Self-Learning Game Bot using

Deep Reinforcement Learning

Thesis submitted in partial fulfillment of the requirement for the degree of

Bachelor of Computer Science and Engineering

Under the Supervision of

Moin Mostakim

By

Azizul Haque Ananto (14301050)

School of Engineering and Computer Science

December 2017

BRAC University, Dhaka, Bangladesh

DECLARATION

We, hereby declare that this thesis is based on the results found by ourselves. Materials of work found by other researcher are mentioned by reference. This Thesis, neither in whole or in part, has been previously submitted for any degree.

Signature of Supervisor	Signature of Author
Moin Mostakim	Azizul Haque Ananto

ACKNOWLEDGEMENTS

All thanks to Almighty ALLAH, the creator and the owner of this universe, the most merciful, beneficent and the most gracious, who provided us guidance, strength and abilities to complete this research.

I am especially thankful to Moin Mostakim, my thesis supervisor, for his help, guidance and support in completion of my project. He helped whenever I needed any kind of help and also with the suggestions. I am also thankful to the BRAC University Faculty Staffs of the Computer and Communication Engineering, who have been a light of guidance for us in the whole study period at BRAC University, particularly in building our base in education and enhancing our knowledge.

Finally, I would like to express my sincere gratefulness to my beloved parents and brother for their love and care. I am grateful to all of my friends who helped me directly or indirectly to complete my thesis.

CONTENTS

LIST OF FIGURES

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 network 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 it for our policy and usage. We applied our method on the python's 'PyGame Learning Environment' and some other classic control games. We found our method learns fast enough but not with best accuracy. Then we tried the batch of input method which results a high score for the Catcher environment. It produced better performance in terms of the learning speed and accuracy.

Chapter 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 the deep Q learning [1].

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 requires 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 [2] and it was literally unbeatable [3]. 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 (1^{st}, 2^{nd}, 3^{rd} 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 improve itself by playing after each stage with higher threshold than the previous one.

Our experiment took place in several domains: PyGame Learning Environment, Atari games of OpenAI environment [4], classic control games like Cartpole, Mountain Car. The Classic control games are comparatively easier ones to learn and perform with 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 successful reading the environment faster and act there, but the moves are not timed perfectly, reflects 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 still not cleared to any scholars [5]. There are researches 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 more complex scenario and environment which beat top class players of the game Dota2 [6]. These kinds of agents can be implemented in real life problem solving like automated managements 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 [7].

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 2

Background

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 bot. In case of two player games, the artificial intelligence was starting to develop from 1951 in the game of Nim and Checkers [8]. 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 task 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 [9]. From 2009 onwards, modern gaming introduces more technical bots who can react according to realistic markers, such as sound made by player and footprints they left behind [10]. Moreover, after a year Nintendo, a video game company started a competition called 'Mario Ai Championship' [11], where the task was to make a game bot that can play Mario, a popular Nintendo Console game. There the concept of new game bot arise when some competitors was able to make one bot that literally knows nothing about the game but completed the game after 34 tries [12] using neural network. Moreover, that bot was like human, cannot see beyond the computer screen. In 2014 Google acquired 'Deepmind', 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 [13].

2.2 Background studies

2.2.1 Deep learning in neural networks

Jürgen Schmidhuber talks about the deep learning policies and underlying mechanics in his research [14]. 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 Sequence to Sequence Learning with Neural Networks

Ilya Sutskever, Oriol Vinyals, Quoc V. Le presented a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure in their research [15]. Their method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Their main result is that on an English to French translation task from the WMT-14 dataset, the translations produced by the LSTM achieve a BLEU score of 34.8 on the entire test set, where the LSTM's BLEU score was penalized on out-of-vocabulary words. Additionally, the LSTM did not have difficulty on long sentences. For comparison, a phrase-based SMT system achieves a BLEU score of 33.3 on the same dataset. When they used the LSTM to rerank the 1000 hypotheses produced by the aforementioned SMT system, its BLEU score

increases to 36.5, which is close to the previous state of the art. The LSTM also learned sensible phrase and sentence representations that are sensitive to word order and are relatively invariant to the active and the passive voice. Finally, they found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier.

2.2.3 On Optimization Methods 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 [16], 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-theart result on MNIST among algorithms that do not use distortions or pretraining.

2.2.4 ImageNet Classification with Deep Convolutional Neural Networks

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 the 1000 different classes [17]. 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 work

2.3.1 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 [18]. 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.

2.3.2 Deep Reinforcement Learning with Double Q-Learning

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 [19]. 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.

2.3.3 Playing Atari with Deep Reinforcement 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 [20]. 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.

2.4 Generalizations of Machine Learning

Machine learning is about learning from data and making predictions and/or decisions. Usually we categorize machine learning as supervised, unsupervised, and reinforcement learning. In supervised learning, there are labeled data; in unsupervised learning, there are no labeled data; and in reinforcement learning, there are evaluative feedbacks, but no supervised signals. Classification and regression are two types of supervised learning problems, with categorical and numerical outputs respectively. Unsupervised learning attempts to extract information from data without labels, e.g., clustering and density estimation. Representation learning is a classical type of unsupervised learning. However, training feedforward networks or convolutional neural networks with supervised learning is a kind of representation learning. Representation learning finds a representation to preserve as much information about the original data as possible, at the same time, to keep the representation simpler or more accessible than the original data, with low-dimensional, sparse, and independent representations. Deep learning, or deep neural networks, is a particular machine learning scheme, usually for supervised or unsupervised learning, and can be integrated with reinforcement learning, usually as a function approximator. Supervised and unsupervised learning are usually one-shot, myopic, considering instant reward; while reinforcement learning is sequential, far-sighted, considering long-term accumulative reward. Machine learning is based on probability theory and statistics (Hastie et al., 2009) and optimization (Boyd and Vandenberghe, 2004), is the basis for big data, data science, data mining, information retrieval, etc. and becomes a critical ingredient for computer vision, natural language processing, robotics, etc. Reinforcement learning is kin to optimal control (Bertsekas, 2012), and operations research and management (Powell, 2011), and is also related to psychology and neuroscience (Sutton and Barto, 2017). Machine learning is a subset of artificial intelligence (AI), and is

evolving to be critical for all fields of AI. A machine learning algorithm is composed of a dataset, a cost/loss function, an optimization procedure, and a model (Goodfellow et al., 2016). A dataset is divided into non-overlapping training, validation, and testing subsets. A cost/loss function measures the model performance, e.g., with respect to accuracy, like mean square error in regression and classification error rate. Training error measures the error on the training data, minimizing which is an optimization problem. Generalization error, or test error, measures the error on new input data, which differentiates machine learning from optimization. A machine learning algorithm tries to make the training error, and the gap between training error and testing error small. A model is under-fitting if it cannot achieve a low training error; a model is over-fitting if the gap between training error and test error is large. A model's capacity measures the range of functions it can fit. VC dimension measures the capacity of a binary classifier. Occam's Razor states that, with the same expressiveness, simple models are preferred. Training error and generalization error versus model capacity usually form a U-shape relationship. We find the optimal capacity to achieve low training error and small gap between training error and generalization error. Bias measures the expected deviation of the estimator from the true value; while variance measures the deviation of the estimator from the expected value, or variance of the estimator. As model capacity increases, bias tends to decrease, while variance tends to increase, yielding another U-shape relationship between generalization error versus model capacity. We try to find the optimal capacity point, of which under-fitting occurs on the left and over-fitting occurs on the right. Regularization add a penalty term to the cost function, to reduce the generalization error, but not training error. No free lunch theorem states that there is no universally best model, or best regularizor. An implication is that deep learning may not be the best model for some problems. There are model parameters, and hyperparameters for model capacity and

regularization. Cross-validation is used to tune hyperparameters, to strike a balance between bias and variance, and to select the optimal model. 7 Maximum likelihood estimation (MLE) is a common approach to derive good estimation of parameters. For issues like numerical underflow, the product in MLE is converted to summation to obtain negative log-likelihood (NLL). MLE is equivalent to minimizing KL divergence, the dissimilarity between the empirical distribution defined by the training data and the model distribution. Minimizing KL divergence between two distributions corresponds to minimizing the cross-entropy between the distributions. In short, maximization of likelihood becomes minimization of the negative loglikelihood (NLL), or equivalently, minimization of cross entropy. Gradient descent is a common approach to solve optimization problems. Stochastic gradient descent extends gradient descent by working with a single sample each time, and usually with minibatches. Importance sampling is a technique to estimate properties of a particular distribution, by samples from a different distribution, to lower the variance of the estimation, or when sampling from the distribution of interest is difficult. Frequentist statistics estimates a single value, and characterizes variance by confidence interval; Bayesian statistics considers the distribution of an estimate when making predictions and decisions.

2.5 Generalization 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 very

long time and the gradient may vanish. Long short term memory networks (LSTM) and gated recurrent unit (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 inputs 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 hierarch 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. The notion of end-to-end training refers to that a learning model uses raw inputs without manual feature engineering to generate outputs, e.g., AlexNet (Krizhevsky et al., 2012) with raw pixels for 8 image classifications, Seq2Seq (Sutskever et al., 2014) with raw sentences for machine translation, and DQN (Mnih et al., 2015) with raw pixels and score to play games.

Chapter 3

Proposed Model

3.1 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 is 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 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 it (there may be some instructions about the game, but for now we are ignoring that because of the complications of NLP) 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 convolution network for the input image and then 20480 fully connected neurons with 4 hidden layers. We have also used the Alexnet implemented by Alex Krizhevsky in [17] which also gave us a good result for defining images. Figure 3.1.1 is the whole idea in short.

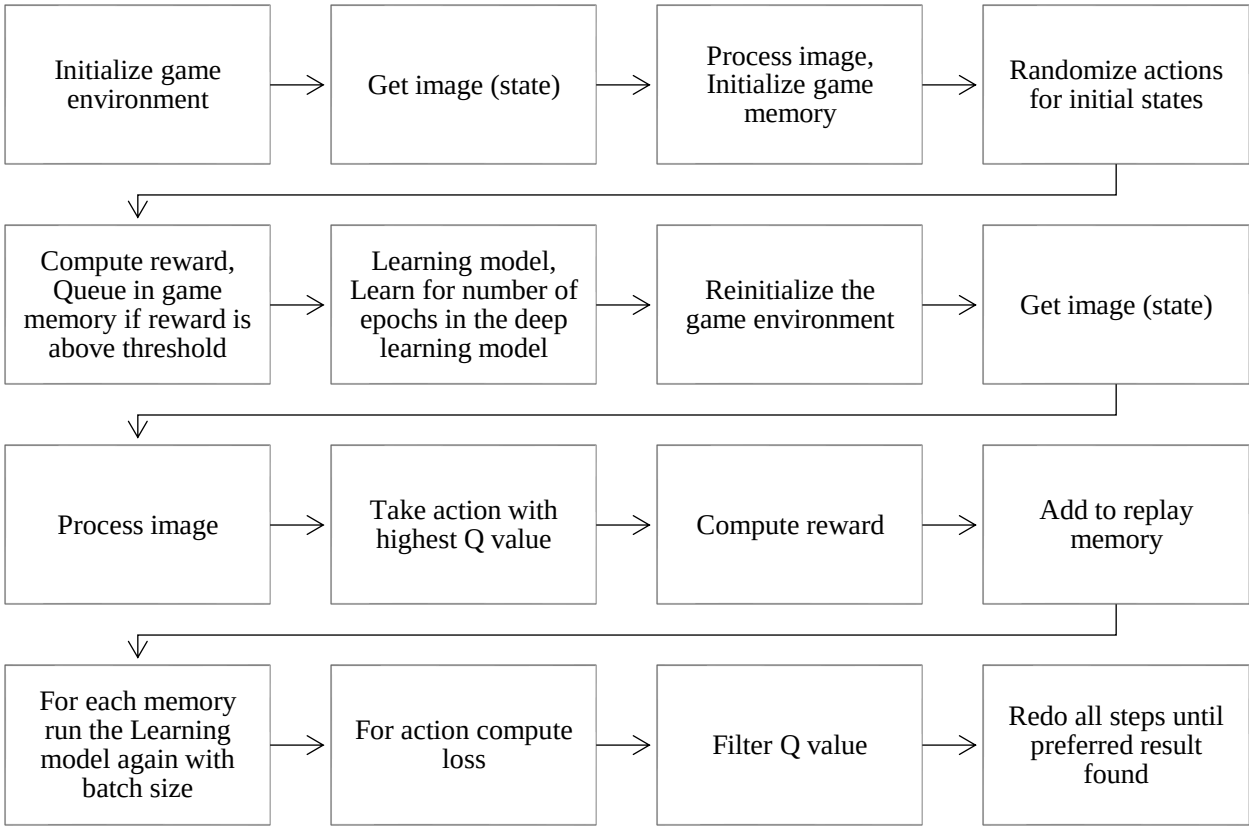

Figure 3.1.1: Proposed model

3.2 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 some cases the game memory, for where the agent is and the opponents are. And for other cases the game environments are setup with the help of simulators and PyGame engine. Figure 3.2.1 shows some examples of game environments we have used.

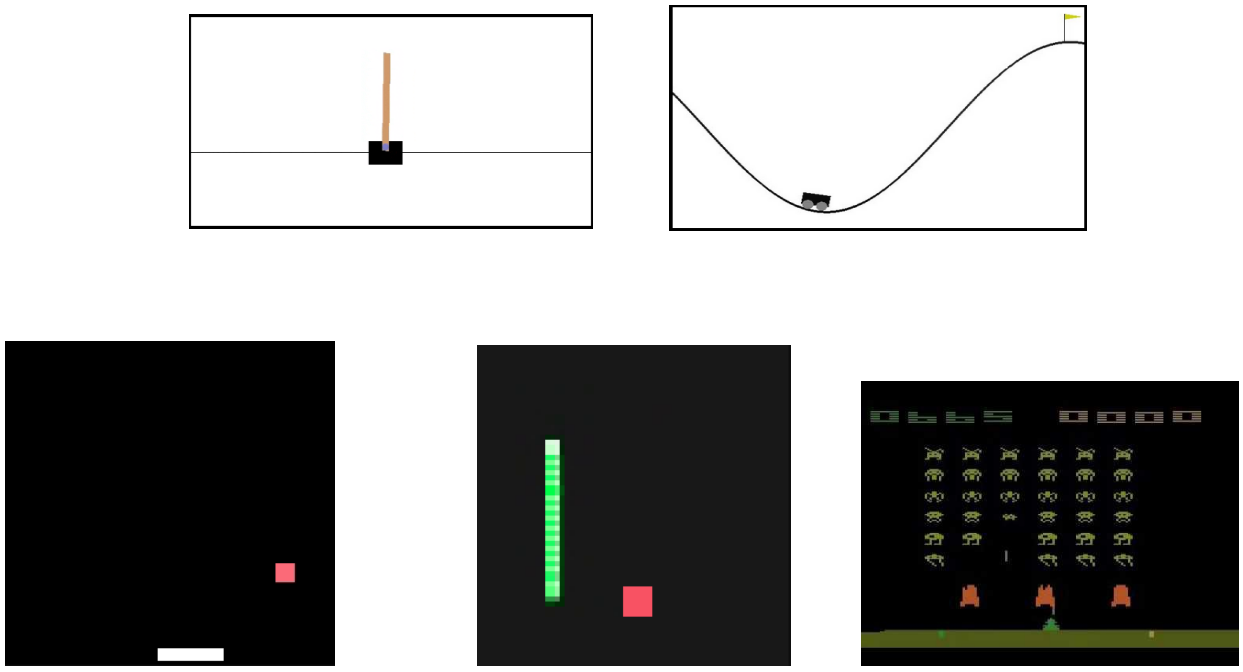

Figure 3.2.1: Cart pole, Mountain car, Catcher, Snake, Space invaders

3.3 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 other 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.

At first, we convert the RGB image to 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 3.3.1 shows the whole process.

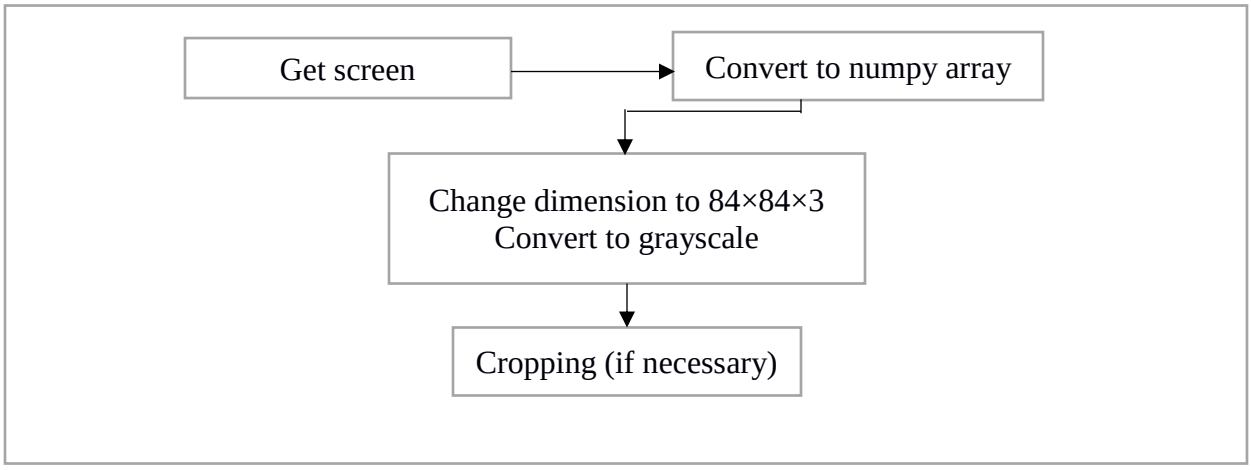

Figure 3.3.1: Preprocessing

3.4 Random playing (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 it 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 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 case of the OpenAI environment, they give us the reward that is showed in the screen. Now we need to save the state (processed image) with the action taken, where the action is converted to 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 3.4.1 is the whole idea.

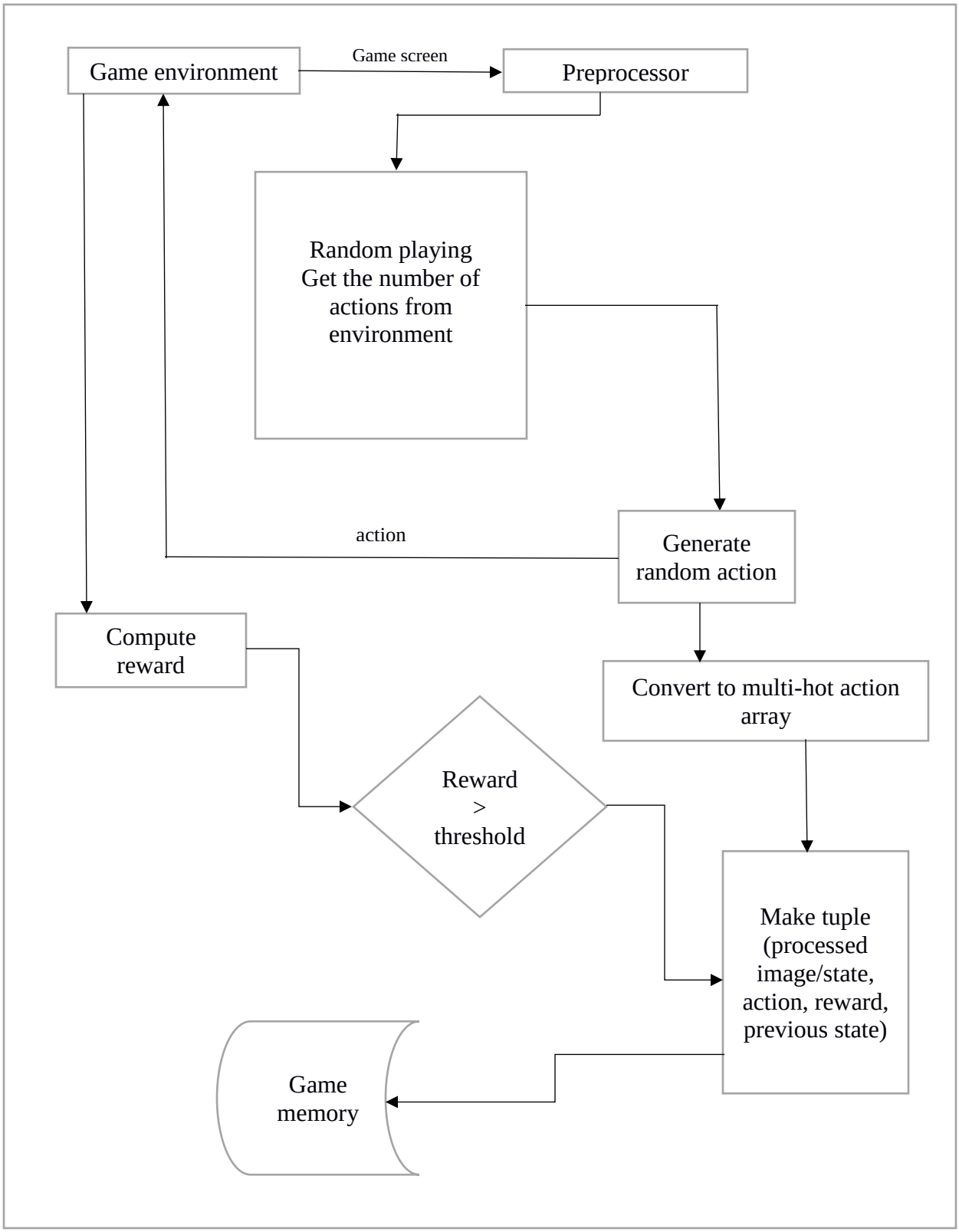

Figure 3.4.1: Random playing

3.5 The Q function

For this project, we adopted a deep reinforcement approach very similar to the one used [20] and [21]. In this reinforcement learning approach, our neural network is used to approximate a function

$$Q*(s, a) = \max_{\pi} E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s, a_t = a, \pi] \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Where s_t is the state at time t, at is the action taken at time t, π is a policy function indicating what action to take given the current state and is followed at every step from t + 1 onwards, r_t is the reward obtained by taking action at given state s_t, and γ is a discount factor. Thus, the Q function represents the highest expected discounted sum of future rewards achievable by following a fixed policy from states to actions. Since we know that the terms $\gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$ represent γ times the highest expected discounted sum of future rewards from the state at time t + 1 and that this value does not depend on s_t, at, or r_t but simply on the state at time t+ 1, the Q function obeys the Bellman equation and can be simplified as

$$Q*(s, a) = E_{s`}[r + \gamma \max_{a`} Q*(s`, a`) \mid s, a] \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Where s` is the state (or possible states, if the process is non-deterministic) that results from taking action a given state s, and a` is the action taken given state s`.

3.6 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:

$$rt+1+\gamma maxaQ(st+1,a)rt+1+\gamma max_{a}Q(st+1,a) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots .(3)$$

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 [20] 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.

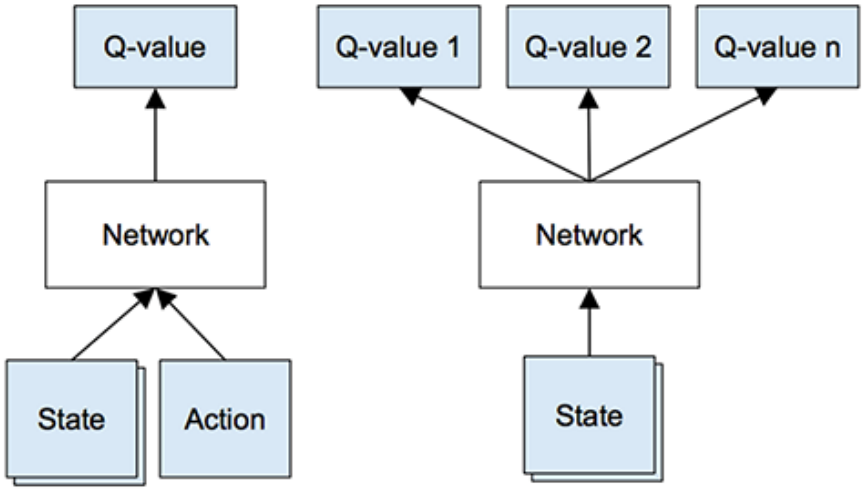

Figure 3.6.1: Left: Naive formulation of deep Q-network. Right: More optimized architecture of deep Q-network, used in DeepMind papers. (Image courtesy of Tambet Matiisen's "Demystifying Deep Reinforcement Learning" [22])

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.

3.6.1 Convolution layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

Overview and intuition without brain stuff: Lets first discuss what the CONV layer computes without brain/neuron analogies. The CONV layer’s 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 (e.g. 12 filters), 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 connectivities, 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.

3.6.2 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 [23]. At that time the LeNet architecture was used mainly for character recognition tasks such as reading zip codes, digits, etc.

Below, Figure 3.6.2.1 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.

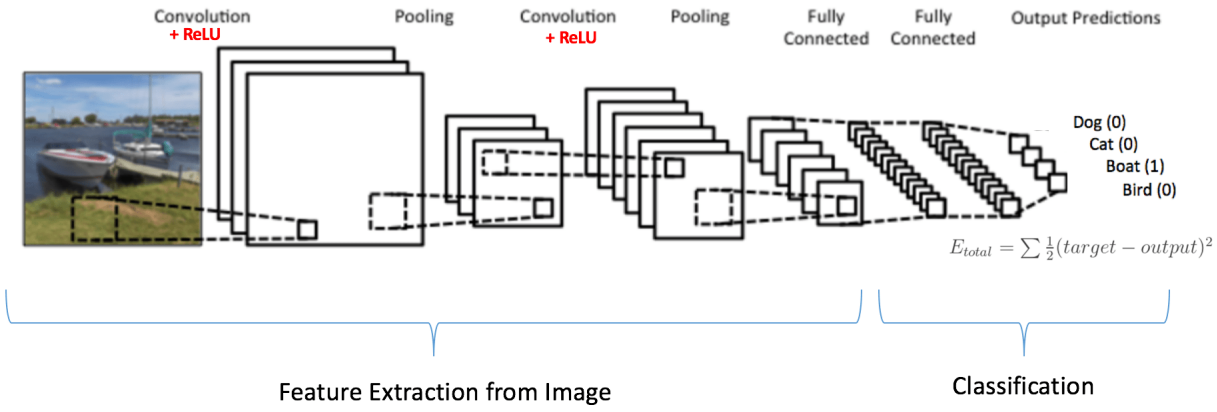

Figure 3.6.2.1: LeNet architecture [23]

3.6.3 The AlexNet Architecture

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 (this is the distance between the receptive field centers of neighboring neurons in a kernel map). 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 $5 \times 5 \times 48$. 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 $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

3.7 Training

Since the Q function obeys the Bellman equation and has the form of Equation 2, we trained our neural network to match:

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

Where Q` is a cached version of the neural network. This cached version updates much more infrequently (once per 1000 iterations instead of every iteration) 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 of 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 a image data. So they are super-fast to train and accuracy is much better than the other models.

For the PyGame environment we are using the learning model with the ConvNet, 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 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.

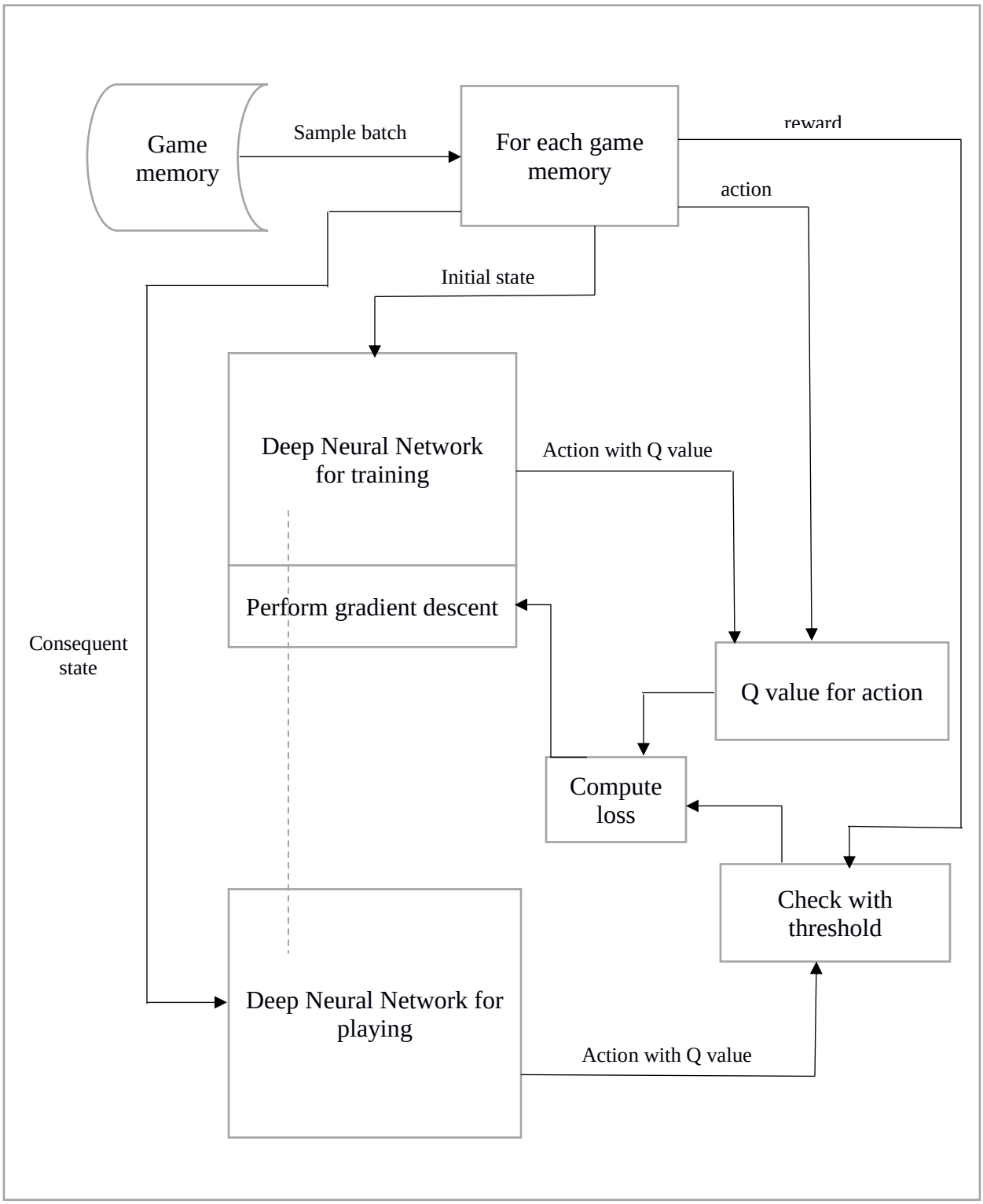

Figure 3.7.1: Training and playing process

3.7.1 Experience Replay

We used a technique called experience replay to create a “game memory” for our neural network to train on. While having the network play games, we saved every “experience”, consisting of a (state, action, reward, next state) tuple, that the network had encountered, and we sample from this full breadth of experience when training the network, not just from the most recent game or frames. This complements the epsilon-greedy algorithm in helping the network continue to search the state space and avoid getting trapped in local optima; since the policy evolves over time, earlier frames will have a different distribution of actions for a given state than will later frames, so the network will still be able to train on off-policy results. Since the network is training on its estimate of the value of the next state, experiences from old policies can be used for training.

Chapter 4

Experimental setup and result analysis

4.1 Initialization of Dataset

First, we play random games as described in 3.4.

```
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 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])
```

4.2 Neural Network Models

For different games we use different networks for training.

4.2.1 Raw input fully-connected model

First, we are using the network described in 3.7

```
neural_network_model():
  input_data
  fully_connected(128, activation='relu')
  dropout(0.8)
  fully_connected(256, activation='relu')
  dropout(0.8)
  fully_connected(512, activation='relu')
  dropout(0.8)
  fully_connected(256, activation='relu')
  dropout(0.8)
  fully_connected(128, activation='relu')
  dropout(0.8)
  fully_connected(numberOfActions, activation='softmax')
  regression()
  model()
```

We used that model for cart pole and mountain car environment. Having a number of 3 actions, we tuned 3 hidden layers for better performance.

4.2.2 AlexNet model

This model has been described in 3.6.3

```
alexnet():
    input_data
    conv_2d(96, 11, strides=4, activation='relu')
    max_pooling (3, strides=2)
    local_response_normalization()
    conv_2d(256, 5, activation='relu')
    max_pooling (3, strides=2)
    local_response_normalization(network)
    conv_2d(384, 3, activation='relu')
    conv_2d(384, 3, activation='relu')
    conv_2d(256, 3, activation='relu')
    max_pooling(3, strides=2)
    local_response_normalization()
    fully_connected(4096, activation='tanh')
    dropout(0.5)
    fully_connected(4096, activation='tanh')
    dropout(0.5)
    fully_connected(numberOfActions, activation='softmax')
    regression()
    model()
```

We used this model for the Catcher, Snake in PyGame environment.

4.2.3 Image input ConvNet and fully-connected model

This model was described in 4.7

```
neural_network():
    input_data
    conv_2d(32, 8, strides=4, activation='relu')
    conv_2d(64, 4, strides=2, activation='relu')
    fully_connected(256, activation='relu')
    dropout(0.5)
    fully_connected(256, activation='relu')
    dropout(0.5)
    fully_connected(numberOfActions, activation='softmax')
    regression()
    model()
```

This model is also used for the PyGame environment. We featured out the objects 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.

4.3 Predictions and experimental results

Datasets are created with the random game architecture in Figure 3.4.1

Below is the collected initial population of Catcher game in PyGame Learning Environment –

```
episodes:1/1000, score:-8.0
episodes:2/1000, score:-6.0
episodes:3/1000, score:-7.0
episodes:4/1000, score:-8.0
episodes:5/1000, score:-8.0
……
episodes:995/1000, score:-8.0
episodes:996/1000, score:-8.0
episodes:997/1000, score:-7.0
episodes:998/1000, score:-8.0
episodes:999/1000, score:-5.0
Average accepted score: -2.3461538461538463
Median score for accepted scores: -2.5
Counter({-3.0: 13, -2.0: 9, -1.0: 4})
```

All the initial population is created in the same way.

Then we run all the games for 50 epochs in all the neural nets described above and the results are -

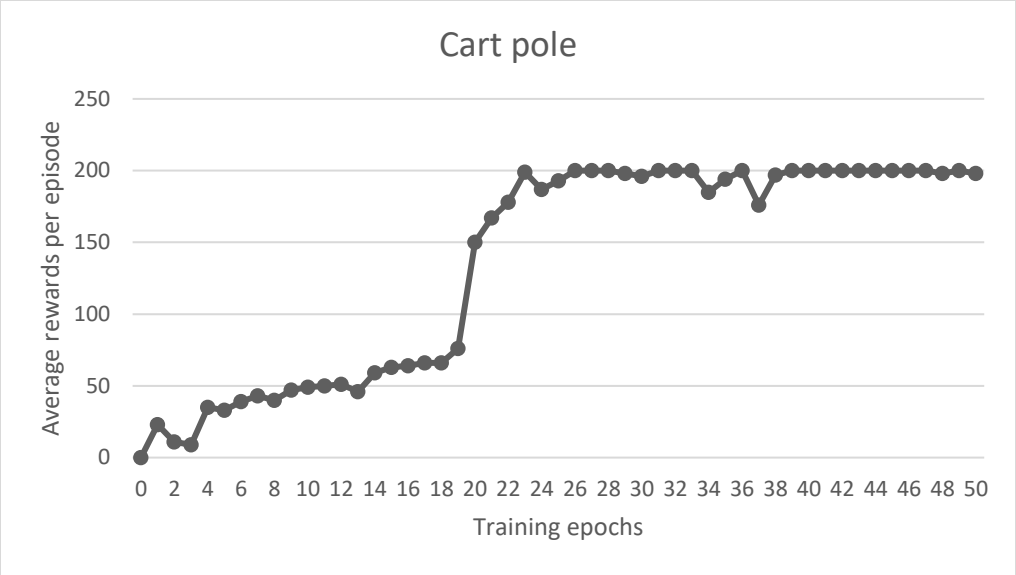

Figure 4.3.1: Cart pole environment results (rewards/epochs)

In the cart pole environment there are only 2 actions, so the maximum score 200 was easy to reach, in the x-axis the epochs are training time considering the training data. Here the training data is less than 1000 and it took only few minutes to reach the maximum score.

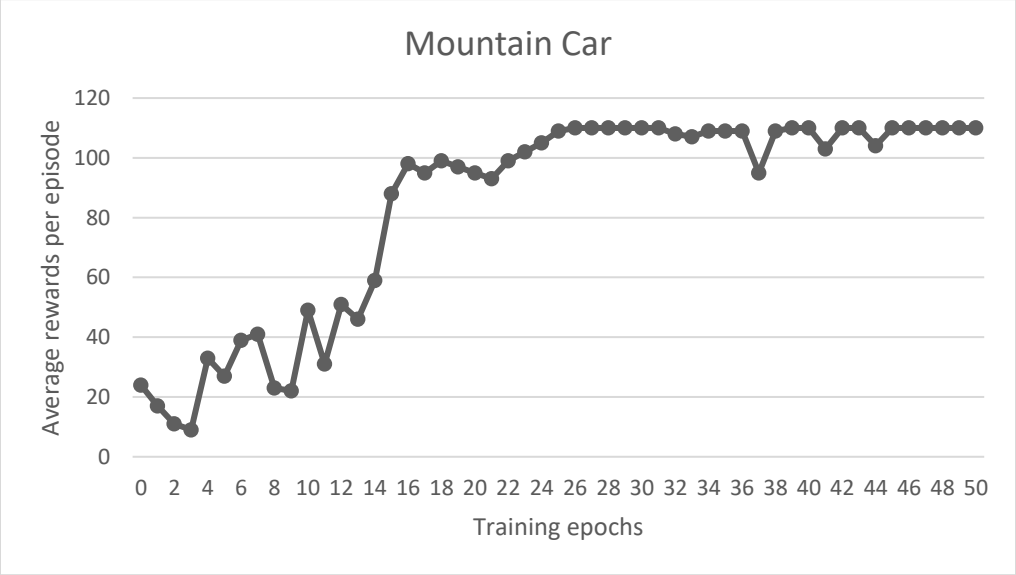

Figure 4.3.2: Mountain car environment results (rewards/epochs)

The mountain car environment is also like the cart pole, 3 actions and simple states. Same as the cart pole the training data size is less than 1000 and it was fast to reach the maximum score.

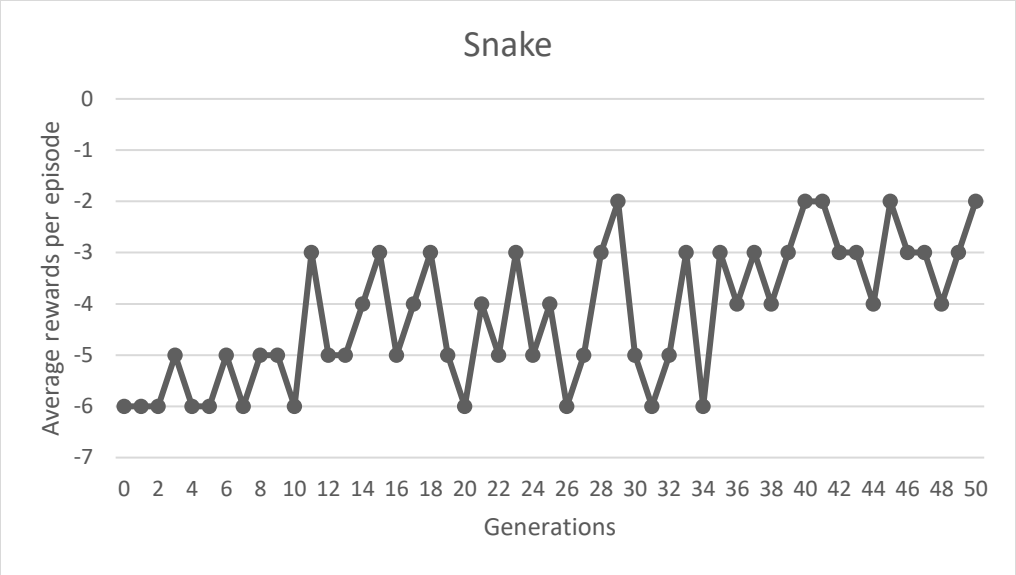

Figure 4.3.3: Snake environment results (rewards/generations)

In the snake environment of PLE, we used generations for presenting the epochs/time. Here each generation constituted with the random games and training epochs. Every time we need play random games based on the scoring ability described in 4.7.1.

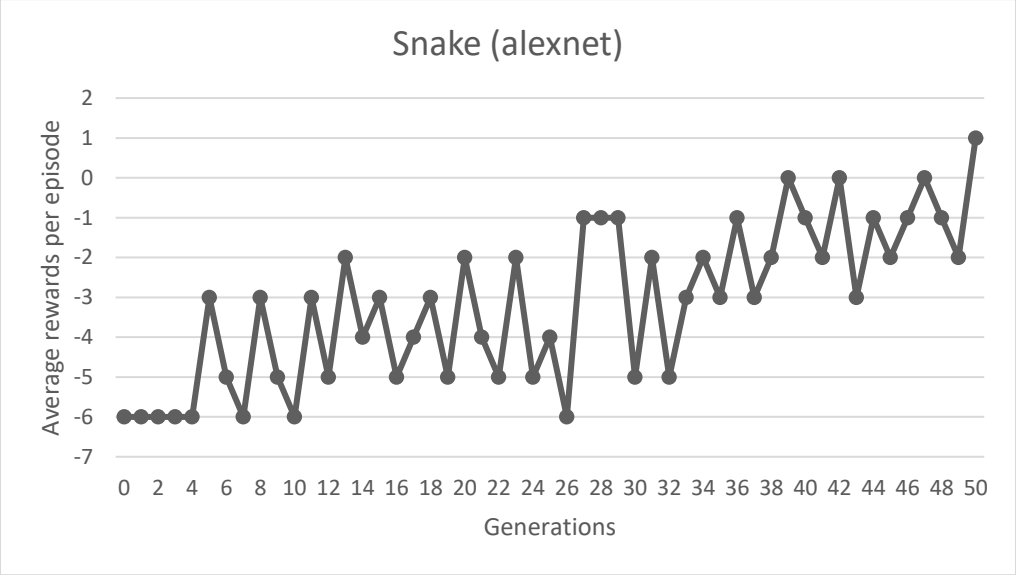

Figure 4.3.4: Snake environment results using AlexNet, slight better results

This shows the result of using the AlexNet described in 4.6.3. We got a bit better result than our own model here.

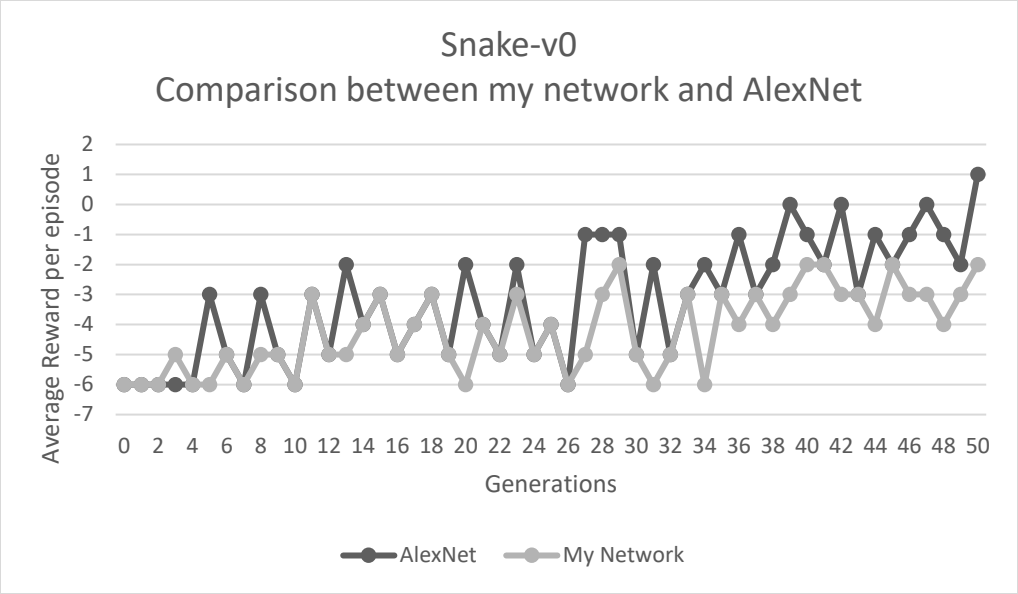

Figure 4.3.5: Comparison between my network and AlexNet in Snake-v0 environment

We can notice slight better performance in the AlexNet but it was not that worthy enough because AlexNet is computational heavy and took a lot of time to train.

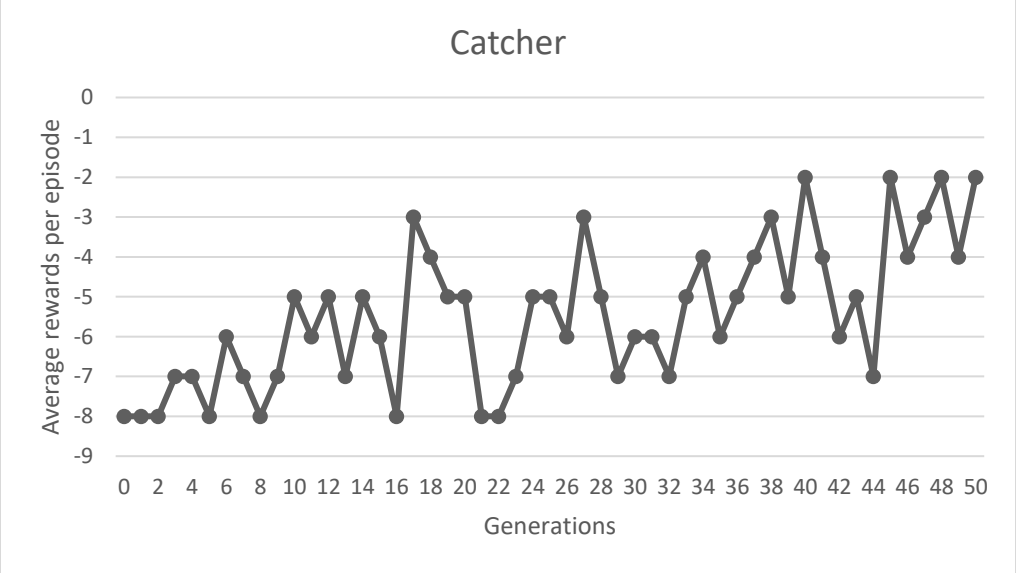

Figure 4.3.6: Catcher environment results (rewards/generations)

In the catcher environment of PLE, we used generations like above snake environment. Here the rewards are getting better but falling in some cases which we predict a drawback of selecting only the scoring states.

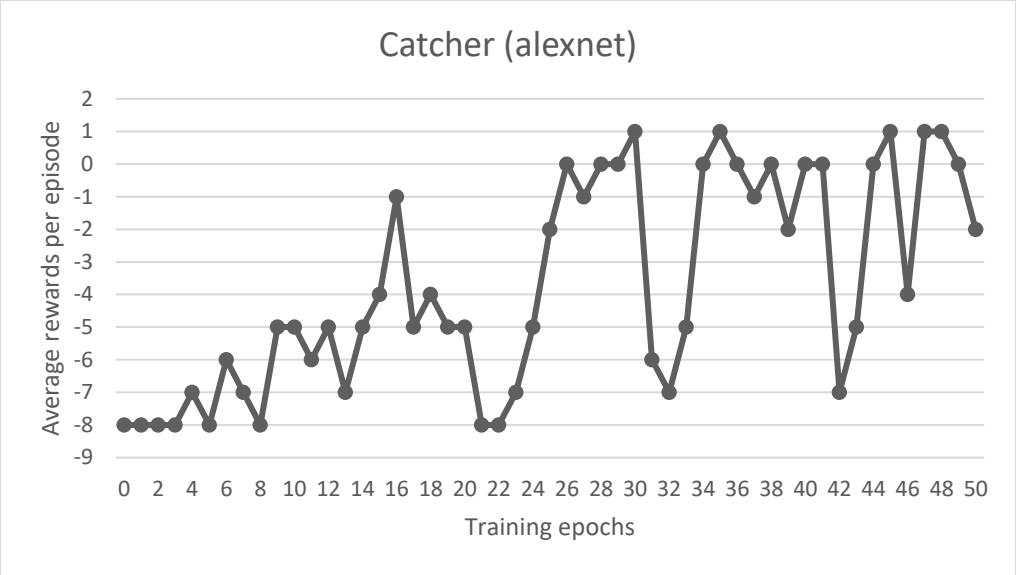

Figure 4.3.7: Catcher environment results using AlexNet, slight better results

This is the result of using AlexNet, and we get a bit better result than our model. But it took more time to run because the AlexNet is computational heavy.

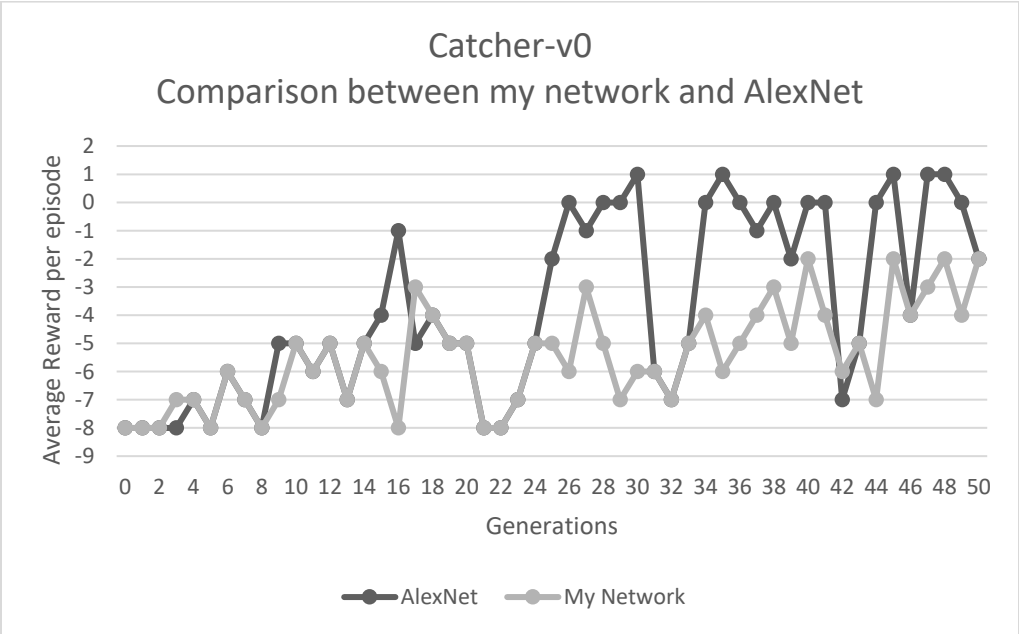

Figure 4.3.8: Comparison between my network and AlexNet in Catcher-v0 environment

In case of Catcher environment AlexNet scored better than my network though dropping in some cases to the same score as my environment.

4.3.1 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 a 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 500states. The graphs are given below –

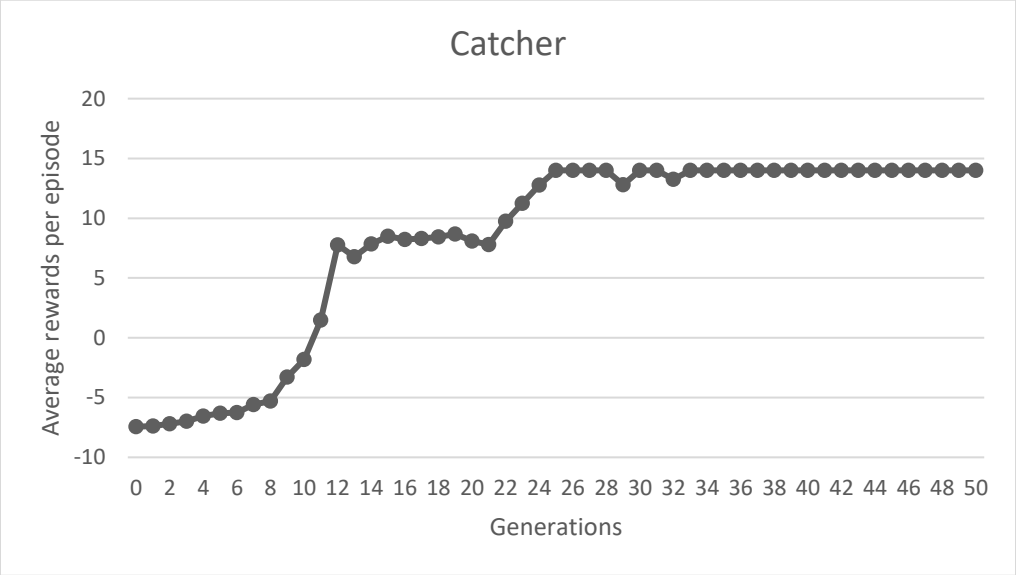

Figure 4.3.1.1: Catcher environment results using batch input technic, much better results

Here we can see the highest possible score 14 is reached in 26 generations. We also changed some policies here. Like we used a series of 10 games for the average reward calculation and played random games on the capability of scoring.

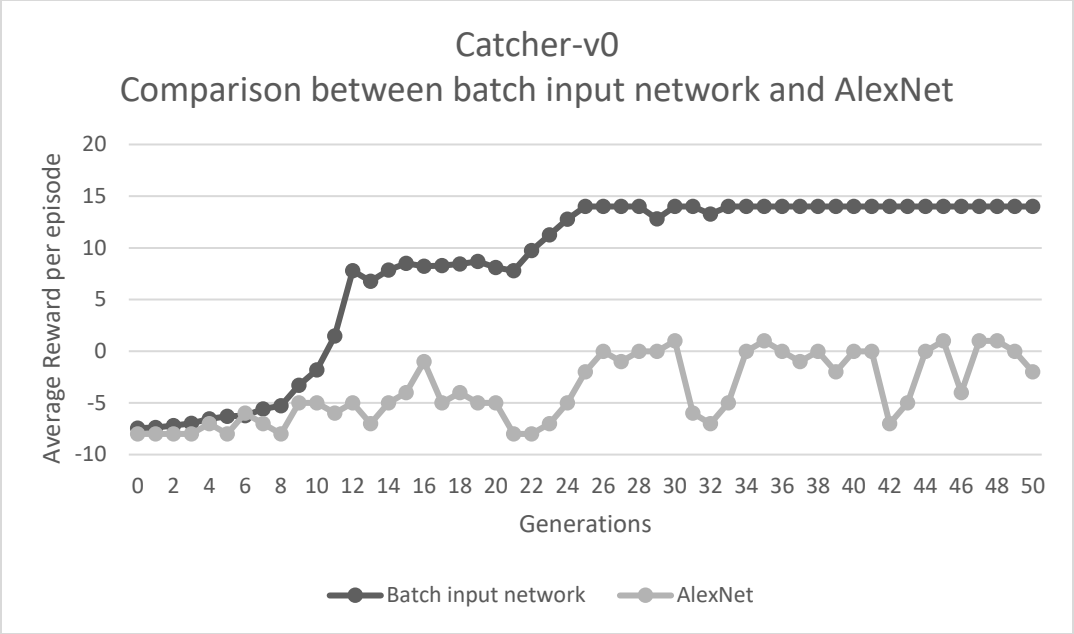

Figure 4.3.1.2: Improvement using the batch of input technique in Catcher environment

We can clearly see a massive improvement in result in terms of using the batch of inputs technique.

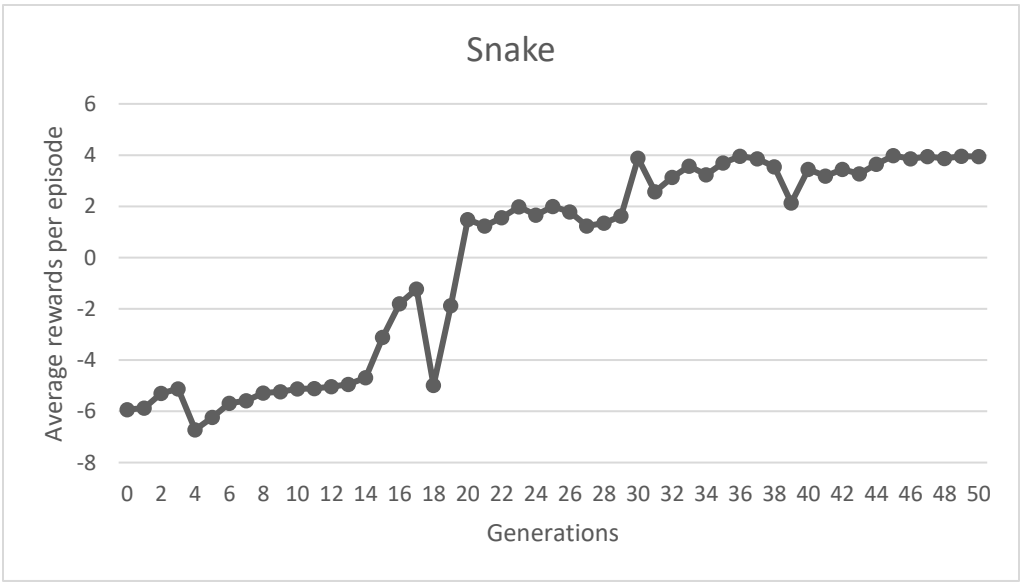

Figure 4.3.1.3: Snake environment results using batch input technic, much better results

In case of the snake environment we were not able reach the high score within 50 generations, maybe we need further training (more generations). That is because the agent need to recognize

the worm/apple that it needs to eat for score, which is not a simple task. So, we couldn't reach the high score possible in 500 states, but we were able to improve score and policy.

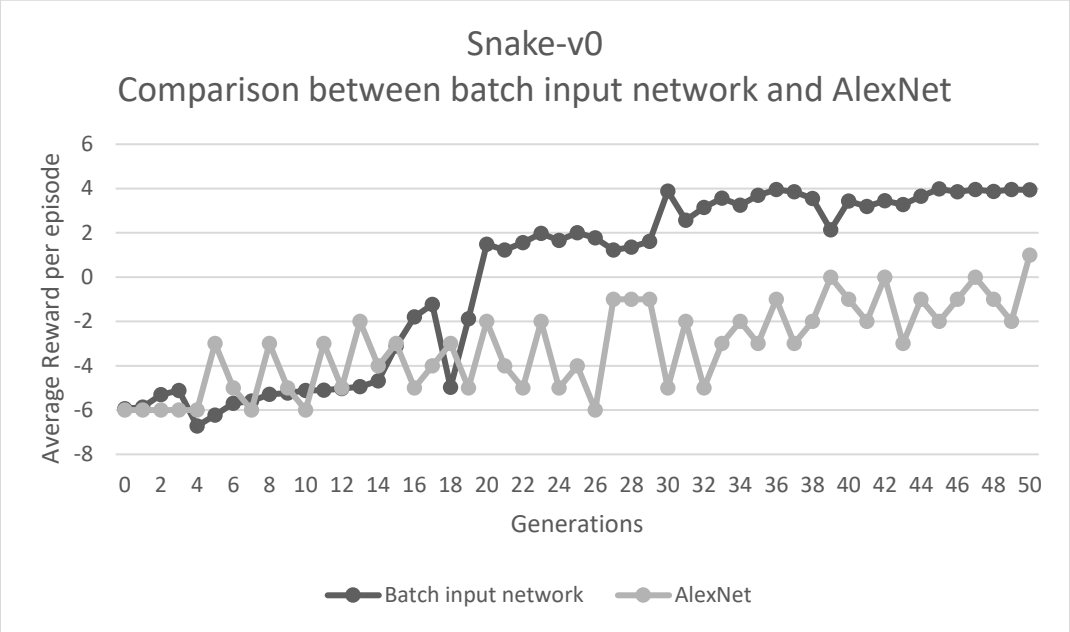

Figure 4.3.1.4: Improvement using the batch of input technique in Snake environment

4.4 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. In other papers and works, 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 come up with a solution of queue for game memory, which also decrease the ram usage.

4.5 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 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.

In case of easier environments like cart pole and mountain car, we reached the highest score within approximately 20 epochs and within a very short time (less than 5 minutes). Because they have less action and easier environment (just the position of pole or car).

However, in Catcher and Snakes the timing is very important. One cannot score without timing it perfectly. So, our model learns from a very unstable randomized data, which results 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 4.3.1 technic, batching a number of sequence of states before a better reward is found.

Chapter 5

Conclusion and future work

5.1 Conclusion

In this paper, we presented a model to solve 2D games with Deep reinforcement learning approach. We introduced a new deep learning model to play game like snake's, catcher etc. using raw pixels for input and also the provided framework by OpenAi. We also presented a AlexNet implementation method for finding satisfied model for playing the game. After all, we used a batching technic for finding better results.

5.2 Future work

We have now the idea of 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 consider can also be taught to the agent.

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-industries-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].