Building a neural network in C#

Creating a neural network with the ability for backpropagation, and evolution based training.

Kip Parker Mar 31, 2019 · 11 min read ★

Introduction

We will be building a Deep Neural Network that is capable of learning through Backpropagation and evolution. The Code will be extensible to allow for changes to the Network architecture, allowing for easy modification in the way the network performs through code.

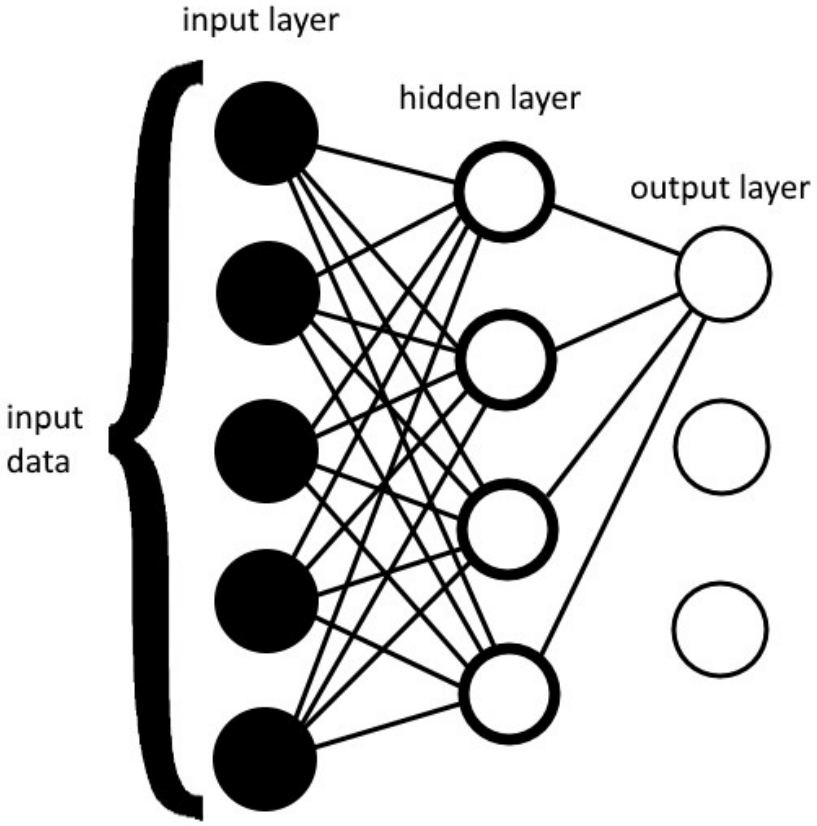
The network is a Minimum viable product but can be easily expanded upon. You can find all the code available on GitHub, This includes the mutation and backpropagation variant.

I will be explaining how we will set up the feed-forward function, setting up all the required arrays and allowing for mutation-driven learning.

You will need to be familiar with some basic coding if you want to understand the workings of the neural network. For backpropagation, you will want to be familiar with gradient descent and calculus. Unless you just want to use the code for your projects.

Concept time!

Our deep neural network consists of an input layer, any number of hidden layers and an output layer, for the sake of simplicity I will just be using fully connected layers, but



A simple neural network model

Neural network Architecture

The model above has 5 neurons on the input layer, as indicated by the first column consisting of 5 solid circles. The second layer has 4 hidden neurons and the output layer has 3 output neurons. The Size of these layers and the number of hidden neurons is arbitrary. But for this visual model, I have chosen a smaller size.

The idea is that a value from each input node gets carried through each dendrite (Connecting line between to nodes on model) and multiplied by a weight that the corresponding dendrite holds, this then gets passed on and held in the neuron in the next layer, then this cycle gets continued for each layer until you get an output;

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \ldots + w_na_n + b)$$

Each node also has a bias attached to it (represented by b), This helps the network perform better. The σ symbol is the **activation function** that the sum of these products

the output layer.

We will go into every step in more detail when we get to the coding section, but if you don't have a basic understanding of it, I recommend you go to youtube or Wikipedia to try and find out some more. **This video might help**

Coding Time!

The prerequisites for making this feedforward function is a way of storing all the data. We will use a series of arrays to store all the data and make sure the network performance is always optimal.

What data-types we will need

An input array to declare the size of the network, this will be labeled Layers. An example of what this might be is {5,4,3}, this would create a network similar to the image above, of the network model.

We will also need a neuron array, this will be used to hold the values generated during the feedforward algorithm. This will be in the form of a 2d jagged array

We will also need another 2d jagged array for storing the biases in each layer.

Finally, we need a 3d jagged array for storing the weights associated with each dendrite. The Biases and weights are learnable, and all these arrays are floats.

Initialization functions

have been defined all the appropriate arrays have been initialized.

The function takes an input array of the network dimensions and populates all the arrays

```
private int[] layers;//layers
private float[][] neurons;//neurons
private float[][] biases;//biasses
private float[][][] weights;//weights
private int[] activations;//layers

public float fitness = 0;//fitness

public NeuralNetwork(int[] layers)
{
  this.layers = new int[layers.Length];
  for (int i = 0; i < layers.Length; i++)
  {
    this.layers[i] = layers[i];
  }
  InitNeurons();
  InitBiases();
  InitWeights();
}
```

First, let's deal with the Neurons, these don't have to have any values assigned to them, we just need to allocate the space in storage, as arrays are static in length. We use lists as a temporary medium to create the arrays.

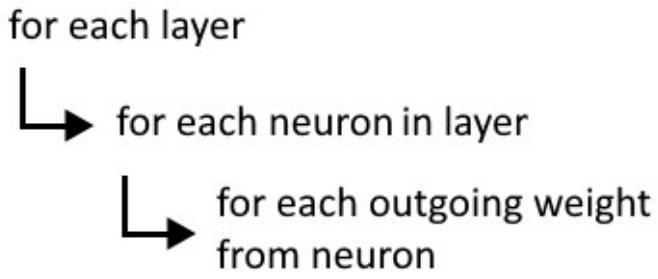
```
//create empty storage array for the neurons in the network.
private void InitNeurons()
{
  List<float[]> neuronsList = new List<float[]>();
  for (int i = 0; i < layers.Length; i++)
  {
    neuronsList.Add(new float[layers[i]]);
  }
  neurons = neuronsList.ToArray();
}
```

Then we can take care of initializing the Biases there, The size of the biases is the same as the size of the neurons, except we need to populate each slot, I will generating each

backpropagation(Vanishing and exploding) but this won't be needed for a genetic implementation.

```
//initializes and populates array for the biases being held within
the network.
private void InitBiases()
{
  List<float[]> biasList = new List<float[]>();
  for (int i = 0; i < layers.Length; i++)
  {
    float[] bias = new float[layers[i]];
    for (int j = 0; j < layers[i]; j++)
    {
      bias[j] = UnityEngine.Random.Range(-0.5f, 0.5f);
    }
    biasList.Add(bias);
  }
  biases = biasList.ToArray();
}
```

Weights are generated similarly to the biases, except we add another dimension to the array for each neuron in the previous layer, such that it follows:



the hierarchy for weight storage

```
//initializes random array for the weights being held in the
network.
private void InitWeights()
{
  List<float[][]> weightsList = new List<float[][]>();
  for (int i = 1; i < layers.Length; i++)
  {
    List<float[]> layerWeightsList = new List<float[]>();
    int neuronsInPreviousLayer = layers[i - 1];
    for (int j = 0; j < neurons[i].Length; j++)
```

```
        {
          neuronWeights[k] = UnityEngine.Random.Range(-0.5f, 0.5f);
        }
        layerWeightsList.Add(neuronWeights);
      }
      weightsList.Add(layerWeightsList.ToArray());
    }
    weights = weightsList.ToArray();
  }
```

Feedforward algorithm

With all previous initialization functions in place, its time to move onto the actual feedforward algorithm and surrounding concepts.

$$\sigma(w_1a_1+w_2a_2+w_3a_3+...+w_na_n+b)$$

As seen earlier this is what is computed for each neuron in hidden and output layers of the network. Let's explain each term. Starting with the **activation function** σ, the idea behind this that you pass in the **weighted Sum** and it returns a nonlinear result which improves the performance as well as keeping the network's neurons under control, in the real world there are not many things that follow a linearity, so a non-linear can help approximate the non-linear phenomenon, as well allowing for backpropagation.

The **Weighted Sum**, this is all the data that gets fed into the activation function. This is each of the nodes in the previous layer multiplied by the weight in the dendrite by which the data gets carried to the current neuron.

The **activation function** can be chosen by you, and depending on the application, a different choice of activation function may be more beneficial.

Popular variants of this function include:

1. Identity
2. Binary Step
3. Sigmoid
4. Tanh

6. Leaky ReLU

7. Softmax

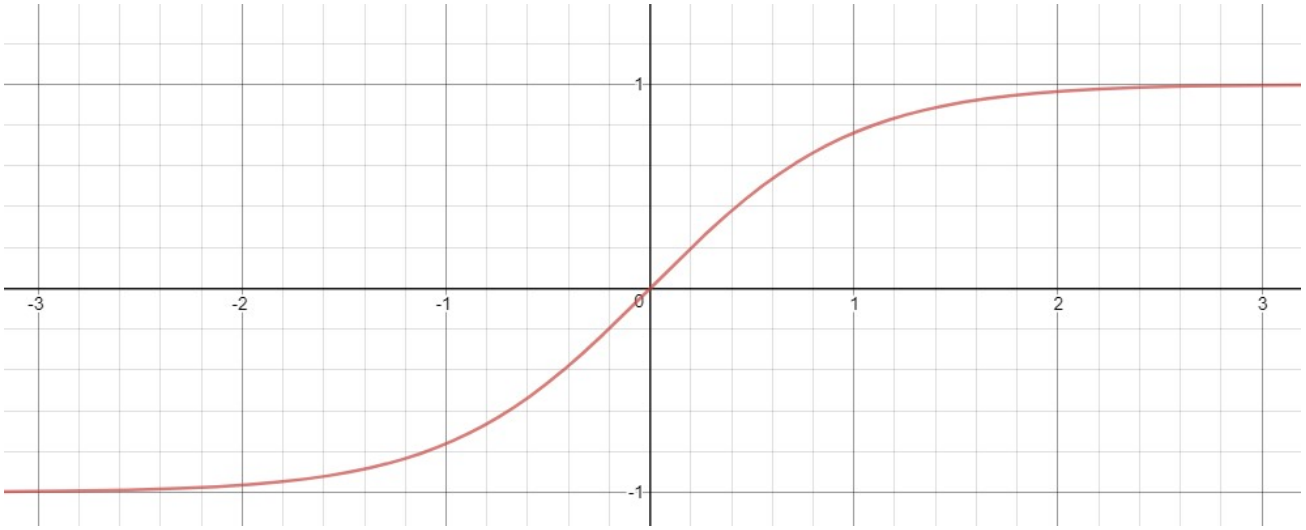
For now, I will be using Tanh as my chosen **activation function** as it allows, for both positive and negative values. Although the other ones would be applicable for different applications.

Tanh can be expressed in the following format

$$tanh(x) = \frac{2}{(1 + e^{(-2x)}) - 1}$$

Tanh function

This non-linear function returns the result of



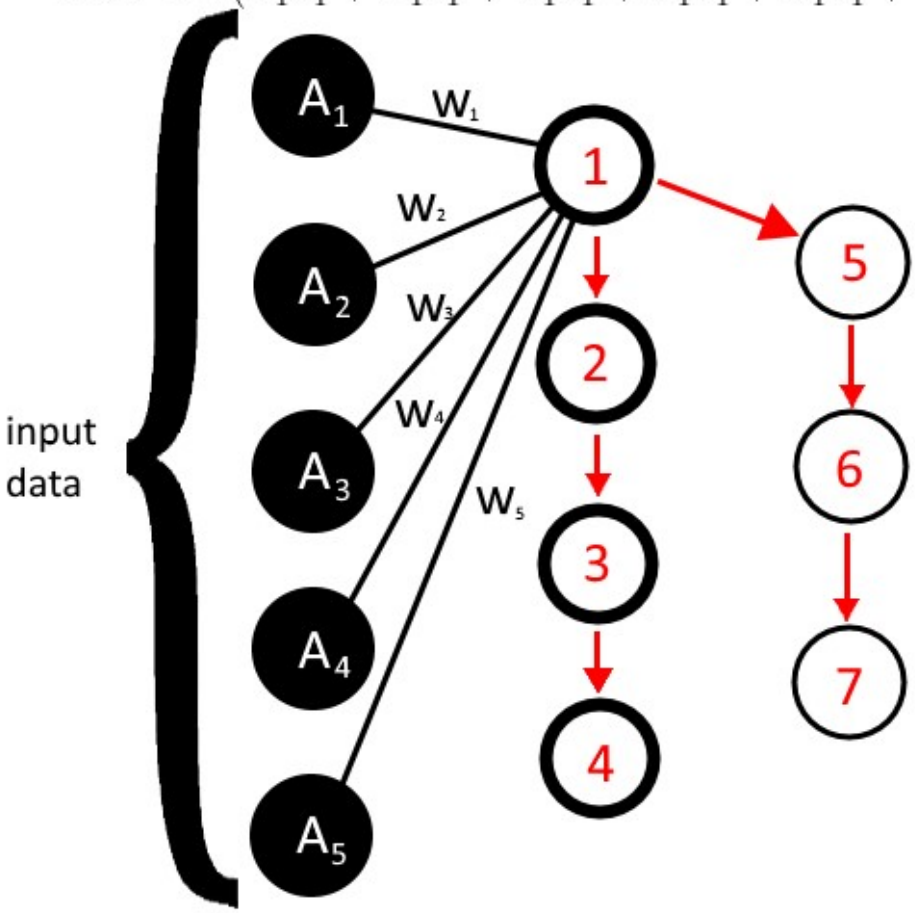
Tanh Graph

```
public float activate(float value)
{
  return (float)Math.Tanh(value);
}
//Luckily Unity provides a built in function for Tanh
```

Now we look at how the feedforward function iterates over the neurons to produce an

The program iterates through each neuron and gets the values in each of the neurons in the previous layer, and multiplies the value by the weight connecting the two, runs this through the activation function, then sets its value to the activation. Here is an example for node 1

$$value = \sigma(w_1a_1 + w_1a_1 + w_1a_1 + w_1a_1 + w_1a_1 + b)$$



Example neuron calculation

```
//feed forward, inputs >==> outputs.
public float[] FeedForward(float[] inputs)
{
  for (int i = 0; i < inputs.Length; i++)
  {
    neurons[0][i] = inputs[i];
  }
  for (int i = 1; i < layers.Length; i++)
  {
    int layer = i - 1;
    for (int j = 0; j < neurons[i].Length; j++)
    {
      float value = 0f;
      for (int k = 0; k < neurons[i - 1].Length; k++)
```

```
            neurons[i][j] = activate(value + biases[i][j]);
            }
        }
        return neurons[neurons.Length - 1];
    }
```

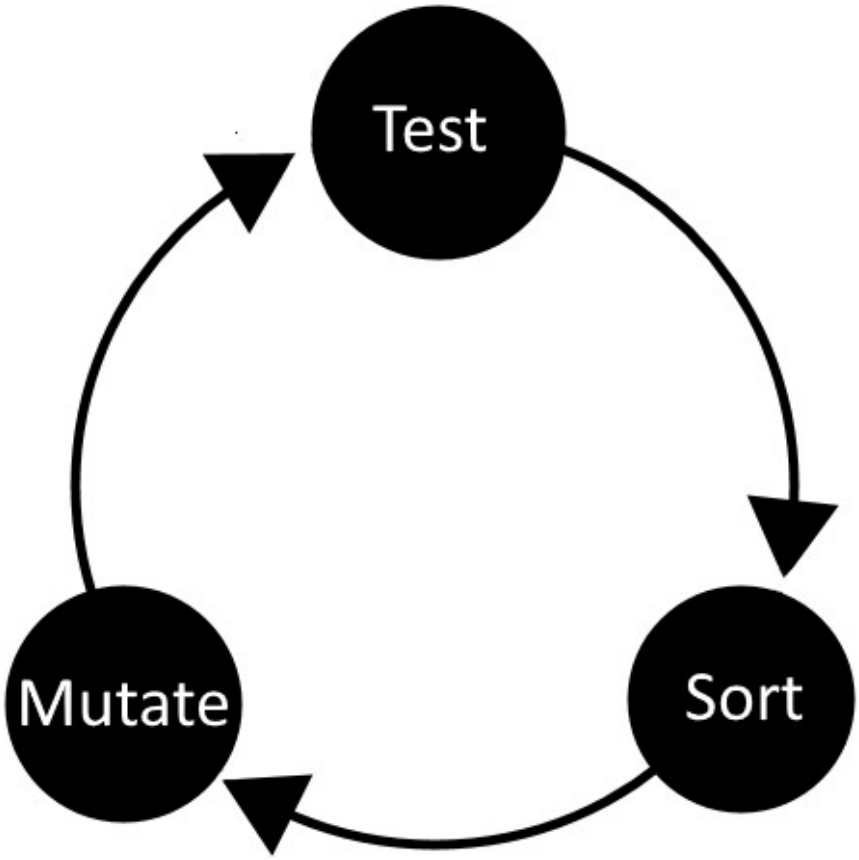
With that, we have a working neural network, but at the moment it is pretty useless until we have a method to train it. Let's get the working next.

Training

For this implementation of the network, we will use a genetic algorithm. They are significantly easier to code, and a lot less involved in the maths side, however, if you are not interested in this implementation, I have included a backpropagation code example and might work on a reinforcement learning approach.

A genetic algorithm is a way of training the neural network to perform a given task well. It works well because you can give it quite a simple fitness function which dictates how 'well' the network performed. Its drawback is that it takes a relatively long time to train when dealing with a large network, and can be quite slow compared to backpropagation, but if you have a large amount of computing power, it can return better results than backpropagation as they should almost always be able to reach the global minimum.

The idea behind the genetic algorithm is based on the **theory of Darwinism** and how biological evolution takes place, although we will be implementing a slightly simplified model, the same overarching concepts apply. A species of creatures and they have a measure of how 'fit' they are, historically this might have been how capable they were of hunting and to the ability to reproduce, but with our network, it will be how far it can walk for example. Then we sort out the population, split them in half, clone the top half into the bottom half and mutates them, this way the performance of the network tends towards a **global maximum**.



Evolution algorithm

Evolution driven learning implementation

The majority of the management of the network will be done from another script. So all we need to add to the network, for now, is a method of sorting the networks, a way to clone a network onto another network and finally a way of mutating the network.

for the sorting We make the code of type IComparable, this will allow us to directly sort it when referencing it. It uses the network fitness as the value on which the network gets sorted

```
//Comparing For NeuralNetworks performance.
public int CompareTo(NeuralNetwork other)
{
  if (other == null)
    return 1;
  if (fitness > other.fitness)
    return 1;
  else if (fitness < other.fitness)
    return -1;
  else
```

We also will need to ability to clone the learnable values (weights and biases) onto another neural network.

```
//this loads the biases and weights from within a file into the
neural network.
public void Load(string path)
{
  TextReader tr = new StreamReader(path);
  int NumberOfLines = (int)new FileInfo(path).Length;
  string[] ListLines = new string[NumberOfLines];
  int index = 1;
  for (int i = 1; i < NumberOfLines; i++)
  {
    ListLines[i] = tr.ReadLine();
  }
  tr.Close();
  if (new FileInfo(path).Length > 0)
  {
    for (int i = 0; i < biases.Length; i++)
    {
      for (int j = 0; j < biases[i].Length; j++)
      {
        biases[i][j] = float.Parse(ListLines[index]);
        index++;
      }
    }
    for (int i = 0; i < weights.Length; i++)
    {
      for (int j = 0; j < weights[i].Length; j++)
      {
        for (int k = 0; k < weights[i][j].Length; k++)
        {
          weights[i][j][k] = float.Parse(ListLines[index]);
          index++;
        }
      }
    }
  }
}
```

Finally, we will need the ability to slightly nudge each learnable value in the network, this is done through mutation.

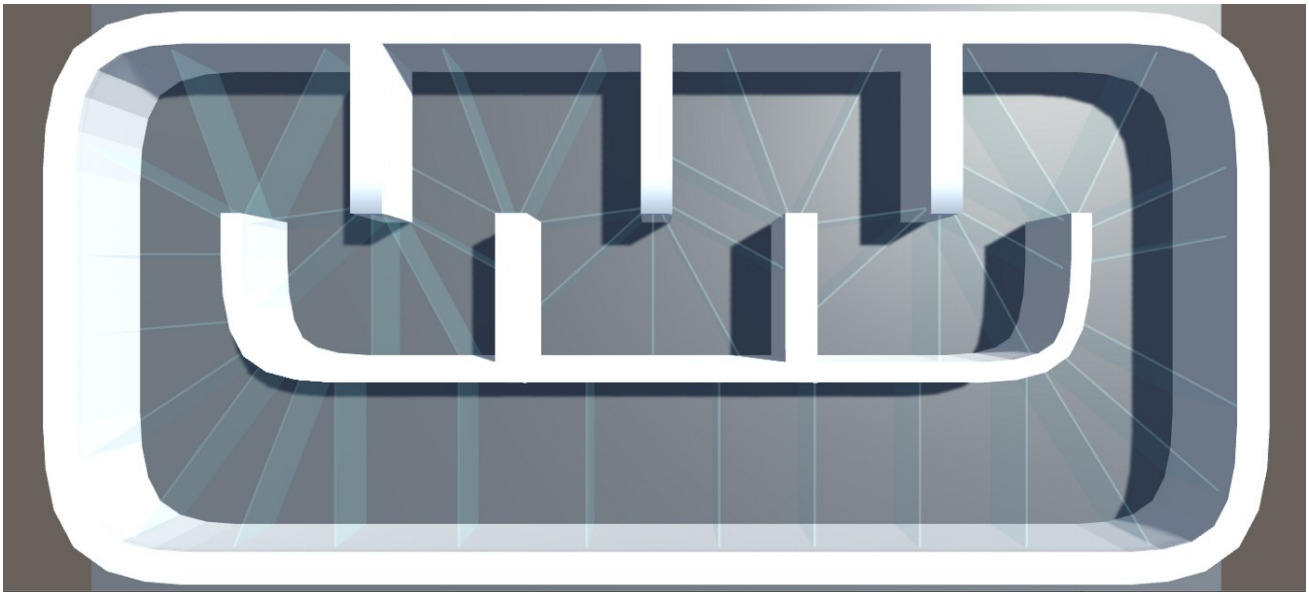
```
//used as a simple mutation function for any genetic
implementations.
```

```
    {
      for (int j = 0; j < biases[i].Length; j++)
      {
        biases[i][j] = (UnityEngine.Random.Range(0f, chance) <= 5) ?
biases[i][j] += UnityEngine.Random.Range(-val, val) : biases[i][j];
      }
    }

    for (int i = 0; i < weights.Length; i++)
    {
      for (int j = 0; j < weights[i].Length; j++)
      {
        for (int k = 0; k < weights[i][j].Length; k++)
        {
          weights[i][j][k] = (UnityEngine.Random.Range(0f, chance) <=
5) ?  weights[i][j][k] += UnityEngine.Random.Range(-val, val) :
weights[i]  [j][k];

        }
      }
    }
}
```

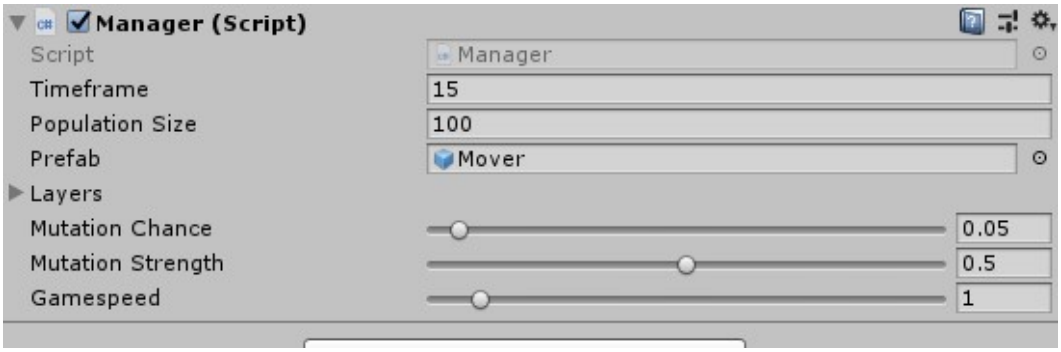
All this code implemented, we should have a working network capable of learning.

Implementing Network in Unity

Setting up the scene, here is a track I created, with checkpoints that our neural network



Within the scene, we have a manager which spawns, mutates clones, and destroy networks. The point of the network is to improve the performance of the network.

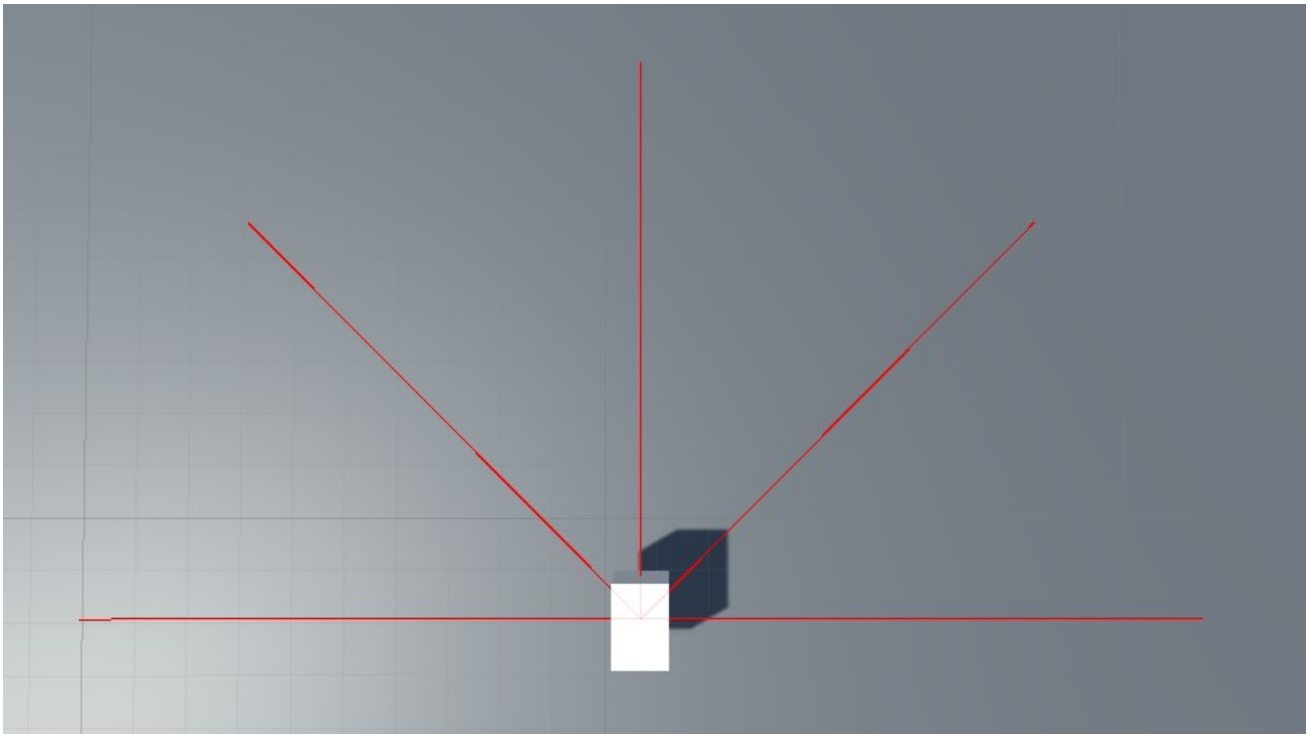


The timeframe is the time for which each generation of bots train, Population size is how many bots train at once, mutation chance is the change of a mutation happening to each weight or bias, the mutation strength is the standard deviation of the nudge and game speed is the in-game tick rate and if you have the appropriate hardware increases the training time of the network.

The Manager creates a population of Prefabs what use the neural network, it then deploys a neural network into each of them. After some time, testing will be completed and the networks will be sorted so that only the best-performing ones get kept, the ones that do not, will get copied over by the best networks and mutated. This is then deployed again and the training cycle gets continued.

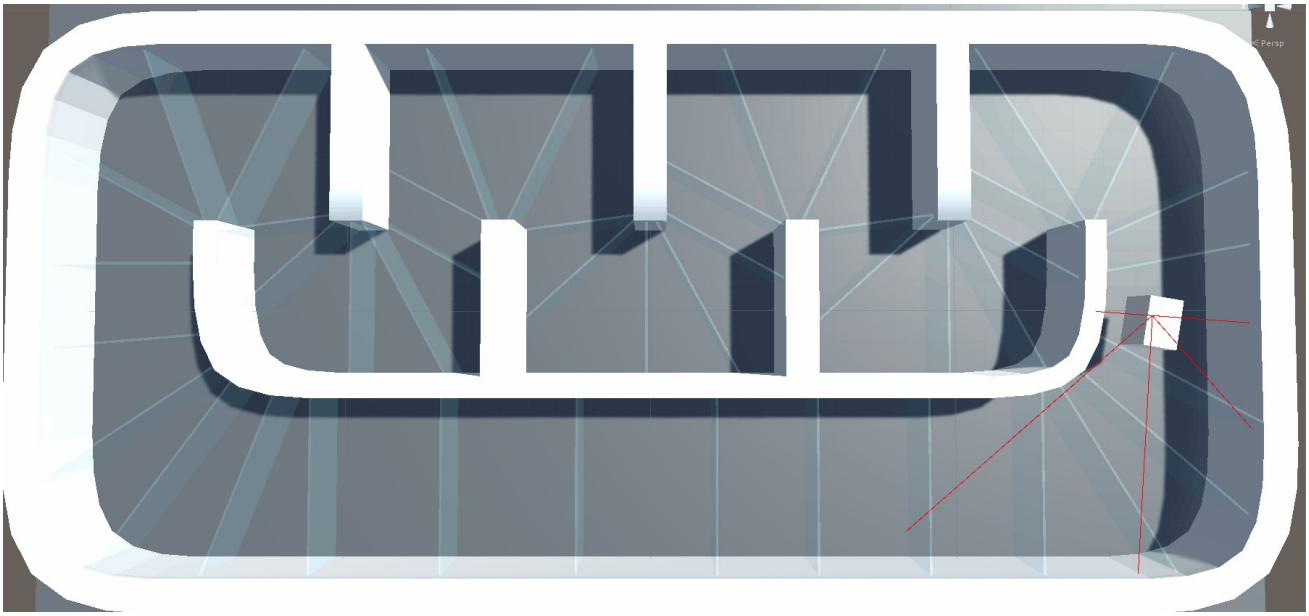
Finally, we need to work on the Learner Prefab, this will function loosely similar to a car

sensors.



After a few minutes of training, we get this result which means that all the code works!

All the code can be found on GitHub and I should note that the code on GitHub also has a save and load function, these functions save and load the weights of the network, to allow the network to resume from where it left off.



Thank you so much for reading, and I hoped it helped. I am open to any questions or feedback. Stay tuned!

Machine Learning Genetic Algorithm Neural Networks Framework