Lesson 2 – Training With Genetic Algorithms & Mutations

Before adding our void BreedNewPopulation method we need to add a library to help us sort objects in lists. Onto the end of the namespace lists:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

add:

using System.Linq;

We can now begin our void BreedNewPopulation:

void BreedNewPopulation()

{

List<GameObject> newPopulation = new List<GameObject>();

List<GameObject> sortedList = population.OrderBy(o => o.GetComponent<DNA>().timeToDie).ToList();

population.Clear();

for (int i = (int) (sortedList.Count / 2.0f) - 1; i < sortedList.Count - 1; i++)

{

population.Add(Breed(sortedList[i], sortedList[i + 1]));

population.Add(Breed(sortedList[i + 1], sortedList[i]));

}

for(int i = 0; i < sortedList.Count; i ++)

{

Destroy(sortedList[i]);

}

generation++;

}

So, what is going here? First, we create a List to hold our newPopulation. This List holds our offspring that we will generate. We then create a sortedList. This holds our population that we had previously clicked on. At the end of the time trial, we will then order this sortedList by a value that is in everyone’s DNA, that being timeToDie. When clicked, timeToDie is set by whatever time has elapsed so by ordering it according to timeToDie we have artificially produced a “fitter” group since those that live longer are considered fittest. So, if a person is clicked on after one second, then their timeToDie is one second. Those on the screen longer will be placed at the end of the List leaving those that died first at the top. Form this, we can say that the fittest individuals in this population are in the bottom of that sortedList.

As to Genetic Algorithms in general, it matters not what we consider to be the fittest. In our little example here, we’re going by the bottom half that are sitting in that array. And we say they are the fittest. But there is no reason we could not pick the middle third of the list, or some other parameter to measure fitness. The most important thing we are interested in is breeding enough people, so the DNA of the individuals is properly “mixed”, so we end up with the same sized population again.

To do this, we need to clear the entire population list since this sorted list has our sorted population and we are going to use this population list to repopulate a new group. And that is what this code does:

population.Add(Breed(sortedList[i], sortedList[i + 1]));

population.Add(Breed(sortedList[i + 1], sortedList[i]));

This code begins by looping around the whole list starting halfway down. From here we get the top part of the list (the fittest people that are going to be bred together) and then use [i], [i + 1] meaning we initially breed the person halfway along and then the next one up from that and we continue along the list, breeding one with the person above it in the list. And so, by the time we get to the end of the looping we have bred all the people in the top half of that list. But to ensure the population stays the same size we must do this twice. Simply put, we bred [i] and [i + 1], then again in reverse with [i + 1] and [i].

This will give us enough of a population size that we can continue forward with different evolutions while the population size stays the same. Next, we need to go through the sorted list and destroy everyone those in there.

for(int i = 0; i < sortedList.Count; i ++)

{

Destroy(sortedList[i]);

}

generation++;

This removes those left in the list and then updates the generation++ count to show we are now in generation two. Now we will build the Breed method just above void BreedNewPopulation let us add:

GameObject Breed(GameObject parent1, GameObject parent2)

    {

      // 1

      Vector3 pos = new Vector3(Random.Range(-7f, 7f), Random.Range(-3f, 3f), 0);

// 2

       GameObject offspring = Instantiate(personPrefab, pos, Quaternion.identity);

      // 3

      DNA dna1 = parent1.GetComponent<DNA>();

      DNA dna2 = parent2.GetComponent<DNA>();

// 4

       offspring.GetComponent<DNA>().r = Random.Range(0, 10) < 5 ? dna1.r : dna2.r;

       offspring.GetComponent<DNA>().g = Random.Range(0, 10) < 5 ? dna1.g : dna2.g;

       offspring.GetComponent<DNA>().b = Random.Range(0, 10) < 5 ? dna1.b : dna2.b;

       return offspring;

    }

This Breed method is returning a GameObject since that GameObject is then going into the population by adding whatever is returned into our population.

// 1

When we create a new little person, we need a random location for them to go.

// 2

We create the actual instance of that person using GameObject called offspring but instead of giving it random red, green, and blue values as we did with its parent, we will instead get the red, green and blue values from the parents.

// 3

Here we provide access to the DNA script.

// 4

Afte getting GameObject parent1 DNA and GameObject parent2 DNA. We perform a swap. This swap is the fundamental basic operation that happens in all genetic algorithms. We run through each value from all the genes we are storing and randomly swap them between the parents.

Here you can we have a Random.Range if it's less than five. In other words, 50% of the time this offspring will get parent1 red channel and the other 50% of the time it will get parent2 red channel. The same is true for the green and blue channels. Finally, note that we are not combining them as far as adding them together and dividing by two as an average, instead we literally swap the combinations. Think about this in human terms; when humans breed, we do not generally inherit a mixture of our parent’s eyes. Instead, we get one colour or the other.

This is what makes genetic algorithms work and is the foundation for all genetic algorithms out there. In our case, this is the swapping of the values in the DNA.

Now it is time to head to Unity and confirm it all works. If not, it is time to debug! So, go back into Unity, press play, and you will have 10 seconds to click on whichever persons you choose. This will train a colour into the generation that you pick last.

To have the algorithm favour those you click on first, we simply need to change sortedList = population.OrderBy to sortedList = population.OrderByDescending in our void BreedNewPopulation method.

Another process in genetic algorithms that are built into these systems is the concept of mutation. In nature, animals and plants will be born such that they have a mutation in them. If the mutation is bad, they may die out quickly, otherwise if it is an advantage for that species then that species becomes fitter and remains in the population to pass on its genetic mutation and all! So, how can we add this mutation into our code?

Head back into our GameObject Breed code. The mutation must happen where the DNA for the offspring is set, and we could have a chance that it will be set by the parents otherwise it will be a mutation. We will put in a mutation that occurs a small number of times so we will adjust the if statement inside the method GameObject Breed:

if(Random.Range(0,1000) > 5)

While we are giving ourselves a decent sample size, Random.Range(0,1000), so as to not make the calculation take longer than needed, we will set an upper limit of > 5. So, for that small percentage, a mutation will occur. This should be enough to see the effect, otherwise you can increase the percentage. In the following lesson we will try to build a challenge!