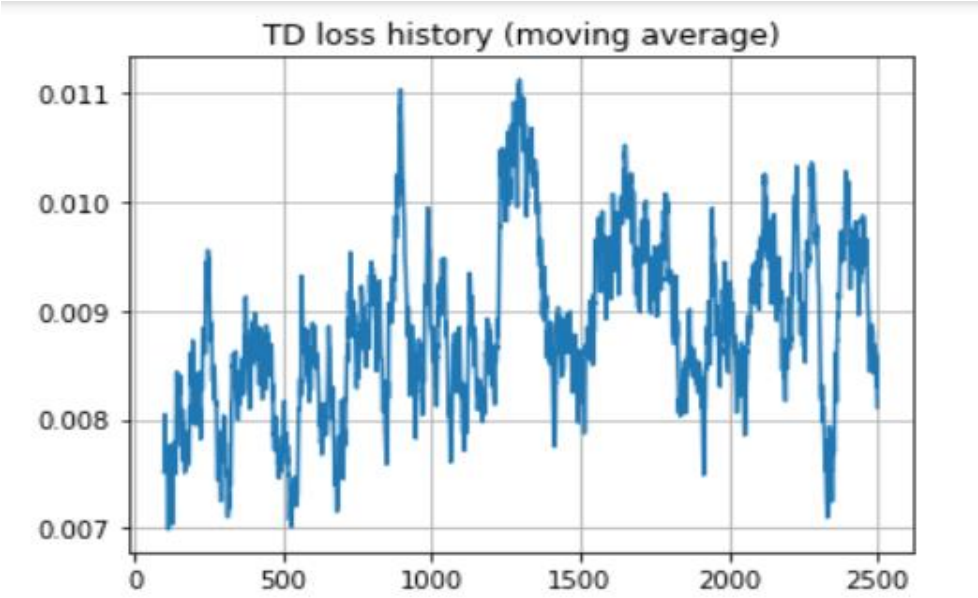

Figure 5.1.1: Breakout Environment Results (rewards/epochs)

In the breakout environment, there are only 4 actions, so the maximum score 358 was reached, in x-axis the epochs are training time considering the training data. Here the moving averages are displayed for the time stamp

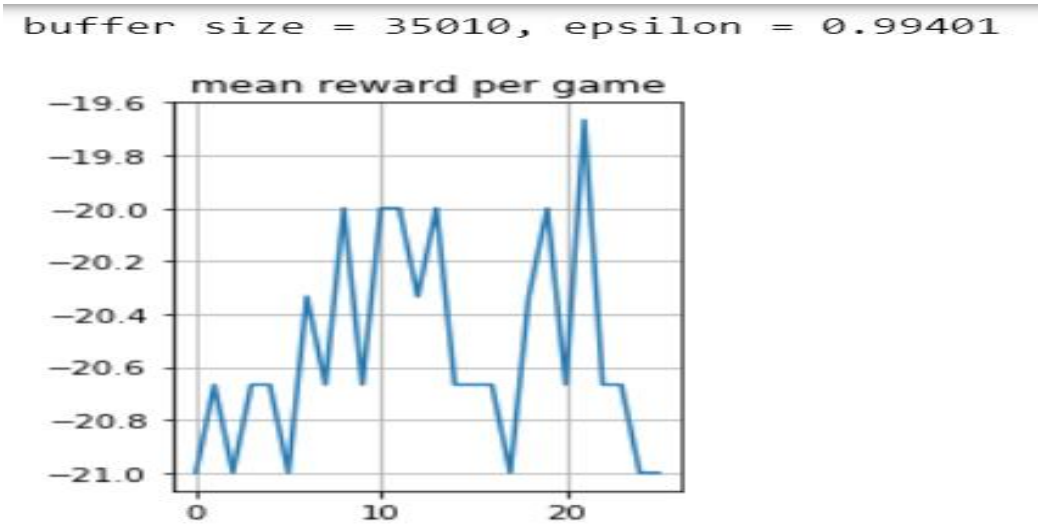

Figure 5.1.2: Breakout Mean Reward Environment Results (rewards/epochs)

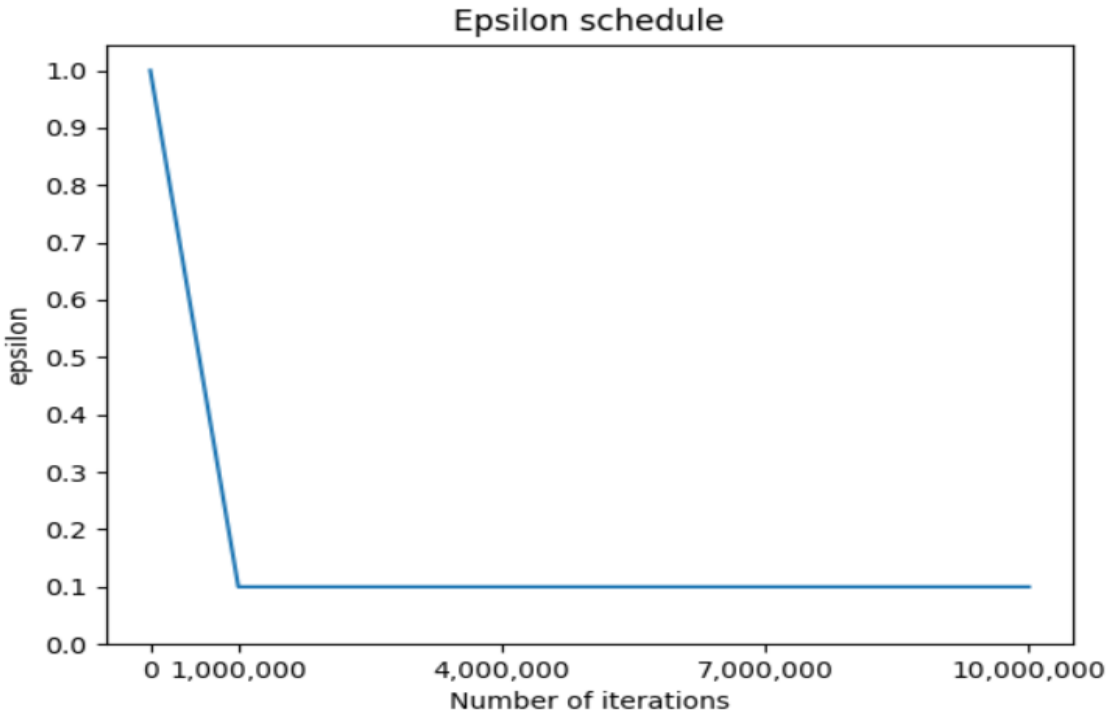

Figure 5.1.3: Environment Results (of Epsilon Greedy)

In the environment the agent will take the values from the neural network if the epsilon value is limited by a particular value and hence it will take values from the transferred domain values. The epsilon value gradually decreases as the number of iterations increases and thus getting values from the neural network rather than random or transferred values.

This shows the result of using the transfer learning. We got a bit better result than our own model here.

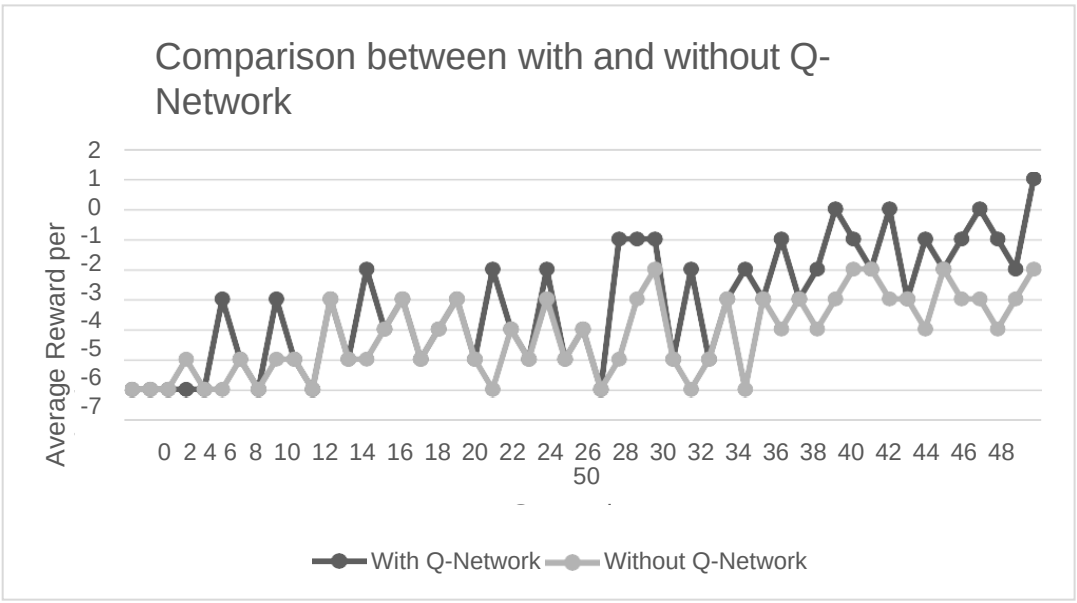

Figure 5.1.4: Comparison between with and without Q-Network

We can notice slightly better performance due to the transfer learning but it was not that worthy enough because still the game takes a lot of time to train

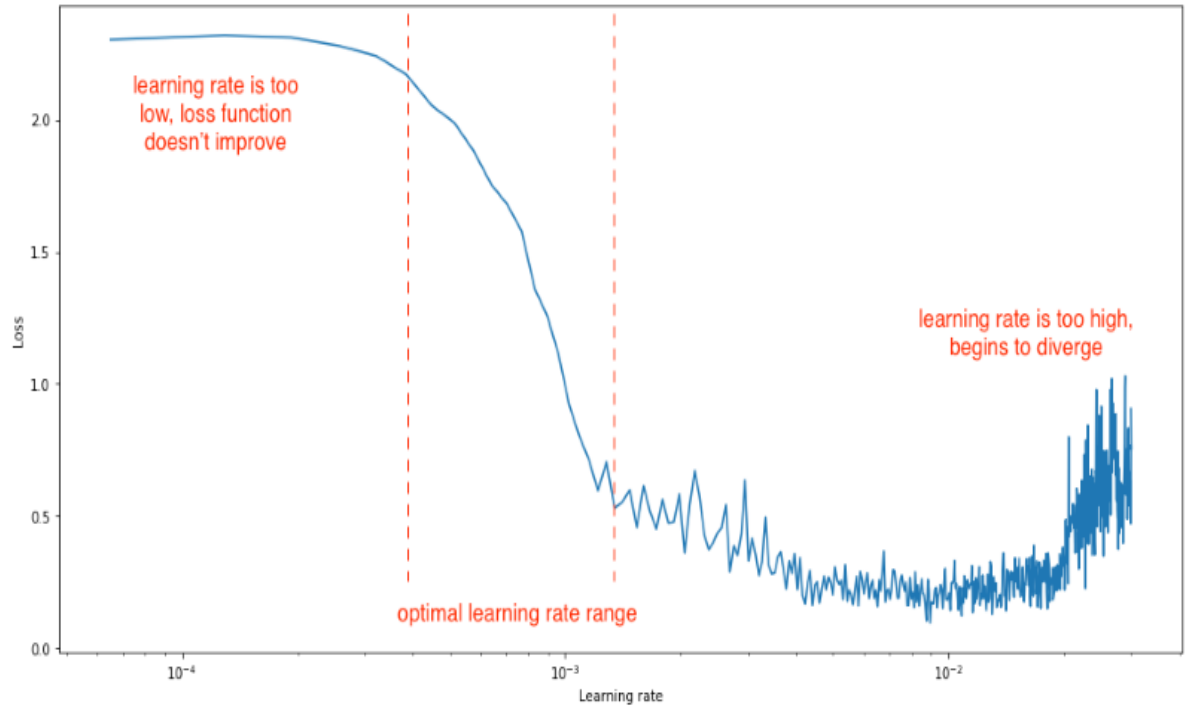

Figure 5.1.5: Learning Rate vs Loss

In the Space Invaders environment of OPENAI GYM, we used generations like the above breakout environment. Here the rewards are getting better but falling in some cases which we predict is a drawback of selecting only the scoring states. After training one game with random initialization we can acquire the weights of the game and then transfer the weights to another game

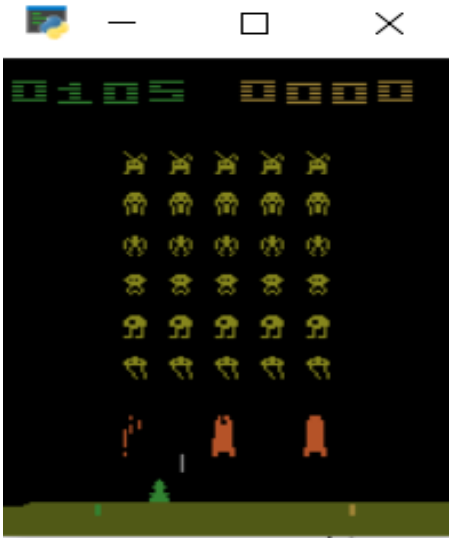

Figure 5.1.6: Space Invaders Environment Results (rewards/generations)

5.2 BATCH OF INPUTS

After getting some average results, we have tried to improve our model and approach. So, we developed a concept of batching the inputs for continuous learning. We took a batch of 4 continuous states, because a rewarding state comes after some series of actions. So, we saved the scoring state and also 3 states before reaching the scoring state and saved them to the game memory and the result was awesome. We reached the highest score possible in 500 states. The neural network has learned a lot by transfer learning and by continuous training. Mini batches allow the gradient descent to reach the global minima faster thus we can improve our learning over a period of time.

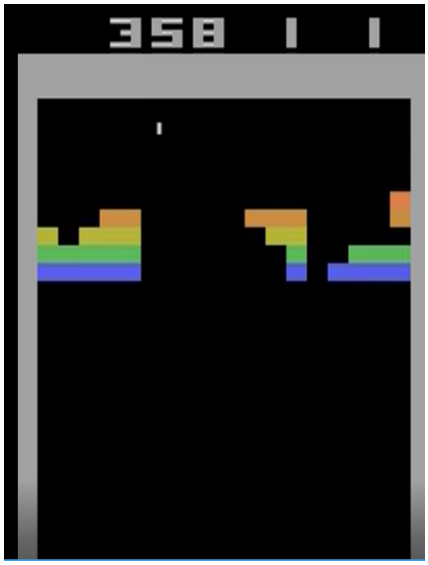

Figure 5.2.1: Atari game score of our model

Here we can see the highest possible score 358 is reached in 100000 iterations. We also changed some policies here. Like we used a series of 3 games for the average reward calculation and played random games on the capability of scoring. The main feasibility of learning is determining the best action possible by weights transfer of a similar game to another. However, our agent is based on the moving weighted averages rather than average because a single action will not have more effect on the other. Hence, we are checking the same for the other games like space invaders and pong. Space invaders achieve better results than pong with transfer learning. We can clearly see a massive improvement in result in terms of using the batch of inputs technique hence will check the results for space invaders also. The results shown for the initial stages of the learning at 1% of the total number of iterations and when the agent has completely learned everything.

There has been a lot of noise in the data while training and the graph depicts the results and noise. While training the data without transfer learning a lot of noise has been encountered as above but after transfer learning even though the rewards are not good the agent tries to maximize the discounted return. Thus, the agent has learned quickly using transfer learning. The phenomena of transfer learning are very useful for this kind of learning where the amount of training data is actually generated by itself by interacting with the environment. The weighted average will not have any effect on single iteration. It will take an epoch into consideration and each epoch is one pass through the training example. The agent takes values from the network by checking the epsilon value and making the value from the replay buffer in the initial stages.

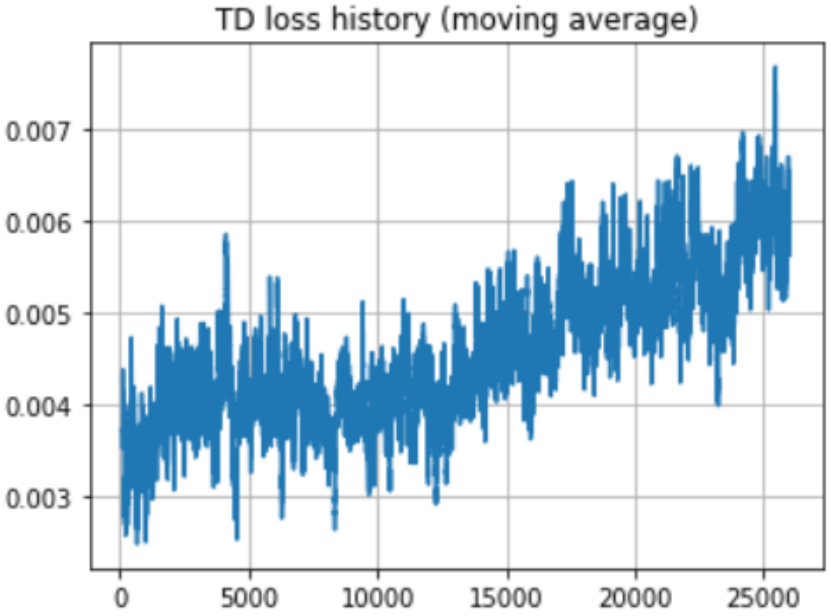

Figure 5.2.2: Space invaders without transfer learning

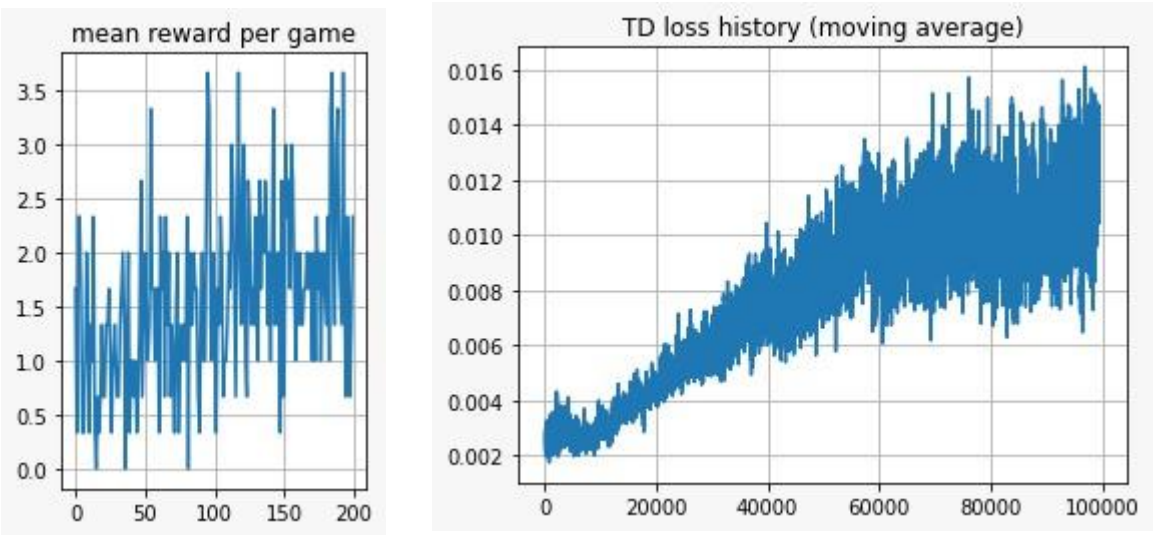

Figure 5.2.3: Space Invaders Environment Results using Batch Input Technique, Transfer Learning No Noise

5.3 CHALLENGES

We have faced several issues while solving the games. The first challenge was determining the neural network for the learning agent as well as the predicting agent. So, we have come up with good results with the above networks mentioned and used. Another challenge was the time to train the model. In most cases, it takes a good amount of time to train any model. So, we focused on short time learning proficiency. For environments like Atari, it takes a huge amount of time for processing the image and learning from it. An agent usually takes 20- 48 hours depending on the processing power and GPU utilization.

Another challenge was the dataset generation and image processing. With our idea of preprocessing we made the image data simple enough to train the agent, but the size of the saved data first started to exceed even several gigabytes. So, we came up with a solution of a queue for game memory, which also decreases the ram usage.

5.4 LEARNINGS

We have come with some ideas from our findings in this research. As long as we are trying to make a human-like network for the agent to learn like a human and act faster, we are quite successful in the case that our agent learned to play, though hasn't got the ability to play with utmost accuracy or beating other hardcoded bots. From the graphs we can see that the performance is not continuous and scores are not that high like a human. Maybe we are stuck in any local maxima in some cases. While training we have found that in some training sessions our agent was standing still in a place because it got the maximum score for consecutive episodes in the same place. However, in breakout and space invaders the timing is very important. One cannot score without timing it perfectly. So, our model learns from a very unstable randomized data, which results in an accuracy loss for the agent. If we can train the model with more continuous and successful batches of game memory, we think the model can perform far better. So, we got a major improvement in our model after trying the transfer learning technique, batching a number of sequences of states before a better reward is found.

CHAPTER VI

CONCLUSION AND FUTURE WORK

CONCLUSION AND FUTURE WORK

Conclusion

In this paper, we presented a model to solve 2D games using Deep reinforcement learning Q-Network and Transfer Learning approach. We introduced a new deep learning model to play games like Snakes, Space Invaders etc. Using raw pixels for input and also the provided framework by OpenAi. We also presented an Transfer Learning implementation method for finding a satisfied model for playing the game. After all, we used a batching technique for finding better results.

Future work

We now have the idea of a neural network for working with 2D games. In future we are planning to work with 3D games and the online ones like Dota2 or PUBG. In these environments we have to consider many aspects and features and there we may need to optimize each action in batches, like moving can be a batch of actions which will depend on seeing the objects or enemies, firing or attacking and other strategies a human player considers can also be taught to the agent.

We would also like to extend this idea to other real-time projects like the Self Driving Cars etc. Where it is very risky to do it manually and may cause severe damage to people and infrastructure.

CHAPTER VII

REFERENCES

REFERENCES

[1] En.wikipedia.org. (2017). Q-learning. [online] Available at: https://en.wikipedia.org/wiki/Q- learning [Accessed 13 Dec. 2017].

[2] Weber, Bruce (1997-05-18). "What Deep Blue Learned in Chess School". The New York Times. ISSN 0362-4331. Retrieved 2017-07-04.

[3] "IBM's Deep Blue beats chess champion Garry Kasparov in 1997". NY Daily News. Retrieved 2017-08-03.

[4] "OpenAI Gym", Gym.openai.com, 2017. [Online]. Available: https://gym.openai.com/. [Accessed: 23- Dec- 2017].

[5] Science, L. (2017). The Spooky Secret Behind Artificial Intelligence's Incredible Power. [online] Live Science. Available at: https://www.livescience.com/56415-neural-networks-mimic- the-laws-of-physics.html [Accessed 13 Dec. 2017].

[6] "Dota 2", OpenAI Blog, 2017. [Online]. Available: https://blog.openai.com/dota-2/. [Accessed: 13- Dec- 2017].

[7] D. Takahashi, "PwC: Game industry to grow nearly 5% annually through 2020", VentureBeat, 2017. [Online]. Available: https://venturebeat.com/2016/06/08/the-u-s-and-global-game-industrie s-will-grow-a-healthy-amount-by-2020-pwc-forecasts/. [Accessed: 23- Dec- 2017].

[8] "A Brief History of Computing", Alanturing.net, 2017. [Online]. Available: http:// www.alanturing.net/.[Accessed: 23- Dec- 2017].

[9] P. McCorduck, Machines who think. Natick, Mass: A.K. Peters, 2004.

[10] "Artificial Intelligence in Game Design", Ai-depot.com, 2017. [Online]. Available: http://ai-depot.com/GameAI/Design.html. [Accessed: 13- Dec- 2017].

[11] "Mario AI Competition 2009", Julian.togelius.com, 2017. [Online]. Available: http://julian.togelius.com/mariocompetition2009/. [Accessed: 23- Dec- 2017].

[12] "Artificial intelligence learns Mario level in just 34 attempts", Engadget, 2017. [Online].

Available:https://www.engadget.com/2015/06/17/super-mario-world-self-learning-ai/.[Accessed: 23- Dec- 2017].

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

[14] Schmidhuber, J., 2015. Deep learning in neural networks: An overview. Neural networks, 61, pp.85-117.

[15] Sutskever, I., Vinyals, O. and Le, Q.V., 2014. Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

[16] Le, Q.V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B. and Ng, A.Y., 2011, June. On optimization methods for deep learning. In Proceedings of the 28th International Conference on International Conference on Machine Learning (pp. 265-272). Omnipress.

[17] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[18] Hausknecht, M. and Stone, P., 2015. Deep recurrent q-learning for partially observable mdps. CoRR, abs/1507.06527.

[19] Van Hasselt, H., Guez, A. and Silver, D., 2016, February. Deep Reinforcement Learning with Double Q-Learning. In AAAI (pp. 2094-2100).

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

[21] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. Nature, 518(7540), pp.529-533.

[22] "Guest Post (Part I): Demystifying Deep Reinforcement Learning - Intel Nervana", Intel Nervana, 2017. [Online]. Available: https://www.intelnervana.com/demystifying-deep- reinforcement-learning/. [Accessed: 23- Dec- 2017].

[23] Yann.lecun.com. (2017). MNIST Demos on Yann LeCun's website. [online] Available at: http://yann.lecun.com/exdb/lenet/ [Accessed 13 Dec. 2017].