**Cruise Battle Game with A.I. Implementation using Deep Reinforcement Learning**

By

M Zaki Al Akkari.

**February 2022**

FYP Declaration

|  |  |
| --- | --- |
| **Student Name:**  **M.Zaki Al Akkari-K00233639** | **Date: 24/04/2022** |
| **Department:**  **Information Technology** | **Award Title**  **B. Sc. in Software Development** |
| **Title:**  **Artificial Intelligence** | **Supervisor:** |

I wish to confirm that my FYP submission document and software is my own unaided effort and that I have acknowledged all sources of information and ideas used in this submission.



Signed: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_24/04/2022\_\_\_\_\_\_\_\_\_\_\_\_



**Acknowledgements**

I would like to thank my family, friends, teachers and classmates who are always there for me and encouraging me to succeed.

**Abstract**

In this report, I am going to develop a 2D battleship game. The game can be played by a single player. In addition, I will develop an AI agent to learn how to win the game using Reinforcement Learning. The AI agent is equipped with a deep learning model to evaluate the game scene and decide a suitable action. At the training phase, the AI agent will explore the game by playing in the random mode. It will then learn from past experiences and eventually develop its own strategy to win the game and gain maximum rewards. My experiments reported that the AI agent after training on thousands of game episodes was able to significantly enhance its strategy and wing the game.

Contents

[Chapter 1 Introduction 8](#_Toc112060345)

[1.1 Overview 9](#_Toc112060346)

[1.2 Aims 10](#_Toc112060347)

[1.3 Objectives 11](#_Toc112060348)

[1.4 Challenges 11](#_Toc112060349)

[1.5 Structure of Report 11](#_Toc112060350)

[Chapter 2 Literature Review 12](#_Toc112060351)

[2.1 Introduction 13](#_Toc112060352)

[2.2 Artificial Intelligence 15](#_Toc112060353)

[2.3 Supervised Learning 16](#_Toc112060354)

[2.4 Unsupervised Learning 17](#_Toc112060355)

[2.5 Reinforcement Learning 17](#_Toc112060356)

[2.6 Gradient Descent Learning Method 20](#_Toc112060357)

[2.7 A.I. in games 24](#_Toc112060358)

[2.8 Neural Networks 25](#_Toc112060359)

[2.9 Deep Learning 28](#_Toc112060360)

[2.10 Genetic algorithms 30](#_Toc112060361)

[2.11 Game Development 32](#_Toc112060362)

[2.12 Minimax algorithm 38](#_Toc112060363)

[2.12.1 Heuristic algorithm 39](#_Toc112060364)

[2.13 Optimization task of Battleship Game 40](#_Toc112060365)

[2.14 Markov Decision Process 41](#_Toc112060366)

[Chapter 3 The Battleship 2D Game 42](#_Toc112060367)

[3.1 Game Design and Rules 43](#_Toc112060368)

[3.1.1 Game rules and setup 43](#_Toc112060369)

[3.1.2 Class Design 44](#_Toc112060370)

[3.2 Python Libraries 45](#_Toc112060371)

[3.2.1 OpenCV 45](#_Toc112060372)

[3.2.2 OpenAI Gym 45](#_Toc112060373)

[3.2.3 Tf\_agent 45](#_Toc112060374)

[3.2.4 Shapley 45](#_Toc112060375)

[3.3 Game Implementation 45](#_Toc112060376)

[3.3.1 ShapedObject Class 45](#_Toc112060377)

[3.3.2 AgentShip 46](#_Toc112060378)

[3.3.3 The Enemy Ship Class: 47](#_Toc112060379)

[3.3.4 The Agent and Enemy Missile Classes 49](#_Toc112060380)

[3.3.5 The BattleShipGame class 50](#_Toc112060381)

[3.4 Running The Game Using Random Strategy Agent 51](#_Toc112060382)

[Chapter 4 Adding AI Agent Using Deep Learning 52](#_Toc112060383)

[4.1 Developing AI Agents with OpenAI Gym Environments 53](#_Toc112060384)

[4.1.1 Interacting with The Gym Environment 53](#_Toc112060385)

[4.1.2 Creating Custom OpenAI Environment 54](#_Toc112060386)

[4.1.3 Defining the Observation Space 54](#_Toc112060387)

[4.1.4 Defining the Action Space 54](#_Toc112060388)

[4.2 Deep Q-Learning 54](#_Toc112060389)

[4.2.1 Markov Decision Process Revisited 55](#_Toc112060390)

[4.2.2 Q-Learning 56](#_Toc112060391)

[4.2.3 Deep Q-Learning 56](#_Toc112060392)

[4.2.4 The discount factor in Q-learning 56](#_Toc112060393)

[4.3 Implementing the AI Agent Class 56](#_Toc112060394)

[4.3.1 Adding Memory Buffer 56](#_Toc112060395)

[4.3.2 Adding Deep Convolutional Networks to Agent 56](#_Toc112060396)

[4.3.3 Applying Q-Learning Training Step 57](#_Toc112060397)

[4.3.4 The Epsilon-Greedy Policy Method 57](#_Toc112060398)

[4.3.5 The Full Picture 58](#_Toc112060399)

[4.3.6 Training Settings 59](#_Toc112060400)

[4.4 Testing and Analyzing the Agent 59](#_Toc112060401)

[Chapter 5 Conclusion and Future Work 60](#_Toc112060402)

[Chapter 6 References 62](#_Toc112060403)

Table of Abbreiviation

A.I.: Artificial Intelligence

ANN/NN: Artificial Neural Networks/Neural Networks

Cruise Battle Game: CBG

Rectified Linear Unit activation function: Relu

**Figures**

[Figure 1: Classic Battleship Tabletop Game. 9](#_Toc101713209)

[Figure 2: Neuron 15](#_Toc101713210)

[Figure 3:Different types of Neural Networks 16](#_Toc101713211)

[Figure 4:Elements of Reinforcement Learning in Auto Racing Car. 21](#_Toc101713212)

[Figure 5:An error surface in gradient descent can be thought of as mountains and valleys as in the image. 23](#_Toc101713213)

[Figure 6: Steps of the gradient descent. 25](#_Toc101713214)

[Figure 7: example of neural network. 28](#_Toc101713215)

[Figure 8: Detailed structure of ANN 29](#_Toc101713216)

[Figure 9: A basic convolutional neural network. 31](#_Toc101713217)

[Figure 10: The artificial neural network is mapped to the environment cells in this diagram. 31](#_Toc101713218)

[Figure 11: Chess pieces. 35](#_Toc101713219)

[Figure 12: Chess pieces on visual display. 35](#_Toc101713220)

[Figure 13: GUI of a chess game. 36](#_Toc101713221)

[Figure 14: Heat map to control the board center. 38](#_Toc101713222)

[Figure 15: Heat map focusing on the king, with the king on square g1. 38](#_Toc101713223)

[Figure 16: tree containing openings. 39](#_Toc101713224)

[Figure 17: Pattern placements conforming to the Battleship game rules (pattern deployment rules) are shown as an example of a problem instance. This example corresponds to our modification (environment of size nxn = 7x7 and two "L"-shaped patterns of size 2x3). 43](#_Toc101713225)

[Figure 18:Use case diagram of an A.I. Battleship Game 47](#_Toc101713226)

[Figure 2Sample of Java Spring Dependencies 54](#_Toc101713227)

[Figure 3When using Spring dependencies the Battleship project becomes unloadable and will not able to load rest of the files of the error in dependencies. 54](#_Toc101713228)

[Figure 21: Battleship game with ship models and two 10\*10 boards. 55](#_Toc101713229)

# Introduction

## Overview

Artificial Intelligence (AI) is becoming an essential part in our daily life. From computer vision applications such as facial recognition security check in our mobile phones to autonomous driving cars. The increased AI capability has reached a level where it can easily compete with human intelligence. One of the main subfields in AI which has a major role in such developments is Reinforcement Learning (RL). RL allows an AI agent to learn any given task by experience. For instance, if we want to teach an AI agent how to professionally play a game, it is very sufficient to let the AI experience the game (even in random mode) and then it will eventually learn from its mistakes. This subfield of AI is the one that is mainly used in developing self-driving cars. For instance, Amazon Web Services (AWS) has launched Deep Racer service which aims to develop self-driving cars using RL based on a 3D virtual track. The trained AI agents are then deployed to drive real AWS deep racer game cars. RL has been also widely used in game development industry such as playing Atari games, 3D games and even chess.

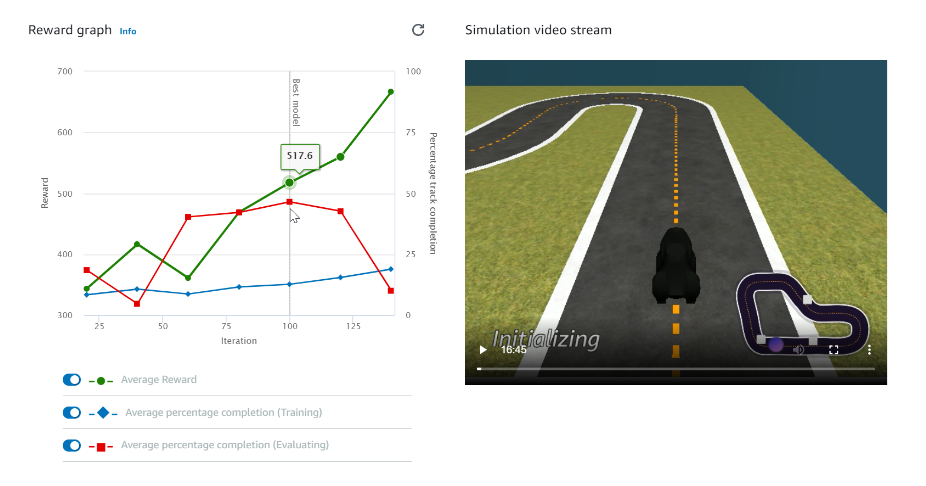


Figure : Training AWS Deep Racer agent..

The interesting thing is that training AI agents using RL does not require any expensive equipment. We can train agents to ride a spaceship using our own machine. This becomes even more accessible with the launch of OpenAI in 2015 by Elon Musk. Open AI contains an RL platform called gym. This platform has thousands of built-in 2D and 3D games and environments that are specially designed to train state-of-the-art AI agents to learn extremely complex tasks. Further to that, it provides a template to easily create a custom environment that would be compatible to train AI agents easily.

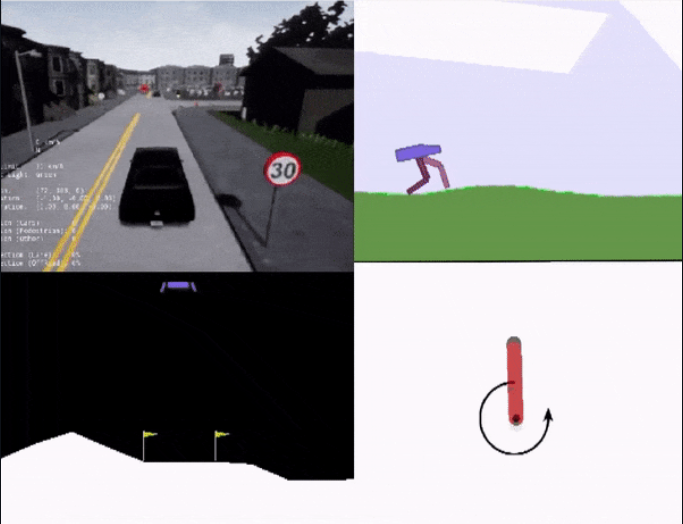


Figure Example of OpenAI gym environments.

## Aims

The aim of this project is to develop a Battleship 2D game and train a deep learning AI agent to professionally learn how to play and win the game. My initial plan was to develop the well-known battleship board game, however, I found that it would be difficult to use AI in this board game due to its stochastic nature and limited strategy development techniques. Hance, I will develop a more advanced 2D game of a navy battle between agent ship and 10 enemy ships that are firing missile towards the agent and the agent has to manipulate such missiles and destroy the enemy ships.

A picture containing text

Description automatically generated

Figure : My battleship game.

## Objectives

There are two main objectives. First, I need to develop the 2D game. The game can be played by a human, AI agent. The game implements the OpenAI gym env template so that it would be easier to develop compatible AI agents. After that, I will develop the AI agent to learn how to play the game using RL. Particularly, the AI agent will use deep convolutional networks to visually evaluate its state in the game and decide the appropriate action. The AI agent will be trained using deep Q-learning approach.

## Challenges

There are several challenges that I faced when working on my FYP. First, I had to develop the game from scratch as this game is not included in the Open AI gym platform. Secondly, developing the AI agent requires knowledge not only in RL but also in computer vision since the AI agent will use deep convolutional networks that will analyze observations from the game. Thirdly, training the AI agent took very long days as training a deep learning modle requires high performance computing using GPU. I used the *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems* book by Aurelien Geron as a guiding source in developing the AI agent.

## Structure of Report

The FY document is organized as follows:

* Chapter 2: Provides a background and literature review on AI topics that are related to my project.
* Chapter 3: Provides details regarding the game design and development process.
* Chapter 4: Provides details on the development of the AI agent and the performance of the AI agent after learning the game.
* Chapter 5: Provides conclusion to this work and draws future works.

# Literature Review

## Introduction

The history of developing A.I. strategy games involves the use of Artificial Neural Networks which are based on nerve cells or otherwise known as neurons in a human brain and a signal. In other words, the main purpose of the ANN is to replicate the nervous system's functioning to bring comparable brain functionality to a computer. ANN is, in essence, nature's coping methods (M. Živković 2017). A human brain nerve cell is made up of dendrites-input branched connections, cell body-processing pulses, and axon-output connections that make up a single neuron (nerve cell).

A picture containing text, vector graphics

Description automatically generated

Figure : Neuron

Neurons converse by transmitting electrical signals, that are short-lived impulses or "spikes" in the cell wall or membrane's voltage (Gurney and York). There are different types of ANNs used in various applications and projects other than games which can be challenging to decide which type of ANN(some of the ANN types and the idea of right tool for the right problem) to use for a strategy game such as Deep Learning and the Three-Layer Neural Networks that has an input, hidden and output layers as seen on Figure 9 **Invalid source specified.**. This is because it was stated that the three-layer neural networks can be used to predict moves that are made in chess (Oshri and Khandwala) as well as simplify the Battleship game that involves the following algorithms of Gradient Descent and, Reinforcement Learning **Invalid source specified.**.

Chart, bubble chart

Description automatically generated

Figure :Different types of Neural Networks

The ANN will also include the following algorithms such as Genetic and, Heuristic algorithms especially Gradient Decent and, Reinforcement Learning which are already mentioned before. Another important fact to point is deciding the types of data between either Qualitative or Quantitative to use for this A.I. project as well as for deciding on a design approach of a board layout for this Battleship game. The reason why data is relevant for A.I. in the first place is that AI is based on the information stored available to it, and it uses this data to create a world in which characters can live and execute fundamental behaviors. At each round, all of the necessary data obtained through AI is used to create a virtual gaming environment with scenarios, motives, and behaviors attributed to the gaming characters that are growing increasingly realistic and lifelike (AIT Staff Writer, 2020).

For any A.I. game project, there would be three game modes to implement which are the single-player mode, multiplayer mode, and an A.I. mode which will be discussed further. The single-player mode is where a human player can play against an A.I. that involves different difficulties between easy, medium and, hard to provide an entertaining challenge to human players who wish to play in a match against a computer program (Shummon 2019). A multiplayer mode for this project, however, is just based human player playing against another human player, and no A.I. is involved in this mode. The last mode which is the A.I. mode as the name states is a mode where an A.I. can play against itself in a game similar to the multiplayer mode.

Despite the involvement of the modes it is also worth considering choosing the game rules to establish in this project whether to use a one-shot rule or multi-shot rule and, these rules will involve shot placement to clarify of the shots take place in different positions that are chosen to strike depending on how the ships are placed due to the placement of the ships.

## Artificial Intelligence

Artificial Intelligence (AI), in simple words, is the ability of machines to perform complex tasks that require human intelligence (Russell, 2002). Such tasks necessitate a range of behaviors that are associated with human intelligence which includes learning, planning, decision making, knowledge representation, and reasoning. The field of AI was founded in the 1950s by John McCarthy, an American scientist of Irish origins, who is widely considered the father of AI (Andresen, 2002). Other early contributors include the British mathematician Alan Turing who started his 1950 paper “Computing Machinery and Intelligence” with the words “I propose to consider the question, ‘Can machines think?’” (Turing, 1950). In this paper, he introduced what is known as the “Turin test” in which two players, a computer and a human, try to answer questions, in the form of text, asked by the interrogator. The role of the interrogator is to determine which of the players is the human and which is the computer. The three players sit inside three separate rooms. Despite that 70 years have passed since the publication of this paper, it has provided fundamental and influential concepts to the philosophy of AI.

The Loebner Prize is an annual Turing Test competition that was first launched in 1990. Up to this date, no computer program has passed the competition, but some were very close. According to Brian Christian, one of the human competitors who was named “The Most Human Human”, to pass the test it took him hours of preparation running through every previous test the question “to show that I’m not just a prepared script of things,” Brian said (Christian, 2011). Similar to us humans, any AI model needs a training phase to learn a particular task. This is the most critical phase in the lifetime of any AI model. There are three main types of learning in AI: Supervised learning, unsupervised learning, and reinforcement learning. Each type will be discussed in the following subsections.

## Supervised Learning

The most used type of learning in AI has supervised learning in which the machine is trained using pairs of inputs and outputs. Based on the training examples, the machine learns a mapping between the input examples and their corresponding outputs and then applies such a mapping to new input samples. The applications of supervised learning are endless in the real world. The main applications of supervised learning lie in the area of computer vision which includes image classification and semantic image segmentation (Krizhevsky, 2012; He, Deep residual learning for image recognition, 2016). In addition, some studies applied supervised game development. For instance, a deep learning model was trained using supervised learning to play the 2048 puzzle game (Kondo, 2019).

## Unsupervised Learning

In the case of unsupervised learning, there are no outputs provided with the input examples. Rather, the AI model will work hard to derive the appropriate output depending on the task on hand. For instance, one of the main applications of unsupervised learning is dimensionality reduction in which the AI model derives a reduced representation for the input dataset which would significantly reduce the size of the input samples. The most well-known dimensionality reduction techniques in the AI community are Principal Component Analysis (PCA) and Autoencoders (AE) (Jolliffe, 2005; Hinton, 2006).

## Reinforcement Learning

Reinforcement learning, in simple words, is a type of learning in AI that is very similar to the “carrot and stick” policy for inducing behavior. The only difference is that in reinforcement learning, there is no stick. Rather, the machine depending on its learning progress is rewarding. The better the progress is, the more carrots the machine will be rewarded.

More technically, let us introduce the following terminology: Agent: In reinforcement learning, the machine is referred to as ‘agent’. Environment: The agent lives in an environment and the goal is to learn how to adapt to this surrounding environment. For instance, in case the agent is a pilotless Formula 1 car, the environment would be the racing track. State: The agent state in the environment, e.g. position, speed, etc. Action space: The action space refers to the possible actions that can be taken by the agent. The agent interacts with the environment based on its action space. This enables the agent to transfer from one state to another. Reward: Every action at a particular state has an associated reward that is obtained using the reward function. Reward function: A function that provides an immediate feedback based on the current state of the agent.

In most cases, the action space contains a limited list of actions. At each state, the agent chooses one action from that list. This action will transfer the agent from the current state to the next state. After that, the reward function is used to instantly evaluate the next state and return a reward value. At the training phase, the machine tries to learn a policy that would maximize the average rewards gained. A good example of reinforcement learning is training autonomous racing cars to complete a lap in the shortest time. The image below shows the main diagram for this problem.

Diagram

Description automatically generated

Figure :Elements of Reinforcement Learning in Auto Racing Car.

The main challenge with reinforcement learning is the fact that training the agent to make decisions in real business scenarios may be risky, particularly during the early training stages. In addition, the training phase takes much longer times compared to supervised and unsupervised learning. Reinforcement learning has many applications in real-world problems. For instance, Amazon Web Services have launched AWS Deepracer which is a cloud-based platform for training auto racing cars in a virtual environment using reinforcement learning. The auto racing car will determine the state based on a built-in camera that analyses the scene using ResNet deep learning model (He, 2016).

It is worth mentioning that amongst all types of learning in AI, reinforcement learning has the widest range of applications in game development. This might be because many games naturally follow the reward-based format in their rules. An interesting survey of applications of reinforcement learning in game development is provided in (Zita, 2012). Many recent studies used early Atari games as a basis for evaluating AI models trained using reinforcement learning (Mnih, 2013; Kaiser, 2019).

## Gradient Descent Learning Method

The gradient descent algorithm and its variants are the most commonly used techniques for training AI models. They are simple, easy to understand, and very practical. The gradient descent algorithm was coined by the French mathematician in the paper by Cauchy, (1847). In this section, the introduction of the gradient descent algorithm helps understand the main intuition behind this technique, discusses its main limitations, and lists other variants that are based on this method. Any AI model can be thought of as a black box that has a set of inputs and outputs. Inside that box exists a set of parameters. The outputs of this black box are determined based on the inputs and the values of these parameters. Generally speaking, the main problem in AI learning is to find the optimal values for such parameters that would lead to the best performance. The term ‘best performance’ is defined depending on the problem that is being solved. For example, in the case of regression problems, the best performance is determined by minimizing the errors between the actual outputs from the training samples and the outputs from the AI model.

In the case of classification problems, the best performance is determined by minimizing the number of misclassifications in the training examples. Such objectives require formulating a ‘loss function’, the minimization of which would lead to the desired best performance. The loss function maps the parameters of the AI model to a single output that defines errors in the training samples. In the standard gradient descent algorithm, the loss function is computed based on the entire training dataset. In case the AI model has only two parameters, the loss function will look like mountains and valleys as in the image below.

A picture containing outdoor, sky, mountain, nature

Description automatically generated

Figure :An error surface in gradient descent can be thought of as mountains and valleys as in the image.

In this simple example, the AI model parameters would be the latitude and longitude and the error (loss) is defined by the elevation. The basic idea of gradient descent is very straightforward, at each position, move one step towards the descending direction of the steepest slope. Keep moving until the slope flattens which is the condition that is satisfied by the lowest point. The image below provides a good demonstration of the gradient descent algorithm. The steepest slope corresponds to the derivative of the loss function at the current position.

More technically, let us denote the loss function as, where and are the two parameters of the AI model and the output of the function corresponds to the error of the AI model on the training samples when using such parameters. Our goal is to find the best values for and that will achieve the minimum error value. Assume that the initial parameter values for the AI model were and. To take the next step by computing the derivative of the loss function concerning and. In addition, to move downhill is to multiply both derivatives by -1. The updated values for and become and respectively where is referred to as the step size. One can arrive at the general update rule for the gradient descent.

For the step size, three common strategies can be used to: Use fixed step size. Decrease the step size by dividing the initial step size by the number of steps moved. Use a customized strategy for the step size. Finding the minimal error for a 2D parameter space may seem an easy task, however, in practice, the parameter space is much higher in dimension. Fortunately, this is not a problem for machines as they can deal with high-dimensional data more efficiently.

In practice, an AI model deals with large-scale data for which computing the loss function may be computationally expensive. In this case, the training samples are grouped into some batches. For each batch, the loss function is computed and then the gradient descent update is performed on the parameters based on this loss function. This is for every batch that is needed to be done and then repeats the process until convergence. This is known as the ‘stochastic gradient descent and is a commonly used technique for neural networks, reinforcement learning, and deep learning models. In many cases, the method may suffer from stability problems.

In particular, the learning path of the algorithm may be of a zigzag shape. Many variants of this method were proposed to solve such a problem and converge better to the optimal solution. Most known variants include gradient-descent with momentum, Adam algorithm, and Root Mean Square Propagation (Ruder, 2016).

A group of people skiing down a mountain

Description automatically generated with medium confidence

Figure : Steps of the gradient descent.

## A.I. in games

Games and AI have a long history together. This is also since the construction of agents for playing games, with or without a learning component, is a focus of much AI for games research. This was the first and, for a long time, the only way to use AI in games. Early pioneers of computer science created game-playing systems to explore if computers could tackle problems that appeared to require "intelligence" even before artificial intelligence was acknowledged as a profession. A. S. Douglas worked on a computerized version of the Tic-Tac-Toe game as part of his doctoral dissertation at Cambridge in 1952 and created the first software that could master a game. Arthur Samuel, using a program that learned to play Checkers by playing against itself, was the first to invent the type of machine learning currently known as reinforcement learning a few years later (Yannakakis and Togelius).

A.I. has been worked on by many researchers and, non-researchers who grow more interested in the field. A.I. has been commonly used by numerous applications (between robotics, marketing, hospitals and, etc). A.I. in-game applications became widespread now that developers in various gaming companies are more focused on working on A.I. implementations on video games and adding more features such as adding non-player characters(e.g. boss, civilian or, party member, etc.).

Although, Laura E. Shummon Maass(2019) stated that sometimes some video game developers worry that when implementing advanced A.I. into their games they might lose player experience which sometimes they do not realize that games with A.I. could help raise more player experience by providing players with more entertaining challenges. Herbert A. Simon mention simultaneously, video games have grown in complexity and variety, with some including AI breakthroughs for controlling non-player characters, producing content, and responding to players. AI technologies are rapidly being used by game producers to examine enormous amounts of user data and optimize game designs. A small but increasing community of researchers and designers is experimenting with ways to use artificial intelligence to design and develop whole games, either automatically or in collaboration with humans (Yannakakis and Togelius).

In Strategy games, the player controls many characters or forces to achieve victory in conquest or war. The narrative and graphics usually, but not always, depict a military fight, using units such as knights, tanks, and ships. Most strategy games contain hidden information (example). In strategy games, the cognitive challenge is laying out and executing intricate strategies involving several units because many units must be moved at every round, this problem is substantially more difficult than the planning (complex) difficulty in traditional board games like Chess. The number of units a player commands can soon exceed the limits of short-term memory. Because a single action may provide new information throughout a move, the sequence in which units are moved can have a major impact on the movement's conclusion, prioritization can be a concern. Furthermore, forecasting the maneuvers of one or more rivals, who typically have several units, is a difficult task.

There are additional perceptual and motoric hurdles in real-time strategy (for example) games due to the game's speed (Yannakakis and Togelius). Most search algorithms struggle with the huge search spaces and large branching factors of strategy games, as scanning a single round forward may already be impossible. Decomposing the problem so that each unit acts independently is one solution; this creates one search problem for each unit, with the branching factor equal to the branching factor of the individual unit. This has the benefit of being tractable, but it also has the problem of limiting coordinated action across units. Nonetheless, many strategy games' built-in AI employs heuristic methodologies in which units are addressed independently. Not surprisingly, many strategy games' built-in AI is largely viewed as inferior (Yannakakis and Togelius).

Some answers to this have been found in studies on playing strategy games, and they include cleverly subsampling the space of turns such that standard search algorithms may be employed. (trees and branching need explanation before talking about them) A Monte Carlo Tree Search (MCTS) variant based on decomposition through Naive Sampling is an example of such a method because It does not thoroughly scan all branches of the search tree; instead, it focuses on the most promising branches. In such scenarios, MCTS not only overcomes Minimax's tree size barrier algorithm but also approximates the game's Minimax tree given enough computation. (Yannakakis and Togelius).

## Neural Networks

Artificial Neural Networks (ANNs) are a family of AI models that mimic the human neural system to solve complex tasks (Yegnanarayana, 2009). Many state-of-the-art AI models belong to this family. The main advantage of such models is the fact that they can be adapted to solve any problem. They can work on both supervised and unsupervised tasks. In addition, ANN is a very general model that has many different types. Now let us have a look at the main structure of an ANN.

Diagram

Description automatically generated

Figure : example of neural network.

The figure shows that ANN consists of multiple layers: A single input layer, a single output layer, and multiple hidden layers. The input layer feeds the input features to the model and the output units produce the output of the network. Each hidden layer contains several units which are referred to as perceptrons. Each perceptron mimics a biological neuron. Considering the biological neural system, each neuron carries a signal coming from other neurons using synapses and passes it to latter neurons until the signal reaches its destination. In ANN, each perceptron is a function whose inputs are the outputs of the units in the previous layer. The perceptron assigns a different weight to each input unit, multiplies the input value by the assigned weight, and then sums the multiplication results of the other units.

After that, the activation function is applied to the result to generate the output of the perceptron. The most commonly used activation functions in neural networks are the sigmoid function and the rectified linear unit. Hence, the output of the network depends on the values of weights of each perceptron and the activation function. Note that ANN requires a large number of weights that correspond to the model parameters. The network in the previous example has an input layer of 4 units, one hidden layer with 5 perceptrons, one hidden layer with 7 perceptrons, and an output layer with 3 output units. Hence, the total number of weights (parameters) in the model is 4\*5+5\*7+7\*3=76. This is a very large number of parameters considering that the model has only 4 input features. Hence gradient descent (or its variants) is a typical option for optimizing the network weights. The below image shows the detailed structure of ANN.

Diagram

Description automatically generated

Figure : Detailed structure of ANN

One needs also to consider determining the network structure before training the model. Hence, there are some hyperparameters to consider before training the network including, the gradient descent step size, the number of layers, the number of perceptron’s for each layer, and the type of the activation function (sigmoid, rectified linear unit, etc.).

## Deep Learning

Deep learning models are a special type of Neural network. One common feature of such models is the fact that the number of hidden layers is very large compared to conventional neural networks (Simonyan, 2014). By increasing the number of hidden layers in the network, the model will be able to perceive more complex patterns and solve more challenging tasks. However, increasing the depth of conventional neural networks has some undesirable side effects. Firstly, since each perceptron in the network is connected to all units in the previous layer, this will significantly increase the number of weights in the network which in turn affects the training time. Secondly, the dense structure in the convolutional neural network makes the mode more prone to overfitting problems.

Deep learning models solve such a problem using a special type of neural network, known as convolutional neural networks (CNNs). Unlike conventional neural networks, where each unit in a layer depends on the entire units in the previous layer, CNN regularizes spatial patterns using a moving window type of processing. In this case, each unit in a layer depends on units in the previous layer within a specific patch. The moving window consists of multiple filters (also referred to as kernels or convolutions) where each filter contains weights for the corresponding patch. One can note that CNN's work only for spatially structured samples such as images, audio, etc. Consider the input samples are of image format. The window moves through the entire image with a specific stride.

After computing the convolutions, the activation function is performed followed or preceded by a normalization process. The activation function is usually a rectified linear unit (ReLU). Pooling layers may also be used in CNNs. These layers operate a moving window in which a single value is chosen per patch per kernel. Two types of pooling layers are commonly used, max pooling and average pooling. The output of each convolutional layer is referred to as a feature map. The final layers usually follow the structure of a conventional neural network. The image below shows a simple convolutional neural network consisting of one convolutional layer, one pooling layer, and fully-connected hidden layers.

Diagram

Description automatically generated

Figure : A basic convolutional neural network.

Like conventional neural networks, CNN is trained using the gradient descent learning algorithm to optimize the filter's weights.

Diagram

Description automatically generated with low confidence

Figure : The artificial neural network is mapped to the environment cells in this diagram.

## Genetic algorithms

Genetic algorithms are optimization methods in the sense that they are used to identify the best solution(s) to a given computing problem that maximizes or minimizes a specific function. In that they emulate the biological processes of reproduction and natural selection to search for the 'fittest' answers, genetic algorithms are one part of the subject of study known as evolutionary computation.

Many of the processes in a genetic algorithm are random, just like in evolution, but this optimization technique allows you to manage the level of randomness and control. These algorithms are significantly more powerful and efficient than random and exhaustive search methods, and they don't require any additional data about the problem. They can use this property to solve issues that conventional optimization methods can't solve because of a lack of continuity, derivatives, linearity, or other properties.. (Carr, 2014)

There were some cases where Genetic Algorithms can be used as useful implementations in games such as optimizing city development for a turn-based strategy game similar to popular titles namely, Civilizations and Age of Empires. This necessitates a great bit of micromanagement, particularly when dealing with the different factors that influence city development. This procedure can be arduous, especially when dealing with huge civilizations with multiple cities, and novice players who are unfamiliar with the game's different intricacies typically encounter a steep learning curve. The goal is to create a genetic algorithm that runs in the background and adjusts city growth factors to optimize city development while leaving the user free to focus on other aspects of the game. Due to the dynamic nature of the gaming world, this optimization process will have to be continual, adjusting to environmental changes as they occur (Azhar, 2008).

A genetic algorithm in a strategy game such as a battleship is composed of five steps: initialization, selection, reproduction, termination, and conclusion. The initialization step is the creation of the starter population. In this algorithm, the population contained 100 members. When playing with a new opponent, eighty percent of the initial population is randomly generated, while the other twenty members are pre-generated at intervals spanning the problem space. the population is then carried over from the previous game or loaded from a saved gene pool (Jeremy G. Bridon, 2009). The selection step determines which members of the population will reproduce and pass their traits on to the next generation. A fitness score determines how successful or poor a gene whether it should be allowed to reproduce by calculating the discrete cross-correlation between the genetic wave and the target distribution for each gene. Genes with the highest fitness are chosen to continue through to next-generation**Invalid source specified.**. The Reproduction step occurs in two genes which are selected in a set of genes which then produces 45 new genes but is applied twice: with or without mutation. The selected group of ten genes is then carried to the next generation unchanged, it gives a total of one hundred members for the next generation.

A crossover occurs when there is a reproduction between two members. The method of a gene the method of a crossover must be dependent upon the entire wave formed by the harmonics of the gene instead of a simple interchange of harmonics of its harmonics. However, the real problem occurs in a crossover between aggregate waves that involves mathematically complex analysis. To begin the analysis by first computing the average of two waves, that is a perfect child, then it composes ten sine waves.

To reduce the count of the component to five, Fourier analysis is utilized for sorting isolate the five harmonics with the highest amplitudes, which are kept as the components of the child gene **Invalid source specified.**. With this, it permits the best approximation to the best child curve that is composed of five harmonics only. To create the mutated population, a copy of each crossover gene is mutated, which involves the introduction of random coefficients to either some or all of the harmonics. Each coefficient has at least a 25% chance to be mutated.

The Fourier analysis can be performed during crossover by using a Fast Fourier Transform (FFT) algorithm, which has a complexity of O(n log); this reduces analysis to a reasonable run-time **Invalid source specified.**. Another problem is when each of the twenty coefficients of the gene is parameterized over a large range, the space becomes extremely large. Despite the issue at hand, however, the limitation for this matter is unimportant since a crossover can be shown to converge on the solution in a reasonable number of generations as stated by Q.C. Meng (1999).

To ensure that the genetic algorithm for this implementation does not converge on local maximum, random coefficient mutations are introduced during the creating of each new generation **Invalid source specified.**. The genetic algorithm runs a large number of generations at the beginning of each game after the first and, in each game after the initial game; one or more generations can run per turn.

The member with the highest fitness in the most recent generation is used when the sinking logic (explained in the following section) calls for a random location. The shooting algorithm also combines a filter that will never permit the algorithm to return a square that will not and cannot contain a ship. This concludes that a checkerboard pattern is applied for targeting, which will adapt depending on the size of the largest enemy ship that is remaining. **Invalid source specified.**.

## Game Development

To develop any strategy game such as chess or battleship there are several steps to be followed. Chess would be an ideal example for a strategy game to discuss the development aspects. There are four aspects to this game: rules, computation, strategy, and playing. In the rules aspect of the game, chess is played by two opponents, one color for each, white and black pieces. The following figure shows the pieces that are used in the game.

Text

Description automatically generated with low confidence

Figure : Chess pieces.

The initial position is always the same, and it is rigorously established at the start of the game. A row of pawns stands in front of the other pieces, followed by the royal couple in the center, bishops, knights, and rooks in a symmetrical order. The board would be a squared grid of 8x8 cases (a two-dimensional array) with dark and bright backgrounds that alternate. In visual display Paquier, (2021) discovered that each symbol is part of Unicode, therefore drawing a piece only requires a basic text label which was then possible to print the board in the console logs for visualization.

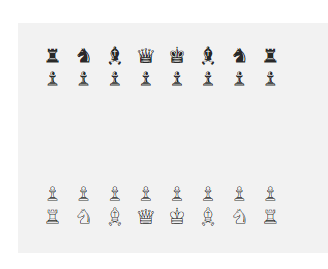


Figure : Chess pieces on visual display.

In chess, the board grid has a coordinate system made of letters (horizontal axis, or files) and digits (vertical axis, ranks) which is used for game notation, i.e. to script the moves. So, it was possible to generate a 10 x 10 grid layout chessboard (Similarly to what will be used for Battleship) with the involvement of MVC architecture and GUI.

A picture containing chart

Description automatically generated

Figure : GUI of a chess game.

There are four fundamental sorts of moves that can be mixed and matched depending on the piece: Straight (used for rooks, queen, and king); Diagonal (bishops, queen, king); «L» (knights); and Pawns moves. A piece, except for knights, cannot jump over another piece, and when a piece approaches an enemy piece, it may capture it and take its place on the board. Since it is now possible to move all the pieces in the game, the following game rules are implemented: turns of the players, Identifying whether a king is in check and, if there is no way out, concluding that the king is checkmate (game lost) and, both kings must always be a square apart; Considering the game's rules: there are things that can be done as well as those that must be done. So, how does the game end now? There are two possible outcomes: either a player wins and the other loses, or if there is a draw on both sides.

When the chess game along with its is complete and then made possible to play for multiplayer mode, the next task would be programming an A.I. based on the level of intelligence needed for a game. Paquier(2021) implemented an A.I. with the lowest that can at least compute all possible moves and randomly select one of them. To evaluate the situation based on the move made A function is required to assess the present position, try a move, and then reevaluate it. When a move leads to checkmating, according to the game rules, then the best move has now been found by a player. A penalty score, on the other hand, is used to see if the move results in a draw, intending to make the A.I. as aggressive as possible. To avoid naïve evaluation such as losing a queer by the opponent player that is the most important piece, A recursive move selection function is required such that after attempting a move, the function is executed again, but this time for the opponent: the goal is to guess what the answer should be. And then assess the situation that has resulted. Predicting one, two, or three moves allows the AI to do a variety of tasks, including:

* Make a game-winning move that will result in checkmate right away (depth of 1)
* Right after, make a move to avoid a loss (depth of 2)

This can be used for testing purposes when creating a game that can lead to checkmate in one or two moves and allow it to identify the optimal move to win or escape losing. (Paquier, 2021) For strategy, Controlling the middle of the board is one of the keys to victory. Placing your pieces in the center, or at least covering the center, aids in the development of the army, both offensively and defensively.

Using the concept of the heat map shows the following implementation: The four squares closest to the center are worth 2 points apiece, the 12 squares surrounding them are for 1 point and, the remaining squares are worth 0 points. Then one must calculate each piece's movement and add all of the points above to gain a final score based on the locations they can reach.

Table

Description automatically generated with medium confidence

Figure : Heat map to control the board center.

Another strategy involves the use of Heuristic algorithms to draw closer to checkmate when surrounding and, capturing the king piece. The following implementation is as follows: 3 points when hitting the king, since having the king in check is very important, 2 points around the king for preventing the king to escape, 1 point around the king region, because it gradually leads to restricting his moves, and finally, 0 points for the remaining squares.

Table

Description automatically generated

Figure : Heat map focusing on the king, with the king on square g1.

It is also worth noting that Many games include a theoretical component that should not be overlooked which is the openings. Numerous libraries provide numerous openings. To make an A.I. comparable to any human player it would require fifteen major openings and for each opening would be two or three moves.

Diagram

Description automatically generated

Figure : tree containing openings.

The following tree structure in Figure 10 describes the classification of the openings: A leaf is the last known move for a particular opening, while a move leads to a node. When the AI must choose a path from a node, it does so at random. It provided alternatives to each game to avoid dull, predictable courses. When the AI doesn't recognize the move that was just made, it reverts to its default calculation(Paquier, 2021).

Now the game is ready to be tested and played using the following A.I. concepts:

* A futile A.I. that plays at random.
* A.I. that computes in depths of 1, 2, and 3.

Another A.I. computing moves with a depth of three, but also has some opening experience. When playing against an A.I. there are some occasions where a human player would win against an A.I. However, it is also possible that an A.I. can win a match against a human player due to the lack of concentration or underestimation (Paquier, 2021).

It is also worth noting that in game development various tools are useful to develop any game for example Pygame as well as there are other methods besides Heuristic that also be used for a strategy game development such as minimax algorithm. Pygame is a set of cross-platform Python tools for creating video games. A Chess game created in Pygame will mostly handle the game's user interface. The user interface will provide the current state of the board, the moves made by the Chess Artificial Intelligence, and move suggestions for the user (Cai et al, 2021).

## Minimax algorithm

The minimax algorithm is a backtracking method used in decision making, game theory, and artificial intelligence (AI). It is used to discover a player's best move, given that the opponent is likewise playing well. This algorithm is used in popular two-player computer or internet games such as Chess, Tic-Tac-Toe, Checkers, Go, and others. A backtracking algorithm is used to find a solution to computing issues by slowly building a candidate towards a solution, one step at a time. And any candidate who does not finish a solution is promptly dropped (Vadapalli 2021).

There are two actors in the AI minimax algorithm: Maximiser and Minimiser. Both of these players compete in the game, with one attempting to achieve the highest score or maximum benefit and the other attempting to achieve the lowest score or minimum benefit. Because each game board includes an assessment score, the Maximiser will choose the highest value, while the Minimiser will choose the lowest value with counter-movements. When the Maximiser has the upper hand, the board score will be positive, but when the Minimiser has the upper hand, the board score will be negative. This is based on the concept of a zero-sum game, in which the total utility score is distributed equally between the two players. As a result, a rise in one player's score causes a decrease in the score of the opponent, resulting in a total score of zero. As a result, for one player to win, the other must lose (Vadapalli 2021).

### Heuristic algorithm

A heuristic algorithm sacrifices optimality, accuracy, precision, or completeness for speed to solve a problem faster and more efficiently than standard approaches. NP-complete problems, a type of decision problem, are frequently solved using heuristic algorithms. Although solutions can be verified when given, there is no known efficient way to identify a solution fast and accurately in these instances. Heuristics can be used to generate a solution on their own, or they can be combined with optimization techniques to offer a reasonable baseline. When approximate solutions are sufficient but exact solutions are computationally expensive, heuristic techniques are frequently used (Kenny 2014).

## Optimization task of Battleship Game

From the standpoint of a single-player solving a single instance of a "shooting" position decision issue at a time, it is referred to as the Battleship game. In this scenario, our optimization goal is to reduce the total number of hit attempts while completely revealing all of the opponent's ships. Before each strike attempt, the player whose mission is to reveal ships on the enemy battlefield chooses a decision. His job is to select a spot on the enemy battlefield to be shot at next. Because the player's decision is independent of the outcome of the preceding iteration's decision, this decision-making is officially described as a Markov decision process (MDP). Because the problem space is only partially visible, this method is referred to as a partially observable Markov decision process (POMDP). In the current condition of the environment, all available information is saved. As a result, no prior decision-making information is required. Apart from the current state of the environment, the order in which past hit attempts were made is meaningless. The player is formally modeled as a representative. The enemy battlefield (including ship dispositions) is considered a problem case. A current state of the environment is defined as an agent's current view (whose view is incomplete knowledge represented by a partially revealed problem instance). A pattern in the problem instance is represented by the ship. Because of inadequate information about the problem instance, represented by the current state of the environment, ship placement may not be known. In a problem instance, ship placement in an environment is formally referred to as pattern placement (rotation and position). It is critical to provide a simplified Battleship game for reasons of description and simulation (shown in figure 14). The change is based on the size of the surroundings, the size of the patterns, and the number of the patterns (Clementis, 2014). The modification can be summarized as follows:

• Only a single player's perspective on optimization.

• Four-cell "L" patterns are organized in 23 different shapes (with all 8 rotation permutations allowed).

• There are two patterns in the environment.

• environment size of n×n = 7×7 = 49 "cells" (MAX. hit

attempts is 49).

All other trivial Battleship game rules (including deployment rules) remain unchanged:

• No pattern overlap (even if it's only partial) is allowed.

• There's a chance that no two patterns will have the same edge.

• While completing a single problem instance, the pattern placement configuration remains consistent (hitting the same position twice is futile).

• An environment's reaction to a hit attempt is always genuine and unmistakable (as if it were an oracle that always answers "YES" or "NO" truthfully), etc.

A picture containing text, crossword puzzle

Description automatically generated

Figure : Pattern placements conforming to the Battleship game rules (pattern deployment rules) are shown as an example of a problem instance. This example corresponds to our modification (environment of size nxn = 7x7 and two "L"-shaped patterns of size 2x3).

## Markov Decision Process

The Markov decision process provides a broad framework for expressing the problem of sequential decision making, but it does not specify the agent's decision-making approach. A policy, which is a function that specifies an action (or a probability distribution over actions) for each conceivable state, determines the approach. The goal of much modern MDP research is to determine the best policy, one that maximizes the (potentially time-discounted) reward. The most basic maintenance policies do not rely on the present memory state. Rather, in every state, they create the same behavior. These maintenance policies are referred to be unconditional in game theory, where a player can adopt a strategy that is independent of the opponent's behavior (e.g., a player in the Prisoner's Dilemma who always defects)(Suchow and Griffiths).

# The Battleship 2D Game

In this chapter, we will discuss the game design and implementation. It is worth mentioning that the game is inspired by the well-known battleship board game. However, I decided to design a more general and challenging 2D game rather than the conventional board game. The main reason for my decision is because the original board game is rather very stochastic in its nature. This will significantly limit the role of the AI agent. The new base game is fully implemented in python.

## Game Design and Rules

Game design is probably the most important phase in developing the game environment in which the agent will live and behave. As mentioned earlier, we will develop a 2D single player game that is inspired by the battleship board game but of much less stochastic nature so that a player will need to use much more intelligent strategies in order to win.

### Game rules and setup

The game involves a battleship facing a set of ten enemy ships launching missiles towards the battleship. The battleship goal is to destroy all enemy ships in order to win the game. The game episode will terminate if one of the following events happens: the battleship destroyed all enemy ships, the battleship is hit by a missile, the battleship collides with an enemy ship or the battleship goes beyond the game margins. At each game iteration the battleship applies an action to the game environment after that, the game applies the action and returns, the resulting observation (or state), and a reward value. The battleship action space is either to rotate left or right by 6 degrees, accelerate by increasing the velocity by a factor of 1, decelerate, or finally launch a missile. Such actions are represented as a discrete number from 0 to 4.

The agent ship will be rewarded 30 points for destroying an enemy ship and 60 points if it destroys all the fleet. In addition, destroying an enemy missile will give a reward of 2. If the enemy is hit, 100 points will be deducted if hit by a missile and 100 points if collided with enemy ship. If the ship exceeds the game boundary, 100 points will be deducted. Such rewards and penalties will significantly help the AI agent to find the optimal war strategy (policy) at the training phase.

A picture containing graphical user interface

Description automatically generated

Figure : A screenshot of our 2D battleship game

### Class Design

In terms of the class design, we will have AgentShip, EnemyShip, AgentMissile and EnemyMissile classes. These classes define the main elements of our game. In addition, these are also subclasses of the ShapedObject class. This class stores the polygon points defining the margins of each navy object. This will make collision detection much easier in the implementation of the game environment. The game environment is represented in the BattleShipGame class which renders the element on the game canvas, applies actions taken by the agent into the environment and evaluate the action by retuning a reward value and new observation. The UML diagram below shows the class design of our game.

BattleShipGame

ShapedObject

Figure : UML class design.

elements:ShapedObject

canvas:ndarray

\_obse:int

Structure:Polygon

Angle:float

x,y:int

render()

reset()

step()

has\_collided()

Set\_shape()

Move()

Rotate()

AgentMissile

EnemyShip

EnemyMissile

AgentShip

Icon:image

vel:float

Icon:image

vel:float

Icon:image

vel:float

Icon:image

Vel:float

Speedup()

slowdown()

launch\_missile()

Speedup()

Slowdown()

launch\_missile()

## Python Libraries

We have used four main python libraries to setup the game environment. The libraries are disused in the following subsections.

### OpenCV

OppenCV is a computer vision and image processing library that is widely used in practical applications. We used OpenCV to properly render the objects on the game canvas.

### OpenAI Gym

Open AI gym is a very well-established RL library created by Elon Musk. Open AI gym provides a plethora of RL environments to train and test state-of-the-art RL AI agents. In addition, it allows for creating customized environments that would be fully compatible with deep learning python libraries such as TensorFlow.

### Shapley

Ahepley is a python library that is used for programming geometrical objects. This library has helped me very much for defining the margins of each navy object and easy detect if a collision between two objects has occurred.

## Game Implementation

In this section, I will walk through the detailed implementation of our game. As mentioned earlier, the core battleship game consists of 6 main classes. All game objects (ships and missiles) are subclasses of the ShapedObject class to better define geometrical boundaries of each navy object and detect collisions. All enemy ships have fixed positions at the right side of the game canvas. At random, these enemy ships fire missiles towards the agent side and the agent need to manipulate and destroy the enemy ships.

### ShapedObject Class

The shape object class mainly stores the polygon points that defines the corresponding object border. These points are represented by the structure attribute. In this way we can easily make primitive geometrical checks using the shapely library. For instance, to check if two ShapedObject instances has collided we can use the shapely function **intersects** which returns true if the two polygons intersects or false otherwise. We will see this more clearly in the BattleShipGame class. The ShapedObject class also stores the direction angle of the object in order to rotate and move the polygon points accordingly. This can be clearly seen in the move function where we used affinity.trnslate shpely function to translate the points based on the input velocity and angle direction. When the angle is 0, the object is directed to the east. The x and y attributes corresponds to the center of the object points.

Text

Description automatically generated

### AgentShip

The AgentShip class defines the player. This class has the velocity and icon attributes corresponding to the velocity of the ship and the image pixel representation. We can see at the clearly init function how we define the points of the object and pass it to the ShapedObject super class using the set\_shape function. Also the velocity init method takes as input the name of the object defines as string, the x and y corresponding to the central position, the direction angle and the velocity. This class has five methods listed below:

* speedup: Increases the velocity attribute by a factor of one. The velocity shall not exceeds 10.
* slowdown: Decelerates the ship by decreasing the velocity by one.
* move: Move the object by calling the move method in the supe class and passing the velocity attribute.
* rotate: Apply rotation to the object polygon points and the pixels in the icon attribute.
* launch\_missile: fire a missile of velocity 30 and the same direction angle as the ship.

Text

Description automatically generated

### The Enemy Ship Class:

This is very similar to the AgenShip class but has different image representing the object pixels and different polygon points. This function has the following methods:

* speedup: Increases the velocity attribute by a factor of one. The velocity shall not exceeds 10.
* slowdown: Decelerates the ship by decreasing the velocity by one.
* move: Move the object by calling the move method in the supe class and passing the velocity attribute.
* rotate: Apply rotation to the object polygon points and the pixels in the icon attribute.
* launch\_missile: fire a missile of velocity 30 and the same direction angle as the ship.

Text

Description automatically generated

### The Agent and Enemy Missile Classes

The agent and enemy missile classes are very similar to the agent and enemy ship classes. In the game environment, the missile always move at a fixed direction. This direction is based on the direction of the launching object. Hence, the rotate method is important to rotate the icon pixels and polygon points from angle zero to the corresponding launching object angle. This can be notes at the init method where in the last line we invoke the rotate method.

Text

Description automatically generated



Text

Description automatically generated

### The BattleShipGame class

Now to the important part, the class that defines our game environment. The BattleShipGame class initializes the game, takes a suitable action from the player (eithe AI player or human player), apply to environment, evaluate the reward that the player deserves and render the resulting observation (state). It is also worth mentioning that this class is a subclass of the gym.Env class. This super class defines the basic template for any OpenAI gym environment so that implementing and developing AI agents on top of this environment would be straightforward. The main methods of this class are listed below:

* reset: Reset the game to its initial state and initializes the main attribute variable such as the episodic return (accumulated rewards along the episode).
* step: This function takes as input an action (that is compatible with the environment defined action space) from the user or the AI and returns the new observation represented as of the game canvas, the reward that the user had after applying the action to the environment and a flag variable indicating whether the episode has terminated due to either the user has won (destroy all the enemy ships) or lost (hit by a missile, ship or exceeds the game boundary).
* render: Render the current observation of the game as an image.
* has\_collided: Used in the step function to check whether two elements have collided using the shapely library.

## Running The Game Using Random Strategy Agent

Before finishing this chapter and moving forward to the AI implementation, let’s implement the game when using a random agent that at each step takes an action at random. Below we show the game implementation with random agent.

Text

Description automatically generated

Here, we run ten episodes using the random agent. We pick a random action using the sample function and after that we apply this action to our game. The game environment will return the new observation, a reward and the done flag to indicate whether the episode has ended. This give very good idea on how interact with our game. We can easily develop further by reading the actions from the keyboard and allowing a human player to play the game. In the next chapter, we will take our game one step further by developing the AI agent which will be able to learn the game after experiencing a huge number of epeisodes.

# Adding AI Agent Using Deep Learning

In this chapter, I will develop the AI agent and train it to play the battleship game. Firstly, the AI agent will be trained on the game by exploring the game when using a random policy and evaluating the resulting rewards. The gradient descent will be applied to enhance the agent experiences from the random policy agent and giving it a chance to play and apply its experience with the random agent using epsilon-greedy policy. After thousands of training epoch, my AI agent will become much more experienced in making decisions.

## Developing AI Agents with OpenAI Gym Environments

OepnAI gym is an RL framework that offers a wide range of ready-to-use environments that has been widely used in the RL research community. This also includes 3D games and physical simulations. All we need is to pick up an environment and then develop and compare AI agents behaviour. Below are some examples of the environments offered by OpenAI gym.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : Atari game env.

A person running on a track

Description automatically generated with low confidence

Figure : Roboschool running env.

### Interacting with The Gym Environment

Any gym environment must have implementation to the following two methods in order to interact with the agent:

* reset:

As the name indicates, the reset method initializes the environment attributes and returns the initial state of the environment. I will use the terms state and observation interchangeably throughout this chapter.

* step:

The step function takes as input an action from the player, applies it to the environment current state, assign a reward based on the new observation and then returns the following: the new observation (state), the reward, a flag indicating whether the current game episode is terminated (win or game over state), and info flag to help in logging and debugging.

### Creating Custom OpenAI Environment

One of the challenges that I faced is that there is no built-in environment for the Battleship game. Hence, I designed my BattleShip game class so that it can easily implement the gym.Env class by implementing the step and reset functions accordingly.

### Defining the Observation Space

One of the most important things in developing environment is to define the observation space. This will play a critical role in efficiently training the AI agent with minimal data consumption. A very good example is the gym environment cart pole in which the AI agent tries to balance the pole. This is a 2D environment as shown in the figure below. We can take the full image as a representation of the current state. However, it would make more sense to use the main physical attributes that corresponds to the pole state (angle, angular velocity, velocity and 1D position).

Diagram

Description automatically generated

Figure : Cart pole env observations.

On terms of the battleship game, the state space is much more complicated since there are a number of missiles the positions of which must be taken into consideration. Hence, it would be more convenient to take the current canvas of the game as a representation of the observations. As discussed in the previous chapter, the original game canvas is of size 600x800x3 pixels which is very excess for the training the AI and storing experiences in the replay memory buffer. For that reason, we resized the resulting canvas to 120x160x3 pixels. This will make the training phase much faster as will be seen later in this chapter.

### Defining the Action Space

Similar to the observation space, the action space also depends on the environment. In our battleship game, the action space will be a discrete number from 0 to 4: 0 to rotate left 6 degrees, 1 to rotate right 6 degrees, 1 to fire missile, 3 to accelerate and 4 to decelerate.

## Deep Q-Learning

In order to train our agent, I will use the deep q-learning technique. Deep Q-learning is a widely used RL training method that involves running a random agent at the early stages of the training phase. The actions, rewards, and observations of the random agent are stored in a replay memory buffer. After that, the AI agent takes a random batch of these experiences to learn the optimal strategy to win the game. This involves predicting the expected accumulated rewards that agent would have if it applied an action at a give state (observation). These accumulated rewards are referred to as quality values (Q-values). The main philosophy of Q-learning is that if the agent is able to predict the accumulated reward for each action at a given state, it can easily win the game.

### Markov Decision Process Revisited

We can think of our game as a Markov Decision Process. At each state we have a number of actions that can be taken. In addition, for each action, there is accumulated reward values (Q-values) that can be gained based on the new state assuming that the agent played optimally for now on. Clearly, if we have all these information, then we can optimally play the game. However, the problem that we have is that there are too many states in our environment and it is impossible to visit each state and know the optimal behaviour and the accumulated rewards. Therefore, we need a more practical way gather such information. Here where Q-learning comes to action.

Diagram

Description automatically generated

Figure : Battle Ship Game as a random desission process.

### Q-Learning

As mentioned earlier, the main problem with RDP is that for many environments it is impossible to know the accumulated reward for an action at a given state. The solution to this problem is to apply q-learning. In this case, we explore the environment using a random policy agent and keep track of the rewards. Based on exploration of sufficient observations, we can then estimate the Q-values for any given state-action pair. The main problem with this approach though is that we still need to explore a very huge number of observations from the environment in order to be able to make a good estimation.

### Deep Q-Learning

Deep Q-learning is a new variant of Q-learning where we use a deep learning model to perform the estimation of the Q-value. It has been widely used in the RL community and has the advantage of being scalable when considering image observation space (as the case in Battle Ship game). In this case, we train the deep learning model to accurately predict the Q-value based on our random exploration to the environment. After training the model, we can then input an observation (state) to the deep learning model and it will give us estimations of the Q-values that can be gained of each possible action. Of course, the optimal policy would be in this case is to choose the action that has the highest q-value.

### The discount factor in Q-learning

One thing that we need to take into consideration is that the outcome of each action in the Battle Ship game is not immediate. For instance, if the agent ship fires a missile, it will take several states to reach the enemy ship or intervene an enemy missile. Therefore, in Q-learning we use what is called the discount factor . This factor takes a value between 0 and 1. This value tells us how immediate the outcome of an action would be. For instance, if the discount factor then we would expect the outcome to be immediate, on the other hand if the value is one, then the outcome would happen any time in the future (long term). In practice, the discount factor takes a value of 0.95.

## Implementing the AI Agent Class

In this section, I am going to walk throughout the AI agent class. This agent will use deep learning to predict the Q-values for the actions of a given state. For my AI agent development, I found the (Géron, 2019) book an excellent source for guiding me on understanding the different AI concepts.

### Adding Memory Buffer

As mentioned in the previous section, in Q-learning we need to randomly explore our environment in order to estimate the Q-values of a given state. Therefore, we need a queue data structure to store the explored states. This can be achieved using the collections.deque class. Due to limited RAM size, the deque will have a limited buffer size of 4000 observations. Hence, it will store the 4,000 most recent observations.



### Adding Deep Convolutional Networks to Agent

The most important part of the agent is the deep learning model. The deep learning model in my implementation consists of three convolutional layers and two dense layers. The deep learning models takes as input an image of size 16x120x3 which is the size of our environment observation space. The structure of the model is given below.

Text

Description automatically generated

The self.n\_outputs corresponds to the number of actions in our action space. There are 5 outputs to our deep learning model. Each output gives a prediction to the Q-value of the corresponding action for the input observation.

### Applying Q-Learning Training Step

This is the most complicated part in our implementation. As mentioned earlier in this chapter, in the training phase the model main goal is to learn how to accurately predict the q-values of each action from a given state. By being able to accurately predict the Q-values, the optimal playing policy will become very straightforward, choose the actions that gives the highest q-value. In the training step of our agent, the agent takes random samples from the replay buffer based on the batch size. Each sample in the replay buffer contains the observation, the action, the reward, the next observation, and the done flag. After that, the agent will predict the Q-values of the next observations multiply by the discount factor and add it to rewards provided by the buffer. This technique is based on the (Géron, 2019) book implementation. Then we apply the mean-squared-error to measure how accurate the model predictions and then apply a gradient descent step to enhance the model prediction experience.

Text, letter

Description automatically generated

Text

Description automatically generated

### The Epsilon-Greedy Policy Method

At the start of the training phase, the AI model has no idea about the environment and will certainly make wrong predictions if we give it the chance to play. Therefore, we use a completely random agent at the beginning of the training phase. However, the AI agent will take turns and involve more in making actions as it gets more experienced to accurately predict. In order to achieve this, we use the epsilon-greedy policy function in which, we pass the observation and a probability (epsilon). Based on this probability, the function will decide to whom it will give the turn, the random agent or the AI agent.

Text

Description automatically generated with medium confidence

At the beginning of the training phase we assign a value of 1 to epsilon and slowly decrease it along the training phase as will be seen in the next section.

### The Full Picture

Now to the train method. The train method takes as input the number of episodes in the training phase. For each episode we play one step using the epsilon greedy function and store the results in the replay buffer. At each fourth step, we apply a training step at which the AI agent takes a batch of 128 random samples from the buffer and train to predict the q-values on these samples using the gradient descent RMSport variant.

Text

Description automatically generated

Text

Description automatically generated with medium confidence

### Training Settings

I personally found that finding the best training setting for the AI agent is the most difficult part. For my agent, I used RMSprop variant of the gradient descent with a learning rate of 2.5. For the loss function we used the mean-squared-error between the estimated Q-values and the predicted Q-values after adding the actual reward, and we trained the model with 12,000 episodes.

## Testing and Analyzing the Agent

As I mentioned earlier, in the training phase, the random agent will be mainly involved in playing turns and exploring the observation space. As the training progresses, the AI model will be involved more and more using the epsilon greedy policy. The graph below shows the accumulated rewards for each episode. Early episodes are played mostly by random agent and later ones are mostly played by the AI agent. We can see at a glance that accumulated rewards are significantly enhanced at the end of the training stage which means the AI model is learning to develop successful strategies to defeat the enemy fleet.

Chart, line chart

Description automatically generated

Figure : AI agent gives much better performance compared to the random policy (at the beginning of the training phase).

# Conclusion and Future Work

## Conclusions

In this document, I developed a 2D battleship game where the player ship tries to manipulate and destroys enemy ships. This can be achieved using deep reinforcement learning. The area of deep reinforcement learning is gaining more attention nowadays due to its practicality for solving real-world problems. There is no better example than the recent advancements in self driving cars as the case with AWS deep racer and Tesla cars. I further developed an AI agent that was able to learn how to make its own strategy and win the game by watching and experiencing the game through a number of training episodes. My model showed significant improvement in rewards gained when compared to the baseline random agent. This was a challenging topic for me as I had to develop the battleship game from scratch and develop the AI model and equip it with a deep convolutional network.

## Future work

In terms of future work, it would be interesting to try to train the Agent using other RL algorithms such as the Proximal Policy Optimization (PPO). In addition, I am also interested in investigating the agent behaviour when using deep convolutional networks.

# References

Gurney, Kevin, and New York. An Introduction to Neural Networks an Introduction to Neural Networks. 1997.

Oshri, Barak, and Nishith Khandwala. Predicting Moves in Chess Using Convolutional Neural Networks.

Summon, Laura E. “Artificial Intelligence in Video Games.” Medium, Towards Data Science, July 2019, towardsdatascience.com/artificial-intelligence-in-video-games-3e2566d59c22.

M. Živković, Nikola. “Introduction to Artificial Neural Networks.” CodeProject, 13 Nov. 2017, [www.codeproject.com/Articles/1215045/Introduction-to-Artificial-Neural-Networks?display=Print](http://www.codeproject.com/Articles/1215045/Introduction-to-Artificial-Neural-Networks?display=Print).

Yannakakis, Georgios, and Julian Togelius. Artificial Intelligence and Games. 2018.

Donges, Niklas. “Gradient Descent: An Introduction to One of Machine Learning’s Most Popular Algorithms.” Built-In, 23 July 2021, builtin.com/data-science/gradient-descent.

Paquier, Micaël. “Implementing a Chess Engine from Scratch.” Medium, 21 July 2020, towardsdatascience.com/implementing-a-chess-engine-from-scratch-be38cbdae91. Accessed 7 Nov. 2021.

Cai, Michael, et al. Chess Teacher. 17 Mar. 2021.

Carr, Jenna. An Introduction to Genetic Algorithms. 2014.

Azhar, Damir. “Download Limit Exceeded.” Citeseerx.ist.psu.edu, 2008, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.567.7035&rep=rep1&type=pdf.

Staff Writer, AIT. “Understanding the Role of AI in Gaming.” authority, 19 Feb. 2020, aithority.com/computer-games/understanding-the-role-of-ai-in-gaming/.

Vadapalli, Pavan. “Min-Max Algorithm in AI: Components, Properties, Advantages & Limitations.” UpGrad Blog, 22 Dec. 2020, www.upgrad.com/blog/min-max-algorithm-in-ai/. Accessed 7 Nov. 2021.

Kenny, Vincent. “Heuristic Algorithms - Optimization.” Optimization.mccormick.northwestern.edu, 25 May 2014, optimization.mccormick.northwestern.edu/index.php/Heuristic\_algorithms.

Andresen, S. L. (2002). John McCarthy: father of AI. IEEE Intelligent Systems, 84--85.

Cauchy, A. (1847). M{\&#39;e}thode g{\&#39;e}n{\&#39;e}rale pour la r{\&#39;e}solution des systemes d’{\&#39;e}quations

simultan{\&#39;e}es. Comp. Rend. Sci. Paris.

Clementis, L. (2014). Study of Game Strategy Emergence by using neural networks. Bratislava:

Faculty of Informatics and Information Technologies.

Christian, B. (2011). The most human: What talking with computers teaches us about what it

means to be alive. Anchor.

He, K. a. (2016). Deep residual learning for image recognition. Proceedings of the IEEE conference on

computer vision and pattern recognition.

He, K. a. (770--778). Deep residual learning for image recognition. Deep residual learning for image

recognition, 2016.

Hinton, G. E. (2006). Reducing the dimensionality of data with neural networks. Science.

Jeremy G. Bridon, Z. A. (2009). An Artificially Intelligent Battleship Player Utilizing Adaptive.

Pennsylvania.

Jolliffe, I. (2005). Principal component analysis. Wiley Online Library.

Kaiser, L. a. (2019). Model-based reinforcement learning for atari. arXiv preprint arXiv:1903.00374.

Kondo, N. a. (2019). Playing game 2048 with deep convolutional neural networks trained by

supervised learning. Journal of Information Processing, 340--347.

Krizhevsky, A. a. (2012). Imagenet classification with deep convolutional neural networks. Advances

in neural information processing systems, pp. 1097--1105.

McCarthy, J. (2007). WHAT IS ARTIFICIAL INTELLIGENCE? Retrieved from

http://35.238.111.86:8080/jspui/bitstream/123456789/274/1/McCarthy\_John\_What%20is%20artifi

cial%20intelligence.pdf

Mnih, V. a. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint

arXiv:1609.04747.

Russell, S. a. (2002). Artificial intelligence: a modern approach. Prentice-Hall.

Simonyan, K. a. (2014). Very deep convolutional networks for large-scale image recognition. arXiv

preprint arXiv:1409.1556.

Turing, A. M. (1950). Computing machinery and intelligence. MIT Press Cambridge, MA.

Yegnanarayana, B. (2009). Artificial neural networks. PHI Learning Pvt. Ltd.

Zita, I. (2012). Reinforcement learning in games. Springer.

Bridon, Jeremy, et al. An Artificially Intelligent Battleship Player Utilizing Adaptive Firing and Placement Strategies. 2009.

Suchow, Jordan, and Thomas Griffiths. Deciding to Remember: Memory Maintenance as a Markov Decision Process. 2015.

IBM Cloud Education. “What Are Neural Networks?” Www.ibm.com, 17 Aug. 2020, www.ibm.com/cloud/learn/neural-networks.

Delgado, Carlos. “How to Fix TensorFlow Warning: Could Not Load Dynamic Library ‘Cudart64\_110.Dll’; Dlerror: Cudart64\_110.Dll Not Found.” Our Code World, 6 Jan. 2021, ourcodeworld.com/articles/read/1433/how-to-fix-tensorflow-warning-could-not-load-dynamic-library-cudart64-110dll-dlerror-cudart64-110dll-not-found. Accessed 16 Feb. 2022.

“Build a Deep Neural Network in 4 Mins with TensorFlow in Colab.” Www.youtube.com, www.youtube.com/watch?v=\_VTtrSDHPwU. Accessed 16 Feb. 2022.

262588213843476. “This Is a Simple Battleship Game, Made with Lists on Python. This Is the First Game I Ever Made.” Gist, gist.github.com/guimaion/9275543. Accessed 16 Feb. 2022.

AryanMurthy. “Let’s Build a Battleship-Game.” GitHub, 29 July 2021, github.com/AryanMurthy/Battleship-game/blob/main/battleship.py. Accessed 16 Feb. 2022.

Brownlee, Jason. “Keras Tutorial: Develop Your First Neural Network in Python Step-By-Step.” *Machine Learning Mastery*, 4 July 2019, machinelearningmastery.com/tutorial-first-neural-network-python-keras/.