GAME PLAYING AGENT USING DEEP REINFORCEMENT LEARNING

A PROJECT REPORT

Submitted by

ANBU SELVAM A	**731119104005**
BALAJI V	**731119104006**
GOBINATH A	**731119104013**

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

GOVERNMENT COLLEGE OF ENGINEERING
ERODE-638316

ANNA UNIVERSITY : CHENNAI 600 025

JUNE 2022

ANNA UNIVERSITY : CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report, **"GAME PLAYING AGENT USING DEEP REINFORCEMENT LEARNING"** is the bonafide work of **ANBU SELVAM A (731119104005), BALAJI V (731119104006), GOBINATH A (731119104013)** who carried out the project work under our supervision.

SIGNATURE
DR.A.SARADHA M.E.,Ph.D.,
PROFESSOR,
HEAD OF THE DEPARTMENT,
Department of CSE,
Government College of
Engineering, Erode-638316.

SIGNATURE
DR.M.MARIKKANNAN M.E.,Ph.D.,
ASSISTANT PROFESSOR,
SUPERVISOR,
Department of CSE,
Government College of
Engineering, Erode-638316.

Submitted for the university examination held on _____at Government College of Engineering, Erode.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We sincerely express our whole hearted thanks to the principal **Dr.R.Murugesan M.E., Ph.D.,** Government College of Engineering, Erode for his constant encouragement and moral support during the course of this project.

We owe our sincere thanks to **Dr.A.Saradha M.E., Ph.D., Professor and Head of the Department**, Department of Computer Science, Government College of Engineering, Erode for furnishing every essential facility for doing this project.

We sincerely thank our guide **Dr.M.Marikkannan M.E., Ph.D., Assistant Professor**, Department of Computer Science, Government College of Engineering, Erode for his valuable help and guidance throughout the project.

We wish to express our sincere thanks to the Project Coordinator and all staff members of the Department of Computer Science for their valuable help and guidance rendered to us throughout the project.

Above all we are grateful to all our classmates and friends for their friendly Cooperation and their exhilarating company.

TABLE OF CONTENTS

ABSTRACT

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. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems.

While reinforcement learning agents have achieved some successes in a variety of domains, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. The proposed system uses 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. The proposed system is tested on the challenging domain of classic Atari 2600 games which includes Breakout, SpaceInvaders and Pong. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	-	Artificial Intelligence
ALE	-	Atari Learning Environment
CNN	-	Convolutional Neural Network
DeepRL	-	Deep Reinforcement Learning
DQN	-	Deep Q-learning Network
LSTM	-	Long Short Term Memory
NN	-	Neural Network
RL	-	Reinforcement Learning
RNN	-	Recurrent Neural Network
ViT	-	Vision Transformer
PER	-	Prioritized Experience Replay

CHAPTER 1
INTRODUCTION

1.1 Artificial Intelligence

Artificial intelligence (AI) is intelligence demonstrated by machines, as opposed to the natural intelligence displayed by animals including humans. AI research has been defined as the field of study of intelligent agents, which refers to any system that perceives its environment and takes actions that maximize its chance of achieving its goals.

AI applications include advanced web search engines (e.g., Google), recommendation systems (used by YouTube, Amazon and Netflix), understanding human speech (such as Siri and Alexa), self-driving cars (e.g., Tesla), automated decision-making and competing at the highest level in strategic game systems (such as chess and Go).

1.2 Machine Learning

A machine learning algorithm is an algorithm that is able to learn from data. But what is learning? Mitchell (1997) provides a succinct definition: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.” One can imagine a wide variety of Experiences E, tasks T, and performance measures P.

There are three main categories of machine learning.

1. In **supervised learning** the agent observes input-output pairs and learns a function that maps from input to output. For example, the inputs could be camera

images, each one accompanied by an output saying “bus” or “pedestrian,” etc. An output like this is called a label. The agent learns a function that, when given a new image, predicts the appropriate label.

2. In **unsupervised learning** the agent learns patterns in the input without any explicit feedback. The most common unsupervised learning task is clustering: detecting potentially useful clusters of input examples. For example, when shown millions of images taken from the Internet, a computer vision system can identify a large cluster of similar images which an English speaker would call “cats.”

3. In **reinforcement learning** the agent learns from a series of reinforcements: rewards and punishments. For example, at the end of a chess game the agent is told that it has won (a reward) or lost (a punishment). It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it, and to alter its actions to aim towards more rewards in the future.

1.3 Deep Learning

Until the last decade, the broader class of systems that fell under the label machine learning relied heavily on feature engineering. Features are transformations on input data that facilitate a downstream algorithm, like a classifier, to produce correct outcomes on new data. Feature engineering consists of coming up with the right transformations so that the downstream algorithm can solve a task. For instance, in order to tell ones from zeros in images of handwritten digits, we would come up with a set of filters to estimate the direction of edges over the image, and then train a classifier to predict the correct digit given a distribution of edge directions. Another useful feature could be the number of enclosed holes, as seen in a zero, an eight, and, particularly, loopy twos.

Deep learning is a field of study of Neural networks. Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f. For example, for a classifier, $y=f(x)$ maps an input x to a category y. A feedforward network defines a mapping $y=f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

There are many types of architectures in neural networks which includes vanilla neural network, Convolutional neural network (CNN), Recurrent neural network (RNN), Gated recurrent unit (GRU), Long short term memory (LSTM), transformer (attention-based), Generative adversarial network (GAN), Autoencoder, Diffusion models.

1.4 Reinforcement Learning

Reinforcement Learning (RL) is one of the paradigms of Machine learning along with supervised and unsupervised learning. RL is a framework to solve problems which requires the algorithm to make decisions at multiple time-steps (sequential decision-making).

Consider the following illustration, As you can see, there are 2 main components in the RL problem setup, the Agent and the Environment. Let's consider that the agent dies after N time-steps. At each time-step t, the agent receives an observation from the environment (or senses the world around it) and it needs to decide which

action to take. The agent chooses an action which gives a high reward in the long run which is the Goal of the agent (reward hypothesis). Action which gives high rewards in the long run are also known as right action. The agent will eventually learn which actions lead to high rewards and which don't, based on the lot of interactions with the environment. Intuitively, at the start of training, the agent chooses actions randomly. After some time, it'll understand which actions lead to high rewards and choose them.

So, RL is nothing but constructing an agent which chooses the right action most of the time, if not all.

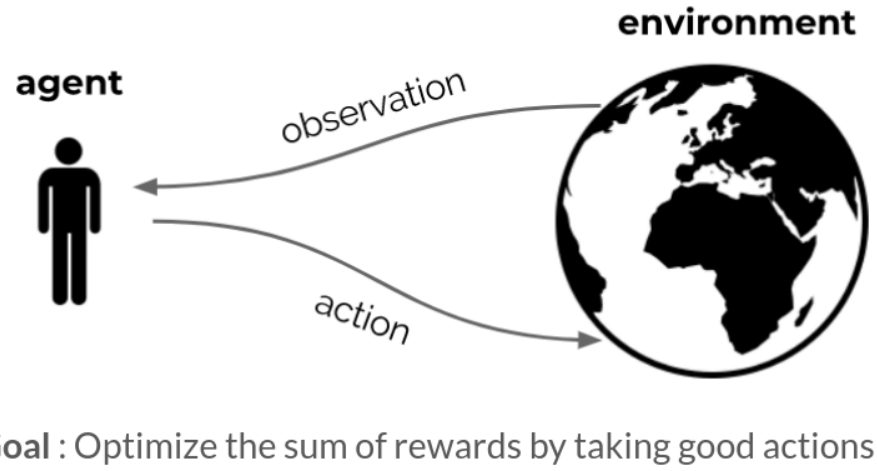

Fig 1.1 RL framework

How to construct an agent which chooses the right action?

There are two ways to construct an agent.

A. Policy function based

B. Value function based

Hereafter, I'll use policy or policy function interchangeably. Same for states or observations.

A. Policy function based approach:

It is natural to think of an agent in a policy based way. In this approach, an agent(also called policy in our case) can be viewed as a mathematical function which takes observations as an input and outputs the action. NN can be used in place of a mathematical function. And the NN can be trained using gradient ascent to maximize our objective which is to get high rewards (that's DeepRL). Again, the policy function can be classified into either deterministic policy or non-deterministic policy.

B. Value function based approach:

Value functions can be classified into 2 types.

a. State value function

State value function takes only state as an input. It calculates how much reward the agent can get if it starts out in this state and takes actions according to the policy. Intuitively, state value function measures how good the state is.

b. State-Action value function

State-Action value function takes both state and action as an input. It calculates how much reward the agent can get if it takes a particular action in a state and after that, it takes actions according to the policy. Intuitively, it measures how good the action is wrt that state.

You might then think about how the agent takes actions given an observation using this approach. The agent calculates values for all the actions wrt the state and then can choose the action which has the highest value. That's it. Deep Q-learning Network (DQN) which is a popular DeepRL algorithm uses this approach.

These approaches are purely reactive, there is no planning involved in achieving the agent's goal. These purely reactive approaches can be good for some problems, but not for problems which require planning. So here comes another thing called model

or transition dynamics function. This model is able to predict the next state, given a state and an action taken in that state. This model can be used along with the above approaches, to further improve the agent. The approaches either use Model (Model-based RL) or not (Model-free RL). At the same time, it uses either policy function or value function orelse both of these functions.

1.5 Deep Reinforcement Learning

Deep reinforcement learning (Deep RL) is a subfield of machine learning that combines reinforcement learning (RL) and deep learning. RL considers the problem of a computational agent learning to make decisions by trial and error. Deep RL incorporates deep learning into the solution, allowing agents to make decisions from unstructured input data without manual engineering of the state space. Deep RL algorithms are able to take in very large inputs (e.g. every pixel rendered to the screen in a video game) and decide what actions to perform to optimize an objective (eg. maximizing the game score). Deep reinforcement learning has been used for a diverse set of applications including but not limited to robotics, video games, natural language processing, computer vision, education, transportation, finance and healthcare.

1.6 Reinforcement Learning and Supervised Learning

At the core of deep reinforcement learning is function approximation. This is something it shares with supervised learning. However, reinforcement learning is unlike supervised learning in a number of ways. There are three main differences:

- Lack of an oracle
- Sparsity of feedback
- Data generated during training

1.6.1 Lack of an oracle

A major difference between reinforcement learning and supervised learning is that for reinforcement learning problems, the "correct" answer for each model input is not available, whereas in supervised learning we have access to the correct or optimal answer for each example. In reinforcement learning, the equivalent of the correct answer would be access to an "oracle" which tells us the optimal action to take at every time step so as to maximize the objective.

The correct answer can convey a lot of information about a data point. For example, the correct answer for classification problems contains many bits of information. It not only tells us the right class for each training example, but also implies that the example does not belong to any of the other classes. If a particular classification problem has 1000 classes (as in the ImageNet dataset), an answer contains 1000 bits of information per example (1 positive and 999 negative). Furthermore, the correct answer does not have to be a category or a real number. It can be a bounding box, or a semantic segmentation, each of which contains many bits of information about the example at hand.

In reinforcement learning, after an agent takes action *a* in state *s*, it only has access to the reward it receives. The agent is not told what the best action to take was. Instead, it is only given an indication, via the reward, of how good or bad a was. Not only does this convey less information than the right answer would have provided, but the agent only learns about rewards for the states it experiences. To learn about (*s, a, r*), an agent must experience the transition (*s, a, r, s0*). An agent may have no knowledge about important parts of the state and action spaces because it hasn't experienced them.

One way to deal with this problem is to initialize episodes to start in the states we want an agent to learn about. However, it is not always possible to do this, for

two reasons. First, we may not have full control over an environment. Second, states can be easy to describe but difficult to specify. Consider a simulation of a humanoid robot learning to do a backflip. To help an agent learn about the reward for successful landing, we can initialize an environment to start just as the robot's feet make contact with the floor after a "good" flip. The reward function is this where the robot may either retain its balance and successfully execute the flip, or fall over and fail. However, it is not straightforward to define the precise numerical position and velocity of each of the robot's joint angles, or the force being exerted, to initialize the robot in this position. In practice, to reach this state, an agent needs to execute a long, very specific sequence of actions to first flip and then almost land. There is no guarantee that an agent will learn to do this, so this part of the state space may never be explored.part of the state space is critical to learn about, since this is where the robot may either retain its balance and successfully execute the flip, or fall over and fail. However, it is not straightforward to define the precise numerical position and velocity of each of the robot's joint angles, or the force being exerted, to initialize the robot in this position. In practice, to reach this state, an agent needs to execute a long, very specific sequence of actions to first flip and then almost land. There is no guarantee that an agent will learn to do this, so this part of the state space may never be explored.

1.6.2 Sparsity of feedback

In reinforcement learning, a reward function may be sparse, so the scalar reward is often 0. This means that most of the time, an agent is receiving no information about how to change the parameters of the network so as to improve performance. Consider again the backflipping robot and suppose an agent only receives a nonzero reward of +1 after successfully executing a backflip. Almost all actions that it takes

will result in the same reward signal of 0 from the environment. Under these circumstances, learning is extremely challenging because an agent receives no guidance about whether its intermediate actions help reach the goal. Supervised learning doesn't have this problem; all input examples are paired with a desired output which conveys some information about how a network should perform.

The combination of sparse feedback and the lack of an oracle means that in reinforcement learning, much less information per time step is received from the environment, compared to the training examples in supervised learning. As a result, all reinforcement learning algorithms tend to be significantly less sample-efficient

1.6.3 Data generated during training

In supervised learning, data is typically generated independently from algorithm training. Indeed, the first step in applying supervised learning to a problem is often to find or construct a good dataset. In reinforcement learning, data must be generated by an agent interacting with an environment. In many cases, this data is generated as training progresses in an iterative manner, with alternating phases of data gathering and training. Data and algorithms are coupled. The quality of an algorithm affects the data it is trained on, which in turn affects the algorithm's performance. This circularity and bootstrapping requirement does not occur in supervised learning. RL is also interactive—actions made by the agent actually change the environment, which then changes the agent's decisions, which changes the data the agent sees, and so on. This feedback loop is the hallmark of reinforcement learning. In supervised learning problems, there is no such loop and no equivalent notion of an agent which could change the data that an algorithm is trained on.

A final, more minor difference between reinforcement learning and supervised learning is that in reinforcement learning, neural networks are not always trained using a recognizable loss function. Instead of minimizing the error of the loss between a network's outputs and a desired target, rewards from the environment are used to construct an objective, then the network is trained so as to maximize this objective. Coming from a supervised learning background, this may seem a little strange at first, but the optimization mechanism is essentially the same. In both cases, the parameters of a network are adjusted to maximize or minimize a function.

1.7 PyTorch

PyTorch is a library for Python programs that facilitates building deep learning projects. It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python. This approachability and ease of use found early adopters in the research community, and in the years since its first release, it has grown into one of the most prominent deep learning tools across a broad range of applications. As Python does for programming, PyTorch provides an excellent introduction to deep learning. At the same time, PyTorch has been proven to be fully qualified for use in professional contexts for real-world, high-profile work.

At its core, the deep learning machine is a rather complex mathematical function mapping inputs to an output. To facilitate expressing this function, PyTorch provides a core data structure, the tensor, which is a multidimensional array that shares many similarities with NumPy arrays. Around that foundation, PyTorch comes with features to perform accelerated mathematical operations on dedicated hardware, which makes it convenient to design neural network architectures and train them on individual machines or parallel computing resources.

CHAPTER 2
LITERATURE REVIEW

Mnih et al proposed the DQN algorithm in his seminal paper "Playing Atari using Deep Reinforcement Learning". This is the first work which successfully combined Deep learning and Reinforcement Learning algorithms. He used a 3-layer Convolutional neural network (CNN) to approximate the value function which predicts how good the action is. He used a Replay buffer to remove the correlation between the different states and target network to remove the divergence during the training period. Then he tested the DQN algorithm on 7 Atari 2600 games without modifying network architecture and Hyperparameters.

Dosovitskiy et al proposed a transformer based neural network architecture to handle images called Vision transformer (ViT). Transformer is a de facto architecture in Natural Language Processing. This is the first work which applied transformers for computer vision tasks and beat CNN in terms of accuracy on image classification task. As a preprocessing step, an image is splitted into patches. Each of those patches is considered to be a “word” or ”token” and projected to a feature space. With adding positional encodings and a token for classification on top, Transformer can be applied as usual to this sequence and start training it for the desired task.

CHAPTER 3
SYSTEM ANALYSIS

3.1 EXISTING SYSTEM:

Researchers had used hand-crafted features for about decades to train a Reinforcement learning agent to play video games. But in 2013, Mnih et al first proposed a technique called Deep Q-learning network (DQN) algorithm which uses a deep learning based system which can automatically learn the features, to train a RL agent to play Atari 2600 video games.

The DQN algorithm also uses a Replay buffer which is used to decrease the correlation between the states during the training, so that the system gets a non-identical and independent (iid) data distribution for the training. It also makes use of a target network which stabilizes the training, otherwise it leads to the divergence of the policy.

The DQN algorithm makes use of the Convolutional neural network (CNN) which was proposed by LeCun et al in 1987. CNNs are also known as Shift Invariant or Space Invariant Artificial Neural Networks, based on the shared-weight architecture of the convolution kernels or filters that slide along input features and provide translation-equivariant responses known as feature maps.

Drawbacks:

- Computationally in-efficient.
- Not sample efficient.

3.2 PROPOSED SYSTEM:

The proposed system makes use of 2 components to combat the drawbacks of the DQN algorithm. One is Vision Transformers (ViT) which was first proposed by Dosovitskiy et al in 2021. ViT makes use of transformer architecture which is a de facto architecture in Natural Language Processing tasks. ViT is used to combat the computational inefficiency of the DQN algorithm.

Another is Prioritized Experience Replay (PER) which was first proposed by Schaul et al in 2015. The DQN algorithm uniformly samples transitions from the replay buffer for training. PER allows the DQN algorithm to sample important transitions more frequently, and therefore learn more efficiently. PER is used to combat the sample inefficiency of the DQN algorithm.

Advantages:

- Computationally more efficient than DQN.
- Sample efficient compared to DQN.

CHAPTER 4
SYSTEM SPECIFICATION

4.1 HARDWARE REQUIREMENTS:

Minimum requirements for Training:

Processor	:	Intel Core i3
RAM	:	4GB
GPU	:	Nvidia GTX 1080

Minimum requirements for Testing:

Processor	:	Intel Core i3
RAM	:	4GB

4.2 SOFTWARE REQUIREMENTS:

Operating System	:	Any OS
Programming Language	:	Python3

Packages:

- OpenAI Gym
- PyTorch
- Torchvision
- Numpy

CHAPTER 5
PROJECT DESCRIPTION

5.1 PROJECT DESCRIPTION:

The Project is about building an artificial agent to play Atari 2600 games such as Breakout, Space Invaders and Pong. The agent is trained on those games using a Deep Reinforcement Learning algorithm called Deep Q-learning network (DQN). The proposed system which includes 2 components Vision Transformer and Prioritized Experience Replay overcomes the computational inefficiency and sample inefficiency of the DQN algorithm. To train on these different games, the system proposed makes use of the same algorithm, neural network architecture and set of hyperparameters. The main objective of the project is to test the proposed system on the above mentioned games. The success of the proposed system tells that it works correctly, so that the proposed system can be applied in real world applications.

5.2 OVERVIEW OF THE PROPOSED SYSTEM:

The Proposed system will overcome the drawbacks of the existing system. The proposed system is able to play any kind of games which has a high dimensional input (e.g. video games) after it has trained on it. At every time step of the game, the agent receives the preprocessed input image of the game and selects the right action from the action space. The environment then executes the selected action and returns the next state and the reward to the agent. The proposed system is able to render the interactions between the agent and the environment in real-time on our computer screen.

5.3 MODULE DESCRIPTION:

5.3.1 Train Agent:

This module takes arguments for environment and neural network architecture and trains the neural network which is our agent on the environment for a fixed number of episodes. This module requires access to the GPU for training purposes. After the training has completed, it returns the trained neural network in the gz file format. This module trains the agent in the environment using the DQN algorithm (along with PER). The Pseudo code for the DQN algorithm is given below:

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Fig 5.1 Pseudo code for DQN algorithm

5.3.2 ViT model

This module implements the Vision Transformer (ViT). This module is implemented using the torch.nn package which provides all the types of layers including Conv layer, RNN layer, LSTM layer, Fully connected layer and many more for building a neural network. The ViT model architecture is illustrated below:

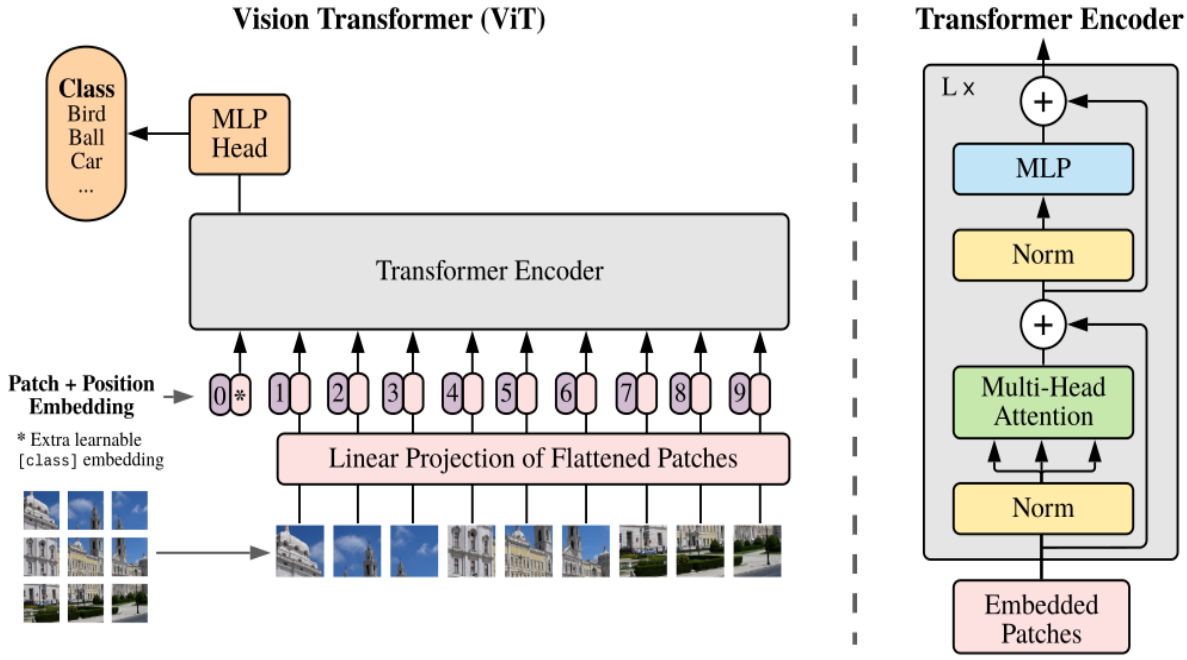

Fig 5.2 ViT model architecture (left) and Transformer encoder architecture (right)

The overview of the ViT model is to split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, the standard approach of adding an extra learnable “classification token” to the sequence is used.

5.3.3 PER

Instead of uniformly sampling a transition from the replay buffer, it is better to sample an important transition more frequently and thus this allows the agent to learn more effectively. This module implements the Prioritized Experience Replay (PER) with ReplayBuffer. Here, each transition is assigned with a priority value.

5.3.4 Play

This module is used to test our trained neural network (agent). This module is implemented using the Atari Learning Environment (ALE) which comes along with the OpenAI Gym. It takes 2 inputs, trained neural network and the environment. Then it renders all the interactions between the agent and the environment, until the agent terminates either by breaking all the blocks or losing all the lives incase of Breakout.

CHAPTER 6
CONCLUSION AND FUTURE ENHANCEMENT

6.1 CONCLUSION:

The proposed system is able to play Atari 2600 games like Breakout, SpaceInvaders, Pong and is able to achieve maximum scores. The proposed system takes less time to train than the existing system, due to the computational efficiency of the proposed system. The proposed system can also be applied for real-world applications.

6.2 FUTURE ENHANCEMENT:

A longstanding goal of the field of AI is a strategy for compiling diverse experience into a highly capable, generalist agent. If we look at humans or animals in general, they are able to perform multiple tasks at hand. But the current AI systems also called “Narrow AI” are only able to perform a single task. When the agent is retrained in another environment, it rapidly forgets everything about the task it had previously trained on, also termed as catastrophic forgetting or catastrophic interference. So one of our future enhancements is to develop a single agent with a single set of weights which can play all these games.

A human is able to learn to play one of these games in about 3 to 5 minutes. But an artificial agent takes about hours to train on these games to be able to play at a level compared to that of a human. Another future enhancement is to increase the sample efficiency of the agent.

CHAPTER 7
APPENDIX

7.1 SOURCE CODE:

vit.py:

```
class AttentionBlock(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
                            nn.Linear(embed_dim, hidden_dim),
                            nn.GELU(),
                            nn.Dropout(dropout),
                            nn.Linear(hidden_dim, embed_dim),
                            nn.Dropout(dropout) )

    def forward(self, x):
        inp_x = self.layer_norm_1(x)
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x

class VisionTransformer(nn.Module):
```

```
    def __init__(self, embed_dim, hidden_dim, num_channels, num_heads,
num_layers, num_classes, patch_size, num_patches, dropout=0.0):
        super().__init__()
        self.patch_size = patch_size
        # Layers/Networks
        self.input_layer = nn.Linear(num_channels*(patch_size**2), embed_dim)
        self.transformer = nn.Sequential(*[AttentionBlock(embed_dim, hidden_dim,
num_heads, dropout=dropout) for _ in range(num_layers)])
        self.mlp_head = nn.Sequential(
                                nn.LayerNorm(embed_dim),
                               nn.Linear(embed_dim, num_classes))
        self.dropout = nn.Dropout(dropout)
        # Parameters/Embeddings
        self.cls_token = nn.Parameter(torch.randn(1,1,embed_dim))
        self.pos_embedding =
nn.Parameter(torch.randn(1,1+num_patches,embed_dim))


    def forward(self, x):
        # Preprocess input
        x = img_to_patch(x, self.patch_size)
        B, T, _ = x.shape
        x = self.input_layer(x)
        # Add CLS token and positional encoding
        cls_token = self.cls_token.repeat(B, 1, 1)
        x = torch.cat([cls_token, x], dim=1)
        x = x + self.pos_embedding[:,:T+1]
        # Apply Transformer
```

```
        x = self.dropout(x)
        x = x.transpose(0, 1)
        x = self.transformer(x)
        # Perform classification prediction
        cls = x[0]
        out = self.mlp_head(cls)
        return out
```

per.py:

```
class ReplayBuffer:
  def __init__(self, obs_dim: int, size: int, batch_size: int = 32):
        self.obs_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.next_obs_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.acts_buf = np.zeros([size], dtype=np.float32)
        self.rews_buf = np.zeros([size], dtype=np.float32)
        self.done_buf = np.zeros(size, dtype=np.float32)
        self.max_size, self.batch_size = size, batch_size
        self.ptr, self.size, = 0, 0

  def store(self, obs: np.ndarray, act: np.ndarray, rew: float, next_obs: np.ndarray,
done: bool):
        self.obs_buf[self.ptr] = obs
        self.next_obs_buf[self.ptr] = next_obs
        self.acts_buf[self.ptr] = act
        self.rews_buf[self.ptr] = rew
        self.done_buf[self.ptr] = done
```

```
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample_batch(self) -> Dict[str, np.ndarray]:
        idxs = np.random.choice(self.size, size=self.batch_size, replace=False,
        return dict(obs=self.obs_buf[idxs], next_obs=self.next_obs_buf[idxs],

    def __len__(self) -> int:
        return self.size

class PrioritizedReplayBuffer(ReplayBuffer):
    def __init__(self, obs_dim: int, size: int, batch_size: int = 32, alpha: float = 0.6):
        assert alpha >= 0
        super(PrioritizedReplayBuffer, self).__init__(obs_dim, size, batch_size)
        self.max_priority, self.tree_ptr = 1.0, 0
        self.alpha = alpha
        # capacity must be positive and a power of 2.
        tree_capacity = 1
        while tree_capacity < self.max_size:
        tree_capacity *= 2
        self.sum_tree = SumSegmentTree(tree_capacity)
        self.min_tree = MinSegmentTree(tree_capacity)

    def store(self, obs: np.ndarray, act: int, rew: float, next_obs: np.ndarray, done:
bool):
        super().store(obs, act, rew, next_obs, done)
        self.sum_tree[self.tree_ptr] = self.max_priority ** self.alpha
```

```
        self.min_tree[self.tree_ptr] = self.max_priority ** self.alpha
        self.tree_ptr = (self.tree_ptr + 1) % self.max_size

    def sample_batch(self, beta: float = 0.4) -> Dict[str, np.ndarray]:
        assert len(self) >= self.batch_size
        assert beta > 0
        indices = self._sample_proportional()
        obs = self.obs_buf[indices]
        next_obs = self.next_obs_buf[indices]
        acts = self.acts_buf[indices]
        rews = self.rews_buf[indices]
        done = self.done_buf[indices]
        weights = np.array([self._calculate_weight(i, beta) for i in indices])
        return dict(obs=obs, next_obs=next_obs, acts=acts, rews=rews, done=done,
        weights=weights, indices=indices)

    def update_priorities(self, indices: List[int], priorities: np.ndarray):
        assert len(indices) == len(priorities)
        for idx, priority in zip(indices, priorities):
        assert priority > 0
        assert 0 <= idx < len(self)
        self.sum_tree[idx] = priority ** self.alpha
        self.min_tree[idx] = priority ** self.alpha
        self.max_priority = max(self.max_priority, priority)

    def _sample_proportional(self) -> List[int]):
        indices = []
```

```
        p_total = self.sum_tree.sum(0, len(self) - 1)
        segment = p_total / self.batch_size
        for i in range(self.batch_size):
        a = segment * i
        b = segment * (i + 1)
        upperbound = random.uniform(a, b)
        idx = self.sum_tree.retrieve(upperbound)
        indices.append(idx)
        return indices

    def _calculate_weight(self, idx: int, beta: float):
        # get max weight
        p_min = self.min_tree.min() / self.sum_tree.sum()
        max_weight = (p_min * len(self)) ** (-beta)
        # calculate weights
        p_sample = self.sum_tree[idx] / self.sum_tree.sum()
        weight = (p_sample * len(self)) ** (-beta)
        weight = weight / max_weight
        return weight
```

train_agent.py:

```
class DQNAgent:
    def __init__(self,env: gym.Env,memory_size: int,batch_size: int,
target_update: int, epsilon_decay: float,max_epsilon: float = 1.0,min_epsilon: float
= 0.1,gamma: float = 0.99,alpha: float = 0.2,beta: float = 0.6,prior_eps: float =
1e-6):
```

```
        obs_dim = env.observation_space.shape[0]
        action_dim = env.action_space.n
        self.env = env
        self.batch_size = batch_size
        self.epsilon = max_epsilon
        self.epsilon_decay = epsilon_decay
        self.max_epsilon = max_epsilon
        self.min_epsilon = min_epsilon
        self.target_update = target_update
        self.gamma = gamma
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.beta = beta
        self.prior_eps = prior_eps
        self.memory = PrioritizedReplayBuffer(obs_dim, memory_size, batch_size,
alpha)
        # networks: dqn, dqn_target
        self.dqn = ViT(obs_dim, action_dim).to(self.device)
        self.dqn_target = ViT(obs_dim, action_dim).to(self.device)
        self.dqn_target.load_state_dict(self.dqn.state_dict())
        self.dqn_target.eval()
        # optimizer
        self.optimizer = optim.Adam(self.dqn.parameters())
        # transition to store in memory
        self.transition = list()
        # mode: train / test
        self.is_test = False
```

```
  def select_action(self, state: np.ndarray) -> np.ndarray:
        # epsilon greedy policy
        if self.epsilon > np.random.random():
        selected_action = self.env.action_space.sample()
        else:
        selected_action = self.dqn(torch.FloatTensor(state).to(self.device).argmax()
        selected_action = selected_action.detach().cpu().numpy()
        if not self.is_test:
        self.transition = [state, selected_action]
        return selected_action

  def step(self, action: np.ndarray) -> Tuple[np.ndarray, np.float64, bool]:
        next_state, reward, done, _ = self.env.step(action)
        if not self.is_test:
        self.transition += [reward, next_state, done]
        self.memory.store(*self.transition)
        return next_state, reward, done

  def update_model(self) -> torch.Tensor:
        # PER needs beta to calculate weights
        samples = self.memory.sample_batch(self.beta)
        weights = torch.FloatTensor(samples["weights"].reshape(-1,
1)).to(self.device)
        indices = samples["indices"]
        # PER: importance sampling before average
        elementwise_loss = self._compute_dqn_loss(samples)
        loss = torch.mean(elementwise_loss * weights)
```

```
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        # PER: update priorities
        loss_for_prior = elementwise_loss.detach().cpu().numpy()
        new_priorities = loss_for_prior + self.prior_eps
        self.memory.update_priorities(indices, new_priorities)
        return loss.item()

    def train(self, num_frames: int, plotting_interval: int = 200):
        self.is_test = False
        state = self.env.reset()
        update_cnt = 0
        epsilons = []
        losses = []
        scores = []
        score = 0
        for frame_idx in range(1, num_frames + 1):
        action = self.select_action(state)
        next_state, reward, done = self.step(action)
        state = next_state
        score += reward
        # PER: increase beta
        fraction = min(frame_idx / num_frames, 1.0)
        self.beta = self.beta + fraction * (1.0 - self.beta)
        # if episode ends
        if done:
```

```
                state = self.env.reset()
                scores.append(score)
                score = 0

        # if training is ready
        if len(self.memory) >= self.batch_size:
                loss = self.update_model()
                losses.append(loss)
                update_cnt += 1
                # linearly decrease epsilon
                self.epsilon = max(self.min_epsilon, self.epsilon - (self.max_epsilon -
self.min_epsilon) * self.epsilon_decay)
                epsilons.append(self.epsilon)
                # if hard update is needed
                if update_cnt % self.target_update == 0:
                self._target_hard_update()
        self.env.close()

  def _compute_dqn_loss(self, samples: Dict[str, np.ndarray]) -> torch.Tensor:
        device = self.device  # for shortening the following lines
        state = torch.FloatTensor(samples["obs"]).to(device)
        next_state = torch.FloatTensor(samples["next_obs"]).to(device)
        action = torch.LongTensor(samples["acts"].reshape(-1, 1)).to(device)
        reward = torch.FloatTensor(samples["rews"].reshape(-1, 1)).to(device)
        done = torch.FloatTensor(samples["done"].reshape(-1, 1)).to(device)
        # G_t   = r + gamma * v(s_{t+1})  if state != Terminal
        #   r                       otherwise
```

```
        curr_q_value = self.dqn(state).gather(1, action)
        next_q_value = self.dqn_target(next_state).max(dim=1,
keepdim=True)[0].detach()
        mask = 1 - done
        target = (reward + self.gamma * next_q_value * mask).to(self.device)

        # calculate element-wise dqn loss
        elementwise_loss = F.smooth_l1_loss(curr_q_value, target,
reduction="none")
        return elementwise_loss

    def _target_hard_update(self):
        self.dqn_target.load_state_dict(self.dqn.state_dict())
```

7.2 SCREENSHOTS:

7.2.1 Breakout

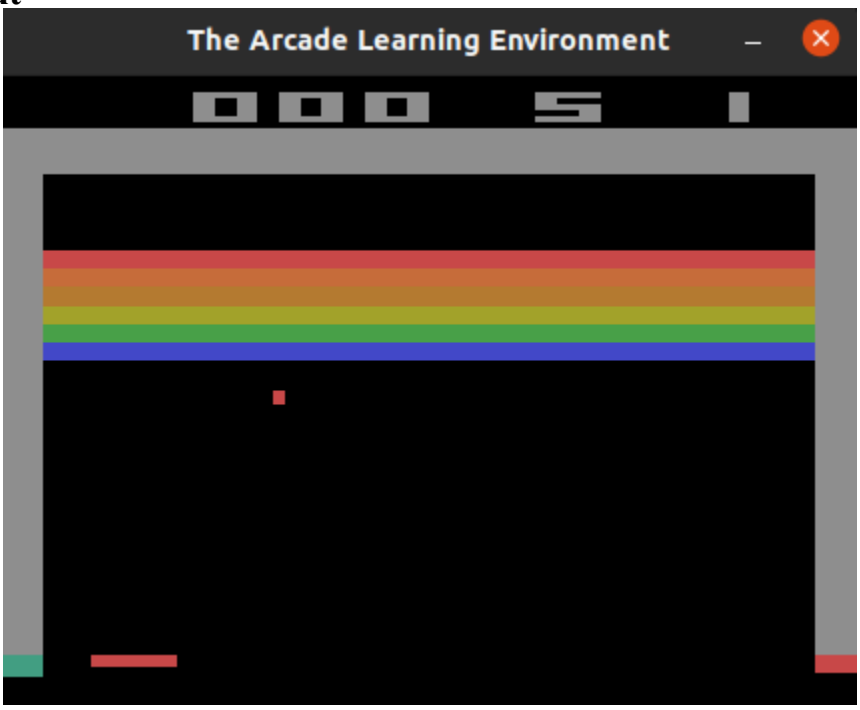

Fig 7.1 - At the start of the game (Breakout)

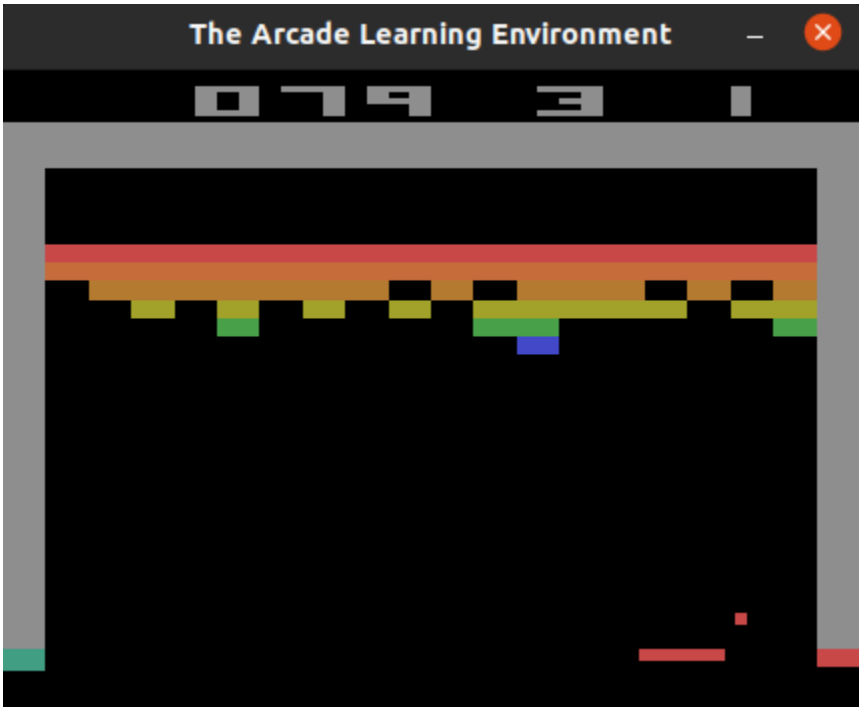

Fig 7.2 - Halfway through the game (Breakout)

7.2.2 Space Invaders

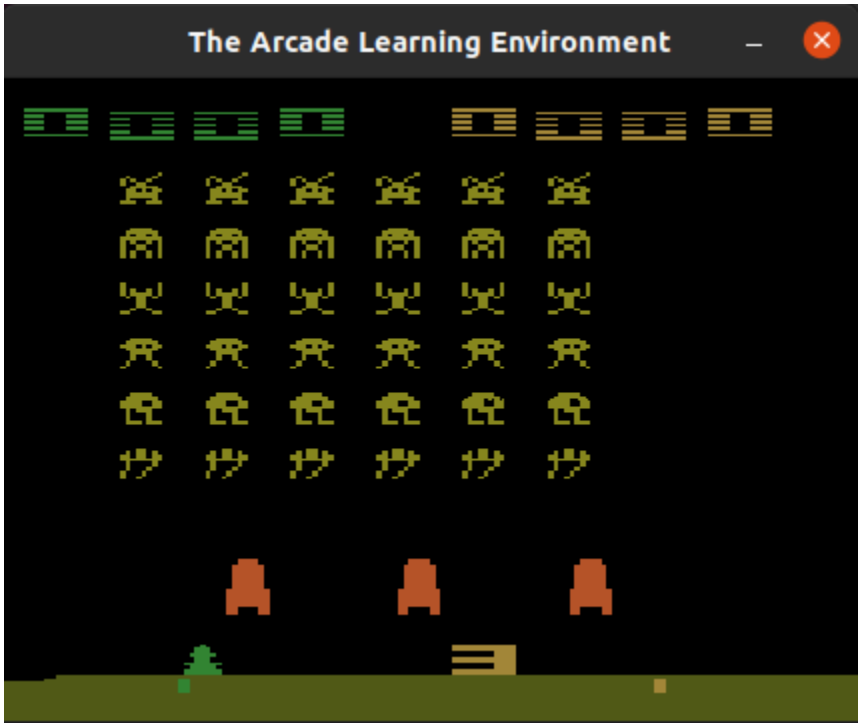

Fig 7.3 - At the start of the game (Space Invaders)

Fig 7.4 - At the end of the game (Space Invaders)

7.2.3 Pong

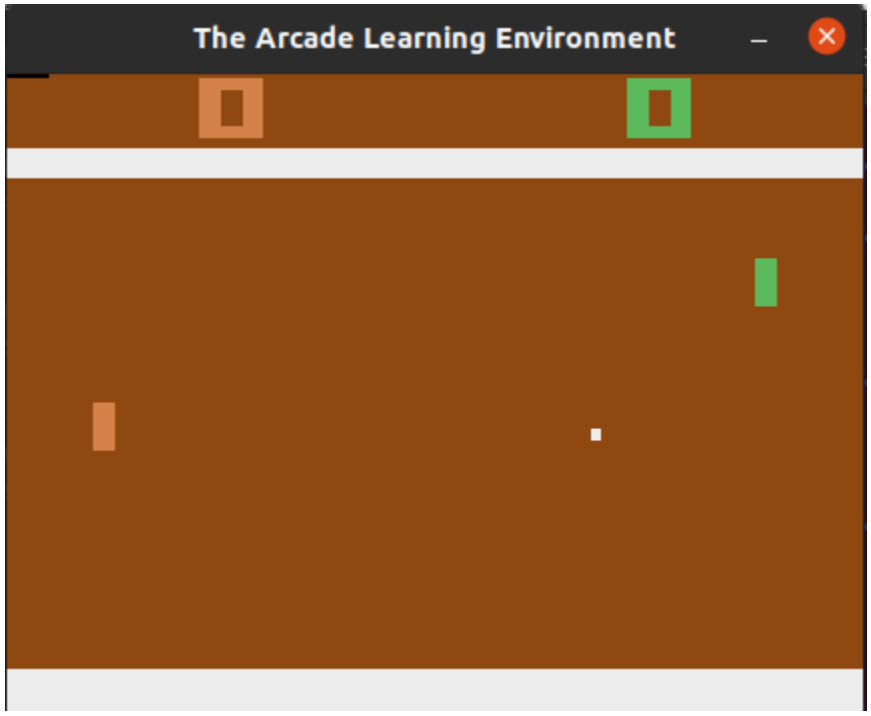

Fig 7.5 - At the start of the game (Pong)

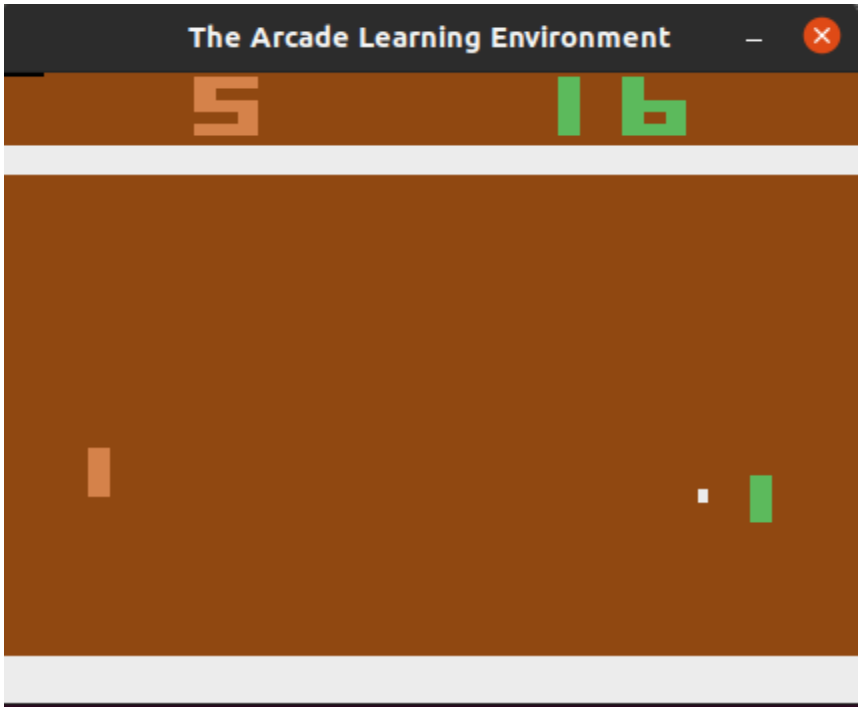

Fig 7.6 - At the end of game (Pong)

CHAPTER 8
REFERENCES

[1] A Generalist Agent, Scott Reed et al, 2022.

[2] Algorithms for Reinforcement Learning, Szepesvari, 2009.

[3] An image is worth 16x16 words: Transformers for image recognition at scale, Dosovitskiy, Alexey, et al 2021.

[4] Attention Is All You Need, Vaswani et al, 2017.

[5] Deep Reinforcement Learning with Double Q-learning, Hasselt et al, 2015.

[6] Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, 2015.

[7] Multi-Game Decision Transformers, Kuang et al, 2022.

[8] Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013.

[9] Policy Gradient Methods for Reinforcement Learning with Function Approximation, Sutton et al, 2000.

[10] Prioritized Experience Replay, Schaul et al, 2015.

[11] Rainbow: Combining Improvements in Deep Reinforcement Learning, Hessel et al, 2017.

[12] Reinforcement Learning: An Introduction, 2nd edition, Richard Sutton and Andrew Barto, 2020.