FINAL REPORT

DINO GAME USING ARTIFICIAL INTELLIGENCE

SUBMITTED BY

UMANG KUMAR GUPTA [1BCE2461]
SHADAB ALAM [18BCE2491]

TO

Anitha A.

CSE3013 : ARTIFICIAL INTELLIGENCE

Slot: E2+TE2

VIT – VELLORE CAMPUS

FALL SEMESTER 2020 – 2021

Abstract

In this project we have utilized the concepts of Genetic Algorithm compiled with Feed Forward Neural Network. The game we have developed is based on the Google Chrome offline dino, However, in our game the player learns every time it collides and dies. The next gen of players is reproduced on the basis of the fitness function. Every player has 8 set of attributes which are used to define their fitness and are used to improve the gameplay.

Some set of players are better than others and reproduce better thus the game is programmed to show the score of the best player of each iteration.

Keywords – Fitness function, Mutation, Neural Network, Genetic Algorithm.

Introduction

Artificial Intelligence is a way to enable people to accomplish more by collaborating with smart software. AI is the kind of technology that can learn from the vast amounts of data that are available in the modern world, understand our kind of language and respond accordingly, and can see and interpret the world the way that we do.

There are different types of AI that vary based on factors such as environment scope, complexity of tasks, knowledge base and performance. The different types are:

- **Reactive Machines AI**

The fundamental types of artificial intelligence systems are quite reactive and they are not able to use previous experiences to advise current decisions and to configure memories. IBM's chess-playing computer called Deep Blue defeated Garry Kasparov who is an international grandmaster in chess in the late 1990s, is one example of this type of machine.

- **Limited Memory AI**

As the name suggests, the memory in such systems is short lived. Such systems can go back to their past for a short time and learn from it. Limited memory AI is mostly

used in self-driving cars. They will detect the movement of vehicles around them constantly. The static data such as lane marks, traffic lights and any curves in the road will be added to the AI machine. This helps autonomous cars to avoid getting hit by a nearby vehicle. Nearly, it will take 100 seconds for an AI system to make considered decisions in self-driving.

- **Theory of Mind AI**

Theory of mind artificial intelligence is a very advanced technology. In terms of psychology, the theory of mind represents the understanding of people and things in the world that can have emotions which alter their own behavior. Still, this type of AI has not been developed completely in the society. But research shows that the way to make advancements is to begin by developing robots that can identify eye and face movements and act according to the looks.

- **Self-aware AI**

Self-aware AI is a supplement of the theory of mind AI. This type of AI is not developed yet, but when it happens, it can configure representations about themselves. It means particular devices are tuned into cues from humans like attention spans, emotions and also able to display self-driven reactions.

- **Artificial Narrow Intelligence (ANI)**

ANI is the most common technology that can be found in many aspects of our daily life. We can find this in smartphones like Cortana and Siri that help users to respond to their problems on request. This type of artificial intelligence is referred to as ‘weak AI’ because it is not strong enough as we need it to be.

- **Artificial General Intelligence (AGI)**

When an AI’s ability to mimic human intelligence and/or behaviour is indistinguishable from that of a human, it’s called AGI (also known as Strong AI or

Deep AI). Most of the robots are ANI, but few are AGI or above. Pillo robot is an example of AGI which answers to all questions with respect to the health of the family. It can distribute pills and give guidance about their health.

- **Artificial Super Intelligence (ASI)**

When an AI doesn't mimic human intelligence and/or behaviour but surpasses it, it's called ASI. ASI is something we can only speculate about. It would surpass all humans at all things: maths, literature, medicine and almost every other aspect of life. To count as an ASI, the AI would have to be capable of things we believe humans will always be able to do better than machines, such as relationships and the arts.

Currently, there are a large number of Weak AI developed to perform specific tasks in different fields including medical diagnosis, electronic trading platforms, robot control, and remote sensing. AI has been used to develop and advance numerous fields and industries, including finance, healthcare, education, transportation, and more.

For this project, we have used the following AI techniques:

- **Backpropagation**

In this technique, an error is computed at the output and distributed backwards throughout the network's layers. It is commonly used to train deep neural networks.

- **Feedforward Neural networks**

Neural networks are computing systems inspired by biological neural networks that constitute animal brains. Neural networks are frameworks for many different machine learning algorithms to work together and process complex data inputs. A feedforward neural network is a neural network wherein connections between the nodes do not form a cycle. Feedforward neural networks are the simplest type of artificial neural networks.

• **Genetic algorithm**

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction. The fitness function of a genetic algorithm determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Related Works

• In the 1990s, IBM developed a chess-playing machine called the Deep Blue based on a series of machines that were a multi-year effort to build a world class chess machine. There were two versions of the Deep Blue (1996 and 1997), both of which competed with the world grandmaster of that time, Garry Kasparov, who lost only to the latter. Deep Blue depended on a "brute force technique": the system relied more on its ability to search millions of examples and evaluate millions of possibilities in a few minutes than on its ability to reason. Specifically, Deep Blue used an approach that looked forward several moves for each reasonable "next move" and then chose the move that would yield the highest number of points. [Deep Blue by Murray Campbell, A. Joseph Hoane Jr. , Feng-hsiung Hsu]

• In 2004, IBM began to consider developing an application that could play Jeopardy!, a very popular TV game that draws large viewing audiences and offers some real challenges for a computer. In Jeopardy! contestants are given "answers" and asked to come up with the "question" that would lead to such an answer. The "questions" and "answers" used on Jeopardy! are drawn from a broad base of general knowledge on topics such as history, literature, science, etc. Moreover, the game format requires that the contestants be able to consider the "answers" provided, which are often subtle, ironic, or contain riddles, and generate responses within about
3 seconds.

A Jeopardy!-playing application posed two different problems: understanding natural language well enough to be able to identify the right "answer" and then

searching a huge database of general information for a “question” that fits the “answer.” Searching a huge database quickly was a more or less physical problem, but “hearing” and then “understanding” spoken English, and finally determining which of several possible answers was the right match for the question being asked, were serious cognitive problems.

In 2007, IBM established a team to develop a machine that could play Jeopardy!. The first version was ready in 2008 and in February of 2010, the software application (named Watson) proved it could beat two former Jeopardy! winners.

The key to Watson’s analytic functionality is DeepQA (Deep Question Analytics), a massively parallel probabilistic architecture that uses and combines more than 100 different techniques—a mixture of knowledge and neural network techniques—to analyze natural language, identify sources, find and generate hypotheses, and then evaluate evidence and merge and rank hypotheses. In essence, DeepQA can perform thousands of simultaneous tasks in seconds to provide answers to questions. Given a specific query, Watson might decompose it and seek answers by activating hundreds or thousands of threads running in parallel.

Watson maintained all its data in memory to help provide the speed it needed for Jeopardy!. It had 16 terabytes of RAM. It used 90 clustered IBM Power 750 servers with 32 cores running at 3.55 GHz. The entire system runs on Linux and operates at over 80 teraflops (i.e., 80 trillion operations per second).

IBM demonstrated that AI-based natural language analysis and generation had reached the point where a system like Watson could understand open-ended questions and respond in real time. Watson examined Jeopardy! “answers,” defined what information was needed, accessed vast databases to find the needed information. It then generated an English response in under 3 seconds. It did it faster and better than two former human Jeopardy! winners and easily won the match.

[Turing’s misunderstood imitation game and IBM’s Watson success by Huma Shah]

Proposed Methodology

We have used Genetic Algorithm to train our neural network

There are Five phases in a genetic algorithm:

1. Creating an Initial population
2. Defining a Fitness function
3. Selecting the parents
4. Making a Crossover
5. Mutation

Creating an Initial Population

In this step, we create a set of *n* elements which is called a Population. Each element from the population is a solution to the problem you want to solve.

In our case, let this population be:

- bahama
- abcdef
- ijklmn
- ….
- All other 6 character words
- mnopqr
- stuvwx
- cabana

Defining a Fitness Function

The fitness function determines how likely an individual is fit to be selected for reproduction, and this is based on its fitness score. Let's suppose our fitness function will assign a fitness score or a probability percentage to each element from the population, for each character matching our target word *banana.*

In our case, let this population be:

Elements, Fitness Score

- Player 1, 100
- Player 2, 200
- Player 3, 95.1

Selecting the Parents

The idea behind this step is to select the fittest individuals and let them pass their genes to the next generation. Two elements of the population are selected based on their fitness scores. In our case, we select individuals with high fitness scores.

In our case, we selected these elements as these words have a high fitness score from the given population.

Elements, Fitness Score

- Player 1, 100
- Player 2, 200

Making a Crossover

It is the most significant phase in a genetic algorithm. In this step, we reproduce a new population of *n* elements from the selected elements. In this step, we have to

permute and combine as many possible words from the characters obtained from the two parent words that were selected in the previous step. Thus the 2 players are used and all their characteristics are permuted and combined to form n number of players.

This then results in a new population which is more fit in comparison to the old one.

Making a Mutation

There are chances that from the crossover phase, we might get a population which will not contribute to the evolution of a new diverse population and our algorithm will converge prematurely. So we need to alter the sequence of words from 1% of the newly created population to maintain this diversity. We can choose any sort of alteration.

For example, suppose from 1% of the previous population we get Players which have different abilities. Some may hold the jump key and try to make things work while other may think let's just run straight ahead. Some think ducking would save them. Sometimes these players are also made after the reproduction process and are no longer useful to the evolution. Thus comes mutation into the picture.

Let's take an example :

For example, we have a population of words, suppose from 1% of the previous population we get words like '***banyan***', and '***yanbac***'. Now we select these elements for creating a new population as these words have good fitness scores of 45 and 49 respectively, and thus have a high probability of being parents. Now if we pick the last 3 and first 3 letters from these two words and combine them, we will get
'***yanyan***' and this word is no longer productive enough to get any new diverse elements.

But if we mutate our 1% of the previous population and alter the letters of '***banyan***' and '***yanbac***' by simply flipping the first and last letters in both words, we get

'***nanyab***' and '***canbay***'. Now if we apply the same combination of thelast 3 and first

3 letters of the mutated elements, we get '***yabcan***' which is quite diverse from '***yanyan***'. The flipping can be done in any random way and a result can be produced.

When does this process stop?

Our population has a fixed size. As new elements are formed, old elements with low fitness score are removed. When the population has converged, i.e., no new elements are reproduced which are significantly different from the previous population, then we may say that the genetic algorithm has provided a set of solutions to our problem.

We have a converged set, i.e., no matter how many times we repeat the entire above process, we are going to get only these set of elements. This can be evident at later stages when all the player perfect their jump to distance ration and jump in synchronous manner. There can't be seen any shadow player who is jumping at different speed. As we progress in the game the algorithm stops as the player no longer dies and has almost perfected the game.

Pseudocode

- *START*

  —create the initial population

- *Compute fitness • REPEAT*

  —Selection

  —Crossover

  —Mutation

  —Compute fitness

- *UNTIL population has converged*

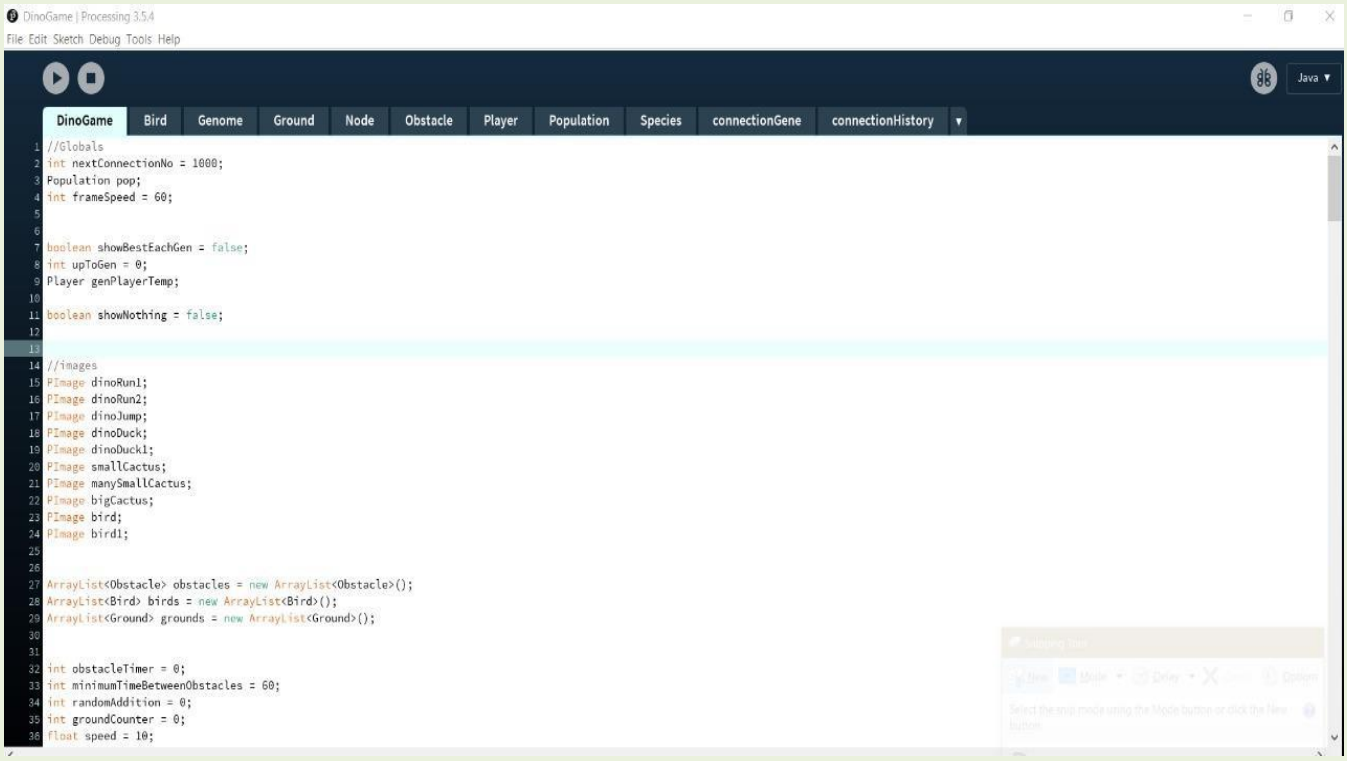
- *STOP*

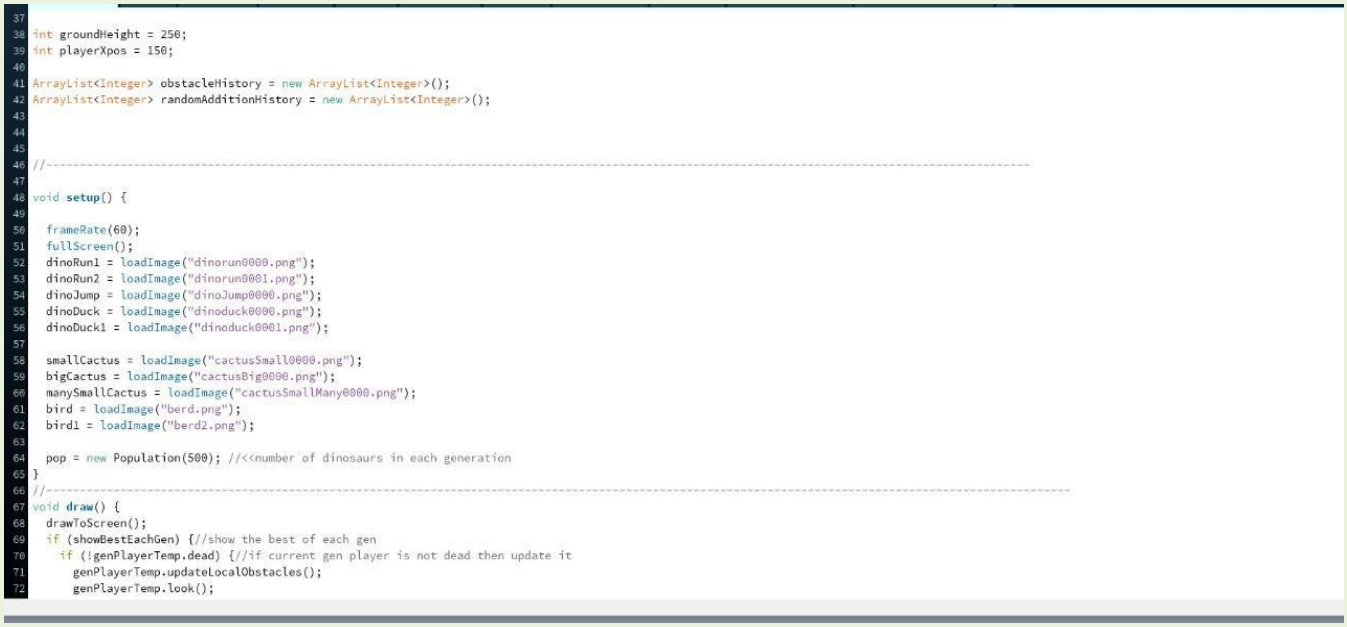
We can implement Genetic Algorithms to learn the best hyper-parameters for a Neural Network. To learn the hyper-parameters, we apply Genetic Algorithms as described in the steps below:

- Create a population of several Neural Networks
- Assign hyper-parameters randomly to all the Neural Networks
- Repeat the following

1. Train all the Neural Networks.
2. Calculate their training cost (Ex- training error and regularization terms)
3. From the cost of previous Neural Networks, calculate a fitness score from that set of hyper-parameters. The best Neural Networks will have the lowest cost. So, its inverse will give a high fitness value
4. Select the two best Neural Networks based on their fitness
5. Reproduce new Neural Networks from these
6. Mutate the genes of the child
7. Perform steps 5–7 for all the Neural Networks in the population. At the end of the latest generation, we have the optimum hyper-parameters.

Code-


DinoGame | Processing 3.5.4
File Edit Sketch Debug Tools Help
Java
DinoGame
Bird
Genome
Ground
Node
Obstacle
Player
Population
Species
connectionGene
connectionHistory
//Globals
int nextConnectionNo = 1000;
Population pop;
int frameSpeed = 60;
boolean showBestEachGen = false;
int upToGen = 0;
Player genPlayerTemp;
boolean showNothing = false;
//images
PImage dinoRun1;
PImage dinoRun2;
PImage dinoJump;
PImage dinoDuck;
PImage dinoDuck1;
PImage smallCactus;
PImage manySmallCactus;
PImage bigCactus;
PImage bird;
PImage bird1;
ArrayList<Obstacle> obstacles = new ArrayList<Obstacle>();
ArrayList<Bird> birds = new ArrayList<Bird>();
ArrayList<Ground> grounds = new ArrayList<Ground>();
int obstacleTimer = 0;
int minimumTimeBetweenObstacles = 60;
int randomAddition = 0;
int groundCounter = 0;
float speed = 10;


int groundHeight = 250;
int playerXpos = 150;
ArrayList<Integer> obstacleHistory = new ArrayList<Integer>();
ArrayList<Integer> randomAdditionHistory = new ArrayList<Integer>();
//---
void setup() {
frameRate(60);
fullScreen();
dinoRun1 = loadImage("dinorun0000.png");
dinoRun2 = loadImage("dinorun0001.png");
dinoJump = loadImage("dinoJump0000.png");
dinoDuck = loadImage("dinoduck0000.png");
dinoDuck1 = loadImage("dinoduck0001.png");
smallCactus = loadImage("cactusSmall0000.png");
bigCactus = loadImage("cactusBig0000.png");
manySmallCactus = loadImage("cactusSmallMany0000.png");
bird = loadImage("berd.png");
bird1 = loadImage("berd2.png");
pop = new Population(500); //<<number of dinosaurs in each generation
}
//---
void draw() {
drawToScreen();
if (showBestEachGen) {//show the best of each gen
if (!genPlayerTemp.dead) {//if current gen player is not dead then update it
genPlayerTemp.updateLocalObstacles();
genPlayerTemp.look();

```
      genPlayerTemp.update();
      genPlayerTemp.show();
    } else {//if dead move on to the next generation
      upToGen ++;
      if (upToGen >= pop.genPlayers.size()) {//if at the end then return to the start and stop doing it
        upToGen= 0;
        showBestEachGen = false;
      } else {//if not at the end then get the next generation
        genPlayerTemp = pop.genPlayers.get(upToGen).cloneForReplay();
      }
    }
  } else {//if just evolving normally
    if (!pop.done()) {//if any players are alive then update them
      updateObstacles();
      pop.updateAlive();
    } else {//all dead
      //genetic algorithm
      pop.naturalSelection();
      resetObstacles();
    }
  }
}



//--------------------------------------------------------------------------------------------------------------------------------------------------------
//draws the display screen
void drawToScreen() {
  if (!showNothing) {
    background(250);
    stroke(0);
    strokeWeight(2);
    line(0, height - groundHeight - 30, width, height - groundHeight - 30);
    drawBrain();
    writeInfo();
  }
```

DinoGame | Bird | Genome | Ground | Node | Obstacle | Player | Population | Species | connectionGene | connectionHistory

```
//--------------------------------------------------------------------------------------------------------------------------------------------------------
void drawBrain() {  //show the brain of whatever genome is currently showing
  int startX = 600;
  int startY = 10;
  int w = 600;
  int h = 400;
  if (showBestEachGen) {
    genPlayerTemp.brain.drawGenome(startX, startY, w, h);
  } else {
    for (int i = 0; i< pop.pop.size(); i++) {
      if (!pop.pop.get(i).dead) {
        pop.pop.get(i).brain.drawGenome(startX, startY, w, h);
        break;
      }
    }
  }
}
//--------------------------------------------------------------------------------------------------------------------------------------------------------
//writes info about the current player
void writeInfo() {
  fill(200);
  textAlign(LEFT);
  textSize(40);
  if (showBestEachGen) { //if showing the best for each gen then write the applicable info
    text("Score: " + genPlayerTemp.score, 30, height - 30);
    //text(, width/2-180, height-30);
    textAlign(RIGHT);
    text("Gen: " + (genPlayerTemp.gen +1), width -40, height-30);
    textSize(20);
    int x = 580;
    text("Distace to next obstacle", x, 18+44.44444);
    text("Height of obstacle", x, 18+2*44.44444);
    text("Width of obstacle", x, 18+3*44.44444);
    text("Bird height", x, 18+4*44.44444);
    text("Speed", x, 18+5*44.44444);
    text("Players Y position", x, 18+6*44.44444);
```

DinoGame | Bird | Genome | Ground | Node | Obstacle | Player | Population | Species | connec

```
    frameRate(frameSpeed);
    println(frameSpeed);
    break;
  case '-'://slow down frame rate
    if (frameSpeed > 10) {
      frameSpeed -= 10;
      frameRate(frameSpeed);
      println(frameSpeed);
    }
    break;
  case 'g'://show generations
    showBestEachGen = !showBestEachGen;
    upToGen = 0;
    genPlayerTemp = pop.genPlayers.get(upToGen).cloneForReplay();
    break;
  case 'n'://show absolutely nothing in order to speed up computation
    showNothing = !showNothing;
    break;
  case CODED://any of the arrow keys
    switch(keyCode) {
    case RIGHT://right is used to move through the generations
      if (showBestEachGen) {//if showing the best player each generation then move on to the next generation
        upToGen++;
        if (upToGen >= pop.genPlayers.size()) {//if reached the current generation then exit out of the showing
          showBestEachGen = false;
        } else {
          genPlayerTemp = pop.genPlayers.get(upToGen).cloneForReplay();
        }
        break;
      }
      break;
    }
  }
}
//--------------------------------------------------------------------------------------------------------------------------
//called every frame
```

DinoGame | Bird | Genome | Ground | Node | Obstacle | Player | Population | Species | connectionGene | connectionHistory

```
      birds.remove(i);
      i--;
    }
  }
  for (int i = 0; i < grounds.size(); i++) {
    grounds.get(i).move(speed);
    if (grounds.get(i).posX < -playerXpos) {
      grounds.remove(i);
      i--;
    }
  }
}
//--------------------------------------------------------------------------------------------------------------------------
//every so often add an obstacle
void addObstacle() {
  int lifespan = pop.populationLife;
  int tempInt;
  if (lifespan > 1000 && random(1) < 0.15) { // 15% of the time add a bird
    tempInt = floor(random(3));
    Bird temp = new Bird(tempInt);//floor(random(3)));
    birds.add(temp);
  } else {//otherwise add a cactus
    tempInt = floor(random(3));
    Obstacle temp = new Obstacle(tempInt);//floor(random(3)));
    obstacles.add(temp);
    tempInt+=3;
  }
  obstacleHistory.add(tempInt);

  randomAddition = floor(random(50));
  randomAdditionHistory.add(randomAddition);
  obstacleTimer = 0;
}
//--------------------------------------------------------------------------------------------------------------------------
//what do you think this does?
void showObstacles() {
```

DinoGame | Bird | Genome | Ground | Node | Obstacle | Player | Population | Species | conne

```
    tempInt+=3;
  }
  obstacleHistory.add(tempInt);

  randomAddition = floor(random(50));
  randomAdditionHistory.add(randomAddition);
  obstacleTimer = 0;
}
//-----------------------------------------------------------------------------------------------------------------
//what do you think this does?
void showObstacles() {
  for (int i = 0; i< grounds.size(); i++) {
    grounds.get(i).show();
  }
  for (int i = 0; i< obstacles.size(); i++) {
    obstacles.get(i).show();
  }

  for (int i = 0; i< birds.size(); i++) {
    birds.get(i).show();
  }
}

//-----------------------------------------------------------------------------------------------------------------
//resets all the obstacles after every dino has died
void resetObstacles() {
  randomAdditionHistory = new ArrayList<Integer>();
  obstacleHistory = new ArrayList<Integer>();

  obstacles = new ArrayList<Obstacle>();
  birds = new ArrayList<Bird>();
  obstacleTimer = 0;
  randomAddition = 0;
  groundCounter = 0;
  speed = 10;
}
```

Result and Discussion

The result of the implemented neural network has gives us an application which uses the neural network and genetic algorithm to train and develop accurate players who can play good and get better each and every turn without human interaction involved in the learning process.

In this game we have initialized 500 players in the beginning and as each iteration or generation of the game goes on until all the 500 players have died. When we reach this situation we start the next generation of the game where all the 500 players start again but with what knowledge they have acquired in the previous generations, this

process continues until all of the 500 players reach a generation where they can all accurately and parallelly take the correct decisions in the game to reach the end and win completely.

Thus, we implement a learning Artificial intelligence which improves itself at each turn without having the inner working of the game.

Below are the running screenshots of the game with their respective generation number mentioned in the corner:

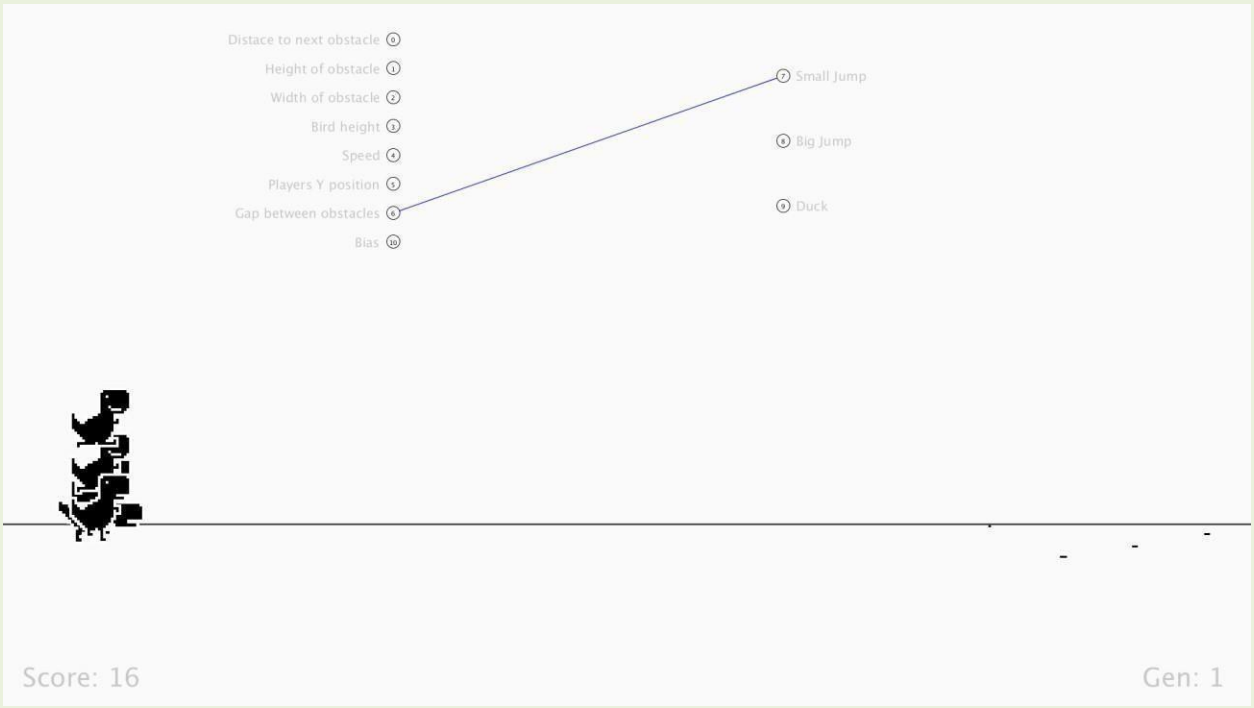


Figure 1.0 **Generation 1**

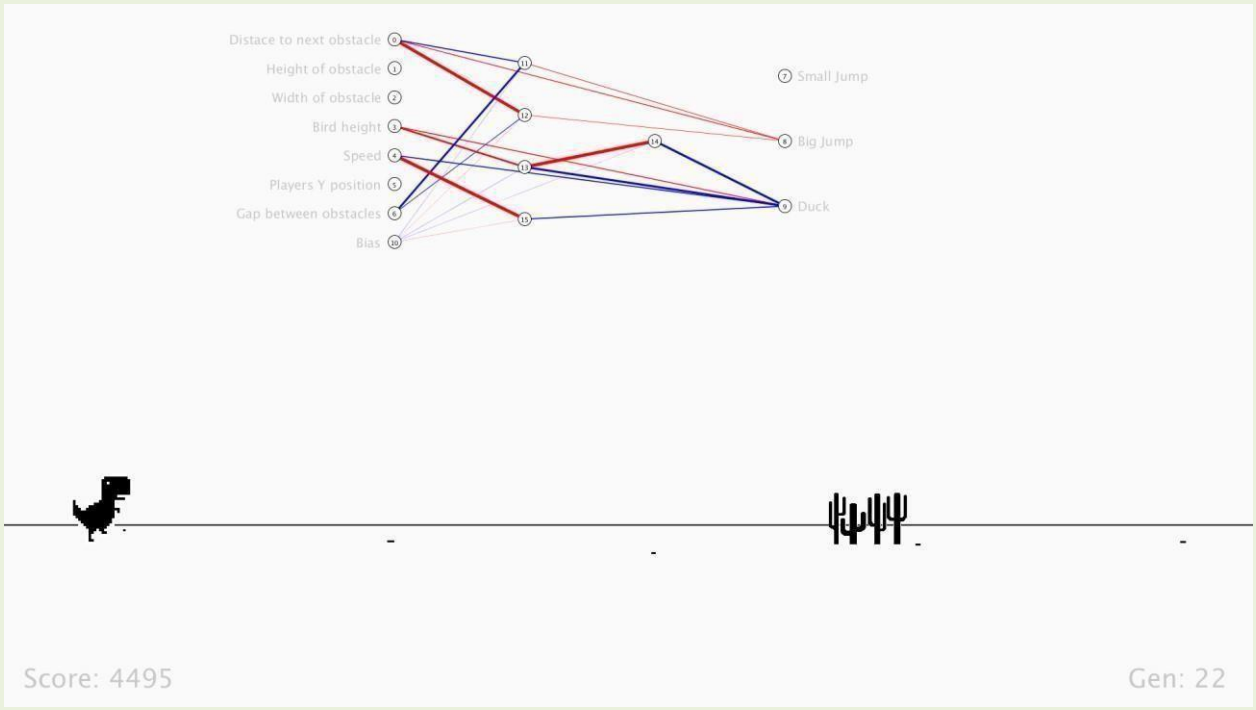


Figure 2.0 **Generation 22**

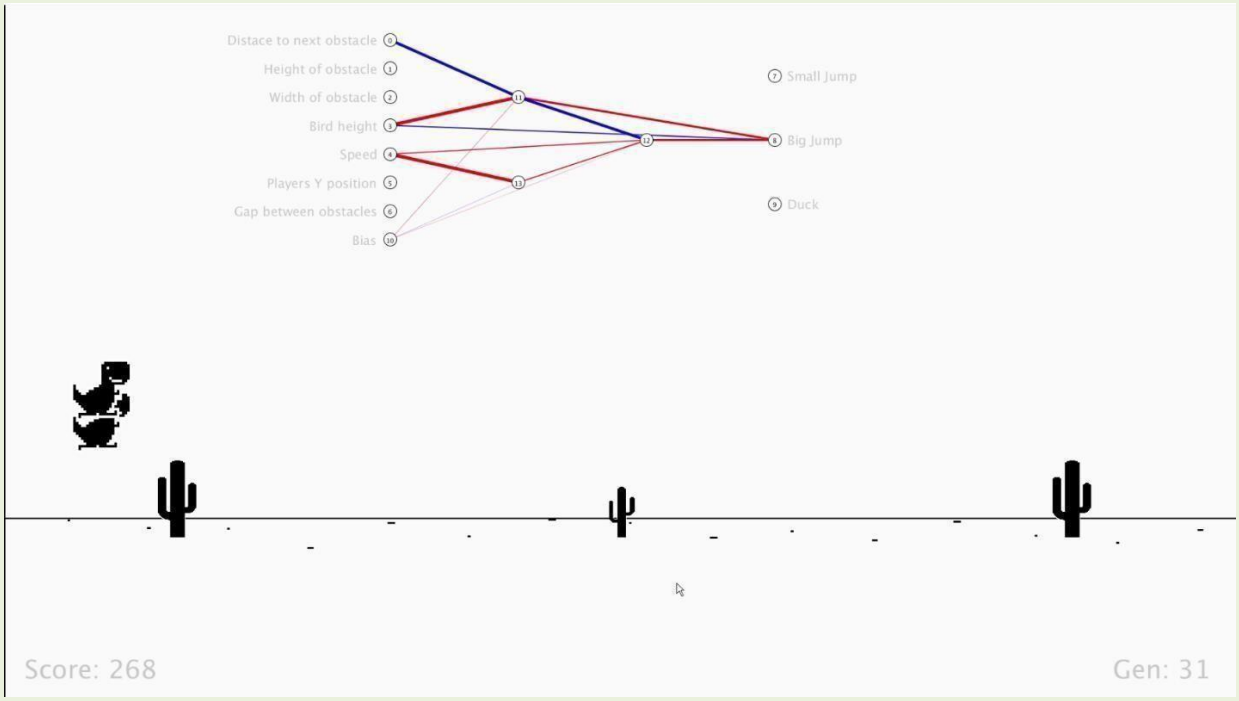


Figure 3.0 **Generation 31**

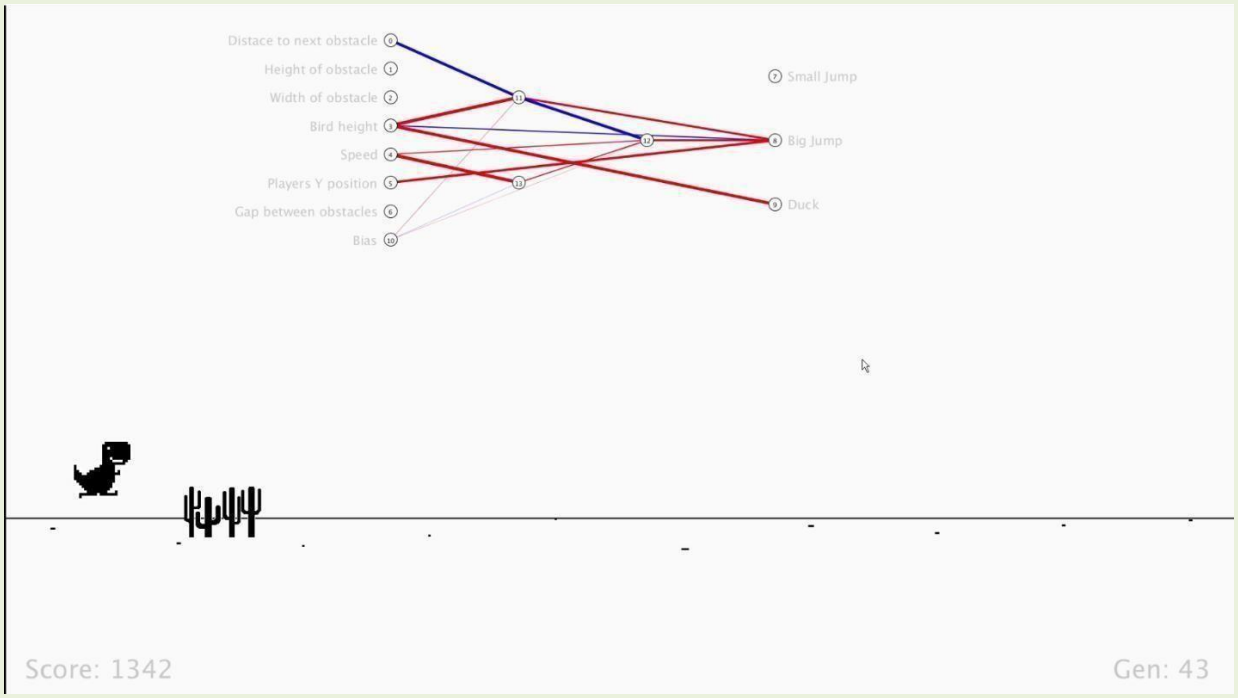


Figure 3.0 **Generation 43**

In figure 1.0, the game is started and shows generation 1. The neural network visible above is the network of the player who is considered best at that moment by the fitness function. It also can be seen that there are multiple player playing the game at that time.

In figure 2.0, it can be seen that the best player of the game has evolved drastically and has reached a high score of 4900+. However, it cannot be said that there is only one player at that moment. There could be multiple player playing but they may be overlapping each other. Moreover, the player hasn't perfected itself yet, it could still die from the speeding of the game.

This iteration process continues until all the players master the game completely, as you can see in the screenshots, they are still learning in the 41st generation of the game cycle.

Conclusion and Future Work

The future work would be to be able to save the progress of each and every player and into a csv file for future use. When the program boots up again the program should be able to read the progress and the game should be able to start from where it was left off. This would allow to monitor the progress of the AI for larger amount of time and help saving computational costs as well.

The work done in this project is only preliminary and the neural network and the bias system could be perfected to provide results at an earlier generation. This would also help to stop the wastage of memory.

References

[1] G. Bradski et al. The opencv library. Doctor Dobbs Journal, 25(11):120–126, 2000.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[3] Baker, J. (1985). Adaptive selection methods for genetic algorithms. In Proceedings of the International Conference on Genetic Algorithms and Their Applications, ed. J. Grefenstette. Lawrence Erlbaum, Hillsdale, NJ.

[4] G. Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3):58–68, 1995.

[5] Shah, Huma. "Turing’s misunderstood imitation game and IBM’s Watson success." Keynote in 2nd Towards a Comprehensive Intelligence test (TCIT) symposium at AISB. 2011.

[6] Campbell, Murray, A. Joseph Hoane Jr, and Feng-hsiung Hsu. "Deep blue." Artificial intelligence 134.1-2 (2002): 57-83.

[7] Brockington, Mark G. "A taxonomy of parallel game-tree searchalgorithms." ICGA Journal 19.3 (1996): 162-174.

[8] Whitley, Darrell. "A genetic algorithm tutorial." Statistics and computing 4.2 (1994): 65-85.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[10] Baker, J. (1987). Reducing bias and inefficiency in the selection algorithm. In Genetic Algorithms and Their Applications: Proceedings of the Second International Conference, ed. J. Grefenstette. Lawrence Erlbaum.

[11] Goldberg, D. (1987). Simple genetic algorithms and the minimal, deceptive problem. In Genetic Algorithms and Simulated Annealing, ed. L. Davis. Pitman, London.