Utilizing a Genetic Algorithm to Implement on a Self-Driving AI Car Simulation and Evaluating the Performance

Kim Marcial A. Vallesteros
College of Engineering, Architecture and Fine Arts
Computer Engineering
Batangas, Philippines
kimmarcial.vallesteros@g.batstate-u.edu.ph

Charlsjon Angelo M. Errazo
College of Engineering, Architecture and Fine Arts
Computer Engineering
Batangas, Philippines
charlsjon.errazo@g.batstate-uedu.ph

Ken R. Tolentino
College of Engineering, Architecture and Fine Arts
Computer Engineering
Batangas, Philippines
ken.tolentino@g.batstate-uedu.ph

I. INTRODUCTION

Artificial Intelligence (AI) has been a relevant and is without a doubt still a large topic nowadays in the field of technology and modern advancement. However, artificial intelligence is not simply based on gathered data which is organized through a database to perform a certain action but shows the nature of augmentation in the context of self-learning and adaptation. To further explain the said topic of interest, the researchers first provide an absolute definition of the analogies used, through which will be used to correlate their understanding to that of their proposed topic. According to IBM Cloud Education (2020), Artificial Intelligence or AI is defined as components that "leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind". Another definition provided by John McCarthy (2004) from his paper states that, AI is the art and science of creating intelligent machines, particularly clever computer programs. It is akin to the task of utilizing computers to study human intellect, but AI does not have to be limited to physiologically observable methods. As aforementioned, the researchers deem the concept to be in a way that a machine learns how to adapt and consequently provide a system where not only to provide ease in the user's everyday living but also the computer or machine's benefit on itself. Truly, the science behind Artificial Intelligence is the key to the future.

Artificial intelligence, in its most basic form, is a field that combines computer science with large datasets to solve problems. It also includes the sub-fields of machine learning and deep learning, both of which are usually referenced when discussing artificial intelligence. AI algorithms are used in these areas to develop expert systems that make predictions or classifications based on input data. To adapt to such capability, it should be noted that the researcher chose a topic that piques their interest. The use of Phyton programming language was inevitably used as it is relatively basic for beginner programmers yet offers a lot of opportunities in the technological setup. Python is a general-purpose programming language which can run on any modern computer. This programming language can process text, numbers, photos, scientific data, and just about anything else that can be saved on a computer. Google's search engine, YouTube's video-sharing website, NASA, and even the New York Stock Exchange all use it on a daily basis. There are other instances where Python plays a significant part in the success of businesses, governments, and non-profit organizations. According to Al Lukaszewski (2019), Python is a free programming language that makes addressing a computer problem virtually as simple as writing down an individual's thoughts about the answer. Without changing the software, the code can be created once and run on practically every computer.

While the programming language is justified, there should also be considerations to the algorithm that makes all of this possible. Genetic algorithm is a method based on natural selection, the mechanism that drives biological evolution, for addressing both limited and unconstrained optimization problems. A collection of individual solutions is repeatedly modified

by the genetic algorithm. At each phase, the genetic algorithm chooses parents at random from the current population and uses them to produce the following generation's children. The population "evolves" toward an optimal solution over generations. Specifically, NeuroEvolution of Augmenting Topologies (NEAT) is used in the simulation. NEAT is a genetic algorithm-based method for evolving artificial neural networks. NEAT is based on the principle that it is most effective to begin evolution with small, simple networks and gradually increase their complexity over generations. As a result, much as organisms in nature have grown in complexity since the first cell, neural networks in NEAT have grown in complexity as well. Finding highly intricate and complicated neural networks is possible thanks to this continuous elaboration process.

The statement of the problem specifically focuses on the method that will be used as to address the needed process or product. Generally, experimental design was done by the researchers to evaluate whether genetic algorithm is effective for training self-driving cars in a given simulation. Genetic algorithm, being the core adaptive process of this AI cars are of great factor in the effectivity of the said setup. This will prove many various applications in life and help disabilities, which in turn can provide aid to differently-abled individuals in the latter. Moreover, the current state of this issue is that it is usually applied to basic or simple contemporaries in our modern times. However, the desired future state that the researchers are hoping for, is that the genetic algorithm would be more complex, adaptive, and interactive to world applications.

The objectives of the said research are as follows: (1) To design a simulation which uses a self-learning neural network to implement in the AI car simulation. The researchers believe that in designing a simulation, they will be able to conveniently use their time and effectively execute the said outcome. Simulations are done prior to actual applications, it does not only ensure the probability of an outcome to be successful, but it also inhibits safety, security, and saving excess time or effort. The next objective is (2) to execute simulations to further evaluate the performance of the AI. The researchers believe that the only way to completely evaluate an action is to run attempts or assess the needed criteria through numerous processes as to test the effectivity of the algorithm. To evaluate the performance of the AI would generally mean that the researchers are too trying to find the probable cause for Artificial intelligence to thrive on with the demands of the ever-changing world. Lastly, is (3) to determine if a self-learning neural network is an effective way to train the AI to perform set actions. Through this matter, the researchers will be able to determine whether NEAT is indeed the needed topology for self-driving cars that inhibits Artificial Intelligence in a given simulation. In addition, this will define the importance of Neural adaptation in the context of Artificial Intelligence. The neural network is relatively pre-existent as to nowadays contemporaries. Determining its effectivity will also allow the researchers to compare it to other topologies present for further evaluation as to assess their capabilities.

Review of Related Literature

Genetic Algorithm Implementation in Python

A genetic algorithm is a probabilistic search algorithm derived from natural selection and genetics mechanics. Developers introduced genetic algorithm implementation methods and defined it as a dynamically typed programming language that is interpreted, interactive, and object-oriented. Python supports a variety of programming paradigms, including procedural, functional, and object-oriented programming. Python's dynamic nature allows it to be more productive than C/C++/Java. Genetic algorithms can be implemented quickly and easily in Python without the use of any third-party libraries. In genetic algorithms, the functional programming style comes in helpful. Python applications are slower to execute than C/C++ applications. However, with tools like Numerical Python, Numarray, SWIG, SIP, Pyrex, and Psyco, one can get high performance comparable to C/C++.

Self-Driving Cars and the Urban Challenge in IEEE Intelligent Systems

Self-driving cars have been a pipe dream for as long as there have been autos. Self-driving cars represent a significant technology advancement that has the potential to provide answers and drastically alter today's transportation network. A self-driving car, also known as an autonomous car, personal automated vehicle, driverless car, or robotic car, is a vehicle that can drive and navigate autonomously without the need for human intervention. In the industrialized world, the automobile is ubiquitous, and it is becoming so in the developing world. The two largest automakers in the world sold almost 18 million automobiles worldwide in 2007. When it comes to domains where intelligent systems can be used, the automotive industry stands out as having the highest potential for influence. The "Winter of AI" is a decades-long period of inaction that followed the implosion of Artificial Intelligence research, which had over-reached its goals and over-hyped its promise until its ultimate decline in the early 1980s.

Ontology based Scene Creation for the Development of Automated Vehicles

A functional system description, including functional system boundaries and a full safety study, is required for the deployment of automated vehicles without persistent human supervision. A scenario-based methodology can be used to identify and assess these technological development inputs. Furthermore, a significant number of scenarios must be established to produce meaningful test results in order to develop an affordable test and release process. Experts are doing a great job of identifying scenarios that are difficult to deal with or unlikely to occur. On the other hand, they are unlikely to be able to identify all probable scenarios based on their current expertise. Expert knowledge modeled for computer-assisted processing could aid in the provision of a diverse set of scenarios. This examines ontologies as knowledge-based systems in the context of automated cars and proposes using natural language to generate traffic scenes as a basis for scenario building.

Python for Data Analytics, Scientific and Technical Applications

Data science and analytics, a branch of computer science, has resurfaced in recent years as a result of significant increases in computer power, the presence of massive amounts of data, and a better understanding of techniques in the areas of Data Analytics, Artificial Intelligence, Machine Learning, Deep Learning, and other related fields. As a result, they have become an important aspect of the technology business, and they are being employed to solve a variety of difficult problems. Python has emerged as a full programming solution in the search for a good programming language on which numerous data science applications can be constructed. Python has become one of the fastest growing languages due to its minimal learning curve and flexibility. Python is a solid option for data analytics because of its ever-evolving libraries. The paper explains the qualities and characteristics of the Python programming language, as well as why it is considered one of the fastest growing programming languages and why it is at the forefront of data science applications, research, and development.

Deepdriving: Learning affordance for direct perception in autonomous driving

There are two main paradigms for vision-based autonomous driving systems today: mediated perception methods, which analyze an entire scene to make a driving decision, and behavior reflex approaches, which use a regressor to directly map an input image to a driving action. The researchers offer a third paradigm for estimating driving affordance: a direct perception method. They propose mapping an input image to a small number of essential perception markers that are closely related to a road/traffic state's driving affordance. To demonstrate this, they use recordings from 12 hours of human driving in a video game to train a deep Convolutional Neural Network, demonstrating that our model can drive a car in a wide range of virtual scenarios. The researchers suggest a revolutionary direct perception-based autonomous driving paradigm. Instead of parsing complete scenes (mediated perception techniques) or blindly mapping an image to driving commands, their representation uses a deep ConvNet architecture to assess the affordance for driving actions (behavior reflex approaches). The results suggest that their direct perception method may be used to real-world driving photos with good results. Experiments demonstrate that their method works effectively in both virtual and real-world settings.

II. METHODOLOGY

This research shall adopt an experimental design as it deals with gathering data from actual simulation of the AI. The approach used by the researchers is qualitative and is not solely based on related literatures, but also the observations from the actual design of the proposed topic. The simulation will test the performance of the AI based on the chosen algorithm that utilizes reinforcing learning through different types of situations or maps. The simulation will compose of a car driving through 3 different maps. By performing the simulation multiple times data will be drawn out to determine if a self-learning AI is an effective way to train self-driving cars to carry out or avoid different obstacles and self-operate.

Paradigm (IPO)

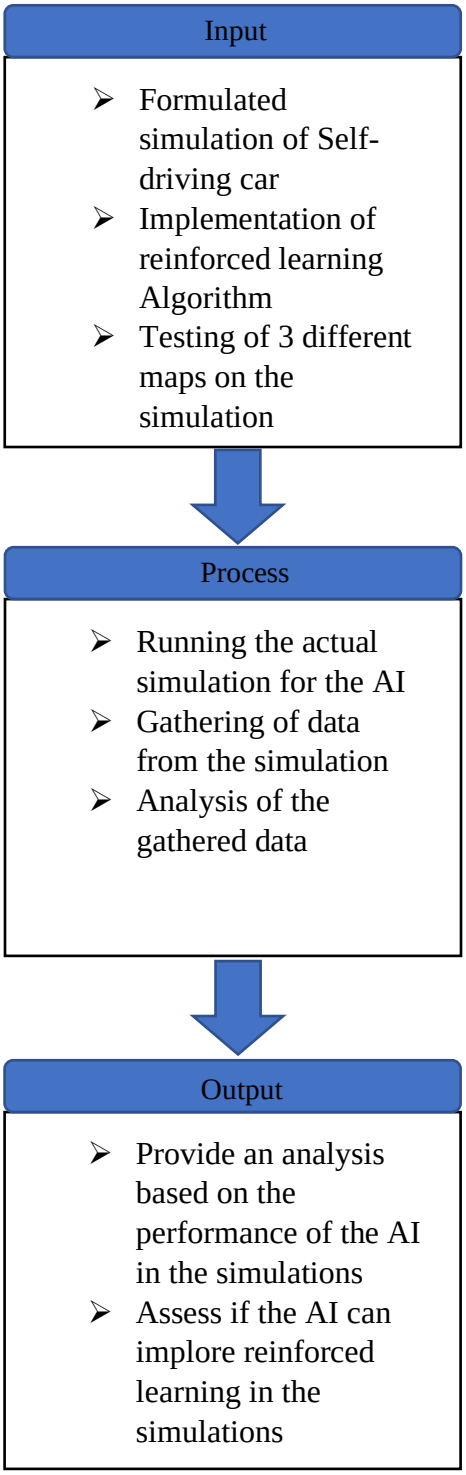


Figure 1
Evaluation of Self-Driving Car Simulation

Figure 1 shows the paradigm on evaluating the performance of the self-driving car simulation.

The input starts off with the formulated or created simulation, along with creating the simulation is implementing the reinforced self-learning Algorithm to be used and after creating the simulation 3 maps and the cars will be loaded into the simulation to determine the performance of the cars on the different tracks. For the process, the researchers will run the actual simulation. The simulation will run for 3 times, 1 run per map and the algorithm that will be implemented will configure it to run the simulation for a set number of generations which will aid in assessing the learning rate of the AI.

Components of Simulation

This research will perform simulations of an artificial intelligence on cars to allow the cars to learn how to self-operate and maneuver through a given course. The simulation will use an algorithm that deals with evolution and which produces artificial neural networks.

Two major components are needed for the simulation, first is the environment in which the simulation will take place and second is the algorithm to be used on the cars. The first component that will be used as the environment for the simulation is using Pygame because to perform the simulation for the artificial intelligence of self-driving cars is that the most optimal choice is to perform the simulation on a cross-platform module that is made for video games. Pygame will be adopted for the sole purpose of writing and designing the simulation environment. To further evaluate the performance of the self-driving

cars three (3) maps will be made of increasing difficulty to determine if the AI will need more time to adopt and overcome the different paths it will encounter in the maps during simulation.

The second essential key component for this simulation and research is the algorithm to be used. One key factor in choosing the algorithm is the ability of reinforcement learning and the form of neural network it uses because neural networks also provide a capable representation of solutions to many kinds of problems. NEAT (NeuroEvolution of Augmenting Topologies) will be the algorithm to be used in this simulation as it is a form of efficient genetic algorithm that creates artificial neural networks. According to Stanley and Miikkulainem (2002) Neuroevolution is fitting for reinforcement learning tasks because it requires no supervision. NEAT aims to show that evolving topologies can increase performance for Neuroevolution. NEAT maintains a set population of individual genomes and each genome contains two sets of genes that describe how to build the artificial network: (1) Node genes, each of which specifies a single neuron and (2) Connection genes, each of which specifies a single connection between neurons (figure 2).

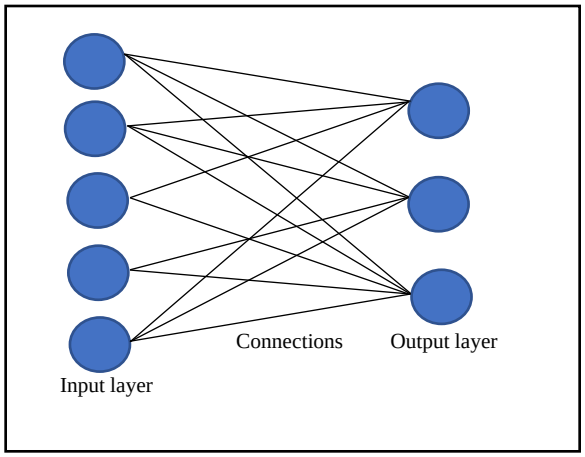


Figure 2
Neural Network Composing of Node Genes and Connection Genes

In figure 2 an example neural network is provided which consist of node genes of input layer and output layer and a connection gene which connects both input and output layer. Hidden layers are also considered as node genes.

In NEAT a fitness function must be set which computes single real number indicating the quality of an individual genome. The better the ability to solve the problem the higher the score. NEAT propagates through a set number of generations which was set by the user and with each passing generation being produced by reproduction (either sexual or asexual) and mutation of the most fit individuals in the previous generations. Reproduction and mutation operations may add nodes and or connections to genomes which means that as the algorithm progresses genomes and the possible neural network it may produce increases in complexity. After running the set number of generations, the algorithm is terminated.

Difficulties may be encountered in this setup, implementation of crossover one difficulty because how does crossover happen between two networks of differing architecture or structure? NEAT handles this difficulty by keeping track of the origin of the nodes with an identifying number this shows that NEAT tracks genes through historical markings. New higher numbers are generated for each individual node and those derived from a common ancestor in which that they are homologous will be matched up for crossover. Connections are

matched if the nodes they connect have the same ancestry. Another potential difficulty that may be encountered is structural mutation. For example, the weights of the connections, such as the addition of a node or connection such as the addition of a node or connection even if it's promisingly beneficial for the performance of the algorithm it might also be disruptive. NEAT handles this by dividing genomes into species, which have a close genomic distance due to its similarity, then by having competition most intense within that species and not in between other species (fitness sharing). Genomic distance is measured by using a combination of the non-homologous nodes and connections with measure of how much homologous nodes and connections have diverged since their common ancestry or origin.

NEAT may increase the complexity of the neural network architecture by adding node genes or connection genes base on the algorithm analysis of the problem it still shows effective results but identifying how the internal neural network structure changes is a challenge as it grows in complexity.

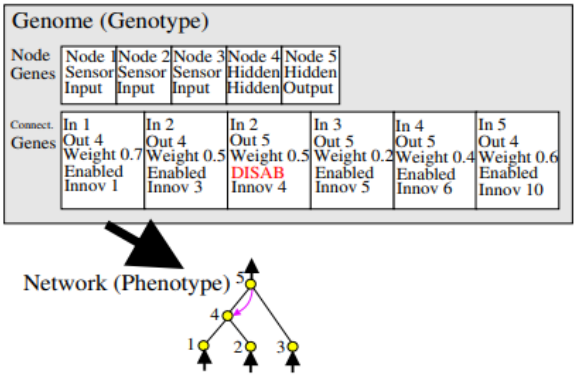


Figure 3

A genotype to phenotype mapping example

The third gene is disabled, so the connection that it specifies (between nodes 2 and 5) is not expressed in the phenotypes

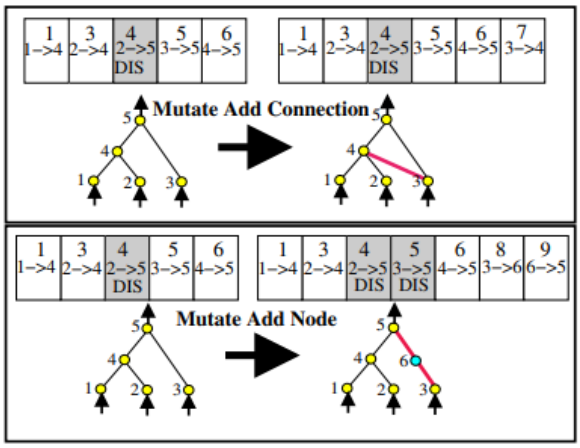
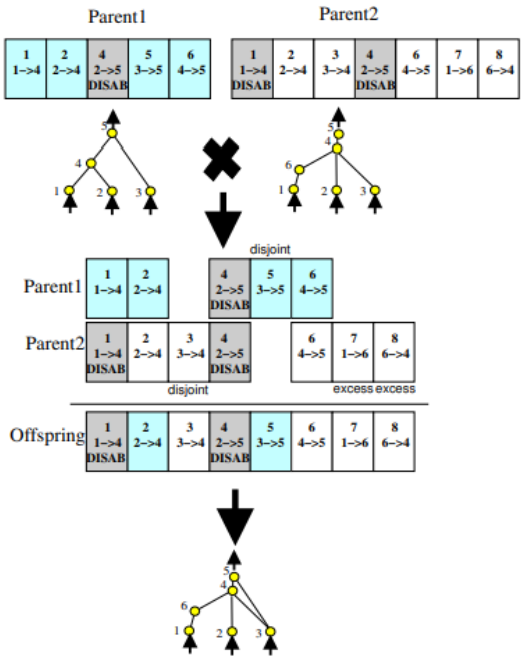


Figure 4

The two types of structural mutation in NEAT

Both types add a connection and adding a node, are showed with the genes above their phenotypes. The top number in each genome is the innovation number of that gene. These numbers identify the original historical ancestor of each gene, making it possible to find matching genes during



crossover. New genes are assigned new increasingly high numbers.

Figure 5

Matching up genomes for different network topologies using innovation numbers.

Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us which genes match up with which without the need for topological analysis.

Figures 3, 4 and 5 are from [1] this shows the process of how NeuroEvolution of Augmenting Topologies work from mapping to reproduction and eventually mutation.

The implementation of NEAT the researchers used was the NEAT-python since Pygame will be also used to create the environment. NEAT-python comes with a configuration file which contains the settings for the NEAT simulation. The NEAT configuration file may be stored as a txt file.

These are the configurations and values set for NEAT that will be used for the simulation. All the information for the functions of each configuration are according to [2].

[NEAT] Section

This section specifies parameters particular to the generic NEAT algorithm or the experiment itself.

- fitness_criterion
  The function used to compute the termination criterion from the set of genome fitnesses.
    - The fitness_criterion is set to *max* value as the configuration.
- fitness_threshold
  When the fitness computed by the fitness_criterion meets or exceeds this threshold, the evolution process will terminate, with a call to any registered reporting class.
    - The fitness_threshold is set with a value of *100000000* as the configuration.
- pop_size
  The number of individuals in each generation.
    - The pop_size is set with a value of *30* as the configuration.
- reset_on_extinction
  If this evaluates to *True*, when all species simultaneously become extinct due to stagnation, a new random population will be created. If *False*, a Complete Extinction Exception will be thrown.
    - The reset_on_extinction is set to *True* as the configuration.

[DefaultGenome] Section

The DefaultGenome section specifies parameters for the builtin DefaultGenome class.

- activation_default
  The default activation function attribute assigned to new nodes. If none is given, or “random” is specified, one of the activation_options will be chosen at random.
    - The activation_default is set to *tanh* as the configuration.

- activation_mutate_rate
  The probability that mutation will replace the node's activation function with a randomly-determined member of the activation_options.
    - The activation_mutate_rate is set with a value of *0.01* as the configuration for the corresponding simulation.
- activation_options
  A space-seperated list of the activation functions that may be used by nodes.
    - The activation_options is set to *tanh* as the configuration.
- aggregation_default
  The default aggregation function attribute assigned to new nodes. If none is given or "random" is specified, one of the aggregation_options will be chosen at random.
    - The aggregation default is set to *sum* as the configuration.
- aggregation_mutate_rate
  The probability that mutation will replace the node's aggregation function with a randomly-determined member of the aggregation_options.
    - The aggregation_mutate_rate is set with a value of *0.01* as the configuration.
- aggregation_options
  A space-separated list of the aggregation functions that may be used by nodes.
    - The aggregations_options is set to sum as the configuration.
- bias_init_mean
  The mean of the normal/gaussian distribution if it is used to select bias attributes values for new nodes.
    - The bias_init_mean is set with the value of *0.0* as the configuration.
- bias_init_stdev
  The standard deviation of the normal/gaussian distribution if it is used to select bias values for new nodes.
    - The bias_init_stdev is set with the value of *1.0* as the configuration.
- bias_max_value
  The maximum allowed bias value. Biases above this value will be clamped to this value.
    - The bias_max_value is set with the value of *30.0* as the configuration.
- bias_min_value
  The minimum allowed bias value. Biases below this value will be clamped to this value.
    - The bias_min_value is set with the value of *-30.0* as the configuration.
- bias_mutate_power
  The standard deviation of the zero-centered normal/gaussian distribution from which a bias value mutation is drawn.
    - The bias_mutate_power is set with a value of *0.5* as the configuration.
- bias_mutate_rate
  The probability that mutation will change the bias of a node by adding a random value.

    - The bias_mutate_rate is set with a value of *0.7* as the configuration.
- bias_replace_rate
  The probability that mutation will replace the bias of a node with a newly chosen random value (as if it were a new node).
    - The bias_replace_rate is set with a value of *0.1* as the configuration.
- compatibility_disjoint_coefficient
  The coefficient for the disjoint and excess gene counts' contribution to the genomic distance.
    - The compatibility_disjoint_coefficient is set with a value of *1.0* as the configuration.
- compatibility_weight_coefficient
  The coefficient for each weight, bias, or response multiplier difference's contribution to the genomic distance (for homologous nodes or connections). This is also used as the value to add for differences in activation functions, aggregation functions, or enabled/disabled status.
    - The compatibility_weight_coefficient is set with a value of *0.5* as the configuration.
- conn_add_prob
  The probability that mutation will add a connection between existing nodes.
    - The conn_add_prob is set with a value of *0.5* as the configuration.
- conn_delete_prob
  The probability that mutation will delete an existing connection.
    - The conn_delete_prob is set with a value of *0.5* as the configuration.
- enabled_default
  The default enabled attribute of newly created connections.
    - The enabled_default is set to *True* as the configuration.
- feed_forward
  If this evaluates to True, generated networks will not be allowed to have recurrent connections (they will be feedforward). Otherwise, they may be (but are not forced to be) recurrent.
    - The feed_forward is set to *True* as the configuration.
- initial_connection
  Specifies the initial connectivity of newly-created genomes.
    - The initial_connection is set to *full* as the configuration.
- node_add_prob
  The probability that mutation will add a new node.
    - The node_add_prob is set with a value of *0.2* as the configuration.
- node_delete_prob
  The probability that mutation will delete an existing node (and all connections to it).
    - The node_delete_prob is set with a value of *0.2* as the configuration.
- num_hidden
  The number of hidden nodes to add to each genome in the initial population.
    - The num_hidden is set with a value of *0* as the configuration.

- num_inputs
  The number of input nodes, through which the network receives inputs.
    - The num_inputs is set with a value of *5* as the configuration.
- num_outputs
  The number of output nodes, to which the network delivers outputs.
    - The num_ouputs is set with a value of *2* as the configuration.
- response_init_mean
  The mean of the normal/gaussian distribution if it is used to select response multiplier attreibute values for new nodes.
    - The response_init_mean is set with a value of *1.0* as the configuration.
- response_init_stdev
  The standard deviation of the normal/gaussian distribution if it is used to select response multiplier for new nodes.
    - The response_init_stdev is set with a value of *0.0* as the configuration.
- response_max_value
  The maximum allowed response multipler. Response multipliers above this value will be clamped to this value.
    - The response_max_value is set with a value of *30.0* as the configuration.
- response_min_value
  The minimum allowed response multipler. Response multipliers below this value will be clamped to this value.
    - The response_min_value is set with a value of *-30.0* as the configuration.
- response_mutate_power
  The standard deviation of the zero-centered normal/gaussian distribution from which a response multiplier mutation is drawn.
    - The response_mutate_power is set with a value of *0.0* as the configuration.
- response_mutate_rate
  The probability that mutation will change the response multiplier of anode by adding a random value.
    - The response_mutate_rate is set with a value of *0.0* as the configuration.

- response_replace_rate
  The probability that mutation will replace the response multiplier of a node with a newly chosen random value (as if it were a node).
    - The response_replace_rate is set with a value of *0.0* as the configuration.
- weight_init_mean
  The mean of the normal/gaussian distribution used to select weight attribute values for new connections.
    - The weight_init_mean is set with a value of *0.0* as the configuration.
- weight_init_stdev
  The standard deviation of the normal/gaussian distribution used to select weight values for new connections.
    - The weight_init_stdev is set with a value of *1.0* as the configuration.

- weight_max_value
  The maximum allowed weight value. Weights above this will be clamped to this value.
    - The weight_max_value is set with a value of *30* as the configuration.
- weight_min_value
  The minimum allowed weight value. Weights below this will be clamped to this value.
    - The weight_max_value is set with a value of *-30* as the configuration.
- weight_mutate_power
  The standard deviation of the zero-centered normal/gaussian distribution from which a weight value mutation is drawn.
    - The weight_mutate_power is set with a value of *0.5* as the configuration.
- weight_mutate_rate
  The probability that mutation will change the weight of a connection by adding a random value.
    - The weight_mutate_rate is set with a value of *0.8* as the configuration.
- weight_replace_rate
  The probability that mutation will replace the weight of a connection with a newly chosen random value (as if it were a new connection).
    - The weight_replace_rate is set with a value of *0.1* as the configuration.
- compatibility_threshold
  Individuals whose genomic distance is less than this threshold are considered to be in the same species.
    - The compatibility_threshold is set with a value of *2.0* as the configuration

[DefaultStagnation] Section

The DefaultStagnation section specifies parameters for the builtin DefaultStagnation class.

- species_fitness_func
  The function used to compute species fitness.
    - The species_fitness_func is set to *max* as the configuration.
- max_stagnation
  Species that have not shown improvement in more than this number of generations will be considered stagnant and removed.
    - The max_stagnation is set with a value of *20* as the configuration.
- species_elitism
  The number of species that will be protected from a stagnation; mainly intended to prevent total extinctions caused by all species becoming stagnant before new species arise.
    - The species_elitism is set with a value of *2* as the configuration.

[DefaultReproduction] Section

The DefaultReproduction section specifies parameters for the builtin Default Reproduction class.

- elitism
  The number of most-fit individuals in each species that will be preserved as-is from one generation to the next.
    - The elitism is set with a value of *2* as the configuration

- survival threshold
  The fraction for each species allowed to reproduce each generation.
  - The survival threshold is set with a value of *0.2* as the configuration.

Getting the results for running NEAT comes in with a method to query the statistics member of the population (a NEAT statistics. Statistics Reporter object).

Simulation Environment

The simulation environment will be designed using Pygame. A car object which comes as a picture will be loaded along the with designed maps creating the environment. All subsequent functions are created within the main python program will be contained in one place along with the algorithm. The program will load the car, the map, and the configuration file.

The Car Object

The car will be the main object that traverses through the maps. The car will start at a specific point. The car will operate on a function to determine how the car will die. When the car makes contact with the white color of the map (255, 255, 255, 255) it will terminate itself or put simply it has crashed. The car is set with a constant speed of 20 and a designated starting on the map.

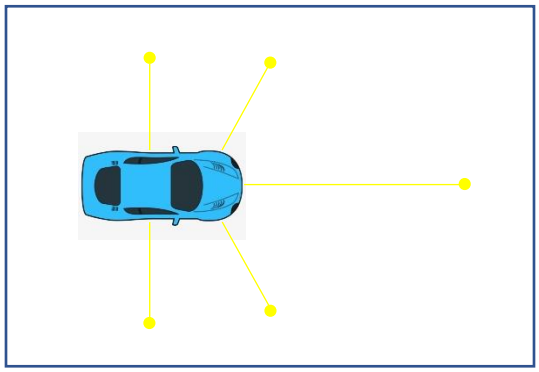


Figure 6

Car in Simulation with Displayed Radar

Figure 6 shows how the car will look in the environment. The radar will be displayed using a function and is an efficient illustration of showing the 5 sensor nodes that will serve as the input nodes in the neural network. The nodes will check for

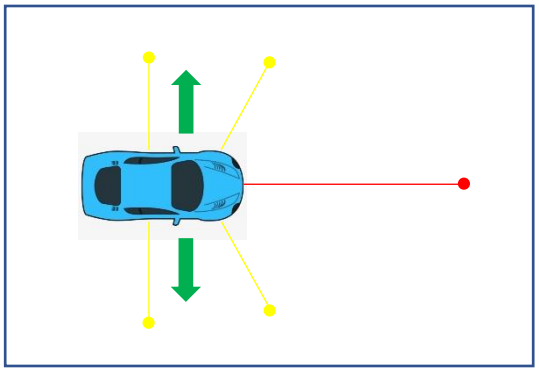


Figure 7

Car in Simulation Showing Possible Action to be Taken

Figure 7 shows the car in simulation showing the possible action it takes. The cars actions depend on the 5 input nodes on what they encounter in the map and along with this the neural network has 2 output nodes which are aligned on what actions the car will do which is to either go left or go right. As the generations progress the better the neural network will determine efficient actions to be taken based on what it has encountered in the previous generations.

Maps

All maps are made using the Paint tool and there are 3 maps in increasing difficulty. This will show the difference on how the neural networks adapts to that map.

Figure 8
Easy Map

Figure 8 shows the easy map which comprises of a simple loop.

Figure 9
Normal Map

Figure 9 shows the normal map which comprises of more vertical and horizontal turns.

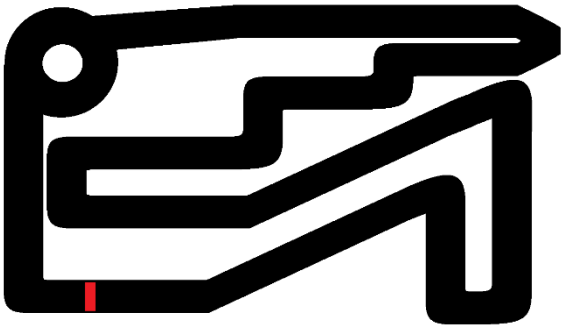


Figure 10
Hard Map

Figure 10 shows the hard map which includes oblique turns, sharp turns, and a circular path.

The car will be put to the test in each map to evaluate the AI's artificial neural networks performance.

The formulation of the simulation and implementation of the algorithm will be created first then the simulation will be conducted. The gathering of data will be conducted solely based on the gathered data in the simulations as it will allow for analysis and evaluation of the performance of the AI which was subjected to 3 different maps or tracks.

The gathered will be analyzed to assess the AI's performance. The implemented algorithm will provide a built-in statistical report based on the simulations that where run through a set of 10 generations. A comparison will be drawn out based on the different levels of difficulty of the maps to analyze if the difficulty of the map affects the nature or self-learning rate of the AI in the simulation.

Visual Studio Code (VS Code) is a lightweight code editor which allows for simple edit, run and debug features and is particularly lightweight compared to Python IDE's like PyCharm. VS code will be used as the code editor for the simulation and specifically for programming the main driver file which will import both Pygame and NEAT-python modules and also other miscellaneous imports that will aid in the simulation.

III. RESULT AND DISCUSSION

In this section, the discussion is elaborated based on the results obtained from the 3 simulations conducted for each map.

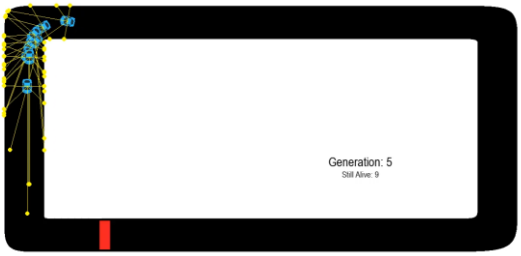


Figure 11
Simulation of Self-Driving AI Cars on Easy Map

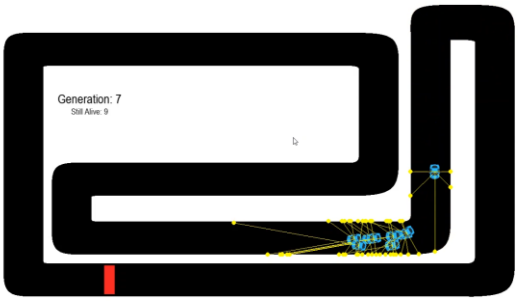


Figure 12
Simulation of Self-Driving AI Cars on Normal Map

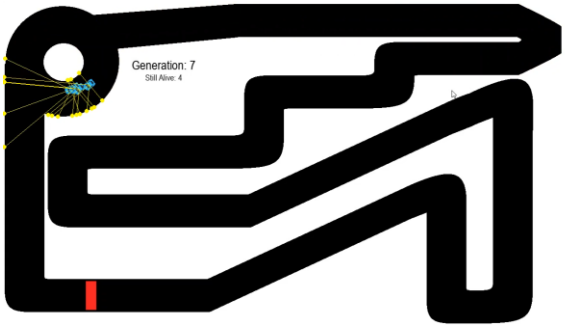


Figure 13
Simulation of Self-Driving AI Cars on Hard Map

The simulation has been successfully simulated and loaded. In figures 11, 12 and 13 the simulation environment which compromises of the cars and the map have been successfully implemented together along with this the NEAT algorithm and configurations was also implemented this allows for the cars to perform differently per generation. Some minor errors have been encountered during the simulation of the hard map which causes the simulation to stop due to being the pixel index being out of range or to close. The problem still persists but it was fixed by lowering the size of the car object in the simulation. By default, the size of the car is set to *48* in both easy and normal but since errors due to pixel index being range out of range the size of the car was lowered to *38* and it can be much lower and it the size of the car can be lowered more.

The result for the simulation is based on one simulation that was recorded by the researchers. If data is gathered on more than one simulation results may still be consistent but the average result will depend as each simulation will have different starting base generations and species reproduction and mutation.

The difference in starting generation and species determines how the species will reproduce and mutate over the course of the generations thus increasing the difficulty in obtaining the average data if multiple simulations are conducted but it does not mean that multiple simulations will produce unreliable data.

Table 1
Simulation Data for Easy Map

Generation	Population Average Fitness	stdev	Mean Genetic Distance	Standard Deviation	ID	Age	Size	fitness	adj fit	stag
1	753.58333	3305.35430	0.998	0.242	1	0	30	18191.7	0.041	0
2	20832.34722	74750.42852	0.991	0.301	1	1	30	300250	0.069	0
3	61366.95833	119530.30877	1.305	0.268	1	2	30	300250	0.204	1
4	85121.63889	132494.85418	1.423	0.262	1	3	30	300250	0.284	2
5	90260.98611	137470.92148	1.424	0.353	1	4	4	300250	0.301	3
					2	0	26	--	--	0
6	81838.58333	131798.95524	1.358	0.447	1	5	7	300250	0.500	4
					2	1	23	300250	0.238	0
7	120781.37500	146578.75865	1.076	0.304	1	6	10	300250	0.143	5
					2	2	20	300250	0.481	1
8	70891.02778	126581.65129	1.169	0.370	1	7	13	300250	0.102	6
					2	3	17	300250	0.303	2
9	50387.09722	111746.51439	1.102	0.392	1	8	10	300250	0.156	7
					2	4	20	300250	0.177	3
10	100360.70833	141345.74232	1.123	0.445	1	9	10	300250	0.102	8
					2	5	20	300250	0.450	4

Table 1 above shows the data that has been generated by the simulation. The 1st generation serves as the first progenitors to be subjected in the map. In the 2nd generation the best fitness value of 300250 has been obtained and remained throughout the generations. In the 3rd generation stagnation 1 occurs and then by the proceeding 4th generation, the stagnation. that occurs is now 2. In the 5th generation another species has been made. The 6th generation's first species stagnates with a value of 4. In the following generations from generation 7 to 10 it continues to add 1 stagnation value per new generation. It is important to note that during the 2nd generation the AI already knows what the best fitness value is. No species extinctions occurred since both of the species retained a population size at the end of the runtime.

Table 2
Simulation Data for Normal Map

Generation	Population Average Fitness	stdev	Mean Genetic Distance	Standard Deviation	ID	Age	Size	fitness	adj fit	stag
1	6711.19444	36098.10247	0.869	0.214	1	0	30	201105.4	0.033	0
2	6721.52778	36096.20037	0.7969	0.220	1	1	30	201105.4	0.033	1
3	7728.77778	36096.45082	1.003	0.327	1	2	30	201105.4	0.038	2
4	47915.08333	105300.00237	1.066	0.376	1	3	30	300250	0.160	0
5	55782	111020.24786	1.032	0.374	1	4	30	300250	0.186	1
6	72186.37500	126014.91778	1.195	0.368	1	5	30	300250	0.240	2
7	93401.19444	135720.32744	1.045	0.330	1	6	30	300250	0.311	3
8	77576.628056	125719.71438	1.035	0.366	1	7	30	300250	0.258	4
9	51879.90278	111287.31817	1.063	0.372	1	8	30	300250	0.173	5
10	56973.61111	111973.50060	0.934	0.315	1	9	30	300250	0.190	6

Table 2 provides the simulation data for the normal map. The 1st generation obtained and retained its fitness value of 201105.4 all the way to the 3rd generation along with this is that the stagnation values increase with a value of 1. A new fitness value of 300250 is obtained in the 4th generation and resetting the stagnation value back to 0. This smoothly transitions through all the way to the 10th generation only adding 1 stagnation value per generation.

Table 3
Simulation Data for Hard Map

Generation	Population Average Fitness	stdev	Mean Genetic Distance	Standard Deviation	ID	Age	Size	fitness	adj fit	stag
1	29.45614	39.75304	1.188	0.281	1	0	30	184.7	0.152	0
2	2830.63158	12757.55497	1.020	0.283	1	1	30	71294.7	0.040	0
3	17633.94737	68768.02017	1.085	0.276	1	2	30	379263.3	0.046	0
4	45042.57895	112654.33319	1.081	0.252	1	3	30	379263.2	0.119	1
	69352.75439	139241.73146	1.136	0.466	1	4	29	379263.2	0.183	2
					2	0	1	--	--	0
6	57785.91228	126885.56021	1.433	0.602	1	5	28	379263.2	0.158	3
					2	1	3	7.9	0.000	0
7	46814.60102	111796.30214	1.449	0.669	1	6	27	379263.2	0.137	4
					2	2	3	71.6	0.000	0
8	82506.82456	141240.98942	1.510	0.631	1	7	27	379263.2	0.242	5
					2	3	3	71.6	0.000	1
9	80831.52632	138230.51706	1.487	0.629	1	8	27	379263.2	0.237	6
					2	4	3	71.6	0.000	2
10	83120.85965	148682.03125	1.447	0.619	1	9	27	379263.2	0.244	7
					2	5	3	71.6	0.000	3

Table 3 shows the simulation data for the hard map. The 1st generation quickly finished therefore all the corresponding values are all low. In the 2nd generation the AI started to gain traction of the map and obtaining a better fitness value of 71294.7. In the upcoming generations a new fitness value of 379263.2 is obtained. Stagnation started to occur in the 4th generation. A new species is made in the 5th generation and now the population is composed of 2 species. In generations 6 and 7 the second species does immediately stagnate, but the first species continues to stagnate by 1. In the 8th generation the second generation started to stagnate by 1. From there on generations 8 to 10 continue to stagnate and retain a fitness value of 379263.2 for the first species and 71.6 for the second species. The second species also retained an adjusted fitness value of 0.000 starting from generation 7 through 10. No extinctions during is simulation for the hard map occurred since both of the species retained a set number of populations at the end of the runtime. If more generations are to be added in the simulation the possibility of a species being extinct will eventually happen.

Evaluation

The AI all performed well for each given map. The most notable attribute or data to take into consideration is the fitness value as this determines if it will be rewarded based on its performance by passing its genes or features on to the next generation by reproduction and eventually mutating while the ones who did not performed so well will go extinct. In the simulation for easy map a consistent fitness value of 300250 has been obtained for both species within two generations of the run on the other hand the simulation for normal map took four generations to find a consistent fitness value of 300250 and finally for the simulation of the hard map it found a consistent fitness value of 379263.2 on the third generation but its second species has a very low fitness value which indicates that probably the second species is not as effective as the first species. The other key attribute is the stagnation value which indicates the number of the population in the particular species which did not show improvement. This implicates that in the easy map stagnation was very high reaching to a value of 8 on the first species this is because the AI adapted to the map at an early rate therefore the room for improvement became narrower making it stagnate. In the normal and hard map stagnation also occurred implicating that some of the species in that population was not improving.

This shows that as the difficulty of the map increases so does the AI's ability to adapt to that certain difficulty. The AI on the easy map easily adapted to it while the AI in the two other maps took more time to get used to their environment. This implies that the performance of the self-learning artificial neural network to adapt to that difficulty is highly dependent on what it encounters in its given environment so if it encounters less obstacles or simple paths in its way the easier to adapt and if there are more obstacles or multiple unfamiliar paths in its way the harder it is for it to adapt. This shows that a self-learning artificial neural network is an effective way to train the AI to perform their designated actions as they displayed these actions according to what their inputs acquire during the simulation.

IV. CONCLUSION

It can be concluded that by using a genetic algorithm specifically NeuroEvolution of Augmenting Topologies (NEAT) that it is effective in training self-driving AI cars in a simulation. The algorithm displayed solutions for the given problems based on what it encounters in its given maps. This shows that this genetic algorithm is fast to learn and fast to optimize a solution for a complex problem.

V. REFERENCES

[1] Stanley and Miikkulainem (2002). "Efficient Evolution of Neural Network Topologies" pp 2-3.

[2] CodeReclaimers, LLC (2018, October 13) NEAT-Python Documentation. https://neat-python.readthedocs.io/en/latest/config_file.html

[3] Deep Learning with Python (2018) http://silverio.net.br/heitor/disciplinas/eeica/papers/Livros/[Chollet]-Deep_Learning_with_Python.pdf

[4] Python for Data Analytics, Scientific and Technical Applications (2019)

https://ieeexplore.ieee.org/abstract/document/8701341

[5] Interface to the Keras Deep Learning Library. Journal of Open Source Software (2017)

https://joss.theoj.org/papers/10.21105/joss.00296.pdf

[6] https://thereaderwiki.com/en/Python_(programming_language)

[7] Python, PyGame, and Raspberry Pi Game Development (2019)

https://link.springer.com/book/10.1007%2F978-1-4842-4533-0

[8] A high fidelity traffic simulation model based on cellular automata and car-following concepts (2004)

https://www.sciencedirect.com/science/article/abs/pii/S0968090X03000573

[9] Development of a fuzzy logic based microscopic motorway simulation model. Proceedings of Conference on Intelligent Transportation Systems (1998)

https://ieeexplore.ieee.org/abstract/document/660454

[10] Deepdriving: Learning affordance for direct perception in autonomous driving (2015)

https://openaccess.thecvf.com/content_iccv_2015/papers/Chen_DeepDriving_Learning_Affordance_ICCV_2015_paper.pdf

[11] Genetic Algorithm Implementation in Python(2005)

https://ieeexplore.ieee.org/abstract/document/1515367

[12] Self-Driving Cars (2017)

https://ieeexplore.ieee.org/abstract/document/8220479/

[13] Ontology based Scene Creation for the Development of Automated Vehicles (2018)

https://ieeexplore.ieee.org/abstract/document/8500632/

[14] Self-Driving Cars and the Urban Challenge in IEEE Intelligent Systems (2008)

https://ieeexplore.ieee.org/abstract/document/4475861/

[15] Python for Data Analytics, Scientific and Technical Applications (2019)

https://ieeexplore.ieee.org/abstract/document/8701341/

[16] Implementation of the NEAT Policy Manager (2019)

https://www.diva-portal.org/smash/record.jsf?pid=diva2:1281308

[17] Report - Public Perceptions of Self Driving Cars (2017)

https://www.ocf.berkeley.edu/~djhoward/reports/Report%20-%20Public%20Perceptions%20of%20Self%20Driving%20Cars.pdf

[18] Creativity and Artificial Intedligence (1998)

https://www.sciencedirect.com/science/article/pii/S0004370298000551

[19] On a Formal Model of Safe and Scalable Self-driving Cars (2017)

https://arxiv.org/abs/1708.06374

[20] Towards the first adversarially robust neural network model on MNIST (2018)