School of Computing

FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Deep Reinforcement Learning On Atari Games

Wenkang Li

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2021/22

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

<As an example>

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (27/04/22)*
Link to online code repository	*https://github.com/wenkangleolee/atariAgent*	*Sent to supervisor and assessor (27/04/22)*
User manuals	*PDF*	*Sent to client and supervisor (27/04/22)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Li Wen Kang

(Signature of student)

Summary

Deep Reinforcement Learning is a subfield of machine learning that combines reinforcement learning and deep learning; the reinforcement learning algorithm is to determine the optimal policy that has a maximum reward and policy gradient methods are modelling and optimising the policy directly. This project is based on the actor-critic algorithm. Actor-Critic methods are the temporal difference (TD) learning methods that represent the policy function independent of the value function. It is a good way to build and train a model to beat computer games (Atari games) and achieve a good game score.

Acknowledgements

A special thanks to Dr. Abdulrahman Altahhan, my supervisor, for his guidance and support throughout the development of my machine learning Agent. Despite the fact that we never met in person, I'd want to express my gratitude to Dr. Matteo Leonetti and Mohammed Alghamdi for teaching the great Machine Learning module; many of the techniques that were used in this project were learnt from that module as well.

Contents

Chapter 1
Introduction and Background Research

1.1 Introduction

In the 20th century, Artificial Intelligence(AI) was first introduced and started used in solving complex problems by researchers, but AI was popularly known from 2016 after alpha go(developed by Google) was exploited and used to beat a human professional player in the Chinese board game “go”. People had seen lots of potential power from AI, and AI was then used a lot in solving complex problems.

AI can be divided into three basic subfields, supervised learning, unsupervised learning and reinforcement learning, Reinforcement learning is different from supervised learning, in which RL users does not need to label input or output data, it can be used more on exploring unknown territory and some fields the human have no much experience about the best approach that should be taken to achieve the best reward.

Atari Games is a platform for computer games, it has a large variety of games that can be used to represent most games patterns. Reinforcement Learning is very effective in training agents in playing computer games, policy-based RL method is focusing on finding the optimal policy by finding optimal value-function. Policy-based RL is effective in sophisticated games that require continuous actions taken and complicated reward policy and can archive better overall performance than those value-based RL methods.

This project is mainly finding a better method of playing the computer game and archiving the best game scores.

1.2 Literature Review

1.2.1 Atari games

The Atari 2600 is a video game console that was released by Atari in 1977. Breakout, Ms Pacman, and Space Invaders were among the games available on the device. The Atari 2600 has been the primary platform for testing new Reinforcement Learning algorithms since Mnih et al. launched Deep Q-Networks in 2013. Due to its high-dimensional video input (size 210 x 160, frequency 60 Hz) and the disparity of demands between games, the Atari 2600 has been a complicated testbed in the open-AI gym.

As a result of limitations in computer resources, Atari games were built in a simple environment, with the majority of games displaying all relevant information about the game state to the player every frame; this is what makes Atari games inputs can be learned and evaluated. Therefore, each frame of the visual output can be used as input to a machine learning agent, which can then learn to perform the optimal moves for a particular game state by utilising the restricted number of pixels available to it.

1.2.2 Machine learning

Machine learning algorithms are capable of performing tasks even when they have not been expressly designed to do so. It entails computers learning from data that is presented in order for them to perform specific jobs. For basic jobs handed to computers, it is possible to build algorithms that instruct the machine on how to perform all steps necessary to solve the problem at hand; no learning is required on the side of the computer in order to accomplish the work. In the case of more complex tasks, it can be difficult for a human to manually develop the algorithms that are required. In fact, it may be more beneficial to assist the machine in developing its own algorithm rather than having human programmers specify each and every step that is required.

Machine learning is a discipline that employs a variety of ways to train computers how to perform tasks for which no entirely suitable solution is currently available. When there are a large number of possible replies, one strategy is to classify some of the accurate answers as valid while rejecting the others. Following that, the data can be utilised as training data by the computer in order to improve the algorithms that is used to identify the right responses. For example, the MNIST dataset of handwritten digits has frequently been used to train a system for the task of digital character recognition.

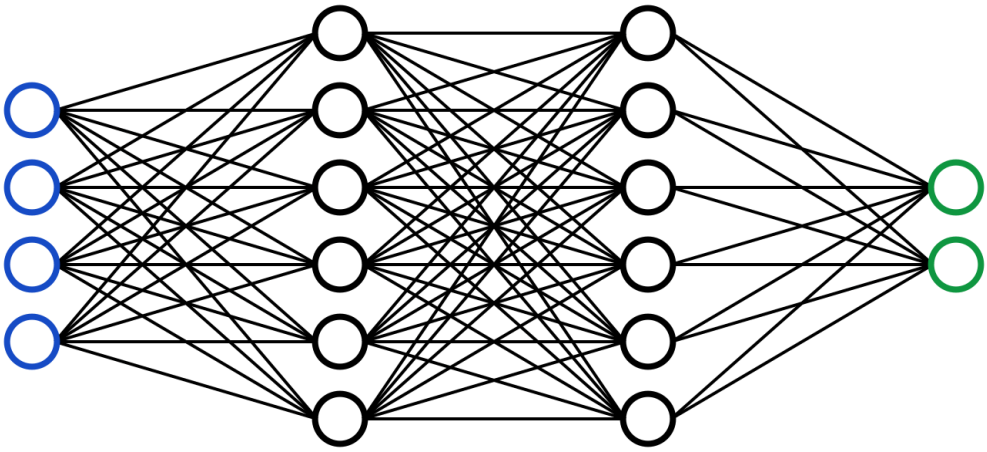
1.2.2.1 Neural network

A neural network can assist in the creation of processes such as time series forecasting, algorithmic trading, securities classification, credit risk modelling, as well as the construction of proprietary indicators and price derivatives in the financial industry.

A neural network functions in a similar way to the neural network of the human brain. In a neural network, a "neuron" is a mathematical function that collects and categorises input in

accordance with a particular architecture. The network resembles statistical procedures such as curve fitting and regression analysis in that it has a high degree of similarity.

A neural network is composed of layers of nodes that are interconnected with one another. Known as the perceptron, each node functions in a manner similar to that of multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may or may not be linear in response to the signal.



1.2.2.2 Optimization

Improve the performance of your neural network by changing its parameters, such as its weights and learning rate, using optimization algorithms or methods (also known as optimization methods).

In the case of optimization, a function has numerous optimal points, only one of which is the global optimal point. It might be extremely difficult to determine the global optima depending on the shape of the loss surface.

The loss surface is the curve or surface that we are referring to when we are discussing a neural network. Considering that we are seeking to reduce the prediction error of the network, we are interested in locating the global minimum on the loss surface — which is the goal of neural network training.

1.2.2.3 Convoluted Neural Networks

In a CNN, the input is a tensor with the following dimensions: (number of inputs) x (input height) x (input width) x (input channels). Once it has passed through a convolutional layer,

the image is abstracted into a feature map, which is also known as an activation map, with the following dimensions: (number of inputs) x (feature map height) x (feature map width) x (number of inputs) (feature map channels).

Convolutional layers convolve the input and send the result to the next layer. This is analogous to the response of a cell in the visual cortex to a specific stimulus in a different area of the brain. Each convolutional neuron processes information just for the receptive field to which it belongs. Although fully connected feedforward neural networks can be used to learn features and categorise data, this architecture is often unfeasible for big inputs such as high-resolution photos because of the enormous number of connections. Even in a shallow architecture, it would be necessary to have a huge number of neurons due to the large input size of images, where each pixel is a relevant input characteristic in and of itself. For example, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for each neuron in the second layer of a completely connected layer for a 100 x 100 image. Convolution, on the other hand, reduces the number of free parameters, allowing the network to be more complex. Consider the following scenario: no matter what the image size is, the use of a 5x5 tiled region, each with the same shared weights, requires only 25 learnable parameters regardless of the image size. Backpropagation in standard neural networks is plagued by difficulties such as disappearing gradients and expanding gradients, which can be avoided by employing regularised weights over a smaller number of parameters. Moreover, because spatial interactions between individual features are taken into account during convolution and/or pooling, convolutional neural networks are well suited for data with a grid-like architecture (such as pictures).

1.2.3 Reinforcement Learning

Reinforcement learning is explored across a wide range of disciplines, including game theory and control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, and statistics, to name a few. Reinforcement learning is referred to as approximation dynamic programming, neuro-dynamic programming, and neuro-dynamic programming in the operations research and control literature. It has also been shown that the problems of interest in reinforcement learning can be studied in the theory of optimal control, which is primarily concerned with the existence and characterization of optimal solutions, as well as algorithms for their exact computation, and less with learning or approximation, particularly in the absence or imperfect knowledge of the environment. Reinforcement learning can be utilised in economics and game theory to

explain how equilibrium can emerge under conditions of constrained rationality, according to the authors.

1.2.3.1 Markov decision process(MDP)

- **S** is a set of states called the *state space*,
- **A** is a set of actions called the *action space* (alternatively, $\mathbf{A}_s$ is the set of actions available from state **S**)
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$
  is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after

transitioning from state s to state s', due to action a

The state and action spaces may be finite or infinite, for example, the set of real numbers. Some processes with countably infinite state and action spaces can be reduced to ones with finite state and action spaces. A policy function π is a (potentially probabilistic) mapping from state space **S** to action space **A.**

The goal in a Markov decision process is to find a good "policy" for the decision maker: a function π that specifies the action $\pi(s)$ that the decision maker will choose when in state **S**. Once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov Chain (since the action chosen in state **S** is completely determined by $\pi(s)$ and $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ reduces to $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s)$, a Markov transition matrix).

The objective is to choose a policy π that will maximize some cumulative function of the random rewards, typically the expected discounted sum over a potentially infinite horizon:

$E\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right]$ (where we choose at = π(st), i.e. actions given by the policy). And the expectation is taken over $s_{t+1} \sim P_{a_t}(s_t, s_{t+1})$

where γ is the discount factor satisfying $0 \leq \gamma \leq 1$, which is usually close to 1 (for example, $\gamma = 1/(1 + r)$ for some discount rate r). A lower discount factor motivates the decision maker to favor taking actions early, rather than postpone them indefinitely.

A policy that maximizes the function above is called an *optimal policy* and is usually denoted π^*. A particular MDP may have multiple distinct optimal policies. Because of the

Markov property, it can be shown that the optimal policy is a function of the current state, as assumed above.

1.2.3.2 Value-based method---Q learning

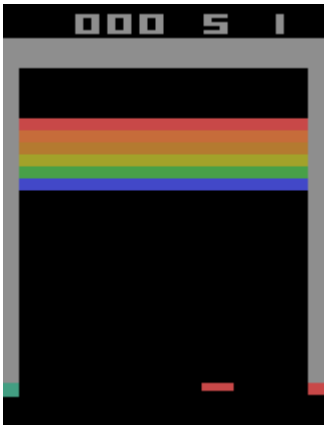
When using an off-policy method, Q learning is used to learn the value of taking action in a state, as well as learning Q value and deciding how to act in the real world. This is defined as the expected result when starting in s, doing action a, and then following pi (state-action value function). The information is presented in tabular format. According to Q learning, the agent employs any policy to estimate Q that maximises the future reward in order to maximise the future reward. As long as the agent keeps updating each state-action combination, Q is a direct approximation of Q*.

$$Q_{t+1}^{\pi}(s_t, a_t) = (1-\alpha)Q_t^{\pi}(s_t, a_t) + \alpha\left(R_t + \gamma \max_a Q_t^{\pi}(s_{t+1}, a)\right)$$

For non-deep learning approaches, this Q function is just a table:

	a_(1)	a_(2)	a_(3)	a_(4)
s_(1)	10	52	15	-2
s_(2)	14	30	8	7
s_(3)	42	0	-5	-10
s_(4)	-3	-1	-7	-20

For each element in this table, a reward value is represented by a value that is modified throughout the training process so that at steady-state, it reaches its expected value with the discount factor, which is comparable to the Q* value. When dealing with real-world situations, value iteration is not an option.



The state of the game in Breakout is represented by screen pixels: Image size: 84x84 pixels, number of images: 4, number of grey levels: 256 As a result, the Q-table contains the following number of rows:

$$256^{84\times84\times4} \approx 10^{69,970}$$

1.2.3.3 Policy gradient Algorithms

REINFORCE

To update the policy parameter, the REINFORCE (Monte-Carlo policy gradient) approach uses an estimated return from Monte-Carlo methods based on episode samples to update the policy parameter $\boldsymbol{\theta}$.

REINFORCE is effective because the expected gradient of the sample gradient is equal to the actual gradient

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a \mid s)] \\ &= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t)];\ \text{Because } Q^\pi(S_t,A_t) = \mathbb{E}_\pi[G_t \mid S_t, A_t]\end{aligned}$$

This allows us to calculate G_t from real-world sample trajectories and utilise the results to adjust our policy gradient as needed. It is based on a whole trajectory, which is why it is referred to as a Monte-Carlo approach.

There are only a few steps to this procedure:

1. Randomly choose the value for the policy parameter θ
2. Create one policy trajectory. $\pi_\theta: S_1, A_1, R_2, S_2, A_2, \dots, S_T$
3. For t=1,2,…T:
    1. Estimate the return G_t

2. Update policy parameters $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t)$

An often-used version of the REINFORCE algorithm is to subtract a baseline value from the return G_t in order to lower the variance of gradient estimates while maintaining the bias (Remember we always want to do this when possible). For example, a frequent baseline is to subtract state-value from action-value, and if this were to be done, we would use advantage $A(s,a) = Q(s,a) - V(s)$ in the gradient ascent update to improve performance. Baseline, in addition to a set of foundations of policy gradient, is effective for minimising variance.

Actor-Critic

The policy model and the value function are the two most important components of the policy gradient. Moreover, learning the value function alongside the policy makes a lot of sense because understanding the value function can aid in policy update efforts in a variety of ways, for as by lowering variance in vanilla policy gradients, which is exactly what the Actor-Critic technique accomplishes.

Actor-critic approaches are comprised of two models, which share parameters:

- The value function parameters w are updated by the critic, and depending on the algorithm, this might be action-value $Q_w(a \mid s)$ or state-value $V_w(s)$, respectively.
- actor makes changes to the policy parameters θ for $\pi_\theta(a \mid s)$ in accordance with the critic

To demonstrate how it works, check out the following simple action-value actor-critic algorithm:

1. Initialize **s**, θ, w, at random; sample $a \sim \pi_\theta(a \mid s)$
2. For t=1…T:
    1. Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s' \mid s,a)$
    2. Then sample the next action $a' \sim \pi_\theta(a' \mid s')$
    3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \ln \pi_\theta(a \mid s)$

4. Compute the correction (TD error) for action-value at time t:

$$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$$

and use it to update the parameters of action-value function:

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$$

5. Update a←a′ and s←s′

Two learning rates, αθ and αw, are predefined for policy and value function parameter updates respectively.

1.2.3.4 Deep reinforcement learning

All of the algorithms discussed above can be implemented and used in a variety of ways. For instance, Q-learning, a well-known form of reinforcement learning, generates a table of state-action-reward values when the agent interacts with the environment. These strategies perform well when dealing with a very simple environment with a small number of states and actions.

However, when a programmer is confronted with a complicated environment, where the total number of actions and states can approach infinity, or when the environment is non-deterministic and contains essentially infinite states, assessing every potential state-action pair becomes difficult.

In these situations, the programmer need an approximation function capable of learning optimal policies from sparse data. This is precisely what artificial neural networks accomplish. With the appropriate architecture and optimization function, a deep neural network can learn the optimal policy for a system without traversing all potential states. While deep reinforcement learning agents still require massive quantities of data (e.g., thousands of hours of gameplay in Dota and StarCraft), they are capable of solving issues that were previously unsolvable with traditional reinforcement learning methods.

For instance, a deep reinforcement learning model can extract state information from visual input such as camera feeds and video game visuals by utilising convolutional neural networks. Additionally, recurrent neural networks can extract meaningful information from sequences of frames, such as the direction of a ball or the presence or absence of a car. This capacity for complicated learning can aid RL agents in comprehending more complex environments and mapping their states to actions.

Comparable to supervised machine learning, deep reinforcement learning is a form of reinforcement learning. The model generates actions and adjusts its parameters in response to feedback from the environment. However, deep reinforcement learning faces several distinct obstacles that distinguish it from ordinary supervised learning.

In contrast to supervised learning problems, where the model is given a set of labelled data to work with, the RL agent only has access to the outcome of its own experiences. It may be able to learn an optimal strategy as a result of the experiences gained over several training episodes. However, it may also overlook numerous different optimal trajectories that would have resulted in more effective policies. Additionally, reinforcement learning requires evaluating the trajectories of state-action pairings, which is far more difficult to learn than supervised learning problems, which couple each training example with its expected outcome.

This increased complexity necessitates an increase in the amount of data required for deep reinforcement learning models. However, unlike supervised learning, which allows for the curation and preparation of training material in advance, deep reinforcement learning models gather data during training. In some forms of reinforcement learning algorithms, the data collected during an episode must be destroyed and cannot be used to speed up the model tuning process in subsequent episodes.

1.3 Software libraries

1.3.1 Tensorflow

TensorFlow is a software library or framework developed by the Google team to simplify the implementation of machine learning and deep learning principles. It integrates computational algebra and optimization approaches to facilitate the calculation of a large number of mathematical equations. TensorFlow is a well-documented framework that comes with a slew of machine learning libraries. It includes a few critical features and approaches for the same. TensorFlow is sometimes referred to as a "Google" product. It incorporates numerous machine learning and deep learning methods. TensorFlow is capable of training and running

deep neural networks for classification of handwritten digits, image recognition, word embedding, and the construction of various sequence models.

1.3.2 Gym and Atari games

Open-ai gym is used in this project. It is a development and comparison toolkit for reinforcement learning algorithms. It makes no assumptions about the agent's structure and is compatible with any library for numerical computing, such as TensorFlow or Theano. The gym library is a collection of test problems – environments — for developing reinforcement learning systems. These environments take actions from the agent, and then the reward of that action and the current state are returned. They are all important to provide the agent feedback for improving its performance and overall score. Atari_py is also needed to be installed alongside with open-ai gym library to provide the required atari games ROM files for the environment to be executed correctly

1.3.3 keras and Stablebaseline library

Keras is TensorFlow 2's high-level API: an approachable, highly productive interface for solving machine learning problems with a particular emphasis on modern deep learning. It provides fundamental abstractions and building blocks for rapidly developing and shipping machine learning solutions. Keras enables engineers and researchers to fully exploit TensorFlow 2's scalability and cross-platform capabilities.

Stable Baselines is a collection of improved Reinforcement Learning (RL) algorithms built on top of OpenAI Baselines.

1.3.4 other libraries

Other less important libraries used in this project are:

Numpy – array and data manipulation

Python 3.8 – python version with new features implemented

Tensorboard – graphical tool to visualize training process

1.4 DeepMind

1.4.1 Introduction

DeepMind Technologies, a British artificial intelligence subsidiary of Alphabet Inc. and research laboratory, was established in September 2010 as a result of the acquisition of DeepMind by Google. DeepMind was acquired by Google in 2014 for an undisclosed sum. The company's headquarters are in London, and it has research centres in Canada, France, and the United States, among other places. In 2015, it was transformed into a wholly owned subsidiary of Alphabet Inc, the parent company of Google.

1.4.2 Technologies

With the help of DeepMind, researchers have created a neural network that learns how to play video games in a manner eerily similar to that of humans. They have also developed a Neural Turing machine, which is a neural network that is capable of accessing external memory in the same way that conventional Turing machines are capable of doing, resulting in a computer that closely resembles the short-term memory of the human brain.

DeepMind made headlines in 2016 when its AlphaGo programme defeated world champion human professional Go player Lee Sedol in a five-game match that was the subject of a documentary film. AlphaZero, a more general programme, used reinforcement learning to defeat the most powerful programmes in go, chess, and shogi (Japanese chess) after a few days of play against itself. DeepMind made significant progress in the problem of protein folding in 2020.

1.4.3 Deep reinforcement learning

In contrast to other artificial intelligence systems, such as IBM's Deep Blue or Watson, which were developed for a specific purpose and only function within that purpose's scope, DeepMind claims that its system is not pre-programmed, but rather learns from experience, using only raw pixels as input data. Technically, it makes use of deep learning on a convolutional neural network, as well as a novel kind of Q-learning, which is a form of model-free reinforcement learning, to achieve its results. They put the system through its paces on video games, particularly early arcade classics such as Space Invaders and Breakout. Without changing the code, the AI learns how to play the game and, over a period of time, for a few games (most

notably Breakout), is able to play more efficiently than any human could ever hope to be.

In 2013, DeepMind presented research on an artificial intelligence system that may outperform human abilities in games such as Pong, Breakout, and Enduro, while also outperforming state-of-the-art performance in games such as Seaquest, Beamrider, and Q*bert, according to the company. According to reports, this work resulted in the company's acquisition by Google. DeepMind's artificial intelligence had already been applied to video games produced in the 1970s and 1980s; development was still in progress on more complicated 3D games such as Quake, which initially emerged in the 1990s and was developed by DeepMind.

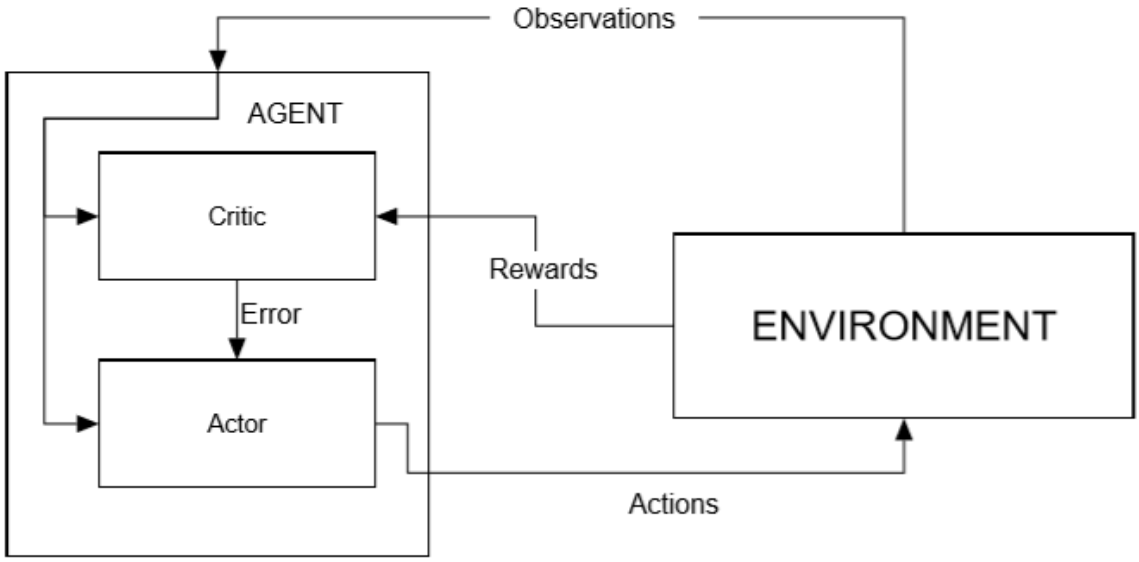
In 2020, DeepMind released Agent57, an artificial intelligence agent that outperforms humans on all 57 games in the Atari2600 suite, according to the company.

Chapter 2
Methods

This Chapter provides an overview of the structure of this programme and the adjustments that have been made to each component of this programme to archive better overall performance on different Atari games.

This programme is developed using Python, and the required library and packages are written in python, too; C++, Java or other languages were not used in this programme.

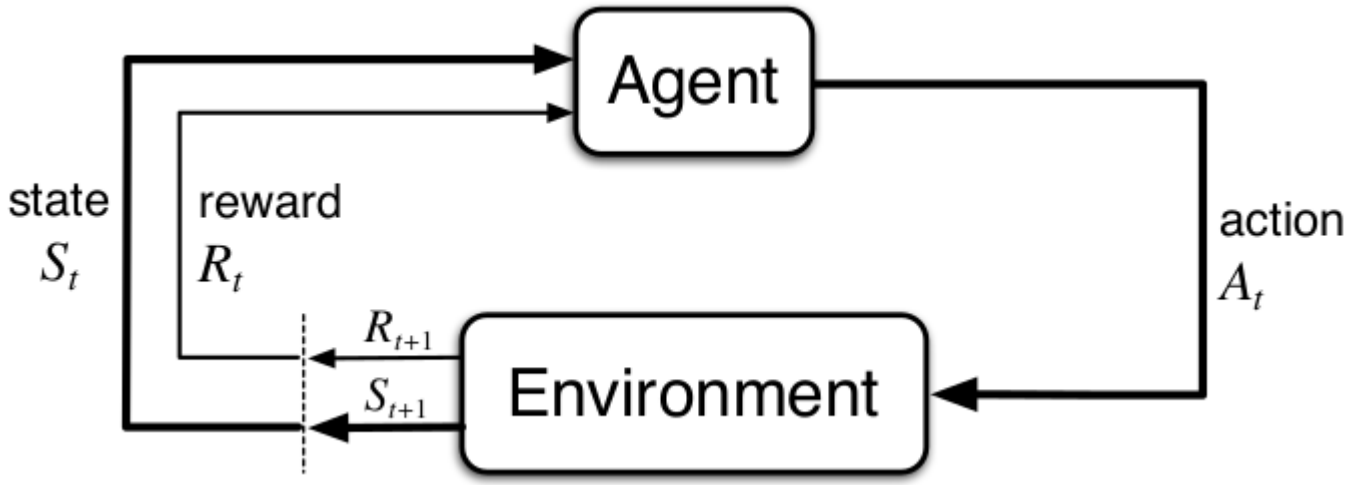
2.1 Structure



Each central element of the model is wrapped up in two classes, which are then instanced with custom parameters defined by the user. Each of these classes provides methods that are called by the main programme when learning begins, and each of these classes contains properties that impact how the learning process is carried out.

The principal algorithm codes are based on stable baseline [1], an open-source package that can be used as a python library.

2.2 Environment

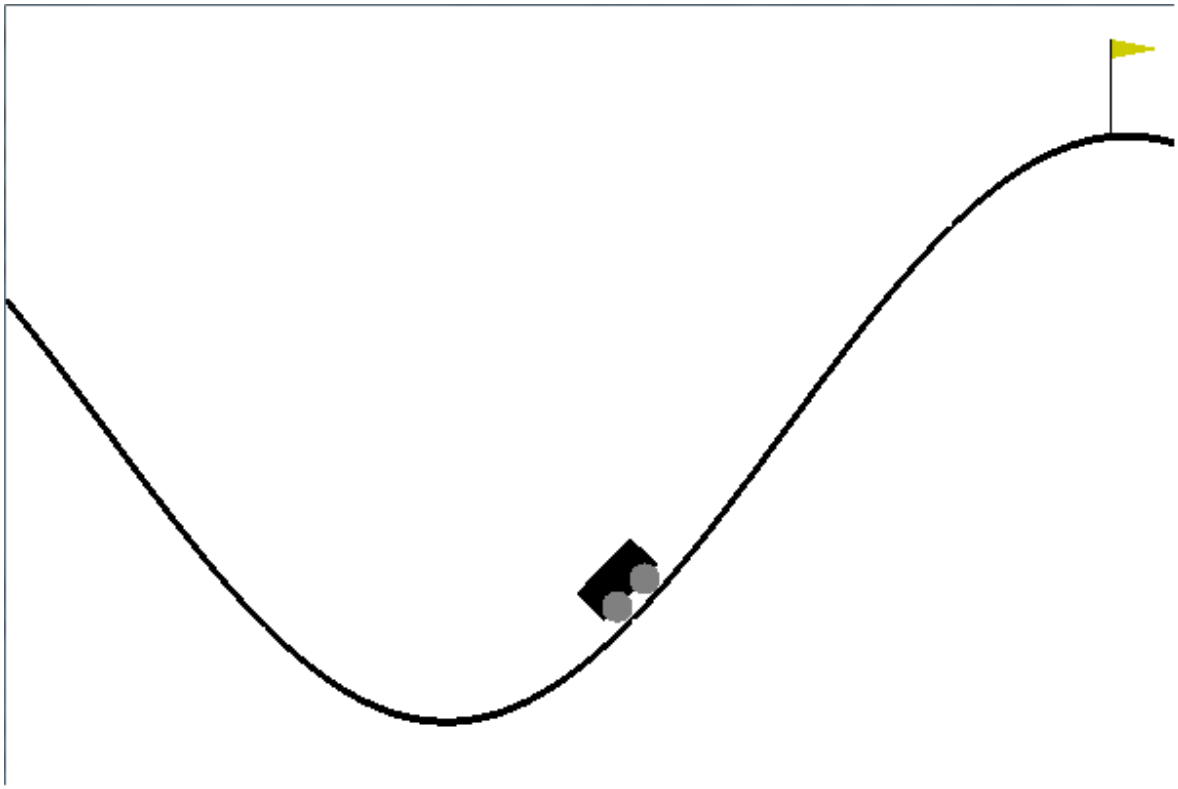


From the perspective of agents, the environment is defined as a universe of agents in which the state of an agent changes when a given action is performed on it. For example, an agent in a current state (St) performs an action (At), to which the environment reacts and responds, resulting in the agent being returned to a new state (St+1) and a reward (Rt+1). The agent chooses the next action based on the updated state and reward, and the loop is repeated until the environment is solved or terminated, whichever comes first.

Agent

The goal of an Agent is to figure out the most efficient way to complete a specific task in a given environment. The agent attempts to complete the task by performing actions (e.g., moving left, right, or stopping) in accordance with the strategy (or Policy). Consequently, after performing a specific action, the agent observes the state of the environment (Observation) and calculates a goodness score (or Reward) based on the action it just completed. In turn, the RL algorithm makes use of the received feedback information in order to upgrade or improve the existing strategy (or policy), thereby ensuring better performance in the future. It is possible for an agent to learn to play and win strategy games, such as Pacman (from the Atari video games), in which case the Pacman is the agent and the gaming construct represents the environment.

MontainCar-v0 is one of many environments from OpenAI Gym [2].



For example:

About MountainCar-v0: A car is travelling down a one-dimensional track, which is sandwiched between two "mountains." While the goal is to drive up the mountain on the right, it is not possible to do so in a single pass due to the car's engine's limitations. As a result, the only way to be successful is to drive back and forth in order to gain momentum. Let's tie together the general RL terms we've learned so far in the context of this straightforward environment.

Agent: the under-actuated motor car. *Observation*: A vector [car position, car velocity] represents the observation space in this case. Due to the fact that this is a one-dimensional problem involving a car moving on a curve-like feature, the car's location is given by a continuous value between [-1.2,0.6] and its velocity is given by a bounded continuous value between [-0.07,0.07]. When it comes to *rewards*, reward functions for these types of simulated environments are already built in the environment code, and in this case, the environment code generates a reward = -1 for any transition from any state, and a reward = +100 when the agent achieves the objective (the flag). *Action*: In this case, the action space is discrete and is denoted by the letters [Left, Neutral, Right]. *Episode*: A number of states and action sequences (in this case, approximately 1000) during which the agent attempts to complete the objective before re-initiating the process. In some cases, a trajectory can be defined as a segment of a full episode, and thus not every trajectory is a complete episode.

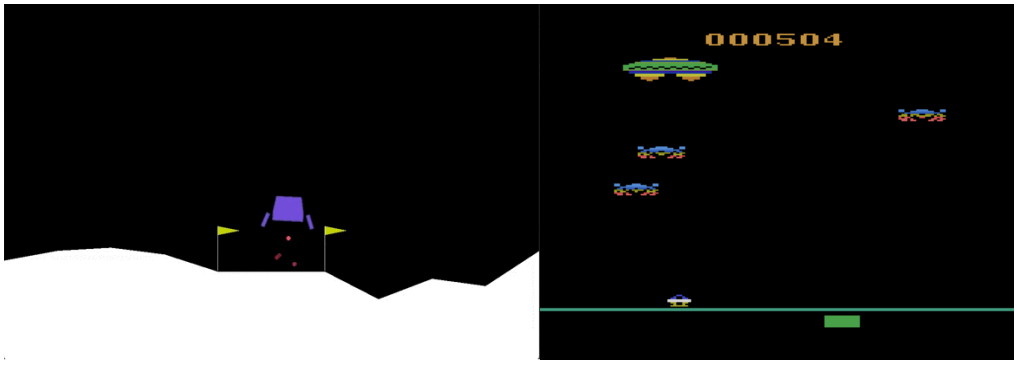


Figure 1 LunarLander V0 and assultV0

2.2 Replay Memory

Replay memory is a data structure that temporarily stores the agent's observations, allowing our learning procedure to update those observations multiple times. Despite the fact that it is straightforward to implement and understand, the role of replay memory within the overall RL procedure is not immediately apparent in many ways.

Experience Play in Actor-Critic with experience replay algorithm is a replay memory technique in reinforcement learning that involves storing the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a data-set $D = e_1, \cdots, e_N$, which is then pooled over many episodes and stored in a replay memory. The memory is then sampled randomly for a minibatch of experience. By treating the problem more like a supervised learning problem, this approach addresses the issue of autocorrelation leading to unstable training

2.3 Neural network and Model set-up

In DQL algorithms, A neural network that takes in a sequence of game states that have been sampled from an environment. At each stage of the sequence, we want our network to predict the action with the highest Q-value at that point in the sequence. Our network will produce an output that will refer to Q-values for each action that can be performed for each potential game state. Thus, we begin by defining a few convolutional layers, each with an increasing number of filters and decreasing stride-lengths as the layers move down the hierarchy. All of these convolutional layers are implemented using a Rectified Linear Unit (ReLU) activation function. Following this, we add a flatten layer to reduce the dimensionality

of the outputs from our convolutional layers to vector representations, which is useful for reducing the size of our convolutional layers.

```
model = Sequential()
model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu', input_shape=(3,height, width, channels)))
model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
model.add(Convolution2D(64, (3,3), activation='relu'))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(actions, activation='linear'))
return model
```

Figure 2 Neural network for DQL

All of the information from these representations is then sent into three densely connected layers that perform the regression of game states against Q-values for the actions that are now accessible. The output layer is densely connected, with a number of neurons that match to the action space of our agent. This layer has a linear activation function.

In ACER algorithm, there are many policy models can be used in stable-baseline library, and their parameters are fully customizable, such as the number of layers, the number of neurons etc. for this project, CnnPolicy has been used in this algorithm for Atari games training; this NN consists of three convoluted layers and one fully-connected layer, three convoluted network layers with the number of neurons set to 32, 64,64 respectively, for the first layer, kernel size has been set to 8, the stride is 4, second layer's kernel size is 4, stride is 2, kernel size of the third layer is 3, the stride is 1, and the one fully-connected layer is set to have 512 neurons. After some testing, it is clear that changing these parameters does not dramatically improve the agent's performance on Atari games, so these parameters are not often changed after it is set up properly.

```
activ = tf.nn.relu
layer_1 = activ(conv(scaled_images, 'c1', n_filters=32, filter_size=8, stride=4, init_scale=np.sqrt(2), **kwargs))
layer_2 = activ(conv(layer_1, 'c2', n_filters=64, filter_size=4, stride=2, init_scale=np.sqrt(2), **kwargs))
layer_3 = activ(conv(layer_2, 'c3', n_filters=64, filter_size=3, stride=1, init_scale=np.sqrt(2), **kwargs))
layer_3 = conv_to_fc(layer_3)
return activ(linear(layer_3, 'fc1', n_hidden=512, init_scale=np.sqrt(2)))
```

Figure 3 Neural network for ACER

2.4 RL algorithms

DQL(Deep Q-learning)

When a given state is reached, a policy network is formed to return the current actions and Q-Values. In addition, a target network is established and updated in accordance with the target update factor in order to return the target Q-Values of a given state. Value-based techniques use the fundamental Mean-Squared loss function, which accepts both the current and intended Q-Values as input parameters. A value assigned to a policy network state is referred to as an action value. The target Q-Values are obtained by collecting all of the states preceding each state and selecting those that are not the episode's final state; final states are activities that lead to the episode's conclusion and, as a result, have a reward of zero, and are therefore excluded from consideration when updating the policy network. The goal Q-Values are calculated by averaging the maximum Q-Values of each subsequent state and dividing them by the total number of states.

ACER(Actor-Critic with experience replay)

ACER is relevantly a more stable way to train an agent than Deep Q-Learning because it not only utilizes the advantage of deep neural networks but also uses variance reduction techniques. ACER combines ideas of parallel agents from A2C with the provision of a replay memory, similar to DQN. Truncated importance sampling with bias correction, stochastic duelling network designs, and a new trust region policy optimization method are all included in the ACER.

2.5 Training

A method that iterates through the number of episodes defined by the user parameters is executed in the main.py file, and this is where the training is carried out. When an episode occurs in the environment, the agent goes through a series of training loops in which each iteration defines a step that the agent must take. As a result of each step, the agent generates an action based on the action strategy that is currently in use, and this action is then implemented in the environment following the completion of the action, the environment returns a reward as well as the next processed state to the caller. Finally, the current state, action, reward, and next state are all stored in replay memory so that they can be sampled for optimization purposes in the future state. The agent retrieves a batch of experiences from the replay memory in order to optimise. Upon completion of an episode, all of the information about the episode is saved for future reference, and the user's performance (total reward, steps taken, moving average, current epsilon, time taken) is displayed to him or her. Furthermore, after a predetermined number of episodes, the current progress of the agent is

displayed on a reward graph, which depicts the agent's performance in real-time. Once the maximum number of episodes has been reached and training has been completed, the stored results of each episode are exported to the logfile(event file) using the tensorboard technique, and the weights of the current agent are saved, allowing the current training agent to be executed again at a later time without interruption, in addition to this, I implemented a callback function that can automatically save all weights information about current agent into a file every 10,000 steps of training.

2.6 User customization

This program has many parameters that can be changed, such as selecting a game on the environment, choosing the visibility of the training process, changing training agent policy, and changing the learning rate. Furthermore, the number of time steps for the agent to train is customizable, and the tensorboard log saving path can also be specified on the parameter list. All available settings are introduced in each comment of the code file.

2.7 Implementation

There were many iterations of each agent executing each environment before it became clear that some agent parameters needed to be kept constant in order for efficient learning to begin. It is advantageous to have consistent parameters not only for the purpose of minimising potential discrepancy errors but also for the purpose of comparing the generalized performance of the agent in different games. All variable parameters are changed for each testing run, in order to archive the best learning efficiency. First and foremost, it was discovered that a learning rate of 0.0007 was the most effective for each of the environments (different games). When we increase the learning rate, it was true that the agent failed to converge on an optimal policy in some environments, and that general performance after training for a long time to the later stage of the game did not come as expected. If we somehow lower the learning rate, on the other hand, it is easy to spend much more time to archive the expected game score, resulting in very low efficiency. In Atari environments, actions taken early in the game have a significant impact on the reward that the agent receives later in the game; In space invader, hitting enemy ships that are closer to the ship we controlled earlier, will make our ship survive longer on the later stage of the game because the enemy ships are going closer and closer to our ship, there is little space for our ship to dodge. After testing this environment many times. We found that there should

not only archive more and more rewards, in the meantime, keeping our ship safe is also important.

2.8 Space invader

Space Invaders is a fixed-frame shooter in which the player controls a laser cannon that moves horizontally across the bottom of the screen, firing at aliens above. In this game, the aliens start out as five rows of eleven that move left and right as a group, shifting downward each time they reach the edge of the screen. The objective is to eliminate all of the aliens through the use of firearms. The game ends immediately if the intruders reach the bottom of the screen, despite the fact that the player has three lives to use. The aliens attempt to demolish the player's cannon by firing projectiles at the player's position. The laser cannon is partially protected by stationary defence bunkers, which are gradually destroyed by the aliens from the top-down, and from the bottom up if the player fires while beneath one.

As the aliens are defeated, the speed of their movement and the speed of the game's music increase. Defeating all of the aliens triggers another wave that starts at a lower level, creating a loop that can go on indefinitely. A special "mystery ship" will occasionally move across the top of the screen, and destroying it will result in bonus points being awarded to the player.

This game provides a very complex environment while archiving higher scores as possible is the default way for an agent to train, we found an issue, that the cannon will wait on the left side of the bottom screen where he can dodge all bullets from aliens and trying to shoot the special "mystery ship" to gain bonus score, but cannon seems can only shoot half of all “mystery ship”, because “mystery ship” moves rapidly from one side to another side of the screen. After 1m steps of training, the agent realizes that it should start to confront these aliens instead of waiting and shooting the unstable “mystery ship”, in order to archive a higher score, the cannon need to learn to dodge the bullet-rains from aliens, of course, from the beginning of dodging, the overall scores are even lower than previous training results (shoot mystery ship) in the result of died so many times under heavy attacks from the enemy. After another 4M of steps in training, the cannon can dodge most of the aliens successfully, so it starts to shoot precisely on aliens while keeping a high dodge rate, when training steps reach 10M, it seems to be able to shoot 75% of all aliens and sometimes caught some “mystery ship”, and though more training, it can archive higher scores by destroying all enemy aliens.

2.9 other game

Lunar Lander is a Lunar Lander-style single-player game in which the player attempts to land a lunar landing module on the Moon. The game is presented in black and white vector graphics and portrays the terrain and landing module from the side. The player is provided information about the module's speed, altitude, and fuel, as well as the score and time spent in the game, at the top of the screen. The ground is rocky, with only a few flat spots suitable for landing. The player steers the module to a safe landing place by controlling its orientation and firing the thruster. The module is always in the centre of the screen, with the landscape scrolling horizontally beneath it as it wraps the single screen width of terrain indefinitely.

If the player successfully lands the module, they will receive points based on how lightly the module landed and the difficulty of the landing spot, as well as a tiny amount of fuel. A flashing bonus multiplier, which is bigger for smaller areas, highlights the safe landing spots. A limited number of points are rewarded if the module crashes—which can happen if it is going too rapidly, is turned too far from vertical when it strikes the ground or lands on an uneven surface. The perspective shifts to a close-up picture of the lander as it approaches closer to the surface. The player has a finite amount of fuel, which is depleted as the module is controlled. Regardless of whether the player lands safely or crashes, the game restarts with a new set of terrain and the player's remaining fuel. When the module runs out of fuel and hits the earth, the game is over.

The agent starts to learn the controls of left and right, in the beginning, the agent it is hard to maintain the balance of the lander, after some steps, balance starts to be maintained correctly and the lander is intact. This is clearly not the end, after many steps of training, the agent can archive a higher score by landing on a flat surface by manipulating the controls of left and right, for example, giving more thrust on left will result in lander going right.

Chapter 3
Results

3.1 Experiment results

The graphs below show the rewards of trained agents can get with training weights after training with certain number of timesteps.

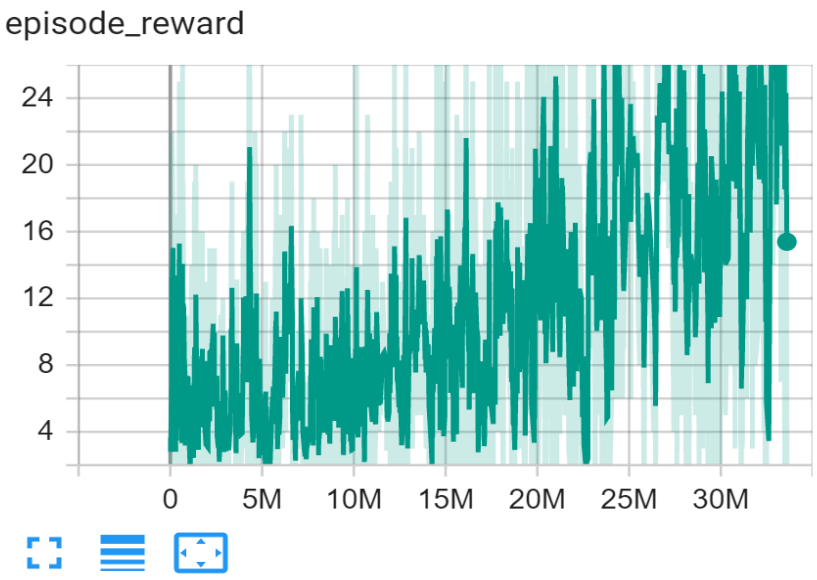


Figure 1 Space Invader

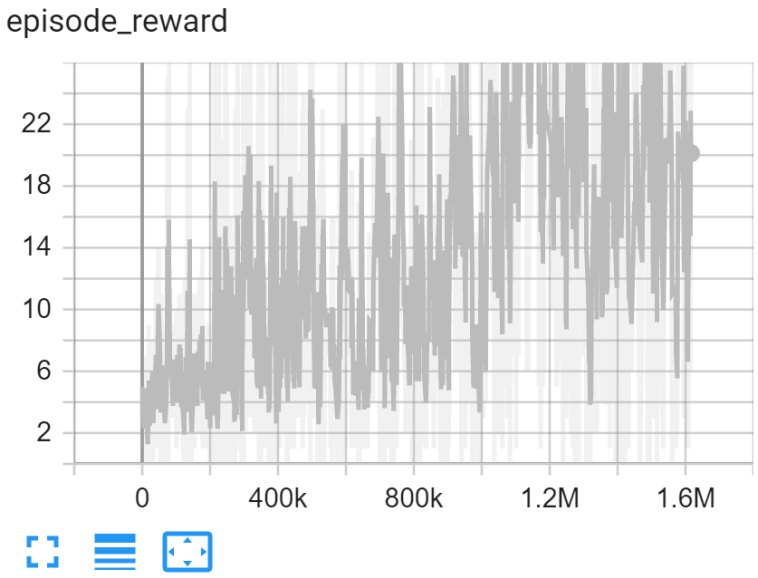


Figure 2 Assault

Figure 3 Lunar lander

Results of trained agents:

Environments	Actor-critic	Random	Human	DQN
Space Invader	750(+-50)	150(+-10)	800(+-100)	300 (+-30)
Lunar Lander	249(+-40)	69(+-20)	224(+-20)	100 (+-10)

Assault	1030(+-50)	130(+-20)	1000(+-10)	300(+-10)

This is the summary table of average scores after running trained agents(DQN, ACER) on different games for ten episodes. The random column represents the result after running on the three Atari games with randomised action inputs, and the human column is the result that the average human player can get from playing the three Atari games.

3.2 Analysis of the results

The lunar lander and Assault took fewer training steps than the space invader to achieve a big high score, while the space invader took much more training steps because the space invader is way more complicated than the other two games. Due to computation power constraints, it may need to take a much longer time to train for better results in space invader.

The space invader training using the ACER algorithm is seemed to be archiving more stable and sometimes higher scores than the average player; it would archive better scores if keep training for longer steps; it is just because space invader is a complex game for either computer or human, humans are seen to be making more mistakes due to their lack of concentration. As long as computers are trained, they will make much fewer mistakes while ranking high scores with stable performance, and under heavy bullets from alien machines will always make the most rational decision without fear and feeling the pressure; compared to the ACER algorithm, the result of DQL algorithm seems to be much worse, agent trained with this algorithms shows much weaker performance while marking dodging action under alien fires, and it has higher variance than ACER algorithm, so it causes weaker stable performance than ACER algorithm.

Assault is the same type of game compared to space invader. Still, it is technically more straightforward than a space invader with much fewer enemy targets. From the perspective of scores, the computer can archive better scores than the human player because the laptop would archive its optimal score much quicker without complicated controls and reward schemes. While lunar lander is also a game where computers can easily archive better results than human players because of its simple reward scheme and rules, computers will easily spot the best spot to land the moon lander module and control the lander in a stable way that is hard for the human player to do.

Chapter 4
Discussion

4.1 Conclusions

Overall, the agents archived expected results, and training on three Atari games proves the generalisation of the agent algorithm, although the results of agents were not much higher than that of normal players due to computation power restriction. The advantage of using the agent is that it will learn and master games with easy controls, and the total amount of time used on training to get optimal results will increase according to the complexity of game controls.

4.2 Possible enhancements can be made

While the current results from two-game agents appear to be better than the results from average human players and random agents, there is undoubtedly significant space for improvement.

First of all, the ACER agent's parameters can be adjusted further to archive higher scores; the ACER algorithm's potential has not been fully exploited. Numerous experiments exploring the effect of varying every parameter are not possible due to computational power constraints, they could be time-consuming.

Secondly, with the knowledge and experience gained by implementing and using DQN and ACER algorithms, developing a new DRL algorithm with improved performance on a wide variety of games and ease of use should be an archivable goal.

Finally, due to the low performance of the computer, there is not enough time to test these two agents on all classic Atari games, such as breakout, PacMan etc.

4.3 Ideas for future work

Except for using reinforcement learning in playing games, we can also use it in different scenarios. For example, in a self-driving car system, we can train an agent for optimizing

trajectories from its environment, the agent can get a reward after every time step by executing an action in the state, then by finding the way to get the best reward, we can get optimized trajectory for the car.

We can also use reinforcement learning on Customizing which advertisements are displayed to the end-user in order to increase the click-through rate. It is possible to use RL in a large-scale ad recommendation system because it allows for dynamic adaptation of advertisements in response to reinforcement signals and because it has proven successful in real-world applications. For example, retargeting users who have previously seen the product and showing the product to users who have not yet seen it are both valid strategies. When a user clicks on an advertisement, they are taken to a landing page. The total click-through rate (CTR) of the advertisement is referred to as the reinforcement signal. When a time step is reached, the reinforcement learning model calculates weights, which are then updated in real-time in response to reinforcement signals received. As a result, it learns how to respond to reinforcement signals in the most effective manner at each time step.

List of References

[1] stable baseline. Reinforcement learning algorithms packages. *https://stable-baselines.readthedocs.io/en/master/modules/policies.html*

[2] open-ai gym. Atari games environment for Reinforcement Learning. *https://gym.openai.com/docs/*

[3] Mohammed Alghamdi. Dr Matteo Leonetti. *Leeds University: COMP3611 Machine Learning-Lecture slides*

[4] Ajitesh Kumar. *Reinforcement Learning Real-world examples. https://vitalflux.com/reinforcement-learning-real-world-examples/*

[5] Víctor Mayoral Vilches. Reinforcement learning in robotics. *Reinforcement learning in robotics | by Víctor Mayoral Vilches | Medium*

[6] Lilian Weng. Policy Gradient Algorithms. *Policy Gradient Algorithms | Lil'Log (lilianweng.github.io)*

[7] Dhanoop Karunakaran. *The Actor-Critic Reinforcement Learning algorithm. https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14*

[8] Nihit Desai. Abhimanyu Banerjee. *Deep Reinforcement Learning to play Space Invaders. https://nihit.github.io/resources/spaceinvaders.pdf*

[9] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, *Nando de Freitas. Sample Efficient Actor-Critic with Experience Replay*

[10] Audrunas Gruslys, Will Dabney, Mohammad Gheshlaghi Azar, Bilal Piot, Marc Bellemare, Remi Munos. *The Reactor: A fast and sample-efficient Actor-Critic agent for Reinforcement Learning*

[11] Space Invaders. *Wikipedia page. https://en.wikipedia.org/wiki/Space_Invaders*

[12] Chloe Wang. *Using Deep Reinforcement Learning To Play Atari Space Invaders. https://chloeewang.medium.com/using-deep-reinforcement-learning-to-play-atari-space-invaders-8d5159aa69ed*

[13] Roland Meertens. *Introduction to OpenAI gym part 3: playing Space Invaders with deep reinforcement learning http://www.pinchofintelligence.com/openai-gym-part-3-playing-space-invaders-deep-reinforcement-learning/*

[14] Deep reinforcement learning. *Wikipedia page.* *https://en.wikipedia.org/wiki/Deep_reinforcement_learning*

[15] callback. *documentation introductions.* *https://stable-baselines.readthedocs.io/en/master/guide/callbacks.html*

[16] How To Think About Replay Memory. *https://jacobbuckman.com/2021-02-13-how-to-think-about-replay-memory/*

[17] Ruishan Liu. James Zou. The Effects of Memory Replay in Reinforcement Learning

[18] Policy Networks. Stable baseline documentations. *https://stable-baselines.readthedocs.io/en/master/modules/policies.html*

[19] M. M. Hassan Mahmud. *Constructing States for Reinforcement Learning*

[20] States, Observation and Action Spaces in Reinforcement Learning. *https://medium.com/swlh/states-observation-and-action-spaces-in-reinforcement-learning-569a30a8d2a1*

[21] Atari 2600. Wikipedia page. https://en.wikipedia.org/wiki/Atari_2600

[22] Machine learning. Wikipedia page. https://en.wikipedia.org/wiki/Machine_learning

[23] JAMES CHEN. *Neural Network.* *https://www.investopedia.com/terms/n/neuralnetwork.asp#:~:text=A%20neural%20network%20is%20a,organic%20or%20artificial%20in%20nature*.

[24] Matthew Stewart. *Introduction to Neural Networks.* *https://towardsdatascience.com/simple-introduction-to-neural-networks-ac1d7c3d7a2c*

[25] Matthew Stewart. *Intermediate Topics in Neural Networks.* *https://towardsdatascience.com/comprehensive-introduction-to-neural-network-architecture-c08c6d8e5d98*

[26] Matthew Stewart. *Neural Network Optimization.* *https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0*

[27] Sanket Doshi. *Various Optimization Algorithms For Training Neural Network.* *https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6#:~:text=Many%20people%20may%20be%20using,help%20to%20get%20results%20faster*

[28] Reinforcement learning*. Wikipedia page.* *https://en.wikipedia.org/wiki/Reinforcement_learning*

[29] Convolutional neural network. *Wikipedia page.* *https://en.wikipedia.org/wiki/Convolutional_neural_network*

[30] Markov decision process. *Wikipedia page.* *https://en.wikipedia.org/wiki/Markov_decision_process*

[31] Barak Or. *Value-based Methods in Deep Reinforcement Learning.* *https://towardsdatascience.com/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1*

[32] Keras-RL DQL algorithms https://keras-rl.readthedocs.io/en/latest/agents/dqn/

[33] Keras deep learning library https://keras.io/

[34] deepmind wiki page https://en.wikipedia.org/wiki/DeepMind

Appendix A
Self-appraisal

A.1 Critical self-evaluation

This project implements a general solution for training an agent to play Atari games and archive high scores as the average player can do. The overall scores archived seem as expected, and this agent can archive similar results in different Atari games without manually modifying much about hyperparameters.

A.2 Personal reflection and lessons learned

Lots of settings on hyperparameters can be further adjusted for specific games to archive better and higher scores; at the beginning of this project, I got some trouble finding the most suitable algorithms for playing different types of Atari games, so I encountered some problems with using some algorithms which may be more effective on Linux system(mine is windows 10) or using some algorithms which may be more effective on playing some games with long term reward strategy. So lots of time can be saved if I deeply learn each type of algorithm beforehand.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

All reading materials and software tools used in this project are open-sourced and can be found on the internet. There are no legal issues with this project.

A.3.2 Social issues

This project is only done for playing Atari games, purely for research and entertainment purposes. Maybe some players can use this to cheat in some player vs player games to destroy the experience of other players, but this project will not be shared on the internet.

A.3.3 Ethical issues

The average player’s results are collected with their agreements; there should be no further ethical issues.

A.3.4 Professional issues

This project follows the guidance from relevant reading materials, but consider this project is more experimental than a formal product, so this project may be hard to use for average users without relevant knowledge backgrounds.

Appendix B

External Materials

Stable baseline-- https://stable-baselines.readthedocs.io/en/master/guide/quickstart.html is a reinforcement learning library with many different algorithms.

OpenaiGym--- https://gym.openai.com/docs/ is a toolkit for testing reinforcement learning algorithms; it includes all Atari games.

Tensorflow--- https://www.tensorflow.org/ free and open-source library for machine learning

Python 3.8--- https://www.python.org/

Atari_py -- https://github.com/openai/atari-py

Tensorboard--- https://www.tensorflow.org/tensorboard/ graphic evaluations tool

Keras-rl -- https://keras-rl.readthedocs.io/en/latest/agents/overview/ machine learning algorithms library