INTRODUCING GRAMMATICAL EVOLUTION TO D.E.A.P. AND APPLYING IT AS AN INTILLEGENT VIDEO GAME CONTROLLER

Interim Report

Ronan McMullen – 0451657 – January 2021

Supervisor: Prof. Conor Ryan

Contents

Table of Figures

Chapter 1 – Introduction and Analysis

1.1 – General Introduction

The title of my final year project is as follows:

"Introducing Grammatical Evolution to D.E.A.P. and Applying it as an Intelligent Video Game Controller."

I think it appropriate at the outset of this report to briefly introduce some of the different components that combine to form this title, namely:

- Grammatical Evolution
- D.E.A.P.
- Intelligent Video Game Controller

Grammatical Evolution (GE) is an evolutionary computing approach that was conceived in University of Limerick in 1998 by Conor Ryan, J.J. Collins and Michael O'Neill. It can be described as an evolutionary computing system that generates working programs. A program can be understood as a series of instructions that need to be carried out in order to complete a task. An essential component of a GE implementation is a defined grammar. A grammar defines the set of rules that illustrate the syntax of sentences and expressions to be evolved in the target language (Noorian *et al.* 2016). GE utilises the search aspects of a simple genetic algorithm to identify individuals in the form of variable length binary strings, these individuals are then mapped into a syntactically legal expression. The syntactic validity of the resultant expression is guaranteed provided the initial grammar specification conforms to the Backus-Naur Form (BNF) of grammar representation (Ryan *et al.* 1998). It is my intention to provide a deeper explanation of GE and BNF as well as evolutionary computing in general in the proceeding chapters of this report.

D.E.A.P. is an acronym that stands for Distributed Evolutionary Algorithms in Python. It is a completely opensource Python framework that exists to provide a user-friendly and abstracted solution for rapid and lightweight implementations of evolutionary algorithms (De Rainville *et al.* 2012). There exists at present functionality within D.E.A.P. to easily implement both classic Genetic Algorithms (GA) and Genetic Programming (GP) algorithms. It is an aim of my project to introduce the functionality of GE to this framework. As will be outlined in later chapters of this report, my proposed GE implementation will be able to utilise much of the existing GA functionality already present in D.E.A.P.

In the context of the title of this final year project the term "Intelligent Video Game Controller" refers to an area of research that utilises video game emulators as training and testing grounds for artificial intelligence and machine learning agents. The closed environment, ease of integration and ready metrics and objectives make video game emulators a particularly suitable vehicle for agent research (Togelius 2015). From Turing's 1953 handwritten MiniMax approach to chess (Turing 1953), right up to the dizzying feats of gaming completed by Google's Deepmind (Vinyals *et al.* 2019), video games and AI research have enjoyed a close relationship over the years.

1.2 – Project Motivations

Personal interest and curiosity were two significant factors the motivated me to pursue this particular project. Since I began my undergraduate program I have readily consumed all course content that was related to AI and machine learning. I find it very exciting to be immersed in a field, that by traditional academic measures can be described as nascent. The blend of bio-inspiration, computer science, mathematics, logic and academia is one that I find intoxicating and stimulating. There is a certain "Lewis and Clark" energy to this field that is very motivating, the sense that the boundary of possibilities and expectations for AI and machine learning is constantly being expanded is one that drives me.

Personal motivations aside I feel that the successful introduction of GE to the D.E.A.P. framework can potentially be of value to researchers and students of Evolutionary Computing. D.E.A.P. is, by nature, intended to facilitate ease of implementation and I am motivated to ensure that my GE addition adheres to this intention. The potential for a user friendly, Python based GE implementation may stimulate entry level research and instruction in this particular field of evolutionary computing.

The second component of my project, that of the intelligent video game controller, comes with its own motivations. The notion of a machine learning agent autonomously controlling a video game is one that has captured my imagination since I first witnessed an application of it. As a lifelong video game player the process of figuring out and mastering video game challenges is one that I am very familiar with. I find it particularly fascinating to consider applying a machine learning agent to tasks that I know so well and I am hopeful that my domain knowledge will be of value to me as this project progresses. I feel that a video game learning environment will provide a controlled arena in which to effectively evaluate GE performance across different problem types.

I would be remiss if I did not note that the opportunity to work with my supervisor Professor Conor Ryan and his research peers was also a motivating factor in undertaking this body of work. Prof. Ryan is a widely regarded and oft cited pioneer in the field of GE specifically, to have access to his input and to that of his research colleagues is a prestigious asset for any would be researcher.

Chapter 2 – Evolutionary Computing

2.1 – Introduction

Evolutionary computing enjoys a rich and recent history. As is commonly known, Charles Darwin first purported the theory of evolution by natural selection in his 1859 work "On the Origin of Species". This theory boils down to a concept of survival of the fittest, in that the fittest or most successful individuals in a population hold the greatest potential to influence the proceeding generation of individuals. This concept is a core tenement of all evolutionary computing.

A next milestone in the lineage of evolutionary computing was presented in Alan Turing's 1948 publication "Intelligent Machinery". After his codebreaking endeavours in World War II, and in the year that he narrowly missed out on a place in the Great Britain Olympic Marathon Team, Turing first broached the concept of machine intelligence. In this paper Turing identified three proposed approaches to possible machine intelligence: intellectual search, cultural search and genetical or evolutionary search (Turing 1948). Turing proceeds to describe his concept of evolutionary search as follows:

"There is genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value."

It is easy to see that parallels that this concept has to today's modern implementations of evolutionary computing.

Although his most recognised contribution to computer science may be the eponymously titled Bremermann's Limit, Hans Joachim Bremermann's 1962 paper "Optimization Through Evolution and Recombination" can be considered as the next milestone in the development of evolutionary computation theory. In the second half of this paper Bremermann analyses the challenge of applying evolutionary learning to the task of minimising the cost function of a set of linear inequalities (Bremermann 1962). He posits on the merit of introducing "*mating*" into the search population alongside the already present mutation operator. Shortly after in 1964, Ingo Rechenberg further advanced the evolutionary computing field with his work on Evolutionary Strategies. This work introduced different approaches to population determination between generations, the "*comma strategy*" which populates the proceeding generation solely with offspring and the "*plus strategy*" which uses both parents and offspring to populate the next generation. The "*comma strategy*" favours greater exploration of the problem space at the risk of losing some valuable genetic material whereas the "plus strategy" prioritises exploitation of the fittest available genetics. These strategies highlight one of the ever present trade-offs that exist in the field of evolutionary algorithms, that of exploration vs. exploitation (Rechenberg 1989).

In 1965 Lawrence J. Fogel and his colleagues at Decision Science again expanded the frontier of evolutionary computing as they became the first research team to specifically apply evolutionary computing to real-world problems (Fogel *et al.* 1965).

A seismic development that would forever change the landscape of evolutionary computing came in 1975 when John Holland wrote his ground-breaking book "*Adaption in Natural and Artificial Systems*". In this work Holland first unveiled his invention of Genetic Algorithms (GA) (Holland 1975). Holland also devised Schema Theorem. This area of study was reprised in 1989 through David Goldberg's work "*Genetic Algorithms in Search, Optimization and Machine Learning*" (Goldberg 1989).

Which brings us to John Koza's influential work from 1992, "*Genetic Programming, On the Programming of Computers by the Means of Natural Selection*". In this volume Koza outlined his work on the conception of Genetic Programming (GP) (Koza 1992). In the opening chapter of this work Koza discusses the pursuit of getting computers to solve problems that they have not been explicitly programmed for, he notes that:

"The attributes of a solution should emerge during the problem solving process as a result of the demands of the problem."

Koza proceeds to introduce GP by presenting the following notion: if one's goal is to enable a computer to generate a program to solve a problem then the optimal solution must reside in the space occupied by all computer programs. In theory this space is infinitely large and complex and thus Koza reasons that in order to search it effectively and intelligent and adaptive approach must be undertaken (Koza 1992).

Evolution of the field of evolutionary computing continues strongly to this day, for the remainder of this chapter I will describe some of the different areas of it that combine to form the current state of the art.

2.2 – Genetic Algorithms

The Simple Genetic Algorithm (SGA) forms the foundation of evolutionary computing. The algorithm takes root in Darwin's theory of evolution, where survival of the fittest dictates that the fitter or more successful an individual in a population is, the more influence its genetics shall have on the following generation of individuals (Goldberg 1989). In turn those individuals that have inherited from successful parentage are in theory equipped with advantageous genes giving them a greater chance of reproductive success and so on. Natural mutation of genes occurs between generations, this, coupled with the splicing of parent genes to produce offspring serves to promote genetic diversity thus expanding the overall pool of genetic material, this process is simulated by an SGA.

Genetic algorithms in modern practice are often applied to problems of optimisation, problems for which there exists an optimal solution. A GA searches the space of all potential solutions to the problem at hand by creating individuals or population members. Any one individual can be considered as a potential solution, their fitness is evaluated based on how appropriate the solution they represent is (Goldberg 1989). Over the course of a number of generations of evolution the genetic algorithm seeks to guide its search for optimality based on the feedback it receives from the fitness evaluation of its individuals.

However, fitness alone is not enough to enable a GA to successfully search a problem domain. Genetic operators such as crossover and mutation must be applied to certain individuals as the population of the next generation is being created. The application of these operators promotes diversity and enhances the search capabilities of the algorithm (1989). These genetic operators mimic the naturally occurring genetic activities that arise through reproduction across generations of a species. Crossover refers to the of crossing over two parent genomes to produce two new child genomes formed from different combinations of the same genetic material as the parents. For the purposes of GA, genomes are often represented by fixed length binary strings. When crossover is applied to two of these bit string genomes they are each split into two parts at a designated crossover point. The first part of parent 1 is combined with the second part of parent 2, similarly the first part of parent 2 is combined with the second part of parent 1. Thus, two different combinations of the parents' genes are created and are ready for fitness evaluation in the coming generation (Goldberg 1989). Mutation refers to the process where a single bit or a number of single bits within an individual are altered, for example a 0 could be flipped to a 1 at a number of points in the genome. Mutation imitates the natural and minor mutations that occur generationally in codons through reproduction (Goldberg 1989). Crossover and mutation serve to increase diversity amongst the populations in a GA. Without these measures to promote random search a GA could likely converge upon a local maximum solution (or minimum as appropriate).

A local maximum is an area of a problem's search space that is not optimal, from where it seems to the algorithm that there is no better solution. To perceive this idea one can consider a small island in the middle of a body of water, to someone standing on a hill on this island it may appear that they are on the highest observable piece of land. To venture in any direction would bring them downward and toward the water. If that person's goal is to stand at the highest place possible then they might well consider this goal satisfied. However, in this example there exists a mountain across the water and out of view of our island dweller, this mountain rises far higher than the hill and is representative of the true maximum or optimal solution to the problem. Crossover and mutation introduce an element of randomness to the genetic pool that is valuable in allowing a GA to maximise its search potential by forcing it to search locations that it may not have reached otherwise. As a GA approaches a potential solution, the relationship between convergence and diversity comes into play. Depending on parameters such as population size and crossover rate as well as which types of crossover, mutation and selection are used, a GA may favour one or the other.

The final component of GA to be addressed is that of selection. Selection is carried out on the population of a GA between each generation and there are a number of approaches to this task. Roulette wheel selection is a selection process whereby the likelihood that an individual will be selected is directly related to its fitness in relation to the entire population, it is a method that prioritises selection of individuals based on their global fitness. Tournament selection sees a group of *k* individuals chosen, the fittest one or two individuals from this local group is selected to propagate into the next generation.

2.3 – Genetic Programming

Genetic programming is a type of evolutionary algorithm that can evolve working programs or expressions. GP shares many foundational concepts with the above described GA. A primary area of difference between the two algorithms is that of representation of the individuals within a population. In GA, an individual is usually represented by a fixed length string of binary digits. In GP, the individuals are represented by variable size syntax trees or as Koza refers to them "*expression trees*" (Koza 1992). A syntax tree is a data structure that describes an executable program or expression. Syntax trees are formed by populating all internal tree nodes with functions or operators and all leaf nodes with variables or terminal values. This difference in representation necessitates a different technique for population initialisation to that of GA.

In his 1992 work Koza illustrates three approaches to population initialisation in GP. The *"Full"* method, requires a user specified maximum depth for the syntax trees to be initialised (Koza 1992). Trees are created by the following algorithm as it proceeds along the tree's nodes at various depths:

- If the current node is at a depth that is less than the maximum specified depth then populate it with a function or operator.
- If the current node is at the maximum specified depth then populate it with a terminal or variable.

Koza proceeds to describe the "*Grow*" method for GP population initialisation, this method differs from the "*Full* " method in that all trees initialised will have a depth up to but no deeper than the maximum specified depth (Koza 1992). It proceeds as follows:

- If the current node is at a depth that is less than the maximum specified depth then randomly populate it with either a function or a terminal.
- If the current node is at the maximum specified depth then populate it with a terminal.

Koza's third initialisation technique is known as "*Ramped, Half and Half* " initialisation (Koza 1992), as the name suggests it is a combination of the "*Full"* and "*Grow*" methods.

The fitness of an individual in GP is determined in a number of different and problem specific ways. For symbolic regression problems mean squared error (MSE) can be used as a loss function to be minimised. Each individual expression is calculated and compared to the benchmark or target output. The MSE does exactly what it says on the tin and keeps track of the squared value of the sum of the differences between an evaluated individual and the target value, divided by the number of individuals evaluated. This fitness function serves to guide the GP search for a more suitable expression. Genetic programming evolves working programs and due to the tree representation of its individuals it must utilise GP specific crossover and mutation operators.

2.4 – Grammatical Evolution

Grammatical evolution is an evolutionary computing technique that is capable of evolving complete programs in an arbitrary language using variable length binary string genomes (Ryan et al. 1998) (O’Neill *et al.* 2001). A defining feature of the algorithm lies in how it maps these binary genotype individuals to an evaluable phenotype form using production rules defined in the Backus-Naur Form (BNF) of grammar notation.

BNF is a notation that uses production rules to define the grammar of a particular language (Naur *et al.* 1976). A grammar can be understood as the entire structure or system of a language, the rules by which valid expressions or sentences can be constructed. A BNF grammar consists of a set of terminal elements and a set of non-terminal elements. Terminals are elements of a language such as operators and variables, they are atomic items in terms of their language in that they cannot be any further decomposed. Non-terminals represent the set of elements in a language that are composed of both terminal and non-terminal items, for example the simple expression ‘x + y’ can be considered a non-terminal element as it can be decomposed into three terminal elements ‘x’, ‘+’ and ‘y’ respectively (Naur *et al.* 1976). In BNF notation a grammar can be represented by the following tuple {S, T, N, P}. ‘S’ denotes the start symbol, this is the symbol from which all expressions begin, it is an element of ‘N’ ($S \in N$). ‘T’ represents the set of all terminal values existent in the language or subset of the language being specified. ‘N’ represents the set of non-terminal values and ‘P’ represents the set of production rules that are used to map the terminal elements to the non-terminal elements. The expressions or sentences that are outputted by a GE implementation are dictated and described by the BNF specification that is being used, this aspect of GE grants it a “*unique flexibility*” (O’Neill *et al.* 2001).

A GE implementation displays many similarities to that of a GA implementation, the main differences being:

- Individuals are represented as variable length binary strings in GE whereas they are represented as fixed length strings in GA.
- Individuals or genomes must undergo a mapping to phenome process in GE before they can be evaluated in terms of fitness.
- A GE implementation requires a BNF grammar specification to define the elements and production rules of the language in which expressions are being evolved. The BNF grammar can define an entire language or just a relevant subset of one (O’Neill *et al.* 2001).

GE specifics will be outlined in more detail in chapter 4.

Chapter 3 – Video Game Training Environment

3.1 – Introduction

As outlined in chapter 1, an aim of this project is to apply grammatical evolution (GE) as a video game playing agent. A more accurate description of this aim however would be; to use GE to search the space of all combinations of valid controller inputs to find an expression that optimises a video game based objective. A video game based objective can be considered in terms of the in-game score or reward earned by a player. In this chapter I will introduce the OpenAI Gym and specifically the Arcade Learning Environment (ALE), training environment that I have chosen for the project, I will go on to explain my reasoning for this choice.

3.2 – OpenAI Gym and Arcade Learning Environment

The OpenAI Gym is a Python based machine learning training environment presented by OpenAI, a San Francisco based AI and machine learning research organisation. The OpenAI Gym serves to provide a benchmark environment setting for machine learning research (Brockman *et al.* 2016). The Gym abstracts many details related to the particular training environment that is being used. It outputs information regarding player actions, observations and rewards on a frame by frame basis. Due to the abstracted nature of the environments presented by OpenAI Gym, researchers can easily apply different types of machine learning agents to a variety of problems. It is an objective of OpenAI to promote reproducibility of machine learning solutions (Brockman *et al.* 2016) as well as a standardised set of environments in which to conduct training research. The potential commonality offered by a widespread uptake of a general training environment would be beneficial to the field of machine learning research. A further goal of the OpenAI Gym is the search for general machine learning solutions (Brockman *et al.* 2016). In recent years AI and machine learning agents have completed feats of staggering complexity, however, it is often that case that these agents do not possess the ability to apply their learning to a variety of problems. OpenAI presents environments that provide a range of different problem types within a common framework, it enables users to easily connect their machine learning solutions to any of these environments and thus seeks to promote a broader approach to problem solving in general.

At the outset of this project, when I was conducting preliminary research, the question of which gaming environment I would use was foremost in my mind. After some searching I identified the Arcade Learning Environment (ALE) as a potential candidate. The ALE is a machine learning training environment that is built upon an Atari 2600 emulator (Bellemare *et al.* 2013). It contains over 50 Atari titles, featuring such now legendary games as Space Invaders, Pacman and Pitfall. The variety of games presented by the ALE as well as the existing community of researchers that have utilised it and published their agent's performance related results were two features that drew me to this environment. The ALE is developed in C++ and targeted at Linux and OSX based operating systems, as I intended to use Python as my language of choice for this project I began to look into what steps I would need to take to incorporate ALE into my Python and Windows based, D.E.A.P. supported, development environment. As I dove deeper into the specifics of this incorporation I began to realise that getting the ALE to work within my setup was to be a complex task that was going to take a lot of time and research to complete! Thankfully, the good people at OpenAI recognised the potential held by the ALE and took it upon themselves to implement ALE within their OpenAI Gym. As stated above, OpenAI Gym is a Python based software, the fact that this package now contained all the functionality of the ALE plus much more was a watershed moment for this project. The installation process for the entire OpenAI Gym library was completed in a few brief Python install commands.

The original Atari 2600 was incapable of generating random numbers, as such all games on the system ran deterministically. If a game runs in a deterministic fashion then the sequence of operations carried out during its course will be the same every time it is run. This makes for a more approachable task for GE as the problem space being searched will remain constant between individuals and between generations. The original ALE stayed true to its Atari based roots and so all the games it offered ran in a deterministic fashion. The OpenAI implementation of this emulator however, presents each game available in both deterministic and stochastic forms.

It is clear that the primary focus of the OpenAI Gym is as a training environment for reinforcement learning (RL) agents. RL agents perform learning by observing the action space of the particular problem they are applied too. Within this action space reside all potential actions the RL agent can take. The agent evaluates the potential actions it can make in terms of their perceived reward. When it performs one of these actions the actual reward gained is compared to the predicted reward. The agent is trained based this reward information and the state of the environment at the time the action was carried out (Mnijh *et al.* 2015). I feel that the reward information returned by OpenAI Gym at each frame of gameplay will provide sufficient feedback to guide a GE driven search.

Chapter 4 – Architecture and Design

4.1 – Introduction

In the following chapter I will outline the progress I have made to date regarding the research and actions carried out in relation to completing the following tasks:

- Development of a working GE implementation featuring a BNF grammar parser, a genome to phenome mapping function and a GE specific fitness evaluation function.
- Development and application of a D.E.A.P. based GE prototype.
- Development of an appropriate BNF grammar specification for a video game agent.
- Application of the D.E.A.P. driven GE prototype to an Atari based problem.

I am of the opinion successful completion of the above tasks will serve as a definitive proof-of-concept for my project. Following on from this progress report, I intend to present some of the project related questions and research goals I have uncovered through prototype observation that I intend to address in the coming semester's work.

It must be stated at this point that the following implementation details are as applied to a prototype development project, as such they are subject to change and refinement as the project progresses and are in no way finalised in terms of design or structure.

4.2 – GE Benchmark Implementation

The following section briefly outlines the development and implementation of a benchmark proof of concept for grammatical evolution using D.E.A.P. Using the following grammar I was successfully able to utilise GE to approximate the following function:

*np.exp(-x) * (x**3) * np.sin(x) * (np.cos(x) * (np.sin(x)**2) – 1)*

4.2.1 – Benchmark Grammar

```
S               ::= <expr>
<expr>          ::= <op>
<sub-expr>      ::= <func> | <var> | <op>
<func>          ::= psin(<sub-expr>) | pcos(<sub-expr>) | pow2(<sub-expr>) | neg(<sub-expr>)
<op>            ::= <push_tree>add(<sub-expr>, <sub-expr>) |
                    <push_tree>sub(<sub-expr>, <sub-expr>) |
                    <push_tree>protected_div(<sub-expr>, <sub-expr>) |
                    <push_tree>mul(<sub-expr>, <sub-expr>)
<var>           ::= x | 1
<push_tree>     ::= -P-
```

4.2.2 – Benchmark Performance Graphs

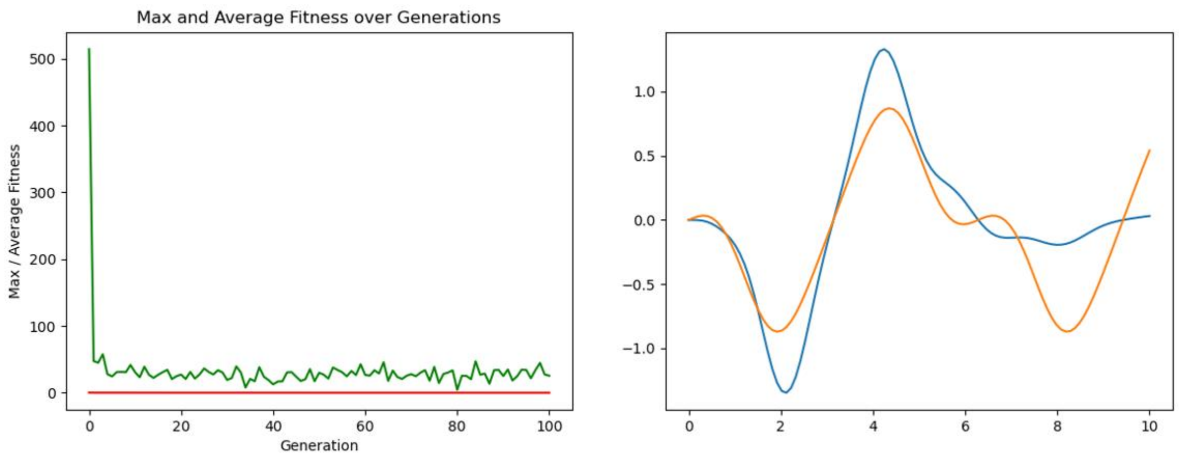


Figure 1 - Benchmark Performance

Fittest Phenotype: mul(psin(x),neg(pcos(add(psin(pcos(1)),pcos(x)))))

4.3 – GE Analysis and Evaluation

4.3.1 – BNF Parser

The first step in my practical implementation task was to begin development on a BNF grammar parser. To do so I identified a simple, Python based grammar specification with which I would work. It is defined below:

```
   S            ::= <expr>
<expr>          ::= <expr><op><expr> | <sub-expr>
<sub-expr>      ::= <func>(<var>) | <var> | pow(<var>,<n>)
<func>          ::= math.log | math.sqrt | math.sin | math.cos
<var>           ::= 100 | pow(100,<n>) | <n> | math.pi
<op>            ::= + | - | * | / | //
<n>             ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

As outlined in Chapter 2, a BNF grammar serves to define the syntax rules of a particular language. In this example the above BNF specification defines a grammar for Python based mathematical expressions. 'S' denotes the start symbol: *<expr>,* this symbol can be recursively called and it is from here that all expressions will evolve. My first task when it came to implementing a BNF parser was to decide upon the structure I would use to store the parsed grammar data. Each production rule in a grammar is composed of a label (*title*) and the values (*components*) it corresponds to. For the purposes of my implementation I decompose the production rules into a string variable *title* and a corresponding list of *components.* For example, for the following rule:

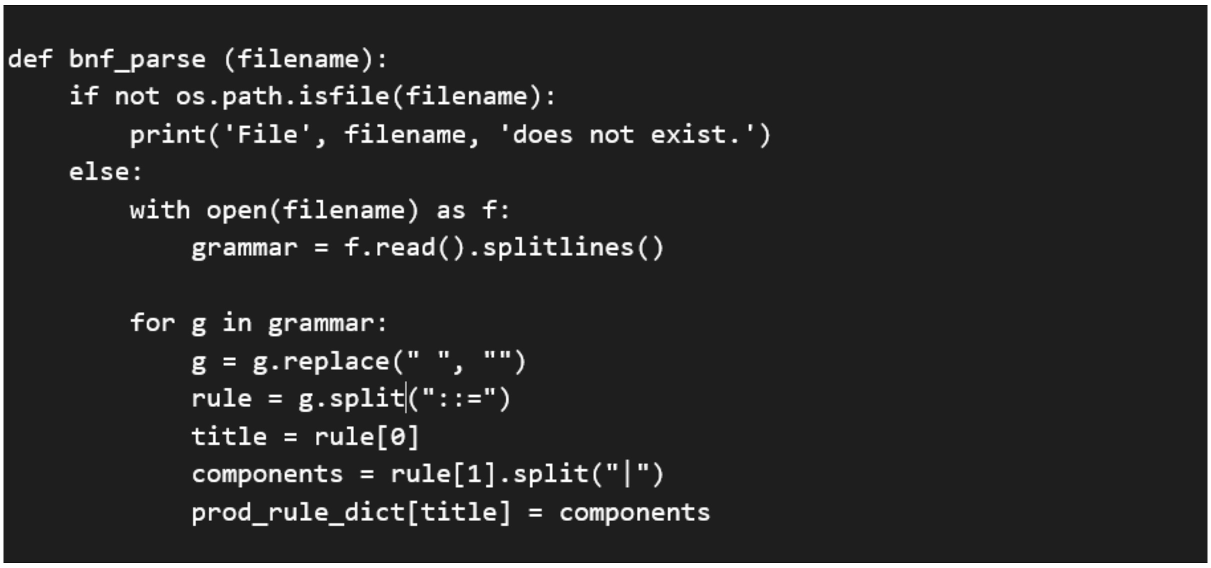
```
<sub-expr>     ::= <func>(<var>) | <var> | pow(<var>,<n>)
```

title = <sub-expr>

components = [<func>(<var>), <var>, pow(<var>,<n>)]

A dictionary makes an appropriate structure for this decomposition. A dictionary is a Python data structure that stores its data in key-value pairs, in this case the key would correspond to the rule title and the value associated with it would be the list of terminal and/or non-terminal values. The creation of a dictionary for BNF production rules will allow the components of any rule to be easily retrieved by referring to the title of said rule.

The next step was to begin work on the parsing function itself. My implementation can be seen below:



```
def bnf_parse (filename):
    if not os.path.isfile(filename):
        print('File', filename, 'does not exist.')
    else:
        with open(filename) as f:
            grammar = f.read().splitlines()

        for g in grammar:
            g = g.replace(" ", "")
            rule = g.split("::=")
            title = rule[0]
            components = rule[1].split("|")
            prod_rule_dict[title] = components
```

Figure 2 - bnf_parse function definition

As you can see in *Figure 1*, it was but a matter of applying some straight-forward Python string functions and the production rules were extracted from file and structured appropriately.

4.3.2 – Genotype -> Phenotype Mapping Process

The next development goal I had was to begin work on my implementation of a genome to phenome mapping function. Before I describe my development efforts in this area I would like to provide some further detailed information on the genome to phenome mapping process as completed by GE. To reiterate from earlier, GE represents its individual population members as variable length binary strings. During the genome to phenome mapping procedure these individual bit strings are decomposed into codons consisting of eight binary digits (8-bit). The mapping function uses the decimal representation of these codons to influence which BNF production rule, terminal or non-terminal element it chooses for the currently evolving expression.

The modulus operator (%) is used in programming to determine the integer remainder value resulting from division, for example the expression 10 % 3 returns a value of 1, as 10 is divided by 3 a total of 3 times with 1 remaining (10 % 3 = 1). GE uses this operator to "mod" the decimal representation of the binary codon by the number of choices available in the currently selected production rule. Allow me to illustrate this process in some more detail with an example.

Consider the below genome, already split into 8-bit codons:

11001100 – 10101011 – 11011100 – 00101100

When converted to decimal representation the codons are as follows:

204 – 171 – 220 – 44

Now, consider the BNF grammar specified in the above section, as denoted by 'S', the start symbol for this grammar is the non-terminal symbol *<expr>* so let us begin our expression there.

<expr>

This particular production rule has 2 available choices, namely:

Choice (0):	<expr><op><expr>
Choice (1):	<sub-expr>

With this in mind the mapping function must evaluate the following expression:

Decimal representation of current codon value % number of currently available BNF choices

The nature of the modulus operator is such that resultant value will never be a number that is greater than the right hand side variable in the expression, that is to say the result will always represent an existing choice. This property guarantees that, regardless of the values of codons from a genome, the GE mapping process will always provide valid identifiers for production rules, terminal and non-terminal elements. This in turn, given that the BNF grammar in use is correctly specified, guarantees the evolution of legal expressions.

In our case, the operation is 204 % 2, it evaluates to 0 so the algorithm selects Choice (0) to replace the non-terminal symbol *<expr>* leaving our expression as:

<expr><op><expr>

The non-terminal elements in an expression are mapped constantly from the left so as the mapping function proceeds, the next element to be examined is again *<expr>*. The next codon value in the current genome is 171 and the *<expr>* production rule holds 2 choices so the result of the modulus operation this time around is 1. Choice (1), as outlined above, is selected to replace the leftmost non-terminal. The expressions is now:

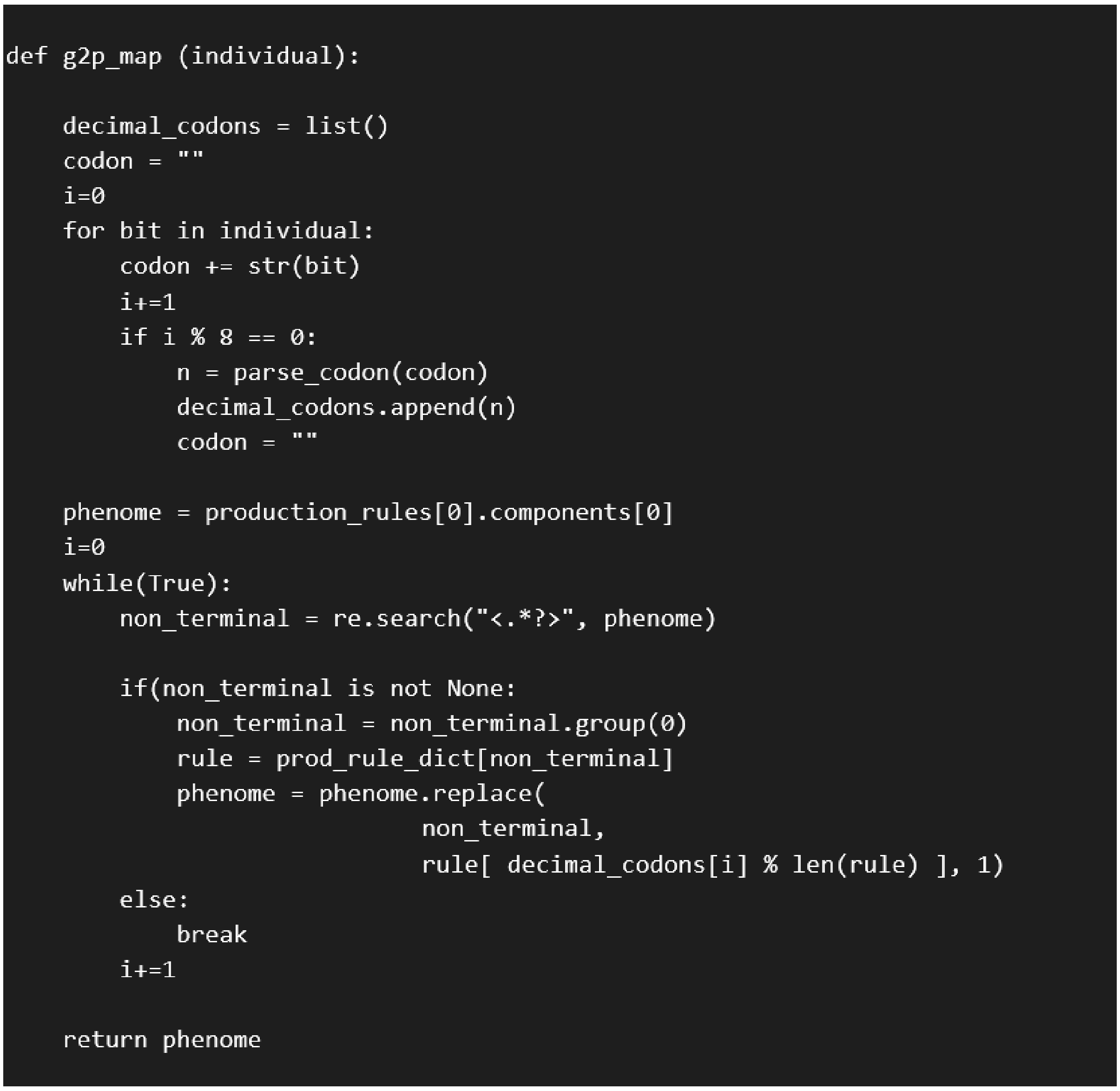
<sub-expr><op><expr>

The mapping process continues in this way until the resulting expression is comprised entirely of terminal values. At this point the genotype to phenotype mapping is complete and the phenome string is returned from the function. See Appendix for a sample mapping process in its entirety.

4.3.3 – Genotype -> Phenotype Prototype

Below is my proto implementation for the *g2p_map* function. It takes as an argument a bit string individual of variable length. Assuming that all codons are 8 bits in length it proceeds to divide the individual into codons and then parse these into their decimal representations. These values are added to the list *decimal_codons.* Next, the phenome is initialised with the component of the first production rule as its starting point.

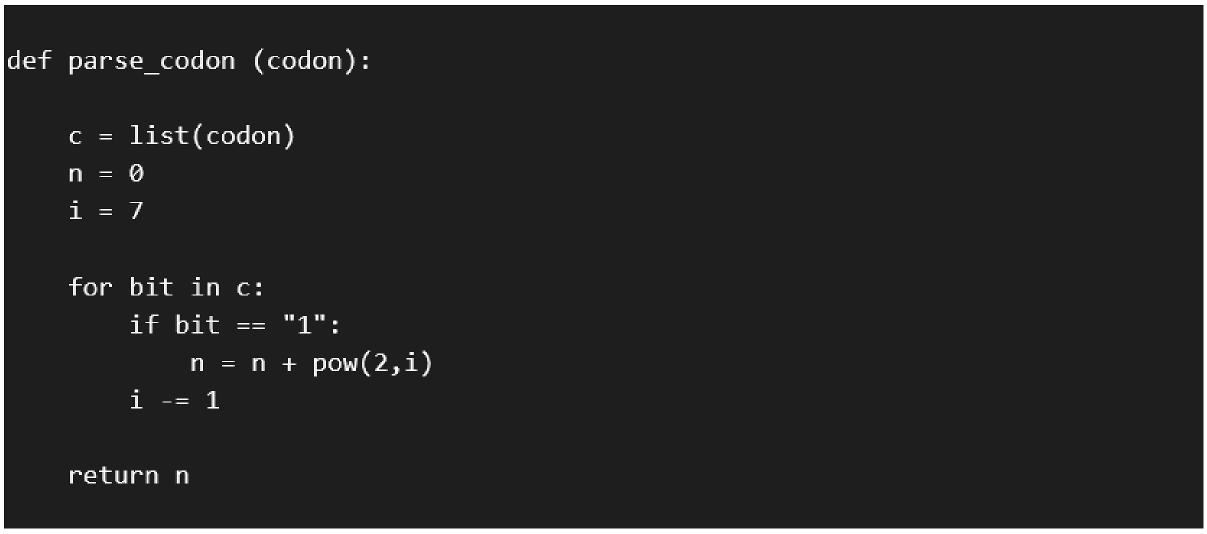
The algorithm enters an infinite while loop. The regular expression function *re.search* is used to find the first instance of a non-terminal element in the phenome. This non-terminal is then mapped to its component parts by way of the *prod_rule_dict* as outlined in the BNF parser. Using the entries in the *decimal_codons* list and the number of choices in the selected production rule, as represented by *len(rule),* the appropriate non-terminal symbol is replaced by the correctly chosen grammar component. This execution proceeds until the expression consists of only terminal values at which point the loop exits and the phenome is returned.



```
def g2p_map (individual):

    decimal_codons = list()
    codon = ""
    i=0
    for bit in individual:
        codon += str(bit)
        i+=1
        if i % 8 == 0:
            n = parse_codon(codon)
            decimal_codons.append(n)
            codon = ""

    phenome = production_rules[0].components[0]
    i=0
    while(True):
        non_terminal = re.search("<.*?>", phenome)

        if(non_terminal is not None:
            non_terminal = non_terminal.group(0)
            rule = prod_rule_dict[non_terminal]
            phenome = phenome.replace(
                                 non_terminal,
                                 rule[ decimal_codons[i] % len(rule) ], 1)
        else:
            break
        i+=1

    return phenome
```

Figure 3 – g2p_map function definition

The following helper function takes as argument a string variable that represents an 8-bit codon. It serves to convert the codon to decimal format.



```
def parse_codon (codon):

    c = list(codon)
    n = 0
    i = 7

    for bit in c:
        if bit == "1":
            n = n + pow(2,i)
        i -= 1

    return n
```

Figure 4 – parse_codon function definition

The final component of my first development goal was that of completing a GE fitness function. Within a GE implementation, the fitness evaluation and the BNF specification provide the specific functionality of that implementation, they are specific to the particular application of GE of which they are a part. With this in mind, I will provide an insight into my fitness prototype later in this chapter when I present my initial application of GE.

4.4 – Prototype Application

My next development goal was to utilise my BNF parser and phenome mapper function within a D.E.A.P. supported implementation. I decided to aim big and began formulating an approach to hooking GE up to an Atari game via OpenAI Gym. The first step in this process was choosing the Atari environment. For this proof of concept implementation I chose the game Space Invaders. All Atari game environments in the Gym take as input single integer values corresponding to the available controller inputs, this is known as the action space of the game. Space Invaders, for example, has the following discrete action space:

No-op = 0, Fire = 1, Right =2, Left = 3, FireRight = 4, FireLeft = 5

As the game is rendered it can perform one of the above actions per frame. The same is true for all the Atari games within the Gym environment, however different games have different action spaces, some containing as many as 18 input options. If I wanted to use GE to play this game then I had to devise a BNF grammar that would result in expressions that could be legally performed by the Gym environment. The following is the grammar specification I defined:

```
S             ::= <loop>
<loop>        ::= <play><wait><play><loop> | <play><loop>
<play>        ::= <move> | <dodge> | <shoot> | <wait>
<dodge>       ::= <move_left> <shoot> <move_right> |
                  <move_right> <shoot> <move_left> |
                  <move_right> <shoot> <move_right> |
                  <move_left> <shoot> <move_left>
<wait>        ::= <pause> <repeat>
<shoot>       ::= <fire> <repeat>
<move>        ::= <direction> <repeat>
<move_left>   ::= <left> <repeat>
<move_right>  ::= <right> <repeat>
<direction>   ::= <left> | <right>
<left>        ::= 3
<right>       ::= 2
<fire>        ::= 1 | 4 | 5
<pause>       ::= 0
<repeat>      ::= -1- | -2- | -3- | -4- | -5-
```

Due to the specific nature of the problem presented by Space Invaders I chose to implement an infinitely recursive grammar structure. The elements of the *<loop>* production rule are all recursive and so it is impossible for an expression in this grammar to be completely composed of terminal elements. This aspect of the grammar meant that every expression would have at least one unmapped non-terminal element at the end of it, thankfully the genome->phenome mapping function is not bothered by such details.

The main building blocks of an expression in this grammar are the production rules titled *<wait>*, *<shoot>*, *<move_left>* and *<move_right>*. Each of these rules is comprised of an action followed by an integer number representative of the number of times to repeat this move. You may notice the *<move_left>* and *<move_right>* are defined independently alongside the more general *<move>* rule. I chose to do this as I wanted to be able to specify the non-terminal elements in the *<dodge>* production rule. These non-terminals aim to replicate some commonly used controller sequences and required specific directions to define. When the mapping process is complete the resulting phenotype for this grammar will be of the following form:

1-2-1-5-2-3-2-10-0-2-0-1 …

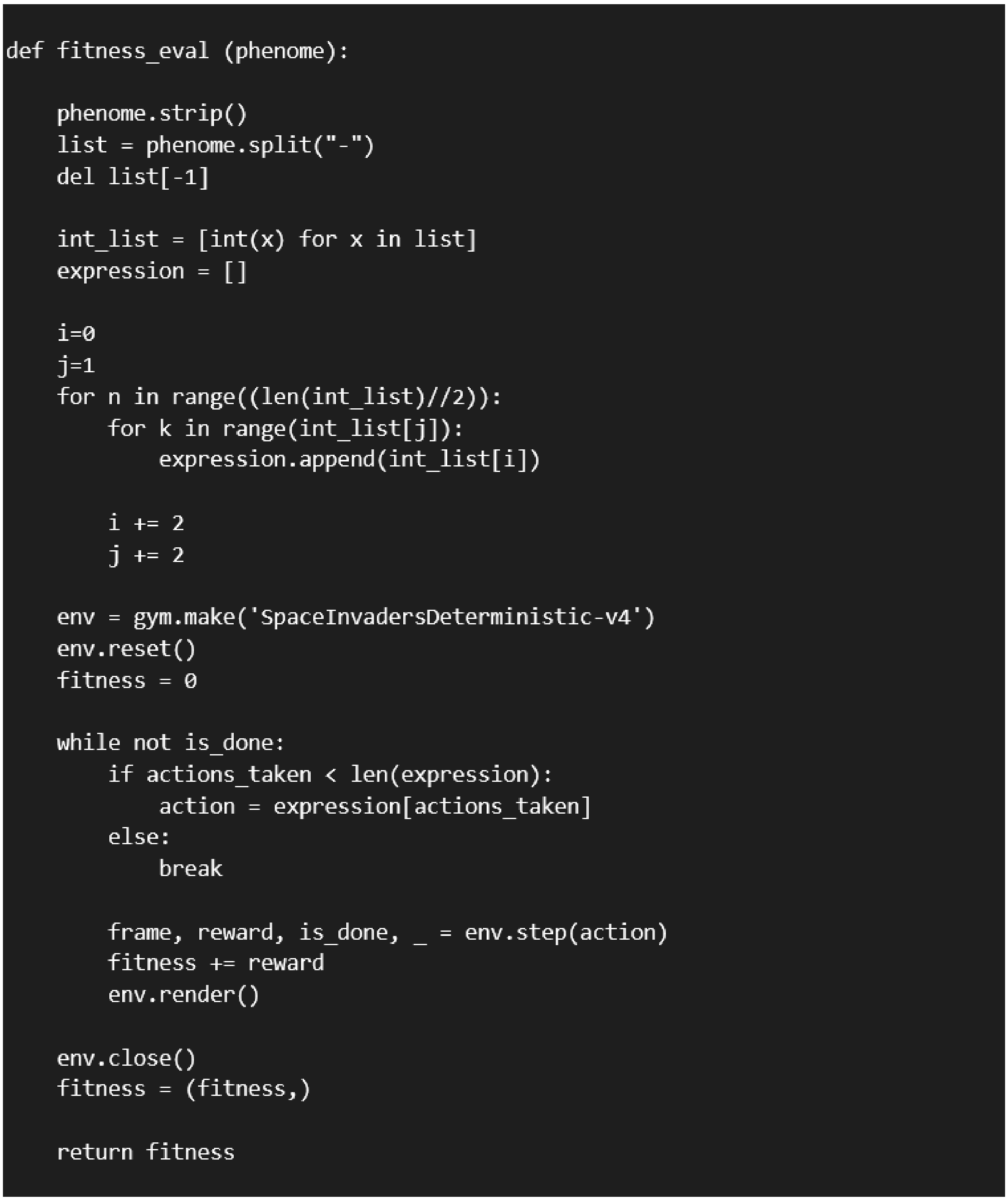
The first digit, and every second digit thereafter corresponds to an action to be inputted into the game environment. The digit that follows each action represents the number of times each action should be repeated. The fitness function is equipped to appropriately parse this input as will be outlined later.

Once I was happy with the grammar specification it was time to hook up the Gym environment. This hook up takes place within the GE fitness function which will be outlined in the next section of this chapter.

4.5 – Fitness Function

The fitness function within a GE implementation is responsible for evaluating the fitness of individual phenotype expressions. It is an aspect of GE that can be different from application to application. The fitness function particular to my prototype takes as an argument the freshly mapped phenotype expression. As stated in the above section, the definition of the grammar for this task will result in a series of integers separated by a hyphen character. The first task the fitness evaluation completes is to clear any leading or trailing whitespace in the phenome. It then splits the phenome at occurrences of " - ". This transforms the phenotype from a string like this: 1-2-3-2-0-6-1-4-0-3-0-6, to a string like this: 123206140306. The phenotype then passes through a pair of nested for loops that apply the action repeat commands resulting in a legal controller expression as such: 1133000000111100000000. At this point in the fitness evaluation the phenotype will be composed of only those values within the game environment's discrete action space as defined in the BNF grammar.

The next step in the fitness evaluation is to make the Gym environment, at this point the value *fitness* is set to 0. Execution then enters the game loop and, provided there are still actions to take, one is assigned to the variable *action* and passed to the game environment with each passing frame of the game. This is repeated until the game ends and returns false for *is_done* or until the phenome runs out of actions. At each frame, the reward gained is added to the *fitness* variable. When the while loop is exited the game environment is closed and the fitness value is returned.



```
def fitness_eval (phenome):

    phenome.strip()
    list = phenome.split("-")
    del list[-1]

    int_list = [int(x) for x in list]
    expression = []

    i=0
    j=1
    for n in range((len(int_list)//2)):
        for k in range(int_list[j]):
            expression.append(int_list[i])

        i += 2
        j += 2

    env = gym.make('SpaceInvadersDeterministic-v4')
    env.reset()
    fitness = 0

    while not is_done:
        if actions_taken < len(expression):
            action = expression[actions_taken]
        else:
            break

        frame, reward, is_done, _ = env.step(action)
        fitness += reward
        env.render()

    env.close()
    fitness = (fitness,)

    return fitness
```

Figure 5 – fitness_eval function definition

4.6 – Prototype Evaluation and Discussion

The following table describes the specifics of a run of the GE prototype.

Objective:	To evolve a controller expression that maximises player score in the Atari 2600 game Space Invaders
Terminal Operands:	Repeat, an integer value between 1 and 10. Determines how many times the terminal operators are repeated.
Terminal Operators:	No-Op, Fire, Move Right, Move Left, Fire Right, Fire Left
Raw Fitness:	Player score, as returned from Gym environment after a game.
Standardised Fitness:	Same as raw fitness.

Parameters:	Population = 100,
	Generations = 50,
	Crossover = Two Point, Crossover Rate = 0.8
	Mutation = Bit Flip, Mutation Rate = 0.05
	Selection = Tournament, Tournament Size = 5

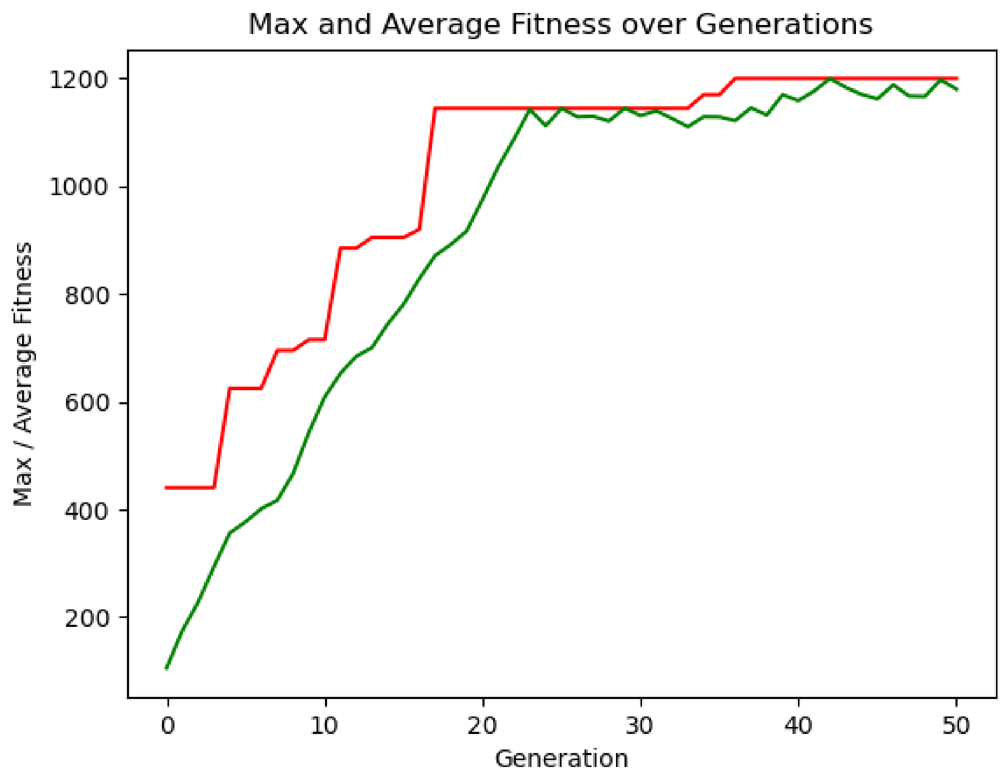


Figure 6 – ge_space_invaders.py sampe run.

Figure 5 above illustrates the algorithm's performance at Space Invaders, the red line indicates the maximum score achieved per generation while the green line represents the average score. You can clearly see the improvement in fitness from generation to generation. The maximum score achieved of 1200 is representative of an averagely performing run for the algorithm and sees the agent make it as far as the final Space Invader in screen one of the game. Based on the above graph, this run of the algorithm might have benefited from a greater number of generations as it does not appear to have fully converged by the time it reaches the 50th and final generation of its run. You may notice that the maximum score seems to plateau for several generations as the evolutionary process proceeds, I feel this phenomenon can be attributed to the fact that as the fitness of potential solutions increases so too does the difficulty of the challenge at hand. The last few Space Invaders to eliminate are proving to be quite difficult for the agent to hit in comparison to the turkey shoot that is the start of a game!

Over the course of the prototype development I have spent a *substantial* amount of time watching it execute and pondering the finer details of the representation and mapping process. My musings often return to the topic of genome length as related to Space Invaders and potentially any other Atari games I intend to tackle. My thoughts on the matter are as such: The games run for varying lengths of time, a fitter individual will usually produce a longer game, a longer game requires a longer control expression, a longer control expression requires a longer phenotype and a longer phenotype requires a longer genotype. That being said it is not sufficient to indiscriminately increase the lengths of individual genomes as this may result in a proliferation of superfluous tailing data which would serve to lessen the effectiveness of crossover and mutation techniques, it could be the case where the "*effective*" portion of an individual repeatedly remains unaffected and unaltered by the genetic operators. If the average individual in a population is too long then only a small percentage of its instructions will be carried out, this leaves the individual with an overly long "*tail*" containing data that has never been evaluated in terms of its fitness. This is an area that I intend to conduct much research on in the coming semester. On a number of occasions, a well performing run resulted in the game playing agent successfully processing every action as expressed by the phenome, when this occurred the algorithm quickly converged as it had essentially reached the end of its search. While I was pleased with agent's performance in these cases I was left wondering just how far it might have progressed if it had a longer individual to begin with. If I can research a solution to focusing crossover and mutation operations to the active and fitness-evaluated areas of individuals, then this issue may be resolved. However, with these measures in place one would still be required to specify an upper limit for individual length, perhaps there is merit in researching an approach that uses dynamically sized individuals? Other areas I intend to explore next semester are those of Intrinsic Polymorphism within GE genomes (O'Neill *et al.* 2003) and of the knock on or "***ripple"*** consequential effect crossover can have on GEs (Keijzer 2001).

I aim to explore development of a somewhat general solution for as many Atari games as possible. I understand that different games can pose vastly different problems for any agent to solve, however they all share one major aspect: the controller. I feel that grammatical evolution can approach the task of a general Atari solution from a unique angle. GE's unique incorporation of BNF provides opportunity for creative interpretation and application of grammar specifications. Each Atari game present in the OpenAI Gym is interacted with in the same way, via a selection of actions taken from the greater set of all actions that can be carried out by an Atari controller. If I can define a BNF grammar that specifies a general controller move-set then in theory it could be sufficient to achieve worthy results across a multitude of different games.

4.7 – Interim Conclusion

I feel that it has been a successful semester in terms of my final year project. Personally, I am satisfied to observe my journey from preliminary research in grammatical evolution to working prototype implementation. I am happy to have been able to successfully search for a useful solution to Space Invaders, this was a reassuring moment as it demonstrated to me that I am on the right track development-wise and that my ultimate goal is an achievable one.

Chapter 5

Preface

Chapter 5 marks the beginning of the second phase of my project, its beginning coincides with the start of the second semester of my final year. It was during this time that I came across a very enlightening paper that fundamentally changed my approach for using GE to evolve a video game controller.

The paper in question is titled "*Evolving a Ms. PacMan Controller using Grammatical Evolution*", it was written by Galván-López *et al.* and released in 2010 as part of the European Conference on the Applications of Evolutionary Computation. In their paper, Glaván-López *et al.* have a similar research objective to my own in this report, that of applying grammatical evolution to the problem of controlling an Atari 2600 game.

Let me explain in some more detail the approach to Ms. Pacman undertaken by Galván-López *et al.* In their own words the approach is described as such:

" In particular we focus our attention on the benefits of using Grammatical Evolution to combine rules in the form of "if <condition> then perform <action>" "

(Galván-López *et al.* 2010)

The research team endeavoured to utilise GE to evolve a working and executable piece of Java code that would have a general application for non-deterministic Ms. Pacman playthroughs. The team used Lucas' Ms. Pacman emulator (Lucas, 2009) as their training ground. They go on to detail how they had implemented some "*high-level functions"* deemed necessary to aid Ms. Pacman on her quest and proceed to outline the BNF grammar they intend to use. The grammar specification was of particular interest to me. As I have gotten quite "*into*" BNF specifications over the last few months I was able to interpret it fairly well at a glance and what I read altered the course of my project.

Galván-López *et al.* specified a set of production rules that would produce a block of code comprised of *if*, *else* and *if-else* statements that, essentially, combine to form a state machine. State machines are a popular approach to video game AI programming, based on the result of the implemented conditional checks different actions will be carried out. The game agent will always be in one state or another and therefore should always be active. GE is used to determine the depth of the conditional checks, their associated parameters and the actions they lead too. Ultimately leading to a general strategy for playing the game. This idea immediately appealed to me and I was very curious to try to apply it to my project.

5.1 – Space Invaders Overview

Below is an example frame from Space Invaders, the non-essential extremities of the board to the left and right and below have been trimmed.

The basic premise of Space Invaders (as the name might suggest) is to defend Earth from an oncoming space invasion. You play as the *Defender*, located at the bottom of the screen. The *Defender* can move from left to right within a defined range and it can shoot straight upwards. The *Invaders* begin in the position shown in the below screenshot. They move together as a block of 36. Beginning on the left side of the screen they track right until they hit the edge of the screen at which point they all drop down a level closer to Earth, speed up and proceed to move to the opposite edge of the screen. This process is repeated until one of three outcomes comes to pass.

1. The *Defender* successfully destroys all *Invaders*, at this point the level is complete and the next level begins.
2. An *Invader* bullet hits the *Defender*. This destroys the *Defender* and will use one of its 3 lives. Once all 3 lives are used the game ends.
3. The *Invaders* manage to reach Earth (the bottom of the game screen). This is a worst case scenario as it will result in game over regardless of how many lives the *Defender* has remaining.

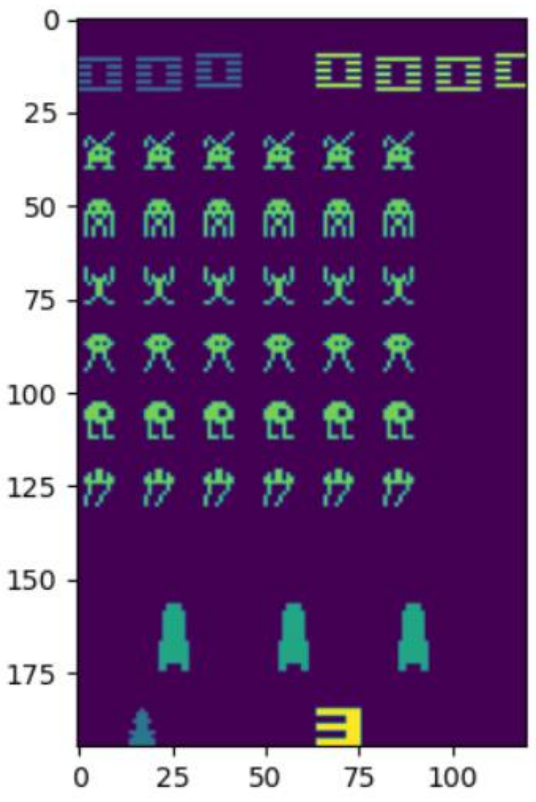


Figure 7 - Trimmed Space Invaders game-screen.

Every *Invader* destroyed by the *Defender* contributes to the overall score achieved. The *Invaders* on the lowest row are worth 5 points each, the *Invaders* on the remaining 5 rows are worth 10, 15, 20, 25 and 30 points, respectively. An entire board of *Invaders* is worth 630 points. This point total does not include the 200 points awarded for destroying the bonus *Mother-ship* which occasionally tracks across the top of the screen. (Space Invaders Manual – 1978).

As *Invaders* are destroyed the movement speed of those that remain increases. By the time there is only one *Invader* left it moves quite quickly and can be rather challenging to destroy. When all *Invaders* have been destroyed the current level is complete. The next level begins with another block of 36 *Invaders*, this time however, they are begin their descent from a lower starting point, increasing the pressure on the *Defender*. Space Invaders is programmed to run indefinitely and as such it cannot be definitively solved.

5.2 – Description of New Approach

I have outlined in the preface of this chapter the approach taken by Galván-López *et al.* in their endeavour to evolve a controller for Ms. Pacman. By defining an appropriate BNF grammar specification they were able to evolve an executable series of conditional statements that could form a strategic approach to game play.

A conditional statement is a type of program control flow syntax. A condition is an expression that contains a conditional operator that can evaluate to either true or false. The commonly known conditional operators are: < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) and == (equal to). As stated, a conditional statement can only evaluate to either true or false, the statement $x < y$ can only be true or false, x is either less than y or it isn't.

The goal of my new approach to this project is to evolve an executable piece of Python code that would successfully control the *Defender's* actions in non-deterministic playthroughs of Space Invaders. Like Galván-López *et al.*, I intend to define a grammar that will produce a syntactically legal expression composed of different conditional checks that can be used to dictate a grand strategy for the *Defender*.

With this in mind I had to consider what information I could extract from Space Invaders in order to from meaningful conditional checks and how I was going to extract it. I also needed to consider just what actions the *Defender* should have the option of carrying out based on the result of these checks.

In order to be aware of the game board in which it operates I needed to grant the *Defender* the gift of vision, this was to be my next development step and is described in the following section of the chapter.

At this stage in the project I now have two parallel development goals, the first, which I shall refer to as Type 1 closely resembles the approach as outlined in my prototype, the second, Type 2 follows the procedures I have just outlined. In this chapter I aim to provide a detailed explanation of some of the functions I have developed to promote the success of each of these approaches. The majority of the proceeding functions are commonly used by both approaches.

5.3 – In the Land of the Blind…

The first step to granting the *Defender* the ability to know its surroundings was to figure out how to retrieve and appropriately process the frame information returned by the OpenAi gym. To address this issue I implemented the function *process_frame*, it is described below.

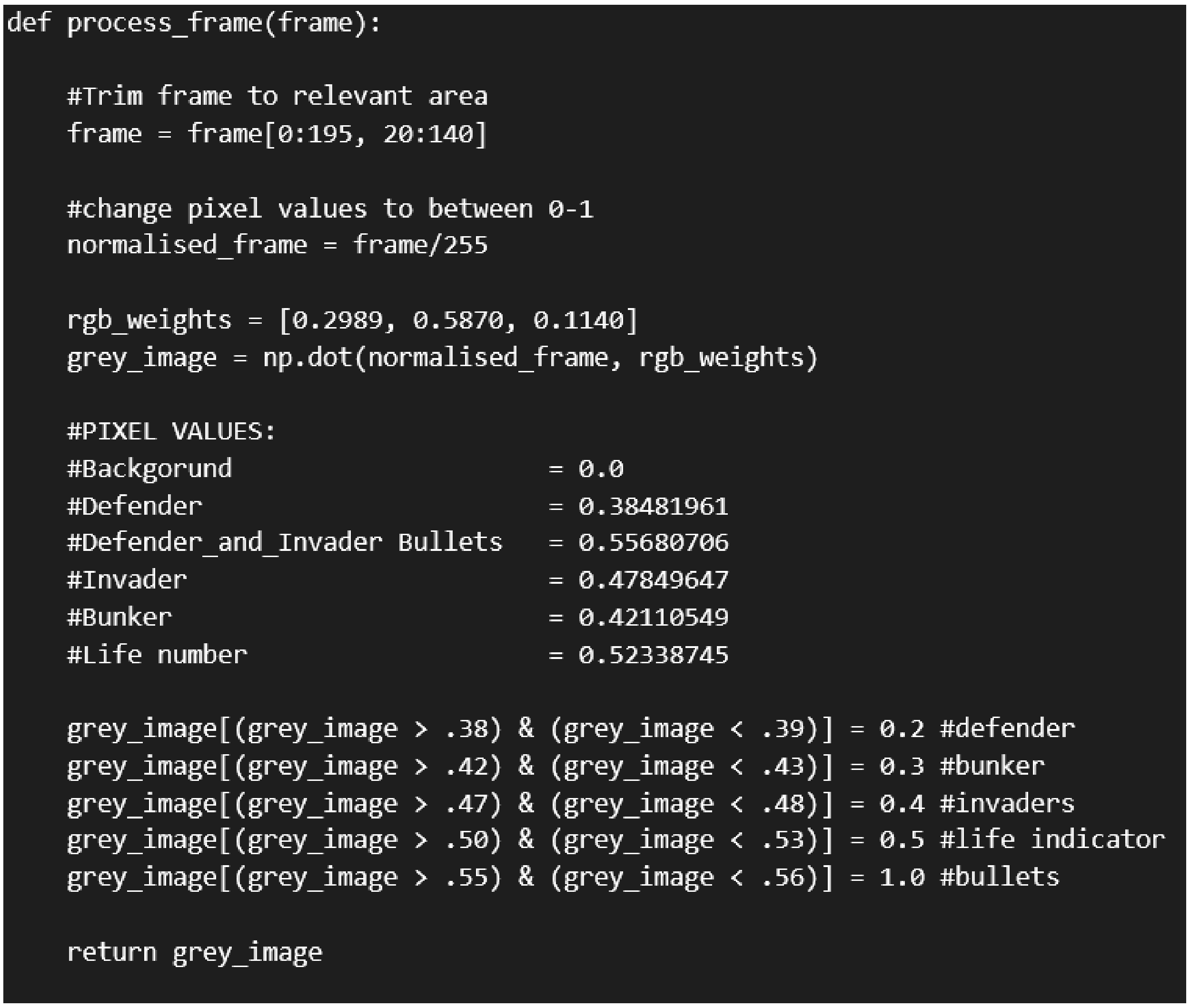
5.3.1 – Process Frame

On a frame by frame basis OpenAI gym will return the current fitness value and the current state of the board. The board state or *frame* is returned in the form of an array of shape 210x160x3. The board itself is of dimensions 210x160 but it is returned in triplicate, one board for each colour red, green and blue. The different values in each of the three arrays correspond to the respective colour value of the pixel they represent, this colour value will be an integer between in the range 0 - 255.

The first task in the *process_frame* function was to trim any superfluous screen pixels from the frame. Some area to the left, the right and the bottom of the game screen can be safely discounted at this point. Next, every pixel value in the frame is divided by 255 (the maximum value a pixel can have), this will normalise the range of pixel values to be within the range 0 – 1. A greyscale image is produced by getting the dot product of the normalised array (dimensions: 195x120x3) and the *rgb_weights* list. This operation results in the variable *grey_image* a numpy array of dimensions 195X120. To clarify, the *grey_image* is of dimensions 195X120, this can be considered in terms of a grid with 195 rows and 120 columns. Each entry on this grid is representative of a pixel displayed on screen. For example, the element at array index [100][100] is a value between 0 and 1 that represents the onscreen colour of the pixel that is in row 100 and column 100.

Within the *grey_image* are a number of values that are of importance. The majority of entries have a value of *0.0*, this is representative of the black background that occupies most of the game screen. The *Defender* has a pixel value of 0.38481961, the *Invaders* have a value of 0.47849647 and, unfortunately, the bullets fired by both the *Invader* and the *Defender* have the same pixel value of 0.55680706. Once these values had been determined I chose to simplify them to one digit values, purely for ease of interpretation on my part. Once this was complete the function returns the processed frame and completes execution. The implementation is displayed in full on the next page.

I had initially considered the merit of down sampling the image to further reduce its demand on processing resources but decided against it as I wanted my agent to have the highest resolution vision possible.



```
def process_frame(frame):

    #Trim frame to relevant area
    frame = frame[0:195, 20:140]

    #change pixel values to between 0-1
    normalised_frame = frame/255

    rgb_weights = [0.2989, 0.5870, 0.1140]
    grey_image = np.dot(normalised_frame, rgb_weights)

    #PIXEL VALUES:
    #Backgorund                     = 0.0
    #Defender                       = 0.38481961
    #Defender_and_Invader Bullets   = 0.55680706
    #Invader                        = 0.47849647
    #Bunker                         = 0.42110549
    #Life number                    = 0.52338745

    grey_image[(grey_image > .38) & (grey_image < .39)] = 0.2 #defender
    grey_image[(grey_image > .42) & (grey_image < .43)] = 0.3 #bunker
    grey_image[(grey_image > .47) & (grey_image < .48)] = 0.4 #invaders
    grey_image[(grey_image > .50) & (grey_image < .53)] = 0.5 #life indicator
    grey_image[(grey_image > .55) & (grey_image < .56)] = 1.0 #bullets

    return grey_image
```

Figure 8 - process_frame

5.4 – Dude, Where's my *Defender*?

The successful implementation of the *process_frame* function opened wide the doors of possibility for this project. Its completion gave the *Defender* eyes, now I had to figure out how to teach it to use them.

The first priority, by my reckoning, was for the *Defender* to be able to determine its own location and so I began development of the function *get_position.* This function takes as an argument a frame that has been processed as described above.

5.4.1 – get_position

get_position is a function that returns the position of the very tip of the *Defender* sprite. The *Defender* is a symmetrical pixel object that is an odd number of pixels wide. As such, it has a conveniently balanced vertical axis running through the central column of pixels in the sprite.

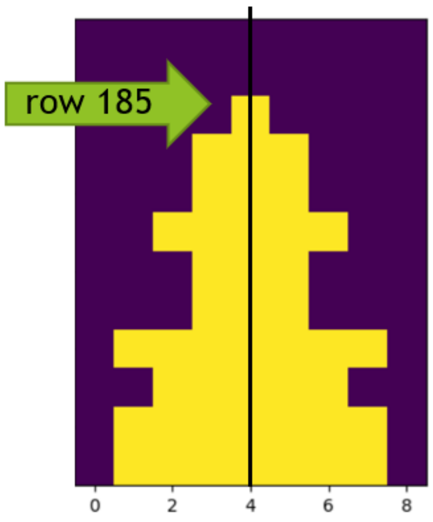
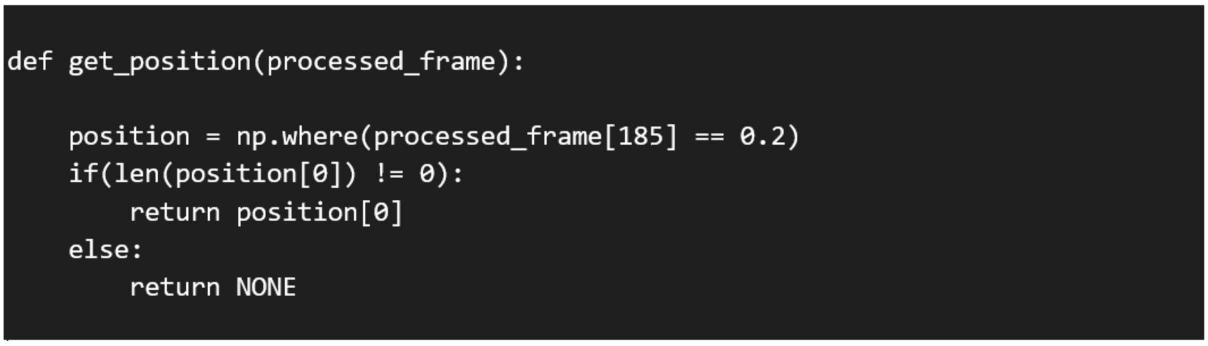


Figure 9 - The tip of the Defender

The tip of the *Defender* is 1 pixel wide, as the *Defender* is limited to horizontal movement this 1 pixel always resides in row 185 of the game screen. *get_position* locates the index in row 185 of the processed frame that has the value corresponding to the colour of the *Defender* sprite (0.2). This index is an integer value that is representative of the current horizontal position of the tip of the *Defender*.



```
def get_position(processed_frame):

    position = np.where(processed_frame[185] == 0.2)
    if(len(position[0]) != 0):
        return position[0]
    else:
        return NONE
```

Figure 10 - get_position

This lightweight function proved to be of great utility moving forward as, once the *Defender's* position had been calculated, it was now possible to determine the relative position of different *Invaders*, the closest, the farthest away, etc.

5.5 – Putting the "Check" in Conditional Check

As stated earlier in this chapter, I intend to evolve an executable block of Python code, comprised of a series of conditional checks that will guide the actions of the *Defender*. As I contemplated this process I tried to consider what meaningful observations the *Defender* should make. To do this, I put myself in the position of a Space Invaders player, I envisioned a playthrough and tried to recognise the things I, myself, would look out for. The majority of my observations revolved around the block of *Invaders*, its position, size and trajectory. In the end I proceeded to develop three functions that would serve in my various conditional checks.

5.5.1 – Check Invader Height

The first of the three checks I implemented is *check_invader_height*. This method returns the value of the lowest row that contains *Invaders*. I felt that it was an appropriate function to include as checking the height of the block of *Invaders* is an intrinsic part of any player's strategy.

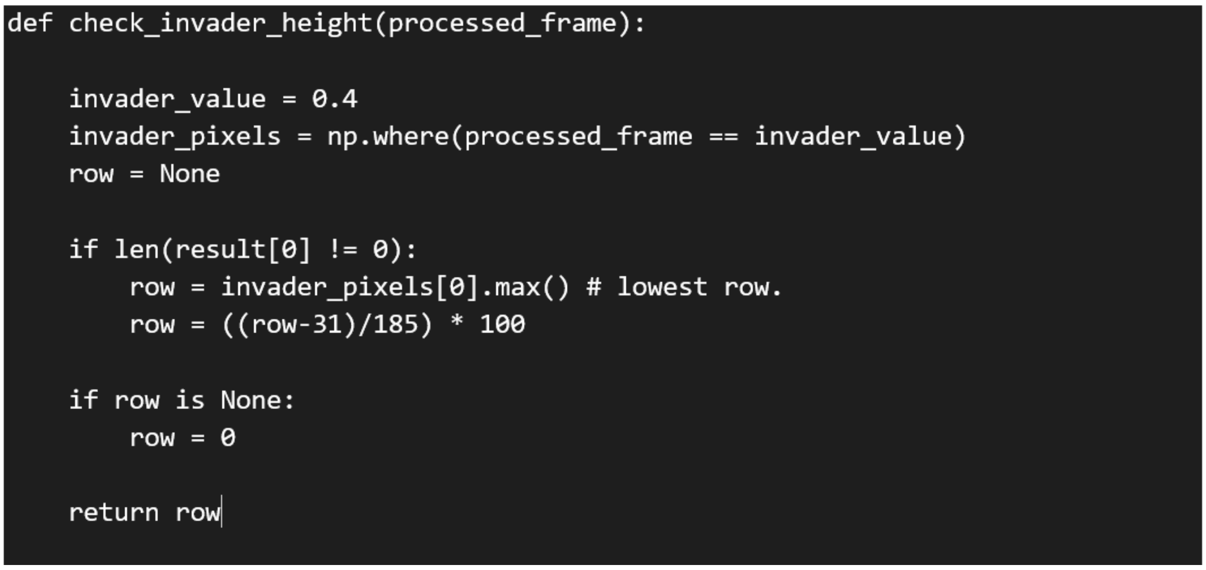
The function is passed a *processed_frame*, it creates *invader_pixels,* a two dimensional array that contains the row and column information of all pixels on screen that are *Invaders* (the pixels whose values match the invader value). *invader_pixels[0]* contains all row information, *invader_pixels[1]* all column.

As outlined in the section describing *process_frame*, the higher the row number, the lower its position on the screen. This function checks to see the maximum value held in *invader_pixels[0]*, this value corresponds to the lowest row in which an *Invader* is present.

The tip of the *Defender* resides in row 185, this is the row from where its bullets originate. As such, no *Invaders* can be killed once they get below this point. The surface of the Earth is on row 195 so there isn't too much of import beyond this depth anyway. With this in mind I defined row 185 as being the lowest row of any consequence that *Invaders* could reach. The highest row in which an *Invader* pixel can be present is row 31.

Once the row in which the current lowest *Invaders* are present is determined I divided its value by 185 and multiplied the result by 100. This product is representative of the percentage of distance descended by the *Invaders*. If the *Invaders* are in row 185 then this function will return 100, signifying that they have completely descended.

The range of values returned from this function were more complex than I had originally envisioned. During my testing process I noticed that the return values did not simply increase toward 100 as the *Invaders* descended. At times, the *Defender* would successfully be able to destroy the entire lowest rows of the *Invader* block, thus making the lowest *Invaders* resident in the upper region of the screen.



```
def check_invader_height(processed_frame):

    invader_value = 0.4
    invader_pixels = np.where(processed_frame == invader_value)
    row = None

    if len(result[0] != 0):
        row = invader_pixels[0].max() # lowest row.
        row = ((row-31)/185) * 100

    if row is None:
        row = 0

    return row
```

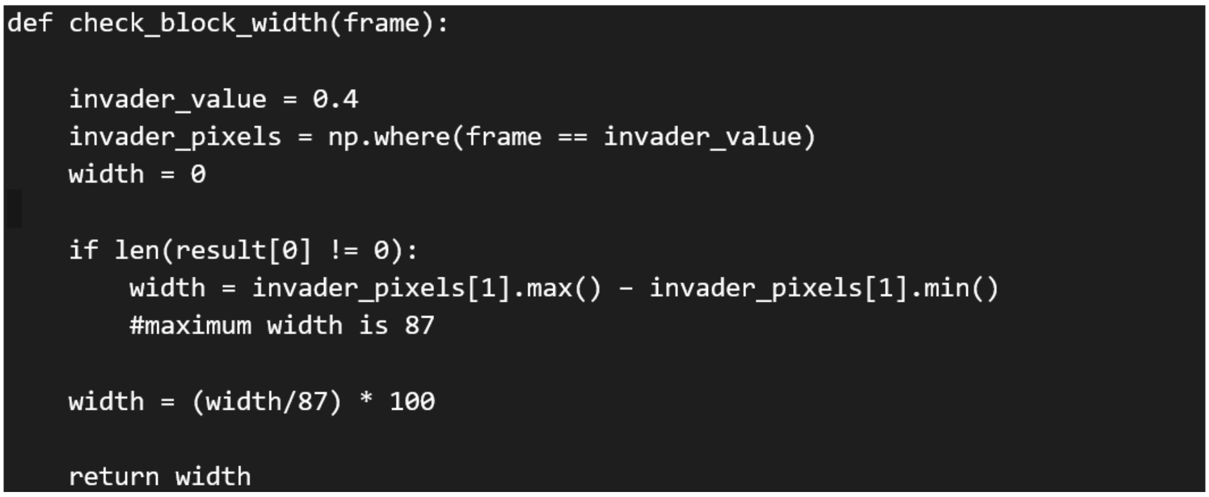
Figure 11 - check_invader_height

5.5.2 – Check Block Width

The second check function I implemented was one that returned a value corresponding to the width of the block of *Invaders*. As a game of Space Invaders progresses, the block of *Invaders* moves from side to side, descending closer to invasion each time it reaches an edge of the screen. This mechanic is an important one and so I felt it appropriate to develop this following check.

The initial lines of this function are similar to those in the function above. An array, *invader_pixels,* is initialised and populated with values corresponding to the position of *Invaders* on screen. In *check_invader_height* I was concerned with the height of the *Invaders* and as such worked with the row information of the pixels (*invader_pixels[0])*. In this function I am concerned with the width of the *Invaders* and so will work with the column information *(invader_pixels[1]).*

The width of the block is determined by subtracting the value of the leftmost column that has *Invader* pixels *(invader_pixels[1].min())* from the rightmost column that has them *(invader_pixels[1].max())*. This width value will never be greater than 87 and so I returned (*width/87)*100* which is representative of the percentage of the maximum width the current block is at. If the block is completely intact then the *width* variable will be 87, the return value will be (87/87)*100 which evaluates to 1*100 or 100.



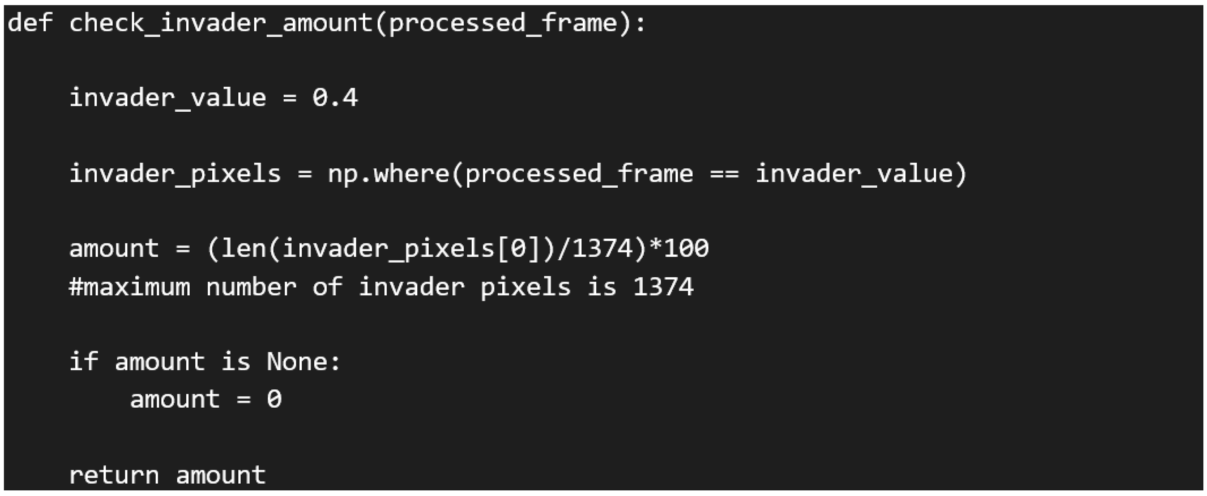
```
def check_block_width(frame):

    invader_value = 0.4
    invader_pixels = np.where(frame == invader_value)
    width = 0

    if len(result[0] != 0):
        width = invader_pixels[1].max() - invader_pixels[1].min()
        #maximum width is 87

    width = (width/87) * 100

    return width
```

Figure 12- check_block_width

5.5.2 – Check Invader Amount

The final check I implemented is independent of the height or width of the *Invader* block. It simply returns a value representative of the percentage of *Invader* pixels currently on screen.

It is simpler in its implementation than the preceding two functions. *invader_pixels* is created and populated as usual. When all *Invaders* are present on screen they occupy a maximum of 1374 pixels, that's 1374 row coordinates and 1374 corresponding column coordinates. With this information in mind it was easy to check the number of *Invader* pixels currently on screen by calculating the length of one of the *invader_pixels* dimensions. This amount was divided by 1374 and multiplied by 100 to return a percentage value, similarly to the other check functions.



```
def check_invader_amount(processed_frame):

    invader_value = 0.4

    invader_pixels = np.where(processed_frame == invader_value)

    amount = (len(invader_pixels[0])/1374)*100
    #maximum number of invader pixels is 1374

    if amount is None:
        amount = 0

    return amount
```

Figure 13 - check_invader_amount

At this stage I had developed three functions that would give similarly ranged, numerical feedback from the game board.

5.6 – To Action!

The next stage in the development process was to consider how I was going to influence the actions taken by the *Defender* via my conditional checks. As described earlier in this report, the Space Invaders game receives input in the form of a discrete range of integer values.

No-op = 0, Fire = 1, Right =2, Left = 3, FireRight = 4, FireLeft = 5

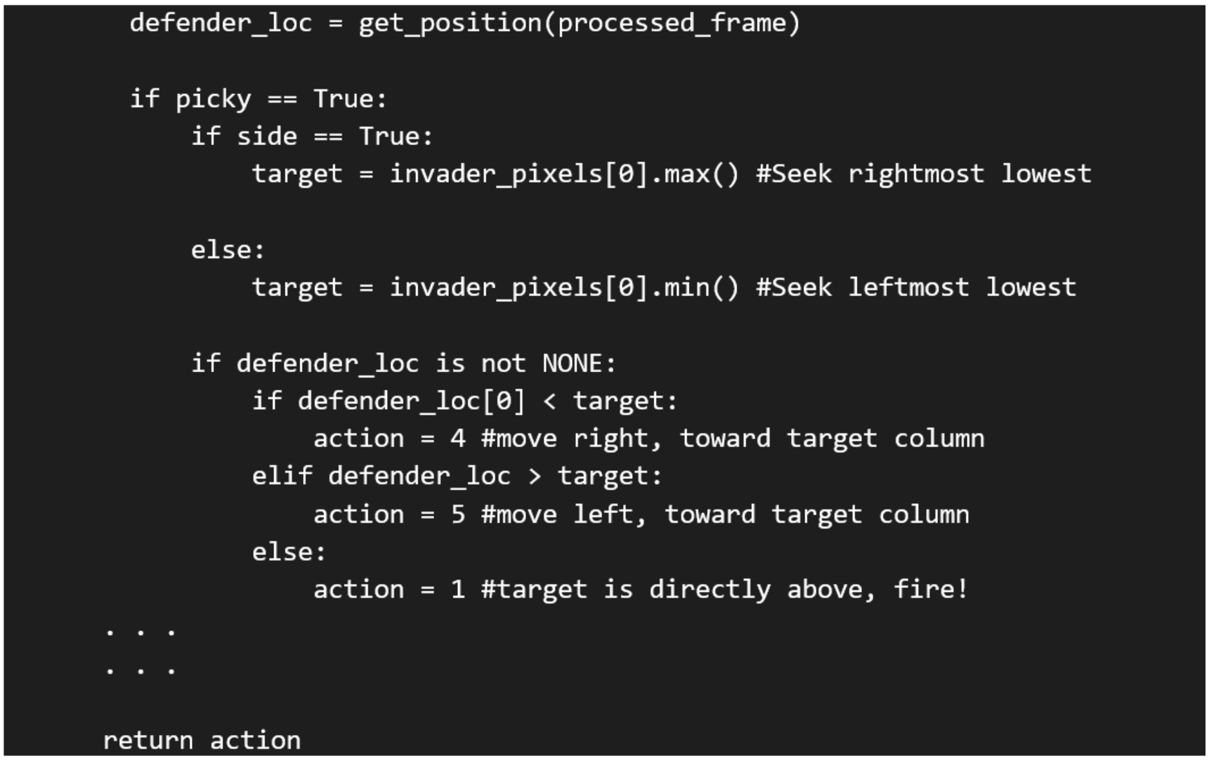
In my prototype approach I specified a grammar that would ultimately evolve a string of these five values that would be fed to the gym emulator on a frame-by-frame, action-by-action basis. With my new approach, that involving a block of conditional checks, I don't think that these actions will be sufficient to from an effective strategy. The commands issued from these actions are atomic, they cannot be decomposed, as such I feel that they are too granular to serve alone in this more general approach to control. To overcome this issue I have developed a number of more generally applicable action functions for the *Defender.* These take advantage of the vison based functions described earlier in this chapter to give the *Defender* the opportunity to react to the changing dynamics of the game screen.

5.6.1 – Seek Lowest Invader

The first of the action functions I implemented was *seek_lowest_invader*. It takes as an argument a *processed_frame.* Similarly to *check_invader_height* this function calculates the lowest row of *Invaders* on screen. It then calls the function *get_position* which returns the current position of the *Defender*. Based on this position, and on some of the other arguments passed to this function, an action is chosen which is returned from the function and passed to the emulator.

This function takes three arguments, a *processed_frame* and two Boolean arguments, *side* and *picky.* Note: a Boolean argument is one that is either True or False. The argument *side* corresponds to which side the *Defender* should target, pass a value of True and it will focus on targeting the lowest *Invaders* on the right, pass a value of False and it will target the lowest on the left. The third and final argument *picky* determines if the *Defender* is picky or not. If a value of False is passed for this argument then the *Defender* is not picky and will simply target the closest, lowest *Invader.* If a value of False is passed for *picky* then the *side* argument is rendered useless.

Below is a code excerpt that highlights a portion of the function.



```
  defender_loc = get_position(processed_frame)

  if picky == True:
      if side == True:
          target = invader_pixels[0].max() #Seek rightmost lowest

      else:
          target = invader_pixels[0].min() #Seek leftmost lowest

      if defender_loc is not NONE:
          if defender_loc[0] < target:
              action = 4 #move right, toward target column
          elif defender_loc > target:
              action = 5 #move left, toward target column
          else:
              action = 1 #target is directly above, fire!
. . .
. . .

return action
```

Figure 14 - seek_lowest_invader (excerpt)

5.6.2 – Seek Side Invader

This function operates in a similar way to *seek_lowest_invader*, it is a little simpler in its execution however. The main difference being that *seek_side_invader* is not concerned with the row value of the *Invader* it is targeting. Without this concern the implementation is lighter. The target in this case is the column on the leftmost or rightmost side of the block of *Invaders*.

The function is passed a *processed_frame,* from which it identifies the location of all *Invader* pixels, and a Boolean value which serves to determine which side of the block the *Defender* will aim for.

As in the above function, *invader_pixels* is a two-dimensional array that represents the row and column values of pixels that make up the *invaders*. The target is determined as either the minimum or maximum value of *invader_pixels[1]*. Minimum corresponds to the leftmost column, maximum the rightmost. This function does not factor in the height of the *Invaders* in these columns, its motivation is to reduce the width of the block of *Invaders*. Doing so extends the amount of time it takes for the block to reach an edge of the screen and drop down, thus prolonging the time until invasion.

5.6.3 – Check Bounding Box

The third action function I implemented is *check_bounding_box*. It is an interesting function that provides the *Defender* with some much needed defensive capabilities and I am quite happy with the results it achieved.

This function is passed three arguments, a *processed_frame* and two integer values representative of the height and width of the bounding box that is to be constructed. Execution begins by getting the current location of the *Defender*. If this location is not *None* then the *Defender* is on screen and the bounding box is constructed. As outlined earlier in this report, the function *get_position* returns an integer value that represents the current location of the very tip of the *Defender* sprite along row 185 of the game screen. It is about this point that the bounding box is constructed.

Location is an integer that corresponds to the position on row 185 of the tip of the *Defender*. This function constructs a bounding box around this point and checks to see if there is an occurrence of a bullet, moving downwards within it. Using the arguments passed and the location value of the *Defender* the bounding box is defined with the following line of code:

b_box = processed_frame[185 - height : 195, location[0] - width : location[0] + (width+1)]

Again, it must be stated that the coordinate system in OpenAI runs contrary to intuition. The higher the number of the row, the lower down on the screen it is. Row 195 is representative of the in-game surface of the Earth, row 185 represents the maximum height of the *Defender*. The image below represents a bounding box of width 4 and height 2. Note: all area directly below the bounding box is also included in the check as I want to avoid the possibility of the *Defender* moving sideways into a low flying bullet.

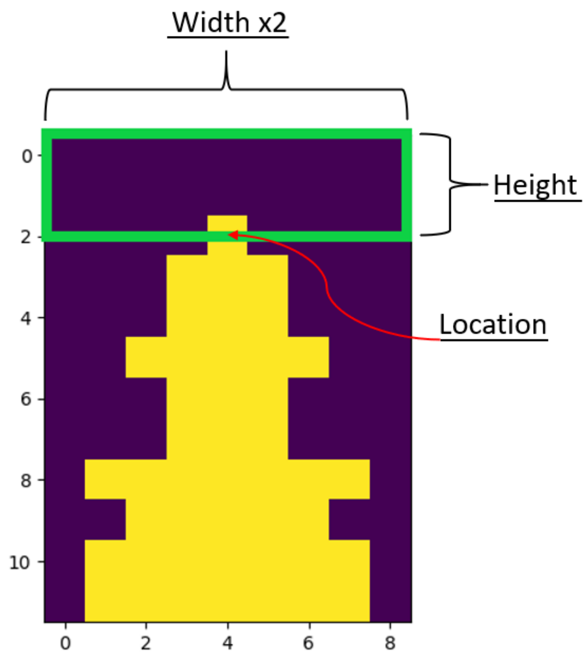
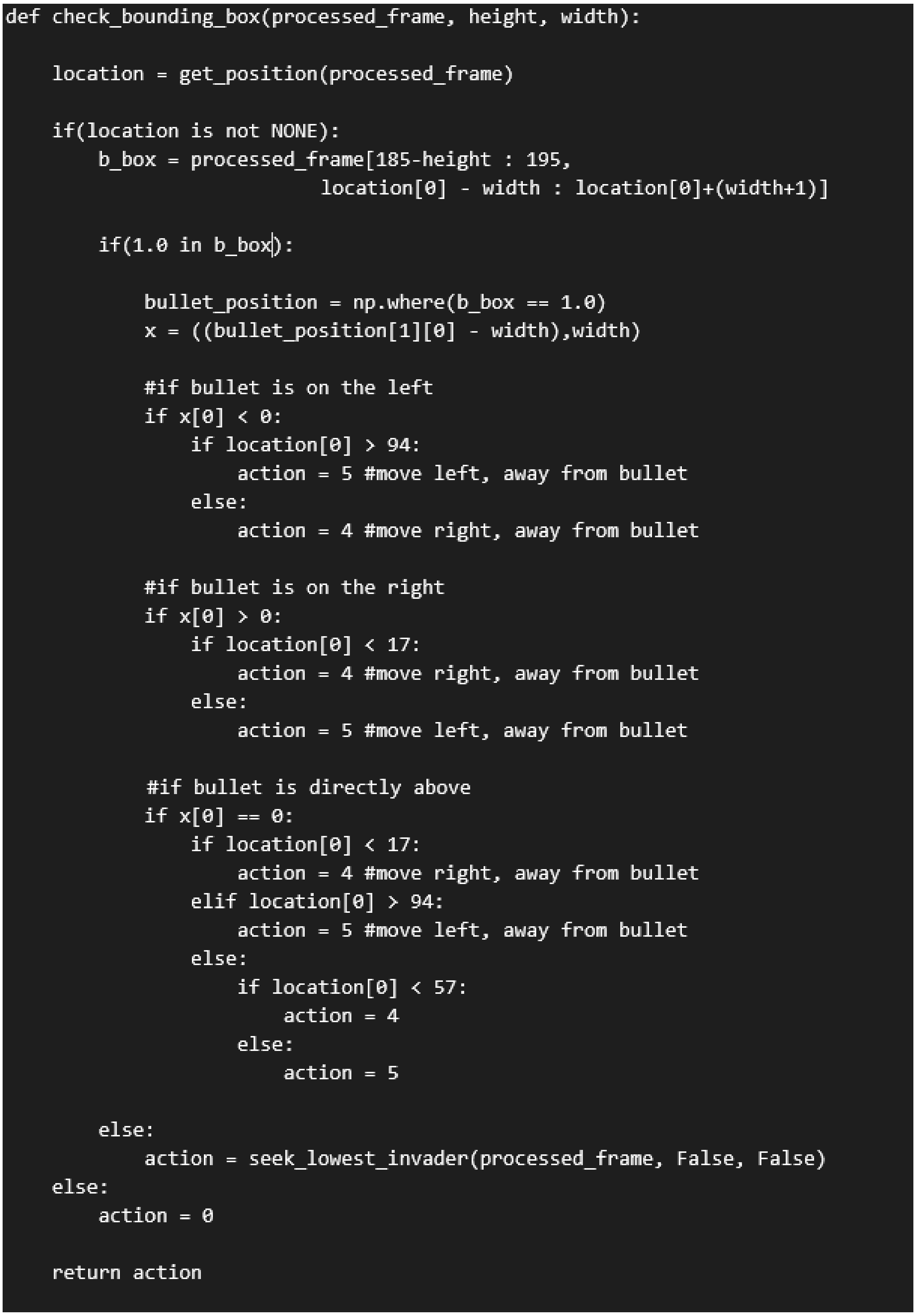


Figure 15 - Bounding Box

Once the bounding box has been established the function checks within it for the presence of pixels with a value matching that of an enemy bullet. If a bullet is found then its location is determined in relation to the *Defender*, the column within the bounding box in which the bullet resides is noted and the difference between this reference and the width of the box is determined. For example, consider the above bounding box, it has a half-width of 4, therefore the box itself is 8 pixels wide. If a bullet is located in column 5 of the box and if we subtract the half-width from this value we get a result that is greater than 1, indicating that the bullet exists to the right of the *Defender*. A negative result indicates a bullet on the left and a result of 0 indicates that the bullet is directly above the *Defender*. Enemy bullets have a straight downward trajectory so once they have been identified in a certain column they will stay within it. If no bullet registers as present within the bounding box the *Defender* will choose to target the closest *Invader*. Some points to note here, the furthest columns to the left and the right that the *Defender* can travel to are 17 and 94 respectively. When its central tip is in column 17 the *Defender* can no longer move left, the same is true in a similar fashion when it its tip resides in column 94. *check_bounding_box* checks to see if the *Defender* is in one of these positions and chances an avoiding move in a less optimal direction if it is the case. The full code for this function is presented on the following page.



```python
def check_bounding_box(processed_frame, height, width):

    location = get_position(processed_frame)

    if(location is not NONE):
        b_box = processed_frame[185-height : 195,
                                location[0] - width : location[0]+(width+1)]

        if(1.0 in b_box):

            bullet_position = np.where(b_box == 1.0)
            x = ((bullet_position[1][0] - width),width)

            #if bullet is on the left
            if x[0] < 0:
                if location[0] > 94:
                    action = 5 #move left, away from bullet
                else:
                    action = 4 #move right, away from bullet

            #if bullet is on the right
            if x[0] > 0:
                if location[0] < 17:
                    action = 4 #move right, away from bullet
                else:
                    action = 5 #move left, away from bullet

            #if bullet is directly above
            if x[0] == 0:
                if location[0] < 17:
                    action = 4 #move right, away from bullet
                elif location[0] > 94:
                    action = 5 #move left, away from bullet
                else:
                    if location[0] < 57:
                        action = 4
                    else:
                        action = 5

        else:
            action = seek_lowest_invader(processed_frame, False, False)
    else:
        action = 0

    return action
```

Figure 16 - check_bounding_box

5.7 – Other Bits and Pieces

As stated in the beginning of this chapter, my development aims for this project diverged somewhat upon reading the aforementioned Galván-López *et al.* paper. My initial development approach (Type 1), which closely resembles the prototype development, is similar enough to my newly proposed approach (Type 2), which involves the evolution of an executable block of conditional statements, that many of the functions outlined already in this chapter can be successfully utilised by it. However, there is significant difference between the approaches to warrant some specific modifications.

The Type 1 approach is limited in its effectiveness to deterministic playthroughs of Space Invaders. Consider a typical training session for the Type 1 approach.

- A random population of say 100 individuals is initialised.
- Each individual is evaluated on **the same** rollout of Space Invaders. By this I mean that the *Invaders* make exactly the same moves and fire at exactly the same frame, from the same position every run.
- The fittest individuals are selected, crossed over and mutated to form the basis of the next generation.
- The proceeding generations continue to search and be evaluated against the same one rollout of Space Invaders.

The nature of this process allows the algorithm to search the determined playthrough of Space Invaders in great detail. Due of the specific nature of the solutions found this way I was able to make some functions that modified the Type 1 fitness evaluation function that would motivate the individuals to perform better.

5.7.1 – Defender Life Tracker and Penalty Application

In the prototype development there was no way, when evaluating a run, for the fitness function to determine if a life had been lost, how many lives are currently remaining or even if a run has ended before the *Defender* used all of its lives. These omissions lead to some imperfections in the application of fitness reward. One individual may have progressed its solution to achieve a high score, removing all of the *Invaders* on the first screen and going so far as to make it to the second screen. If this individual has lost two of its three lives in the first level then it will be under-equipped to make any significant dent in the fresh new block of *Invaders.* A second individual may have also been successful, may have achieved the same points in the first round and may also find itself at the second level. However, this individual did not lose any of its lives in the first round. Objectively, even if both of these individuals fail to score any points in the second round and both end with the same point total, the second individual should be classed as fitter as it is better positioned to evolve effectively.

In Space Invaders the player has three lives, every time the *Defender* is hit by an *Invader* bullet it loses a life, if all three lives are lost then the game is over. The only other way the game can end is if the *Invaders* manage to make it to the Earth's surface, in this scenario it is instant game over, regardless of the lives remaining. When a player life is lost the *Defender* explodes, flashes for a couple of frames and then is returned to its starting position. A number is displayed on screen at these times to indicate the count of lives remaining. These numbers are displayed in the exact same region of the screen every time. The count of lives remaining is also displayed at the start of a game screen.

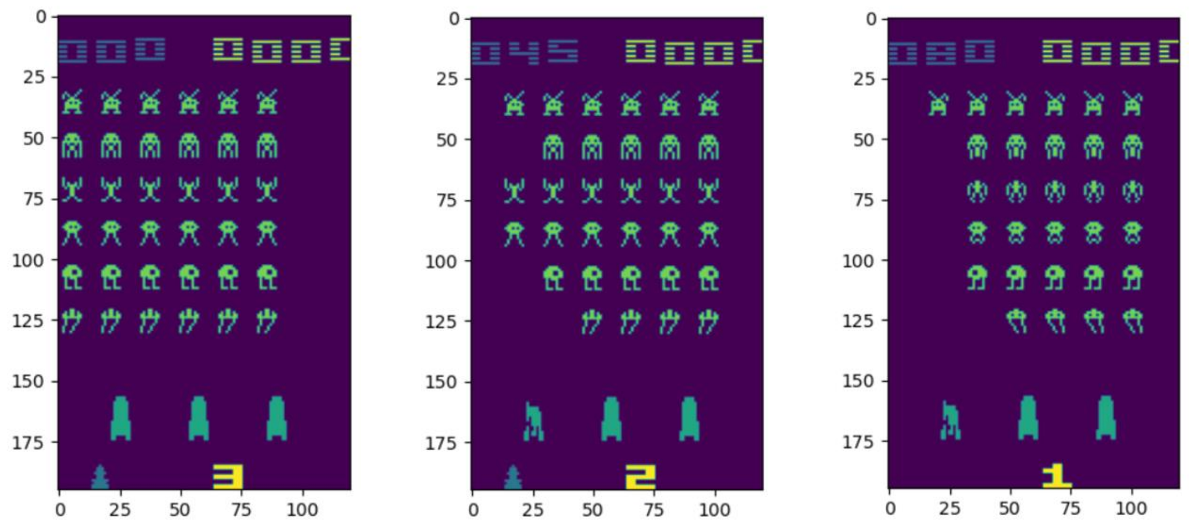


Figure 17 - Three Space Invaders life screens

I implemented a function that would look to this area of the screen and be able to recognise when a life had been lost. I had to differentiate between when a life is lost and when the *Defender* is at the start of a fresh game screen. I applied a fitness penalty relative to the current fitness at the time of life loss. An individual that loses a life while on a low score will receive a harsher punishment than one that loses a life while on a higher score. This penalty seeks to promote conservation of lives amongst the individuals and had an immediate impact on general performance. Some of the code for this method is outlined below. Firstly a copy of the current life number being displayed is made by taking a slice out of the *processed_frame*. This slice is checked for the presence of pixels that represent the life numbers. If it finds one of these then the function knows that a life number is on screen. The various slices of pixels can be seen below.

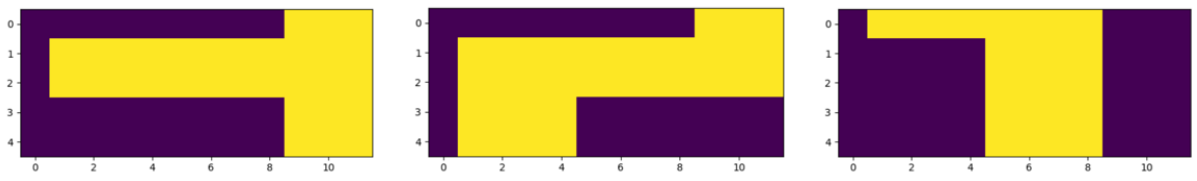
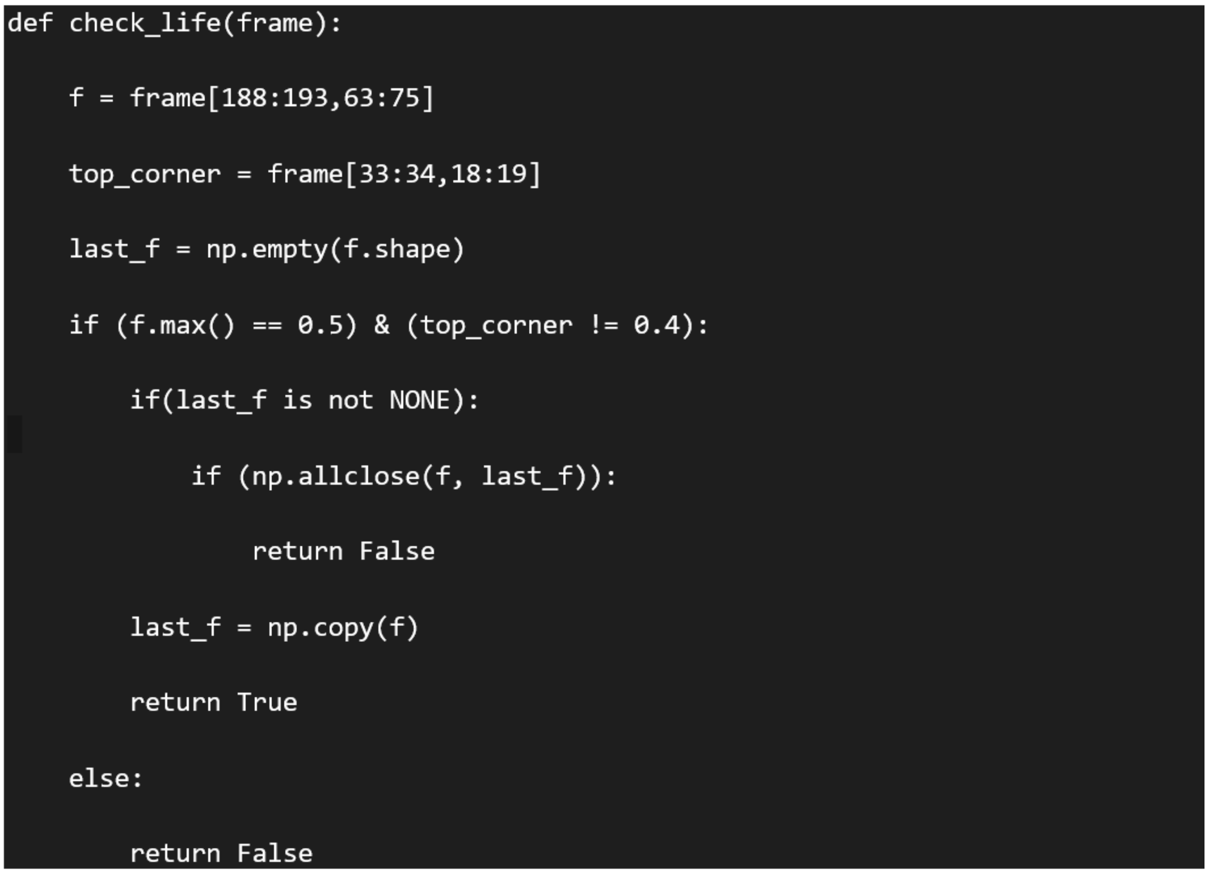


Figure 18 - Life Number segments

This method only returns true when the first and second lives have been lost. At the start of every level, including the preliminary one, a number will flash on screen that indicates the amount of lives the currently held. I enabled the function to differentiate between these displays and the displays that occur when a life is lost by using the array *last_f*. This array is created as a copy of the previous life number that triggered the penalty. If *last_f* is representative of 2, then the *Defender* has lost a life. If the *Defender* proceeds to the second level it will begin by flashing the number 2 on screen to indicate the number of lives currently held. The function will not return True as the current slice matches *last_f.* When the *Defender* (inevitably) loses its next life the current slice with represent the number 1, this will not match the currently stored *last_f* and so the function returns True.



```
def check_life(frame):

    f = frame[188:193,63:75]

    top_corner = frame[33:34,18:19]

    last_f = np.empty(f.shape)

    if (f.max() == 0.5) & (top_corner != 0.4):

        if(last_f is not NONE):

            if (np.allclose(f, last_f)):

                return False

        last_f = np.copy(f)

        return True

    else:

        return False
```

Figure 19 - check_life

check_life is called every frame, from within the fitness evaluation function. If it returns True then a penalty is applied to the in the form of a negative reward, its amount determined by the current fitness level.

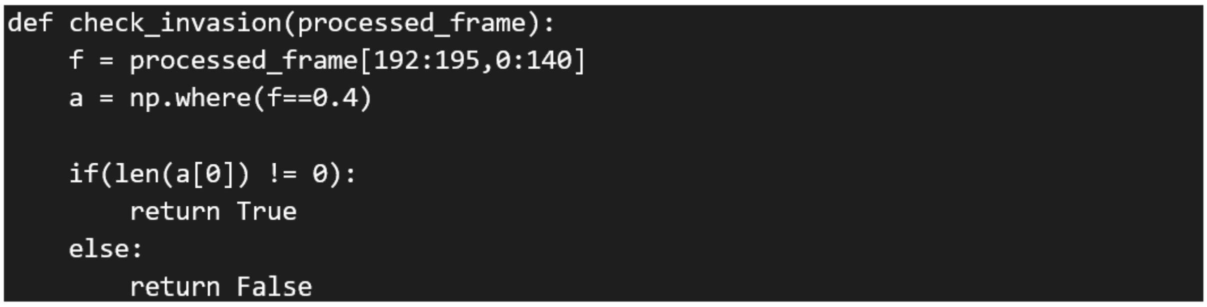
5.7.2 – Check Invasion

I have mentioned an end game scenario in this report one or twice that I have deemed as potentially the worst outcome for a Type 1 individual. This is the scenario when an *Invader* manages to successfully descend to Earth level, completing its objective and potentially dooming the inhabitants of our planet.

This particular scenario usually arises when a *Defender* concentrates its fire on the centre of the block of *Invaders*. An individual will often rack up a decently high score by playing this way but it often leads to many *Invaders* remaining in the furthest left and the right columns. The *Defender* finds itself in a quandary in these situations and is inevitably unable to destroy both extreme columns before an *Invader* touches down, ending the game.

I found that this scenario began to play out more often after I had introduced the *life_check* as outlined above. It seemed that the individuals were "*learning*" to avoid the life penalty by letting the game end like this. Often times an individual would concentrate its fire centrally, clear a space above it and then survive until an *Invader* reached ground level. I write "*learn*" as such because in reality, it was actually a matter of selection. Individuals in early generations of a training run tended to receive much harsher *life_check* penalties. As such a mediocrely performing individual that was free from *life_check* penalties quickly rose in terms of fitness and dominated selection, leading the march of evolution into a dead end.

The function *check_invasion* is actually quite lightweight in its implementation as can be seen below. It begins by taking a slice of the passed *processed_frame*. This slice represents a thin strip, three pixels wide that runs across the screen at the in-game ground level. *check_invasion* is called every frame from the *fitness_eval* function. If there is an occurrence of an *Invader* in the sliced pixel strip then an invasion is underway and the function returns True, otherwise it returns False.



```
def check_invasion(processed_frame):
    f = processed_frame[192:195,0:140]
    a = np.where(f==0.4)

    if(len(a[0]) != 0):
        return True
    else:
        return False
```

Figure 20 - check_invasion

Upon a return of True a penalty is applied to the overall fitness for that individual. This penalty is determined in proportion to the *life_check* penalty as both penalties use the common variable *penalty_amount.*

5.7.3 – Effective Crossover

During the evaluation stage of my prototype development I got the impression that the individual genomes were potentially not being crossed over correctly. Due to the continuous nature of the phenotype expressions that are evolved by my Type 1 implementation I was forced to initialise my population with extremely long individuals, usually in the 50,000 to 75,000 bits long range.

I concluded that in general, a fitter individual for Space Invaders will be one that is active long enough to accumulate a relatively high score. By this logic, one can assume that a fitter individual will utilise more of its genomic sequence than a less fit one. The problem that I believed I was encountering was this: less fit individuals, in the early generations of a run were not being diversified by crossover due to the relatively short length of their effective area. Consider the following figure:

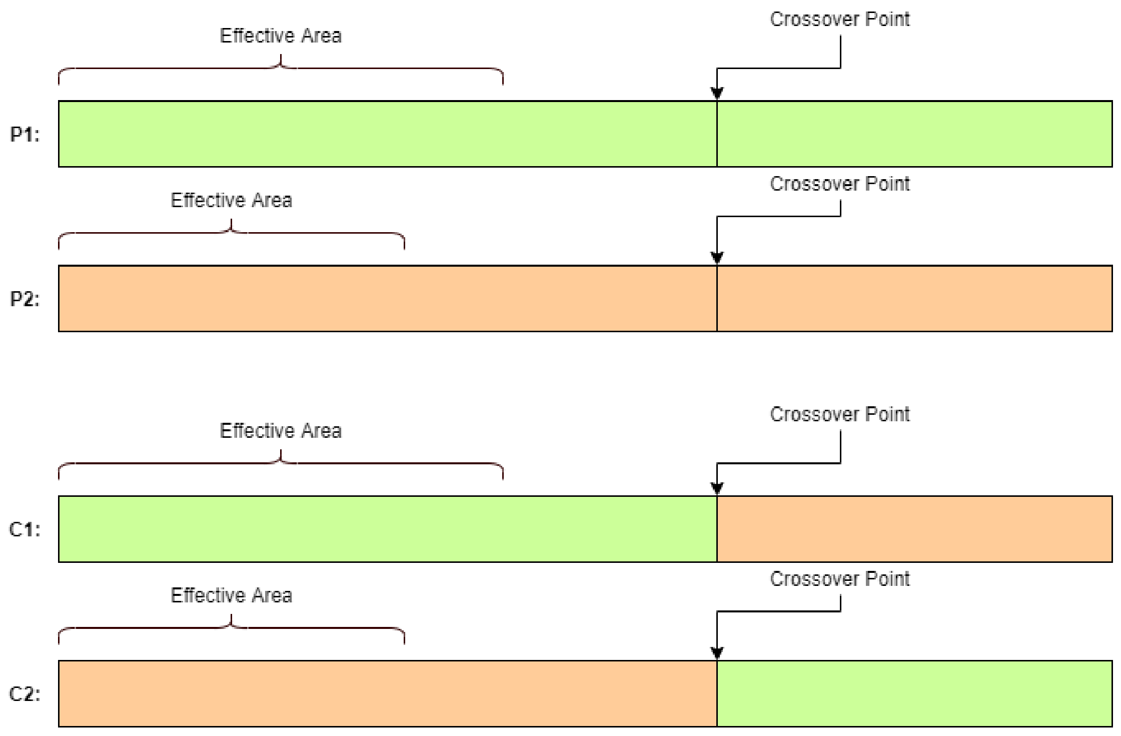


Figure 21 - 1 Point Crossover

The "Effective Area" as displayed above corresponds to the portion of the genome that gets executed before the game ends. It is the only portion of the genome that has contributed to the individual's fitness evaluation. If the crossover point is chosen to be at a point in the individual that is outside the effective area then the crossover procedure will have no impact on either individual. This is because the series of commands that are carried out in the effective area will be unchanged, they will be unchanged and thus will reach to the same point in the game, whereupon the *Defender* loses all of its lives, and achieve the same fitness score.

To counteract this issue I implemented the functions cx_one_point_effective and cx_two_point_effective to only choose crossover points that reside within the effective area.

The below figure illustrates this process:

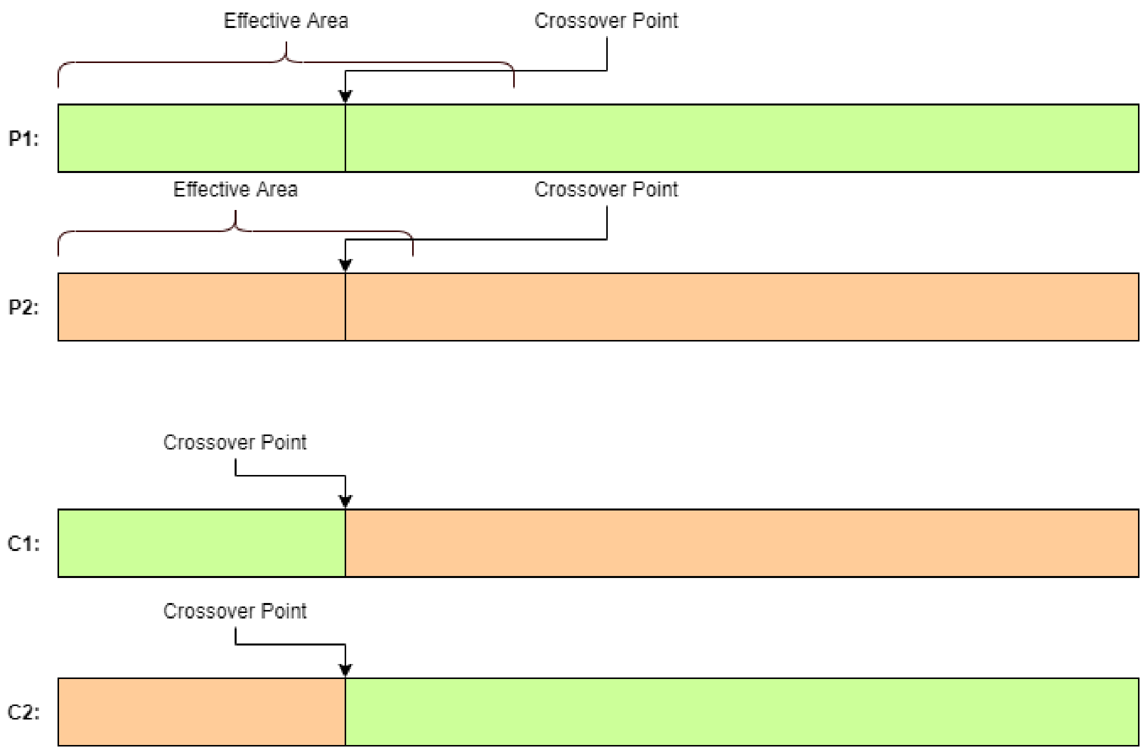


Figure 22 - Effective 1 Point Crossover

Because the position of the crossover point was restricted to be within the effective area, the children produced are unevaluated in terms of fitness and could very well be the fittest individuals in the population. At the very least, diversity is promoted. The above figure represents an ideal situation, where the crossover point resided within the effective area of both parents, this is not always guaranteed to be the case, but it is much more likely than it would be with the native D.E.A.P. crossover functions.

Chapter 6 – Execution (Let Them Eat Cake!)

In this chapter I will detail my execution procedures for both types of implementation I have developed. The primary differences between these implementations lie in their grammar specifications and fitness evaluation functions, both of these areas will be described in this chapter. I will outline my training procedure and present statistical feedback on the performance of the various experimental runs I have carried out.

6.1 – Type 1

This part of the chapter will focus on the Type 1 implementation. This implementation closely resembles my prototype development in its application. The advancements in functionality I have made since the prototype have been outlined in the previous chapter.

The genome to phenome mapping process for this particular implementation remains unchanged from that which was outlined in chapter 4 of this report, with that in mind I do not feel it necessary to revisit it now in any great detail. The main point I would like you to consider at this stage is that of the state of the mapped phenotype expression.

I have explained how the Space Invaders emulator receives action instructions in the form of one of five discrete commands.

$$No\text{-}op = 0,\ Fire = 1,\ Right = 2,\ Left = 3,\ FireRight = 4,\ FireLeft = 5$$

Regardless of implementation, approach or even machine learning style, these five digits are the extent of legal input. As such, any evolved expression will ultimately translate into a series of choices from the above set. I outline the most successful grammar for my Type 1 implementation below. This BNF specification does not result in a programming language based executable expression but rather a string of integer digits that I parse in the *fitness_eval* function to either directly assign a gameplay action or call an action function which will, in turn, return a choice of action.

Over the course of fine tuning my Type 1 implementation I carried out multiple training runs in order to be able to methodically approach the task of hyper parameter tuning and evaluation of my individual hand-coded functions. My process was as follows:

- Initiate multiple training runs with each parameter/grammar/logic set to determine an average performance evaluation across different random seeds.
- Training schedule begins with the least complex, or vanilla, strategies. These runs did not include any of the action methods such as *seek_lowest_invader*
- Run the training course, note the parameters, the grammar and the performance feedback.
- Add new parameters or update the grammar accordingly.

- Run the training course with new settings and note everything.
- Compare training feedback to determine next parameters/functions to add/remove/change.
- Rinse/repeat.

At the outset of my training program I defined a list of parameters and functions that I intended to introduce over the course of the process in an attempt to find the optimal combination for search. I intended to introduce a new parameter change or function with each training run and to note the positive or negative impact they had on the maximum score achieved. The list is as follows:

- Population Size: 50, 100, 150
- Crossover: 1 Point, 1 Point Effective, 2 Point Effective
- Function: *check_bounding_box*
- Function: *seek_lowest_invader*
- Function: *seek_side_invader*
- Fitness Penalty: *life_penalty, invasion_penalty*
- Generations: 10, 15, 25

Each training session carried out consisted of three differently seeded runs. A steady progression in average max score can be observed from the below collection of tables. With Training Run 6 displaying the most effective feature and parameter combination.

Training Run 1:

Population:	50
Generations	10
Crossover Type:	1 Point Crossover
Functions:	None
Fitness Penalties:	None
Run 1 Max:	545
Run 2 Max:	665
Run 3 Max:	765
Average Max Score:	**658.33**

Training Run 2:

Population:	100
Generations:	10
Crossover Type:	1 Point Crossover
Functions:	None
Fitness Penalties:	None
Run 1 Max:	900
Run 2 Max:	625
Run 3 Max:	825
Average Max Score:	**783.33**

Training Run 3:

Population:	100
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	None
Fitness Penalties:	None
Run 1 Max:	1080
Run 2 Max:	1125
Run 3 Max:	725
Average Max Score:	**976.66**

Training Run 4:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	None
Fitness Penalties:	None
Run 1 Max:	1170
Run 2 Max:	930
Run 3 Max:	1120
Average Max Score:	**1073.33**

Training Run 5:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	*check_bounding_box*
Fitness Penalties:	None
Run 1 Max:	1415
Run 2 Max:	970
Run 3 Max:	1075
Average Max Score:	**1154.33**

Training Run 6:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	*check_bounding_box* *seek_lowest_invader*
Fitness Penalties:	None
Run 1 Max:	1400
Run 2 Max:	1275
Run 3 Max:	1250
Average Max Score:	**1308.33**

Training Run 7:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	*check_bounding_box* *seek_side_invader*
Fitness Penalties:	None
Run 1 Max:	940
Run 2 Max:	865
Run 3 Max:	975
Average Max Score:	**926.66**

Training Run 8:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	*check_bounding_box* *seek_lowest_invader* *seek_side_invader*
Fitness Penalties:	None
Run 1 Max:	1000
Run 2 Max:	1000
Run 3 Max:	820
Average Max Score:	**973.33**

Training Run 9:

Population:	150
Generations:	10
Crossover Type:	2 Point Crossover Effective
Functions:	*check_bounding_box* *seek_lowest_invader*
Fitness Penalties:	None
Run 1 Max:	1320
Run 2 Max:	1370
Run 3 Max:	945
Average Max Score:	**1211.66**

Training Run 10:

Population:	150
Generations:	10
Crossover Type:	1 Point Crossover Effective
Functions:	*check_bounding_box* *seek_lowest_invader*
Fitness Penalties:	*life_penalty* *invasion_penalty*
Run 1 Max:	1375
Run 2 Max:	955
Run 3 Max:	1365
Average Max Score:	**1231.667**

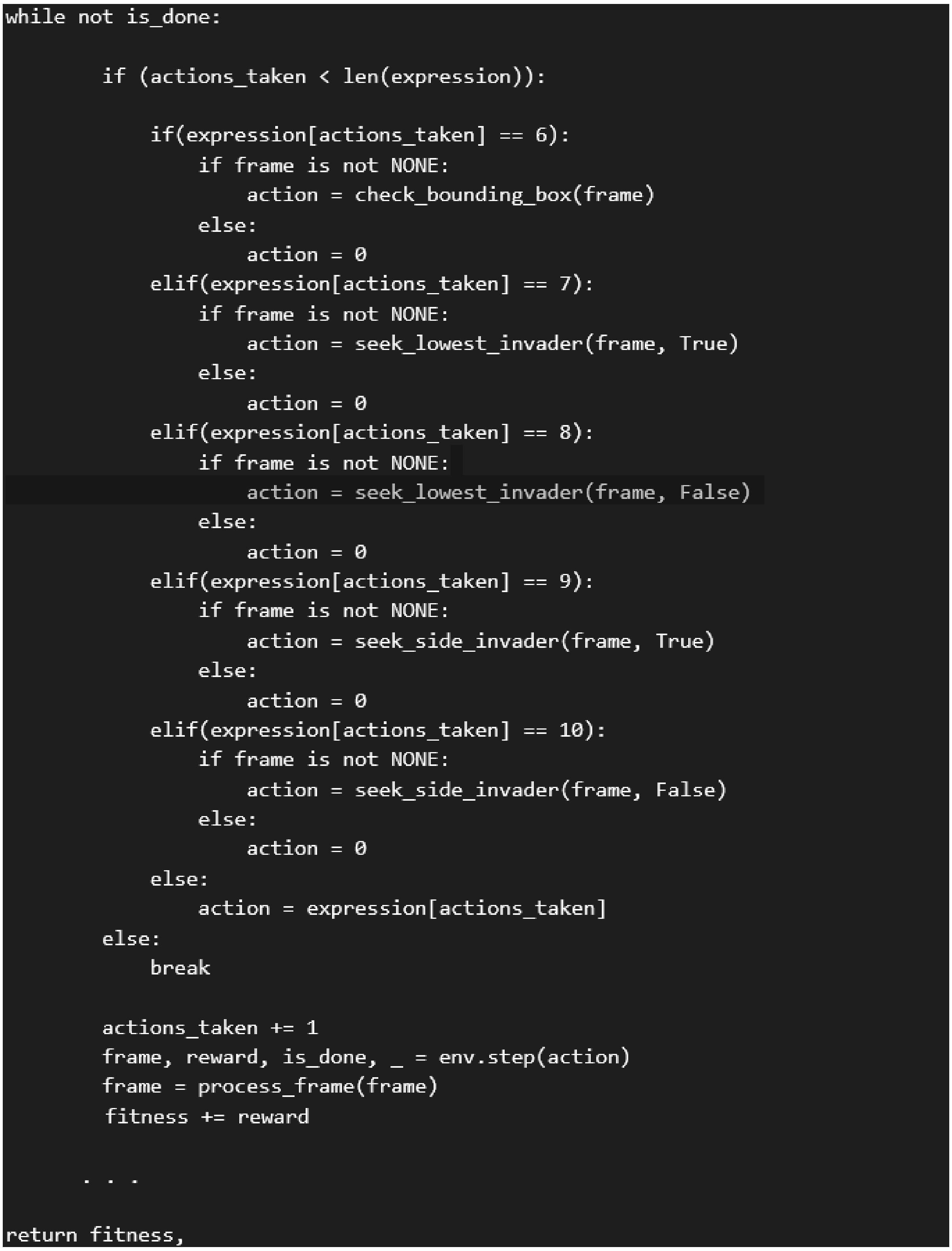
6.1.1 – Type 1 Grammar Specification

The following grammar is the ultimate iteration of a lengthy tuning process. Through testing it became clear that the function *seek_side_invader* was not contributing positively to the overall performance of the agent so it was removed. The mapping process for a grammar such as the one below has been covered in some detail in chapter 4 of this report so I shall not revisit it again. I have added two extra terminal values to this grammar, <check> which resolves to 6 and <find_lowest> which resolves to 8. These numbers, when parsed in the fitness function will call *check_bounding_box* and *seek_lowest_invader* respectively.

```
--------------------------------------------------------------------------------------------------------------------

S               ::= <loop>

<loop>          ::= <play> <loop>

<play>          ::= <move> | <shoot> | <wait> | <scout> | <seek1>

<wait>          ::= <pause> <repeat>

<scout>         ::= <check> <repeat>

<seek1>         ::= <find_lowest> <repeat>

<shoot>         ::= <fire> <repeat>

<move>          ::= <direction> <repeat>

<direction>     ::= <left> | <right>

<left>          ::= 3 | 5

<right>         ::= 2 | 4

<fire>          ::= 1

<pause>         ::= 0

<check>         ::= 6

<find_lowest> ::= 8

<repeat>        ::= -1- | -2- | -3- | -4- | -5- | -6-

--------------------------------------------------------------------------------------------------------------------
```

6.1.2 – Type 1 Fitness Evaluation

The only major changes that were made to the Type 1 fitness function come in the form of additional entries to the main conditional block. These additional entries correspond to calls to the hand-coded action functions described in the previous chapter.



```
while not is_done:

        if (actions_taken < len(expression)):

            if(expression[actions_taken] == 6):
                if frame is not NONE:
                    action = check_bounding_box(frame)
                else:
                    action = 0
            elif(expression[actions_taken] == 7):
                if frame is not NONE:
                    action = seek_lowest_invader(frame, True)
                else:
                    action = 0
            elif(expression[actions_taken] == 8):
                if frame is not NONE:
                    action = seek_lowest_invader(frame, False)
                else:
                    action = 0
            elif(expression[actions_taken] == 9):
                if frame is not NONE:
                    action = seek_side_invader(frame, True)
                else:
                    action = 0
            elif(expression[actions_taken] == 10):
                if frame is not NONE:
                    action = seek_side_invader(frame, False)
                else:
                    action = 0
            else:
                action = expression[actions_taken]
        else:
            break

        actions_taken += 1
        frame, reward, is_done, _ = env.step(action)
        frame = process_frame(frame)
        fitness += reward

      . . .

return fitness,
```

Figure 23 - Type 1 fitness_eval (excerpt)

6.1.3 – Type 1 Demonstration Run

Based on the feedback observed in the multiple training runs I deemed the best parameter set to be the following:

- Population Size: 150
- Crossover: 1 Point Effective
- Function: *check_bounding_box*
- Function: *seek_lowest_invaders*
- Fitness Penalty: *None*
- Generations: 25

I initiated a demonstration run with the above settings included as well as the following hyper parameters:

Hyper Parameters:

- Game: SpaceInvadersDeterminstic-v4
- Grammar: space_invaders_grammar_type_1.pybnf
- Seed: 809
- Number of Runs: 1
- Individual Length: 75000
- Crossover Rate: 0.8
- Mutation Rate: 0.03

Runtime: 7:29:56.526319

MaxFitness Value: 2020

This run performed quite well as can be observed from the following table and plot. It looks like it could have continued to grind out points for a number of generations yet as it did not seem to plateau in its 25 generation training run. It does show some signs of convergence due to the ever decreasing gap between the maximum and average scores achieved. At nearly 7 hours and 30 minutes execution time however my central processing unit was happy it ended when it did.

The following links to a Youtube video playback of the best performing individual from this training run: https://youtu.be/KUfBZr0584Q

Generation	Num Evals	Max	Min	Avg	Std. Dev
0	150	855	15	234.367	158.52
1	115	865	60	386.033	169.386
2	122	1055	30	530.767	192.546
3	128	1055	55	663.033	244.238
4	115	1080	60	852.167	145.998
5	128	1080	75	929.8	140.381
6	118	1365	110	987.267	178.899
7	120	1365	180	1039.7	142.126
8	132	1365	95	1031.6	217.115
9	128	1365	105	1011.73	226.368
10	122	1365	55	990.667	285.609
11	123	1365	45	1070.27	306.855
12	123	1365	75	1230	274.445
13	120	1630	35	1355.1	111.022
14	128	1630	60	1328.3	230.258
15	122	1630	50	1331.47	268.015
16	115	1630	120	1424.83	216.816
17	128	1660	45	1538.67	274.896
18	129	1870	95	1586.27	251.276
19	124	1870	35	1568.4	320.258
20	110	1870	85	1658.67	233.615
21	105	1965	90	1720.53	330.296
22	120	1965	50	1834.6	237.292
23	122	1965	30	1823.5	303.857
24	115	2020	140	1822.17	370.858
25	129	2020	165	1937.6	200.682

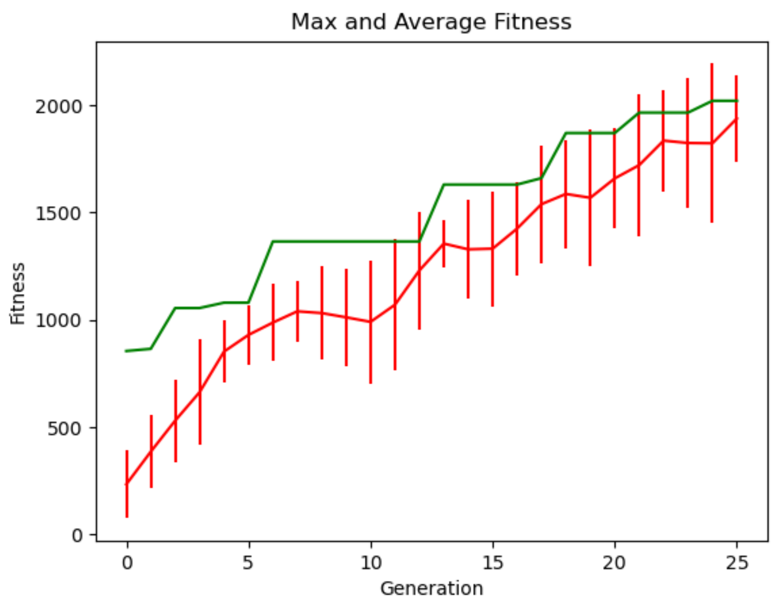


Figure 24 - Type 1, 25 Generations

6.2 – Type 2

Type 2, in my opinion, is a much more *"on brand"* approach to GE than Type 1. As described, the Type 2 approach centres around evolving a syntactically correct and executable block of Python code. This block of code serves as a general controller for Space Invaders. This is different from the Type 1 approach in that, with Type 1, the phenotype expressions that are ultimately evolved are essentially long sequences of commands to be carried out at each frame of a playthrough. Type 2 phenotypes however are reactive, they can dictate which move to make based on observation of the game board and its current state. This reactive quality allows the Type 2 solution to be applied in a more general sense to Space Invaders.

The blocks of code to be evolved are composed of *if, if-else* and *else* statements. These conditional statements can be structured together with appropriate conditional checks to form potentially detailed strategic approaches to game play. For example, the logic of one such controller might run as such.

If there are less than half of the invaders left on screen then: focus fire on the lowest invaders.

Otherwise: focus on the side invaders.

In reality the blocks of code evolved are more complex than the example described above, the conditional statements often forming nested chains with many different scenarios being checked for.

Using Python to develop and evaluate these phenotypes presented one major benefit and one inconvenient drawback. The major benefit arose in the form of the life-savingly useful *exec* function. *exec*, as described in the Python documentation, "*supports dynamic execution of Python code*". This function can be passed a Python string object, and provided that the string is syntactically correct it will be executed within the currently running program. If I'm honest this discovery blew my mind and was a firm reminder of the fantastic utility of the Python language. The one caveat in the above function description was at the core of the inconvenient drawback I have just mentioned. The string passed to *exec* must be syntactically correct to be successfully executed. Python may take a laissez faire approach to object typing and general punctuation, but it is uncompromising in its policing of code indentation. This proved to be a rather substantial hurdle for me in my Type 2 development arc. My intention was to specify a grammar that would evolve an executable block of nested conditional Python code. With every interior *if* statement that was added to the phenotype and new level of indentation would be required, these levels of indentation would then need to be tracked and appropriately reversed as the partnered *else* statements were stated. Through trial and error, and with a little ugly "*hacking*" I managed to get the indentation of the phenotypes under control.

I will now outline the specifics of the grammar I used for Type 2, before proceeding to explain the enhancements required by the mapping function in order to map binary genotypes to tree-structured executable phenotypes. I will then briefly highlight some of the differences between the Type 1 and Type 2 fitness functions before giving a training process overview.

6.2.1 – Type 2 Grammar Specification

```
-------------------------------------------------------------------------------------------------------------------

S                     ::= <setup><main>

<setup>               ::= <newline> bound_box_height =  <bound_box_height>
                          <newline> bound_box_width  = <bound_box_width><newline>

<main>                ::= if (<condition>): <statements> <newline> <reset_indent>
                          else: <statements>

<statements>          ::= <newline><indent><ifs> |
                            <newline><indent><push_tree><ifs>
                            <newline><pop_tree><match_tree><elses>

<ifs>                 ::= if (<condition>): <newline> <action> |
                           if (<condition>): <statements>

<elses>               ::= else: <newline> <action> |  else: <statements>

<condition>           ::= <check> <comparison> <nums>

<check>               ::= check_invader_amount(frame) |
                          check_invader_height(frame) |
                          check_block_width(frame)

<action>              ::= <indent><seek_lowest><unindent> |
                          <indent><seek_sides><unindent> |
                          <indent><check_b_box><unindent> |
                          <indent><camp><unindent>

<seek_lowest>         ::= set_action(seek_lowest_invader(frame, <boolean>, <boolean>))

<seek_sides>          ::= set_action(seek_side_invader(frame, <boolean>))

<check_b_box>         ::= set_action(check_bounding_box(frame, bound_box_height,
                                                               bound_box_width))

<camp>                ::= set_action(camp( ))

<bound_box_height>    ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

<bound_box_width>     ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

<nums>                       ::= 10 | 25 | 50 | 75

<boolean>             ::= True | False

<comparison>          ::= -le- | -ge-
```

<newline>	::= -NL-
<indent>	::= -I-
<unindent>	::= -X-
<reset_indent>	::= $
<match_tree>	::= -M-
<pop_tree>	::= -PP-
<push_tree>	::= -PH-

The grammar displayed above is clearly more complex than the Type 1 grammar shown earlier in this chapter. All expressions in this grammar begin with the non-terminal member <setup>. This non-terminal sees to the random initialisation of the variables *bounding_box_height* and *bounding_box_width*. Following these initialisations the <main> non-terminal is parsed. This non-terminal element forms the overall structure of any evolved phenotype. At a macro level every phenotype will be of the structure:

if (<condition>): <statements> else: <statements>

The non-terminal <statements> serves to form the bulk of most evolved phenotypes, technically this element can grow infinitely due to its recursive nature. It is composed of standalone <ifs> and also <ifs> followed by <elses>. Each of these three non-terminal elements include the <statements> non-terminal and so if left unchecked this grammar could evolve infinitely nested expressions. I handled this by implementing functionality to track and cap the depth of the current tree being evolved.

With every consecutive *if* statement added to the expression the depth of the tree increases by one. This depth is decreased after a corresponding *else* statement is parsed. Any action statements (*seek_lowest_invader, check_bounding_box, seek_side_invader)* added to the expression can be considered as the leaves of the tree. Keeping an accurate track of the tree depth as well as which *else* statement was paired with which *if* statement proved to be a trying task!

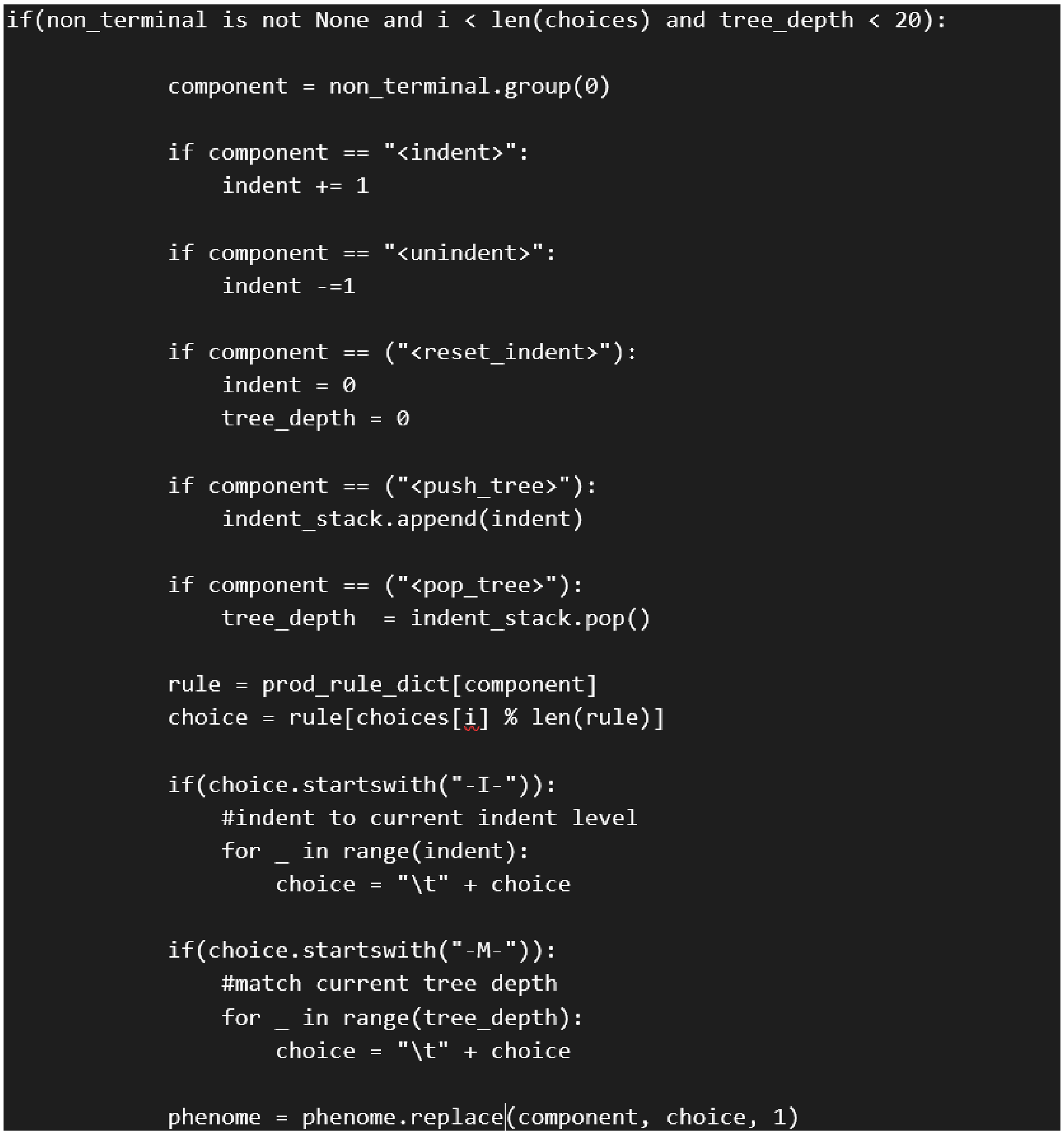
You might notice, within the Type 2 grammar, the existence of an ugly proliferation of non-terminals such as <newline>, <indent>, <push_tree>, <pop_tree> and <reset_indent>. These were, I'm afraid, a necessary evil. The non-terminals in question are translated to their respective in-phenome symbols which in turn served great utility during the revised genome to phenome mapping process. They allowed me to keep track of various indentation rules and ultimately enabled the entire Type 2 approach.

6.2.3 – Type 2 Phenome to Genome Mapper

The primary difference between the Type 1 and Type 2 mappers is illustrated in the below code snippet. It is the case that non-terminal elements will find their place in an expression long before they are fully decomposed into terminal elements. It is at their initial placing in the expression that I chose to input the required indentation information. An indentation can be represented in a Python string with the following characters "\t". At the occurrence of the <indent> non-terminal, the global indent level is incremented by 1. If you look a little further down the code you can see that on the occurrence of the string "`-I-`", a number of indents, equal to the global level is added before the current non-terminal symbol. This works similarly with the <unindent> non-terminal. Tree depth is monitored throughout the expression by the *indent_stack*. <push_tree> and <pop_tree> serve to keep this stack in order and reflective of the current tree depth. When an *else* statement is being stated it will be preceded by the <match_indent> non-terminal, this element leads to the terminal string "`-M-`". When this string appears in the mapper the indentation level is set to that pushed onto the stack by the most recent *if* statement, provided it was a part of an *if/else* clause and not a standalone *if*. Following the *else* declaration the stack is popped and the next indentation level awaits.

The non-terminal <reset_indent> is used when the top *level else* statement from the <main> non-terminal is called. At this point, regardless of the current depth, *indent* and *tree_depth* are reset to 0.

The code for this part of the function is displayed on the following page.

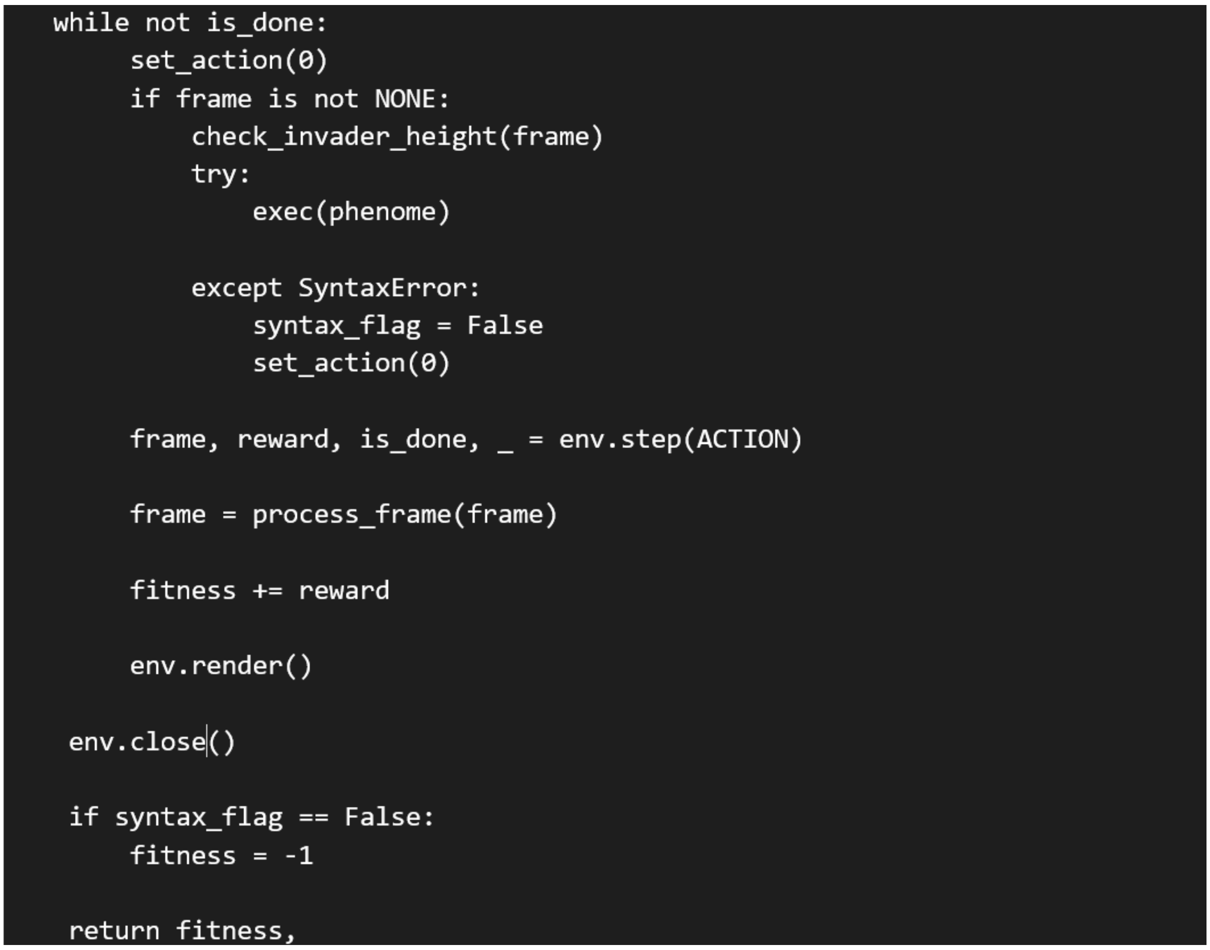


```
if(non_terminal is not None and i < len(choices) and tree_depth < 20):

            component = non_terminal.group(0)

            if component == "<indent>":
                indent += 1

            if component == "<unindent>":
                indent -=1

            if component == ("<reset_indent>"):
                indent = 0
                tree_depth = 0

            if component == ("<push_tree>"):
                indent_stack.append(indent)

            if component == ("<pop_tree>"):
                tree_depth  = indent_stack.pop()

            rule = prod_rule_dict[component]
            choice = rule[choices[i] % len(rule)]

            if(choice.startswith("-I-")):
                #indent to current indent level
                for _ in range(indent):
                    choice = "\t" + choice

            if(choice.startswith("-M-")):
                #match current tree depth
                for _ in range(tree_depth):
                    choice = "\t" + choice

            phenome = phenome.replace(component, choice, 1)
```

Figure 25 - Type 2 g2p_map (excerpt)

6.2.3 – Type 2 Fitness Evaluation Function

The Type 2 fitness evaluation function is a less complex entity than its Type 1 counterpart. Thanks in a major way to the ingenuity of the already described *exec* function. In this implementation *exec* is called from within a try/catch statement. This is to catch any phenomes that have unmapped non-terminal elements still present from the genome to phenome mapping procedure. The maximum tree depth is specified in the mapping function and once this level is reached no further non-terminals will be mapped. Any individuals of this nature are given a score of -1 and the action 0 is fed to the emulator ensuring that the *Defender* does not accumulate any points. A harsh but fair punishment.



```
while not is_done:
     set_action(0)
     if frame is not NONE:
         check_invader_height(frame)
         try:
             exec(phenome)

         except SyntaxError:
             syntax_flag = False
             set_action(0)

     frame, reward, is_done, _ = env.step(ACTION)

     frame = process_frame(frame)

     fitness += reward

     env.render()

 env.close()

 if syntax_flag == False:
     fitness = -1

 return fitness,
```

Figure 26 - Type 2 fitness_eval (excerpt)

6.2.3 – Type 2 Training Run

```
===================2021-04-23 09:16:39.072562===================

Run 1 of 1


Hyper Parameters:

    -    Game: SpaceInvaders-v4
    -    Seed: 683
    -    Pop: 250
    -    Number of Runs: 1
    -    Generations: 25
    -    Individual Length: 4000
    -    Crossover Rate: 0.7
    -    Crossover Type: cx_one_point
    -    Mutation Rate: 0.04
    -    Mothership Curb: False
    -    Life Penalty: False
    -    Invasion Check: False


Runtime: 6:37:36.083266

MaxFitness Value: 1740.0


CPU Info:

Intel64 Family 6 Model 142 Stepping 10, GenuineIntel

Physical cores: 4

Total cores8

Max Frequency: 1992.0Mhz
```

Generation	Num. Evals.	Max. Score	Min. Score	Avg. Score	Std. Dev.
0	250	650	-1	216.024	152.941
1	184	805	-1	338.524	146.533
2	175	785	-1	364.61	168.166
3	167	785	-1	412.328	176.852
4	161	920	-1	429.8	192.651
5	144	920	-1	464	199.168
6	161	920	-1	472.096	199.168
7	178	920	-1	451.34	211.103
8	172	970	-1	470.888	231.596
9	182	970	-1	484.308	238.239
10	175	970	-1	520.776	228.234
11	173	1180	-1	532.404	260.605
12	185	1240	-1	538.256	247.127
13	174	1240	-1	549.408	251.405
14	183	1240	-1	545.296	255.093
15	176	1250	85	577	285.23
16	196	1240	-1	551.036	285.028
17	166	1240	-1	622.592	328.217
18	172	1740	-1	666.628	361.915
19	191	1740	-1	604.828	352.611
20	194	1740	-1	600.452	350.827
21	175	1740	-1	633.78	367.981
22	166	1740	110	670.94	385.329
23	182	1740	195	667.42	394.246
24	186	1740	-1	642.636	407.493
25	174	1740	-1	678.656	428.013

See the next page for a printout of the best performing phenotype from this training run.

```
bound_box_height=5
bound_box_width=3

if(check_block_width(frame) >=50):
        if(check_invader_amount(frame) >=50):
                set_action(seek_side_invader(frame,True))
        else:
                if(check_block_width(frame) <= 50):
                        set_action(seek_side_invader(frame,False))
                else:
                        if(check_block_width(frame) >=10):
                                if(check_block_width(frame) <= 75):
                                        set_action(seek_side_invader(frame,False))
else:
        if(check_invader_height(frame) <= 50):
                set_action(check_b_box(frame,bound_box_height,bound_box_width))
```

This phenome was run 100 times repeatedly on differing non-deterministic playthroughs of Space Invaders.

It achieved a maximum score of 970 and an average score of 450.98

6.2.5 – Example Phenotype 1

```
bound_box_height=9
bound_box_width=5

if(check_invader_height(frame) >=25):
        if(check_invader_height(frame) >=50):
                set_action(seek_side_invader(frame,True))
else:
        if(check_block_width(frame) <= 50):
                set_action(seek_side_invader(frame,True))
        else:
                if(check_invader_amount(frame) <= 50):
                        set_action(seek_lowest_invader(frame,True,True))
                else:
                        set_action(seek_lowest_invader(frame,False,False))
```

This phenome was run 100 times repeatedly on differing non-deterministic playthroughs of Space Invaders.

It achieved a maximum score of 885 and an average score of 396.8

6.2.6 – Example Phenotype 2

```
bound_box_height=4
bound_box_width=8

if(check_invader_height(frame) <= 75):
        if(check_invader_amount(frame) >=10):
                set_action(check_b_box(frame,bound_box_height,bound_box_width))
        else:
                if(check_invader_amount(frame) >=75):
                        set_action(check_b_box(frame,bound_box_height,bound_box_width))
else:
        if(check_invader_amount(frame) <= 75):
                set_action(seek_lowest_invader(frame,True,True))
        else:
                set_action(check_b_box(frame,bound_box_height,bound_box_width))
```

This phenome was run 100 times repeatedly on differing non-deterministic playthroughs of Space Invaders.

It achieved a maximum score of 800 and an average score of 327.9

6.2.7 – Example Phenotype 3

```
bound_box_height=6
bound_box_width=6

if(check_invader_amount(frame) >=75):
    if(check_invader_amount(frame) >=10):
        set_action(check_b_box(frame,bound_box_height,bound_box_width))
    else:
        if(check_block_width(frame) >=10):

            set_action(check_b_box(frame,bound_box_height,bound_box_width)

else:
    if(check_invader_height(frame) <= 25):
        if(check_block_width(frame) <= 75):

            set_action(check_b_box(frame,bound_box_height,bound_box_width)
    else:
        set_action(seek_lowest_invader(frame,False,True))
```

This phenome was run 100 times repeatedly on differing playthroughs of Space Invaders.

It achieved a maximum score of 800 and an average score of 387.95

Chapter 7 – Evaluation, Critique and Conclusion

7.1 – Evaluation

In terms of a general evaluation I feel that my implementations performed well. By way of a benchmark I observed the maximum and average scores obtained by an agent that solely makes random moves. The scores were as follows:

Generation	Num. Evals	Max	Min	Avg.	Std. Dev
0	100	620	20	157.3	110.203

It can be seen that both the Type 1 and Type 2 demonstrations significantly outperform a random agent. One of the best performing runs of my Type 1 implementation achieved a maximum score of 2020. This score outperforms numerous attempts at Space Invaders, conducted with differing machine learning approaches. My Type 1 approach would rank a respectable 30th position overall on the PapersWithCode Atari leader board, nestled between MEFC (Model Free Episodic Control) which achieved a score of 1990 (Blundell *et al. 2016)* and DREAMER V2 which achieved a score of 2112 using a discrete world model approach (Hafner *et al. 2020)*. Let it be noted that the GE approach to Space Invaders significantly outperforms the only other evolutionary based approach that appears on the leader board. The Cartesian Genetic Programming approach to Space Invaders managed to achieve a high score of 1001 (Wilson *et al.* 2018)

While the scores achieved by the Type 1 implementation are impressive it is worth noting the deterministic quality of the game in which it trains. The Type 1 implementation can be considered as a fine tipped paint brush, a fantastic choice when one wants to apply specific, fine detail, each move carefully measured, evaluated and made. The non-deterministic, Type 2 implementation however, paints in broader strokes, ultimately covering a lot more canvas.

The effectiveness of the Type 2 solution is a little harder to evaluate, as Type 2 evolves a general solution to Space Invaders it is natural that its performance is not as immediately impressive as the results achieved by Type 1. The best performing individual presented in this report achieved an average score of 450.98 over 100 non-deterministic playthroughs of the game. While this is significantly higher than the average score achieved by a completely random agent it appears slightly underwhelming in comparison to the scores achieved by the Type 1 deterministic approach. An unfair comparison, it is true but one that is hard to refrain from all the same.

7.2 – General Critique

One of the main and most consistent issues I encountered with this project centred around the very long execution times associated with performing a training cycle. OpenAI Gym, as has been covered in this report, renders each playthrough on a frame by frame basis and no quicker. Even with no extra processing overhead, every single frame of a playthrough must be rendered and displayed on screen, there are often thousands of frames per game, couple that with 150 individuals, training over 25 generations, repeated in triplicate and you have a lot of processing to get through. Add on top of all this the large demand the increasingly complex GE framework was putting on the system and execution times rose to over 10 hours for one 25 generation training iteration. This proved to be an issue that I feel prevented me from fully exploring the complete potential of my solutions, particularly that of Type 1.

Another, and perhaps more serious critique I have is the lack of commonality that exists between the mapping functions of each implementation type. At their respective core they are the same function, however I deigned to slightly modify them individually to make my life a little easier. This is something that I would put more planning into if I were to do this project again. The late introduction of a different approach to the video game controller problem, while ultimately rewarding, upset the applecart somewhat in terms of planning.

7.3 – Type 1 Critique

The solutions discovered by the Type 1 implementation are extremely limited in their application. As this implementation is limited to performing on deterministic playthroughs any evolved phenotype will only be able to reproduce its fitness levels on the specific run of Space Invaders on which it was trained. The series of actions evolved were hyper detailed and at times allowed the game agent to perform moves with an almost super-human dexterity, but what the solution offered in complexity, it lacked in generality.

Due to the nature of the grammars used with this approach the individuals used are extremely long. The optimal individual length I worked with for Type 1 was 75,000 bits. This was a lot for my cpu to handle, especially so with the already long execution cycles the project was suffering from.

A further critique I have is regarding the *life_penalty* applied in some Type 1 training runs. This is an imperfect and potentially ineffective adaption to the solution. In hindsight, I can see that the number of lives used or not used is of little importance to the solution. I do not believe that the *life_penalty* stifled the algorithm's progress in any way, just that it may have been an unnecessary expenditure of development time.

My final critique of the Type 1 approach is the lack of any evolved code at the end of a training sequence. I feel like I missed the point of GE somewhat in this regard, evolved code is a defining feature of the algorithm and I felt it was sorely lacking from my project. This however did provide me with the motivation to delve further and explore a more definitively GE solution.

7.4 – Type 2 Critique

The main critique I can offer on my Type 2 implementation is that the strategies it evolves do not seem to perform very well when ran over a number of games. The grammatical evolutionary side of the implementation operates just fine, however the suitability of the problem it is being applied to may be called into question. Over repeated training runs the Type 2 algorithm achieves decent fitness scores but it seems unable to converge on a solution. I feel that the problem may stem from the selection of hand-coded functions provided to the *Defender*, they are overly offensive and lead to repeated and unavoidable loss of life which stifles evolutionary progress. The only defensive function Type 2 can use is *check_bounding_box*, this proved to be insufficient protection for the *Defender* when it came to non-deterministic play.

7.5 – Conclusion

I very much enjoyed working on this project, I have learned that the topic of Grammatical Evolution is one that appeals to me. I feel that I have gained a respectably in-depth understanding of the intricacies of the topic and the power it holds and overall I am proud of the work I have completed in this area.

The goals of this project were to

I shall continue my research in this field and will try to develop a more effective grammatical approach to a different Atari game.

I have spent a considerable amount of time working with Python since the beginning of this project cycle, as this is the first project I began in Python I feel it has been extremely helpful in getting a deeper understanding of the language.

References

Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M., 2013. ‘The Arcade Learning Environment: An evaluation platform for general agents’, *Journal of Artificial Intelligence Research*, *47*, pp.253-279.

Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J.Z., Rae, J., Wierstra, D. and Hassabis, D., 2016. ‘Model-free episodic control.’ *arXiv preprint arXiv:1606.04460*.

Bremermann, H.J., 1962. ‘Optimization through evolution and recombination’, *Self-organizing systems*, *93*, p.106.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym.

De Rainville, F.M., Fortin, F.A., Gardner, M.A., Parizeau, M. and Gagné, C., 2012. ‘DEAP: A python framework for evolutionary algorithms.’, *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pp. 85-92.

Fogel, L.J., 1965. ‘Artificial intelligence through a simulation of evolution.’, In *Proc. of the 2nd Cybernetics Science Symposium*.

Galván-López, E., Swafford, J.M., O’Neill, M. and Brabazon, A., 2010. ‘Evolving a Ms. Pacman controller using grammatical evolution’, *European Conference on the Applications of Evolutionary Computation* (pp. 161-170). Springer, Berlin, Heidelberg.

Goldberg, D.E., 1989. ‘*Genetic algorithms in search, optimization and machine learning*’.

Hafner, D., Lillicrap, T., Norouzi, M. and Ba, J., 2020. ‘Mastering atari with discrete world models.’ *arXiv preprint arXiv:2010.02193*.

Holland, J.H., 1992. ‘*Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.*’, MIT press.

Keijzer, M., Ryan, C., O’Neill, M., Cattolico, M. and Babovic, V., 2001. ‘Ripple crossover in genetic programming.’, In *European Conference on Genetic Programming*, Springer, Berlin, Heidelberg, pp.74-86.

Koza, J.R., 1992. ‘*Genetic programming: on the programming of computers by means of natural selection*’, MIT press.

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

Naur, P., Backus, J.W., Bauer, F.L., Green, J., Katz, C. and McCarthy, J., 1976. '*Revised report on the algorithmic language Algol 60*.', State University of New York at Buffalo, Computing Center.

Noorian, F., de Silva, A.M., Leong, P.H., 2016. 'Grammatical Evolution: A Tutorial using gramEvol', MIT Press

O'Neill, M. and Ryan, C., 2001. 'Grammatical evolution.', *IEEE Transactions on Evolutionary Computation*, *5*(4), pp.349-358.

O'Neill, M., Ryan, C., Keijzer, M. and Cattolico, M., 2003. 'Crossover in grammatical evolution.', *Genetic programming and evolvable machines*, *4*(1), pp.67-93.

Rechenberg, I., 1989. 'Evolution strategy: Nature's way of optimization.', *Optimization: Methods and applications, possibilities and limitations*, Springer, Berlin, Heidelberg, pp.106-126.

Ryan, C., Collins, J.J., Neill, M.O., 1998. 'Grammatical evolution: Evolving programs for an arbitrary language.', *European Conference on Genetic Programming, Springer, Berlin, Heidelberg, pp. 83-96.*

Togelius, J., 2015. 'AI researchers, Video Games are your friends!.', *International Joint Conference on Computational Intelligence* Springer, Cham, pp.3-18.

Turing, A.M., 1948. '*Intelligent machinery;*.

Turing, A.M., 1953. 'Digital computers applied to games'. *Faster than thought*.

Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P. and Oh, J., 2019. 'Grandmaster level in StarCraft II using multi-agent reinforcement learning.', *Nature*, *575*(7782), pp.350-354.

Vose, M.D., 1999. '*The simple genetic algorithm: foundations and theory*.', MIT press.

Wilson, D.G., Cussat-Blanc, S., Luga, H. and Miller, J.F., 2018. 'Evolving simple programs for playing Atari games', In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 229-236).

Appendix

i – Sample Mapping Procedure

The following section provides a blow-by-blow account of a sample genome to phenome mapping procedure as carried out by my prototype mapping function.

i.i – Grammar Specification

```
S           ::= <expr>
<expr>      ::= <expr><op><expr> | <sub-expr>
<sub-expr>  ::= <func>(<var>) | <var> | pow(<var>,<n>)
<func>      ::= math.log | math.sqrt | math.sin | math.cos
<op>        ::= + | - | * | / | //
<var>       ::= 100 | pow(100,<n>) | <n> | math.pi
<n>         ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

i.ii – Sample Individual

```
01111010110110111010101100000111100001010000110011010111000101
01001000010011010010011111100101011001011010101001100000000110
01110000111010000011100001101000110100001100100010111001101110
11010101101000111101011110011110100011101101000011111110111101
00
```

i.iii – Decimal Representation of Codons

```
122, 219, 171, 7, 133, 12, 215, 21, 33, 52, 159, 149, 150, 169,
128, 103, 14, 131, 134, 141, 12, 139, 155, 181, 104, 245, 231,
163, 180, 63, 189
```

i.iv – Mapping Process

Note: Phenomes are mapped from the leftmost component at each stage.

```
Component: <expr>
Rule: ['<expr><op><expr>', '<sub-expr>'] , len(rule): 2
Codon to decimal element 0 = 122
122 % 2 : 0
Choice: <expr><op><expr>
Phenome = <expr><op><expr>


Component: <expr>
Rule: ['<expr><op><expr>', '<sub-expr>'] , len(rule): 2
Codon to decimal element 1 = 219
219 % 2 : 1
Choice: <sub-expr>
Phenome = <sub-expr><op><expr>


Component: <sub-expr>
Rule: ['<func>(<var>)', '<var>', 'pow(<var>,<n>)'] , len(rule): 3
Codon to decimal element 2 = 171
171 % 3 : 0
Choice: <func>(<var>)
Phenome = <func>(<var>)<op><expr>


Component: <func>
Rule: ['math.log', 'math.sqrt', 'math.sin', 'math.cos'] , len(rule):
4
Codon to decimal element 3 = 7
7 % 4 : 3
Choice: math.cos
Phenome = math.cos(<var>)<op><expr>


Component: <var>
Rule: ['100', 'pow(100,<n>)', '<n>', 'math.pi'] , len(rule): 4
Codon to decimal element 4 = 133
133 % 4 : 1
Choice: pow(100,<n>)
Phenome = math.cos(pow(100,<n>))<op><expr>


Component: <n>
Rule: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10'] ,
len(rule): 10
Codon to decimal element 5 = 12
12 % 10 : 2
Choice: 3
Phenome = math.cos(pow(100,3))<op><expr>
```

```
Component: <op>
Rule: ['+', '-', '*', '/', '//'] , len(rule): 5
Codon to decimal element 6 = 215
215 % 5 : 0
Choice: +
Phenome = math.cos(pow(100,3))+<expr>


Component: <expr>
Rule: ['<expr><op><expr>', '<sub-expr>'] , len(rule): 2
Codon to decimal element 7 = 21
21 % 2 : 1
Choice: <sub-expr>
Phenome = math.cos(pow(100,3))+<sub-expr>


Component: <sub-expr>
Rule: ['<func>(<var>)', '<var>', 'pow(<var>,<n>)'] , len(rule): 3
Codon to decimal element 8 = 33
33 % 3 : 0
Choice: <func>(<var>)
Phenome = math.cos(pow(100,3))+<func>(<var>)


Component: <func>
Rule: ['math.log', 'math.sqrt', 'math.sin', 'math.cos'] , len(rule):
4
Codon to decimal element 9 = 52
52 % 4 : 0
Choice: math.log
Phenome = math.cos(pow(100,3))+math.log(<var>)


Component: <var>
Rule: ['100', 'pow(100,<n>)', '<n>', 'math.pi'] , len(rule): 4
Codon to decimal element 10 = 159
159 % 4 : 3
Choice: math.pi
Phenome = math.cos(pow(100,3))+math.log(math.pi)
```

Final expression: `math.cos(pow(100,3))+math.log(math.pi)`

Value: `2.081482013382545`