Training an AI to make a humanoid body to stand using Genetic Algorithm

Submitted by

Adithya Jayan | 181EC102

Anvith M | 181EC105

Under the Guidance of

Prof. Shyam Lal

Department of Electronics and Communication Engineering,

NITK Surathkal

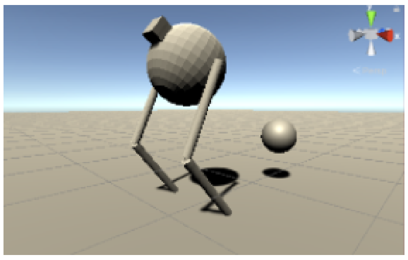
Date of Submission: 27-11-2021

National Institute of Technology Karnataka, Surathkal

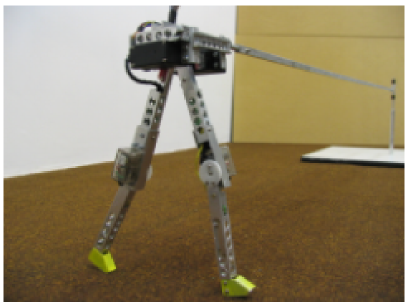
Contents

1 Introduction

The idea of the project was to create a simulation of a 3D shape with a fixed number of limbs and joints, and then train an AI (via Genetic Algorithm) to learn how to make the body stand upright. This model can then be used on a physical body with similar structure to allow it to stand using physical actuators.



(a) Simulated model



(b) Real robot (https://en.wikipedia.org/wiki/RunBot)

Figure 1.1: Virtual simulation and possible robot body design for later application

2 Implementation Methodology

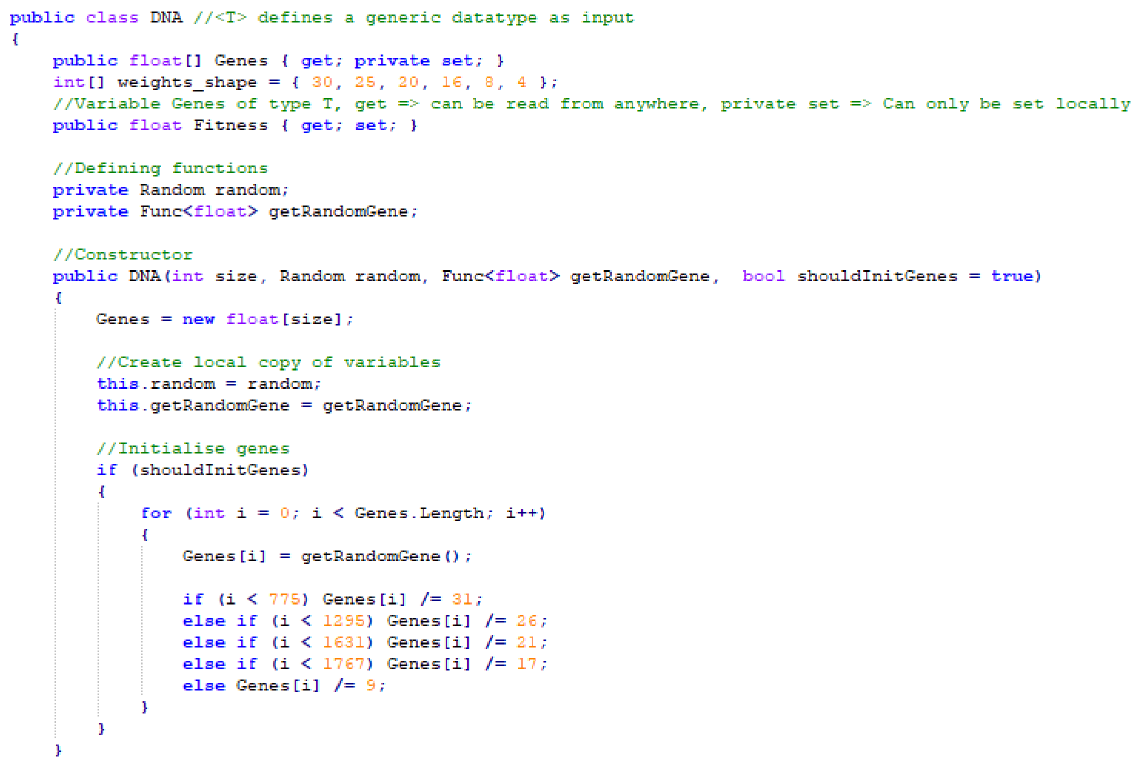
- The planned model is a bipedal body with a sphere for the torso. Because both limbs have knee joints, the total number of actuators is four (two for knees, two for torso-limb contact).
- The implementation was done in C#, and the simulation physics was done using the Unity Game Engine.
- The rotation and speed values of the torso and legs are taken as the inputs to the neural network.
- The magnitude and direction of force that the actuators must provide will be the outputs.

3 Genetic Algorithm

Genetic algorithm makes use of a population of individuals with specific characteristics (DNA) competing. The individuals with the most promising results are used to generate the next generation of individuals through crossover and mutation. Thus the performance of the population improves over multiple generations.

- The brain will be a neural network initialised with random weights.
- Individuals of each generation would be penalised if the torso touches the ground. Scoring is a combination of survival duration, standing duration and height ,and factors such as planted feet and floor contact.
- The most fit individuals form the basis for the next generation.

3.1 DNA and Population



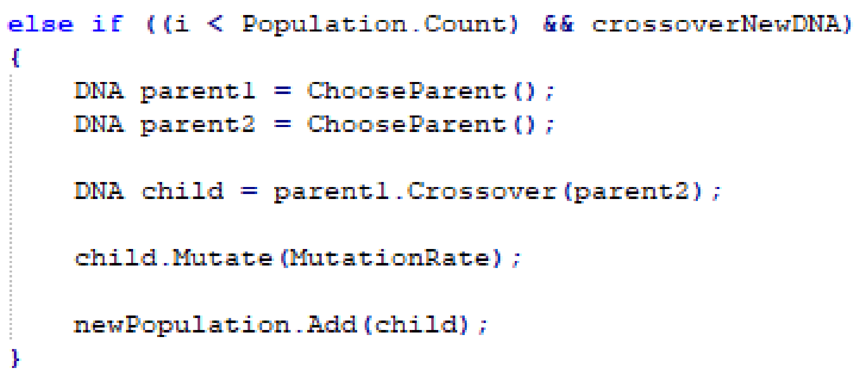
```
public class DNA //<T> defines a generic datatype as input
{
    public float[] Genes { get; private set; }
    int[] weights_shape = { 30, 25, 20, 16, 8, 4 };
    //Variable Genes of type T, get => can be read from anywhere, private set => Can only be set locally
    public float Fitness { get; set; }

    //Defining functions
    private Random random;
    private Func<float> getRandomGene;

    //Constructor
    public DNA(int size, Random random, Func<float> getRandomGene,  bool shouldInitGenes = true)
    {
        Genes = new float[size];

        //Create local copy of variables
        this.random = random;
        this.getRandomGene = getRandomGene;

        //Initialise genes
        if (shouldInitGenes)
        {
            for (int i = 0; i < Genes.Length; i++)
            {
                Genes[i] = getRandomGene();

                if (i < 775) Genes[i] /= 31;
                else if (i < 1295) Genes[i] /= 26;
                else if (i < 1631) Genes[i] /= 21;
                else if (i < 1767) Genes[i] /= 17;
                else Genes[i] /= 9;
            }
        }
    }
```

Figure 3.1: DNA class and constructor

The DNA is the values/weights that run the simulation's brain. The ability of the body to stand and balance depends on the values present here.

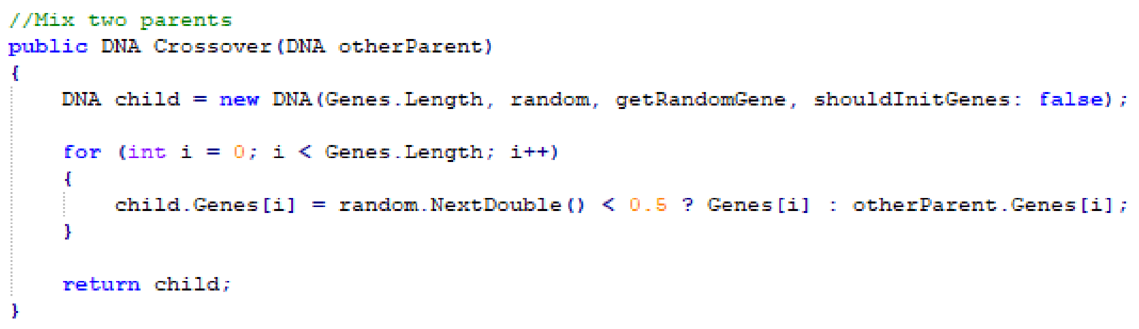
The DNA is a list of 1803 float values (Genes), and are used as weights for forward propagation by the brain of each individual. The DNA values are initially generated randomly, but over generations become better suited for the task through crossover and mutation.

3.2 Mutation and Crossing



```
else if ((i < Population.Count) && crossoverNewDNA)
{
    DNA parent1 = ChooseParent();
    DNA parent2 = ChooseParent();

    DNA child = parent1.Crossover(parent2);

    child.Mutate(MutationRate);

    newPopulation.Add(child);
}
```

(a) Function call for crossover



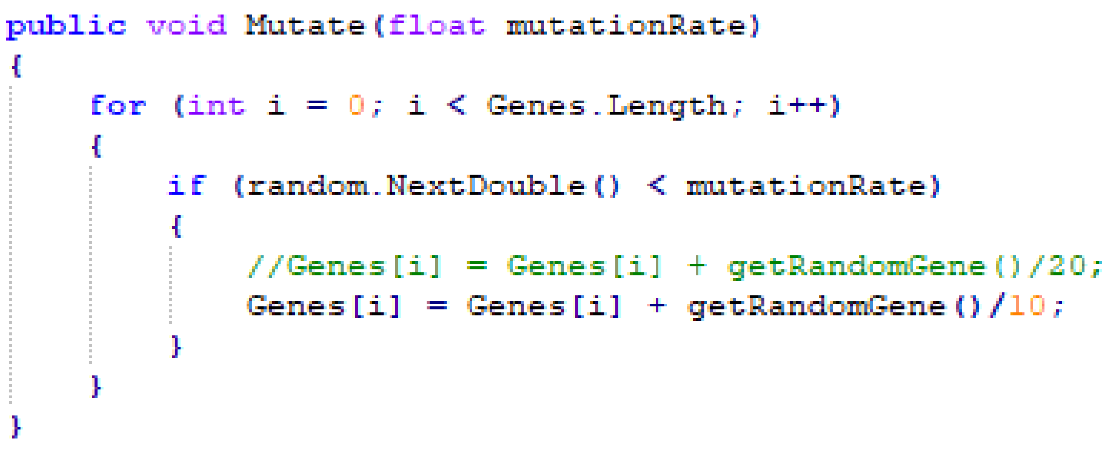
```
//Mix two parents
public DNA Crossover(DNA otherParent)
{
    DNA child = new DNA(Genes.Length, random, getRandomGene, shouldInitGenes: false);

    for (int i = 0; i < Genes.Length; i++)
    {
        child.Genes[i] = random.NextDouble() < 0.5 ? Genes[i] : otherParent.Genes[i];
    }

    return child;
}
```

(b) Function Definition

Figure 3.2: Gene Crossover

Each generation is obtained through the crossover of the best individuals of the previous generation. The best is classified based on the fitness and the *eliteness* parameter.

In our simulation, *eliteness* was set to 10. i.e.. the top 10 members of the generation were carried over without change to the next generation, and the remaining 90 members of the population were produced by crossing the elite 10 among themselves followed by mutation.

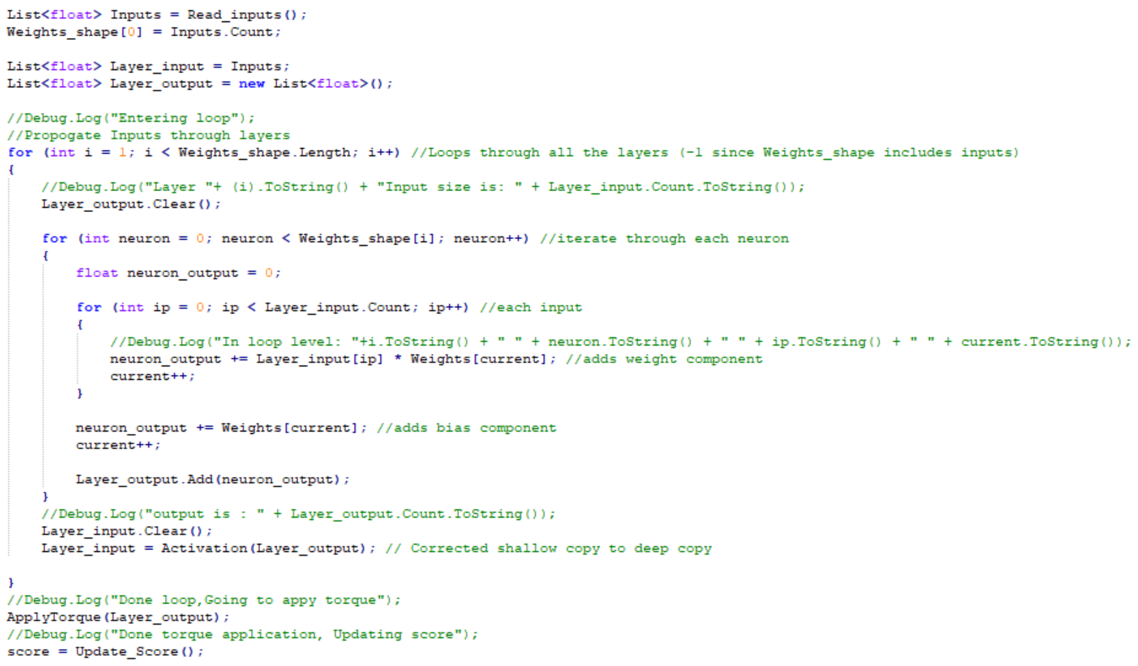


```
public void Mutate(float mutationRate)
{
    for (int i = 0; i < Genes.Length; i++)
    {
        if (random.NextDouble() < mutationRate)
        {
            //Genes[i] = Genes[i] + getRandomGene()/20;
            Genes[i] = Genes[i] + getRandomGene()/10;
        }
    }
}
```

Figure 3.3: The Mutation function

3.3 The Brain

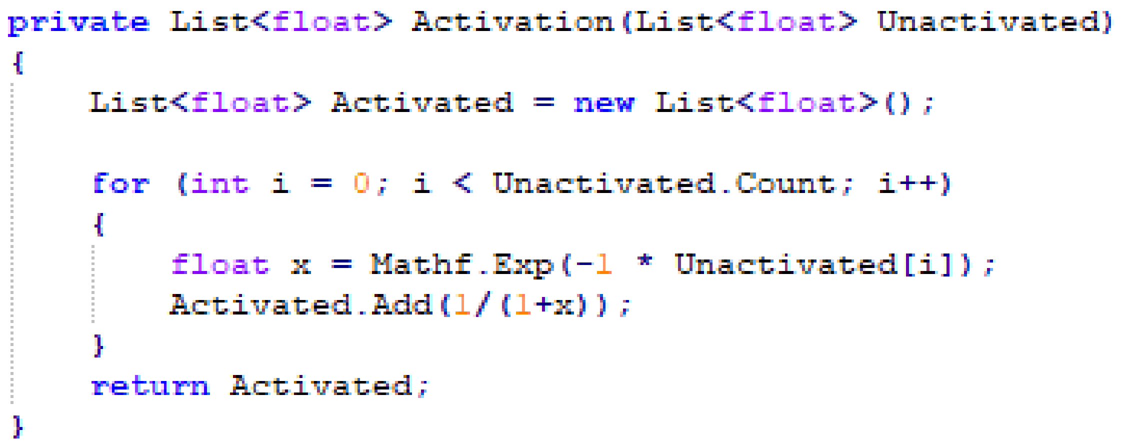
The brain of the individual is what controls how it moves and performs. Here, the brain was taken as a simple network of 5 fully connected layers. The layers contained 25, 20, 16, 8, 4 neurons respectively. This adds up to a total of 1803 trainable parameters, which are obtained from the individual's DNA.



```
List<float> Inputs = Read_inputs();
Weights_shape[0] = Inputs.Count;

List<float> Layer_input = Inputs;
List<float> Layer_output = new List<float>();

//Debug.Log("Entering loop");
//Propogate Inputs through layers
for (int i = 1; i < Weights_shape.Length; i++) //Loops through all the layers (-1 since Weights_shape includes inputs)
{
    //Debug.Log("Layer "+ (i).ToString() + "Input size is: " + Layer_input.Count.ToString());
    Layer_output.Clear();

    for (int neuron = 0; neuron < Weights_shape[i]; neuron++) //iterate through each neuron
    {
        float neuron_output = 0;

        for (int ip = 0; ip < Layer_input.Count; ip++) //each input
        {
            //Debug.Log("In loop level: "+i.ToString() + " " + neuron.ToString() + " " + ip.ToString() + " " + current.ToString());
            neuron_output += Layer_input[ip] * Weights[current]; //adds weight component
            current++;
        }

        neuron_output += Weights[current]; //adds bias component
        current++;

        Layer_output.Add(neuron_output);
    }
    //Debug.Log("output is : " + Layer_output.Count.ToString());
    Layer_input.Clear();
    Layer_input = Activation(Layer_output); // Corrected shallow copy to deep copy

}
//Debug.Log("Done loop,Going to appy torque");
ApplyTorque(Layer_output);
//Debug.Log("Done torque application, Updating score");
score = Update_Score();
```

Figure 3.4: Forward propogation in Brain

Each layer is followed by an activation function. The sigmoid activation function was used here.



```
private List<float> Activation(List<float> Unactivated)
{
    List<float> Activated = new List<float>();

    for (int i = 0; i < Unactivated.Count; i++)
    {
        float x = Mathf.Exp(-1 * Unactivated[i]);
        Activated.Add(1/(1+x));
    }
    return Activated;
}
```

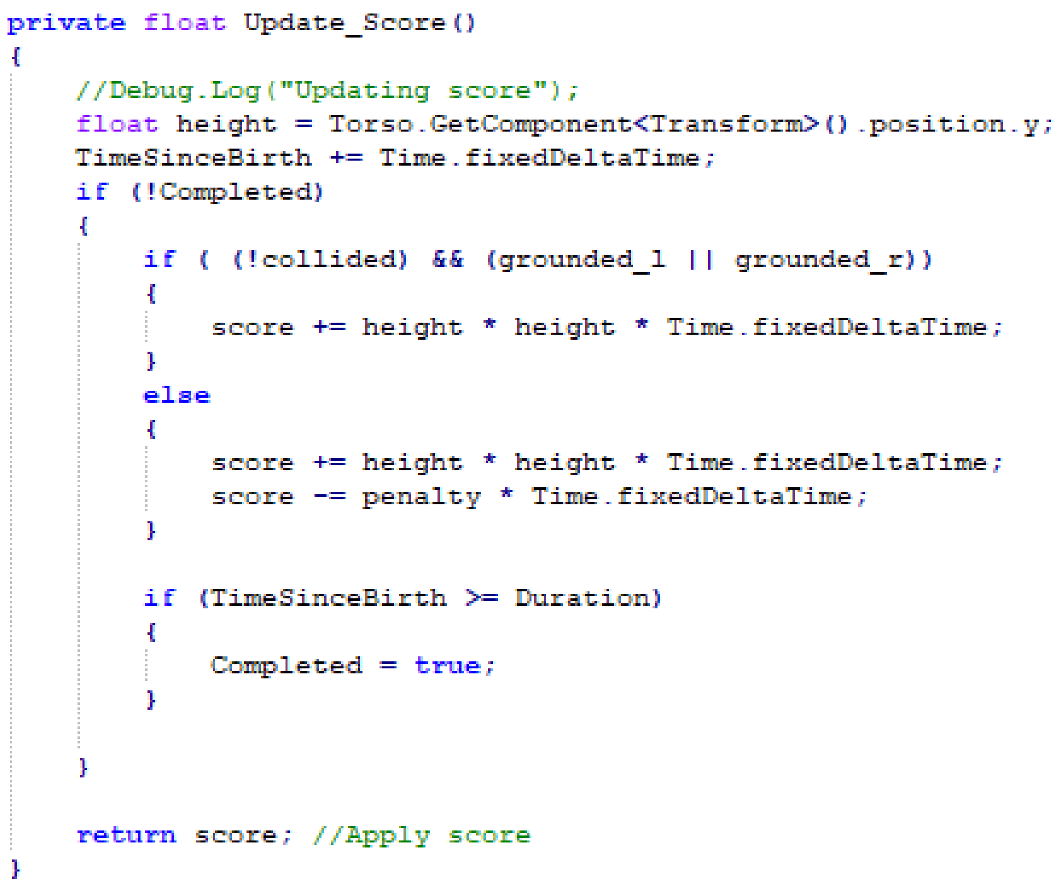
Figure 3.5: Activation Function

The inputs are taken from the rotation and position values of the limbs and torso. This gives us a total of 30 inputs (6 * 5 parts). The output of the brain is used to control the torque applied by the actuators on the hinges.

3.4 Fitness Function

For checking the performance of the population the following fitness rules were considered:

- At every time instance, the score is incremented by the product of the square of the vertical distance of the torso from the floor (the spherical part of the robot model) and the times step.
- If the torso is touching the floor or if none of the lower limbs are in contact with the floor the particular individual is penalized.



```
private float Update_Score()
{
    //Debug.Log("Updating score");
    float height = Torso.GetComponent<Transform>().position.y;
    TimeSinceBirth += Time.fixedDeltaTime;
    if (!Completed)
    {
        if ( (!collided) && (grounded_l || grounded_r))
        {
            score += height * height * Time.fixedDeltaTime;
        }
        else
        {
            score += height * height * Time.fixedDeltaTime;
            score -= penalty * Time.fixedDeltaTime;
        }

        if (TimeSinceBirth >= Duration)
        {
            Completed = true;
        }

    }

    return score; //Apply score
}
```

Figure 3.6: Score calculation

4 Simulation

This script acts as the main controller for the simulation. It invokes the scripts necessary to run the simulation in a structured manner.

- Initialising Population : The first generation of 200 (the population size we chose) individuals is initialised and given random weights.
- Running the physics simulation : Every individual of the population is assigned to a robot model and the Unity physics is allowed to run.
- Calculate Score : After all the physics simulation is completed, the score of each individual is obtained. This is used to sort the population from best to worst individual.
- Generating the next generation : The next generation is generated using the sorted

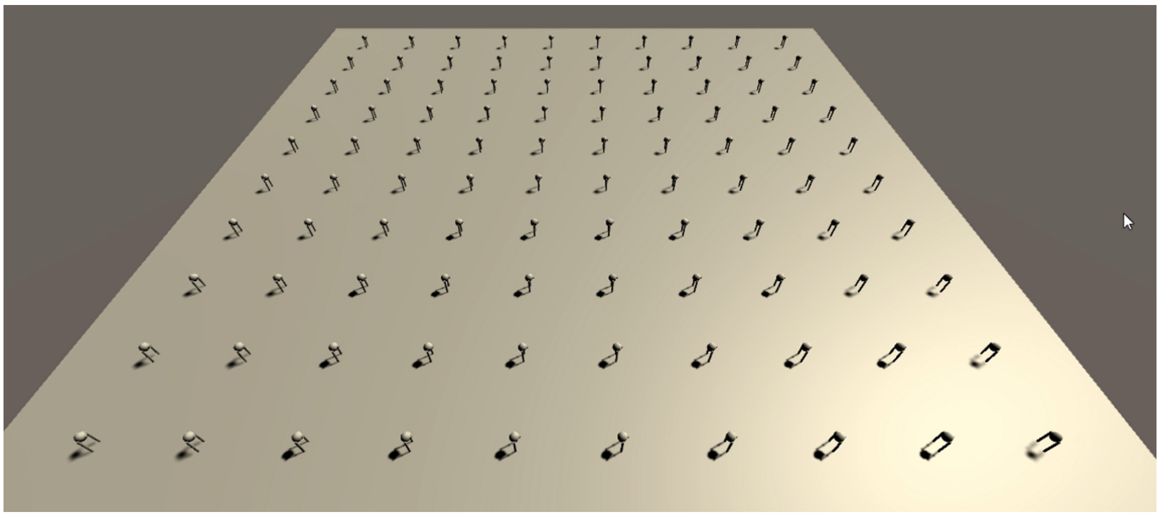


Figure 4.1: Simulation setup

population. Mutation and Crossover are implemented here.

5 Observations

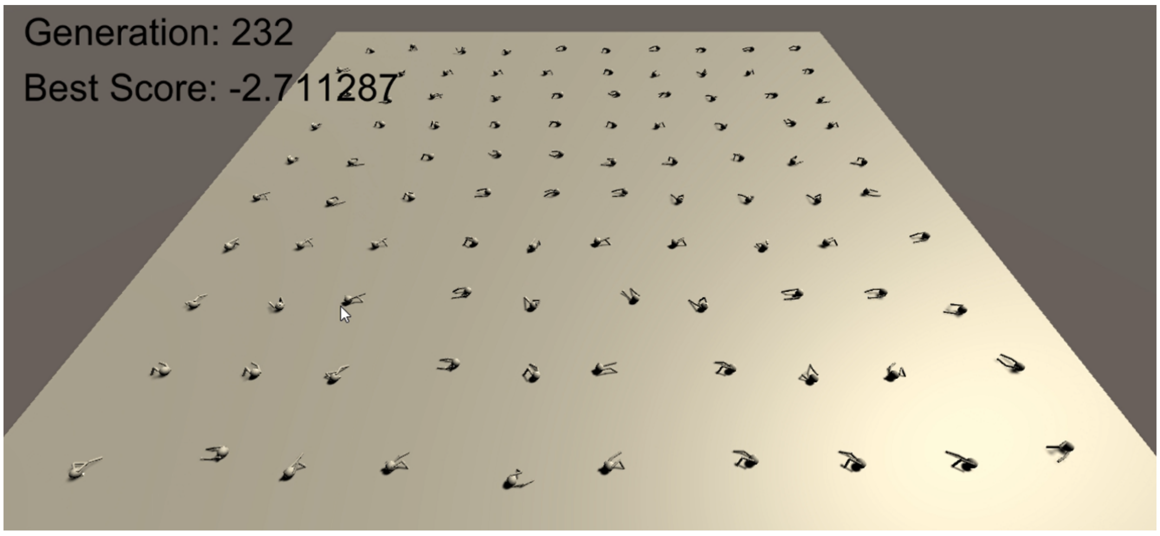
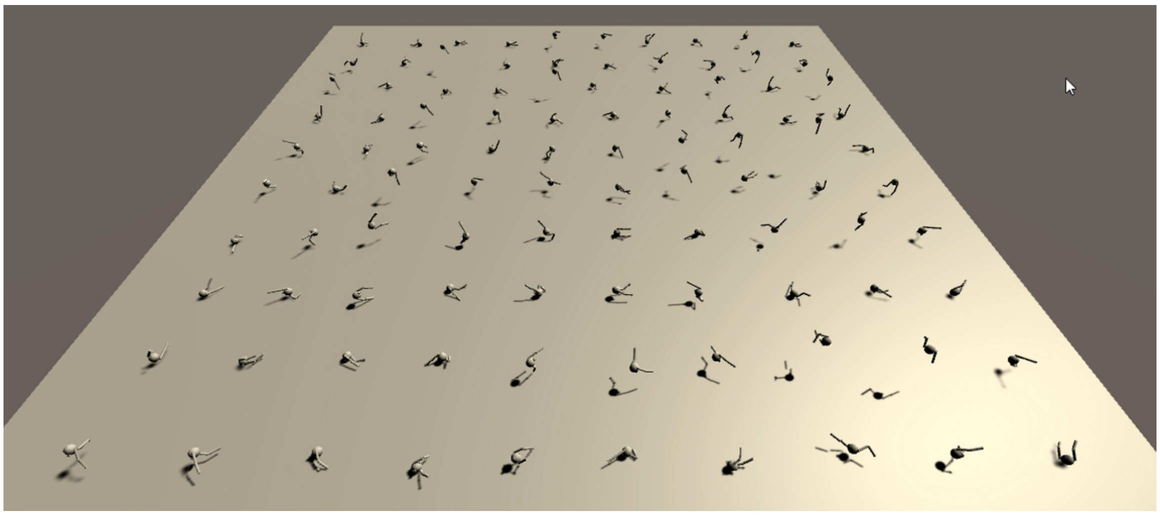


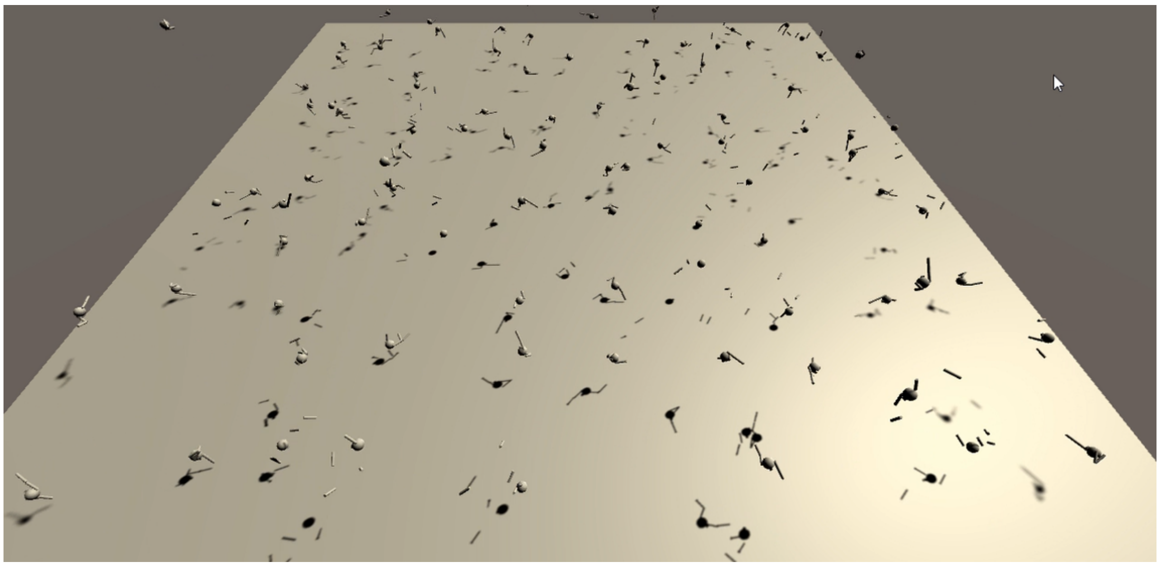
Figure 5.1: Failures during early iterations

For the initial simulations, we had not included a limitation for the body detaching from the floor,i.e.. the score was directly proportional to the height from the ground. This resulted in the population learning to fly/ jump very high in to maximise score.

In the next set of simulations we incorporated a script to make sure bodies have at



(a) Small jumps initially



(b) Extreme case

Figure 5.2: Models learnt to jump to maximize height and thereby the score

least on of their limbs in contact with the floor. This change made sure the bodies don't launch themselves up, but the population wasn't improving fast enough.

We finally tweaked the fitness script to incorporate the score to increment as a square of the height. This resulted in the population learning faster and have more individuals standing erect.

6 Conclusion

After 4318 generations we were able to obtain individuals that were capable to stand for the entire duration of the simulation.

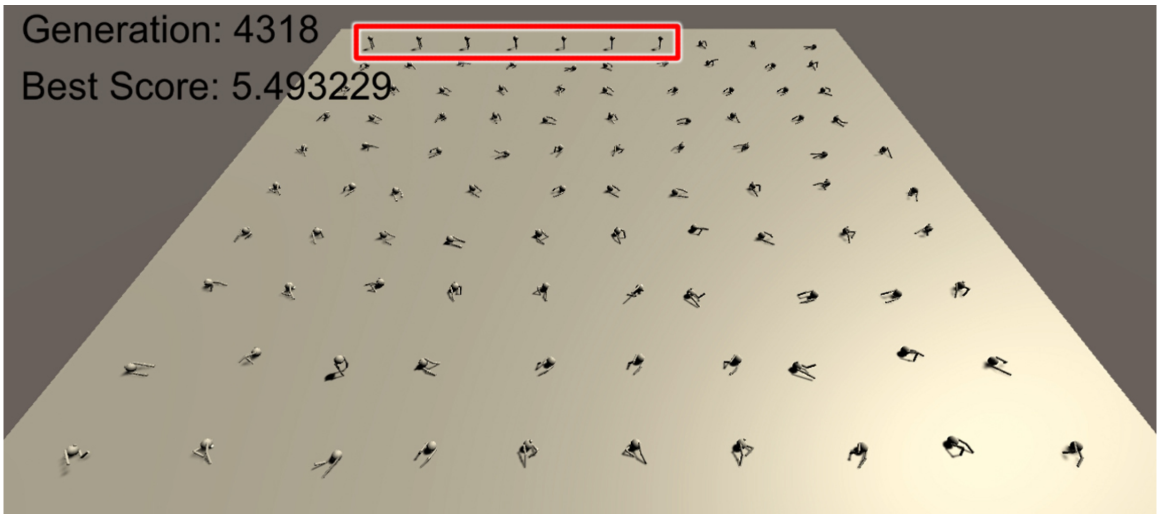


Figure 6.1: The best (*elite*) survivors after 4318 generations.

We can further develop the script (By modifying the fitness evaluation method) to implement walking in the robots.

References

[1] https://forum.unity.com/threads/tutorial-genetic-algorithm-c.479062/

[2] https://robotics.ee.uwa.edu.au/theses/2002-BipedSim-Boeing.pdf

[3] Wang, Shouyi, Wanpracha Chaovalitwongse, and Robert Babuska. ”Machine learning algorithms in bipedal robot control.” IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42.5 (2012): 728-743.

[4] https://medium.com/analytics-vidhya/genetic-algorithm-in-unity-using-c-72f0fafb535c