Systemy Sztucznej Inteligencji

dokumentacja projektu Kulki XDXDXD

Chłąd Paweł
Grupa 2D

Meller Bartłomiej
Grupa 2D

28 maja 2020

Część I

Opis programu

Instrukcja obsługi

Aby uruchomić projekt, należy:

1. Uruchomić Unity
2. Wybrać projekt z listy projektów
3. Wybrać scenę główną z listy scenę
4. Uruchomić projekt, klikając przycisk *"Play"*

Gdy projekt został uruchomiony, zacznie się proces uczenia. Uczenie może zająć długi okres czasu.

Dodatkowe informacje

Projekt działa na silniku Unity (ver. 2018 LTS)

Część II

Opis działania sieci neuronowej

Jak zostało wcześniej wspomniane program opiera się na sztucznej sieci neuronowej (SSN), czyli matematycznym modelu sieci neuronów działającej układach nerwowyc istot żywych. Podobnie jak ludzka sieć neuronowa, SSN zbudowana jest z neuronów ułożonych w warstwy. Każda komórka nerwowa danej warstwy połączona jest ze wszystkimi komórkami warstwy poprzedniej i warstwy następnej za pomocą synaps posiadających pewne losowo zainicjowane wagi w postaci liczb. Są one modyfikowane w procesie uczenia sieci neuronowej.

Pierwszą warstwę sieci, odpowiedzialną za przyjmowanie danych wejściowych, nazywamy warstwą wejściową. Analogicznie ostatnia warstwa sieci to warstwa wyjściowa, odpowiadająca za zwracanie wyniku. Pomiędzy nimi mogą (lecz nie muszą) znajdować się warstwy ukryte. Zadaniem projektanta sieci neuronowej jest znalezienie optymalnej ilości i wielkości tych warstw, dzięki czemu nauczanie będzie przebiegało efektywnie. Z kolei ilość neuronów na warstwach skrajnych zależy od tego, ile cech obiektu lub sygnału wejściowego jest badane oraz rodzaju i ilości klas do których można przypisać wyjście - w przypadku tego projektu jest to $n + 1$ neuronów wejściowych (gdzie n jest liczbą promieni będących symulacją lasterowych czujników odległości) oraz dwa neurony wyjściowe decydujące o kierunku poruszania się agenta w dwóch wymiarach.

Każdy z neuronów przyjmuje pewną wartość na wejściu, a następnie przetwarza ją dzięki funkcji aktywacji. Sygnał wejściowy i-tego neuronu k-tej warstwy można opisać równaniem:

$$s_i^k = \sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b,$$

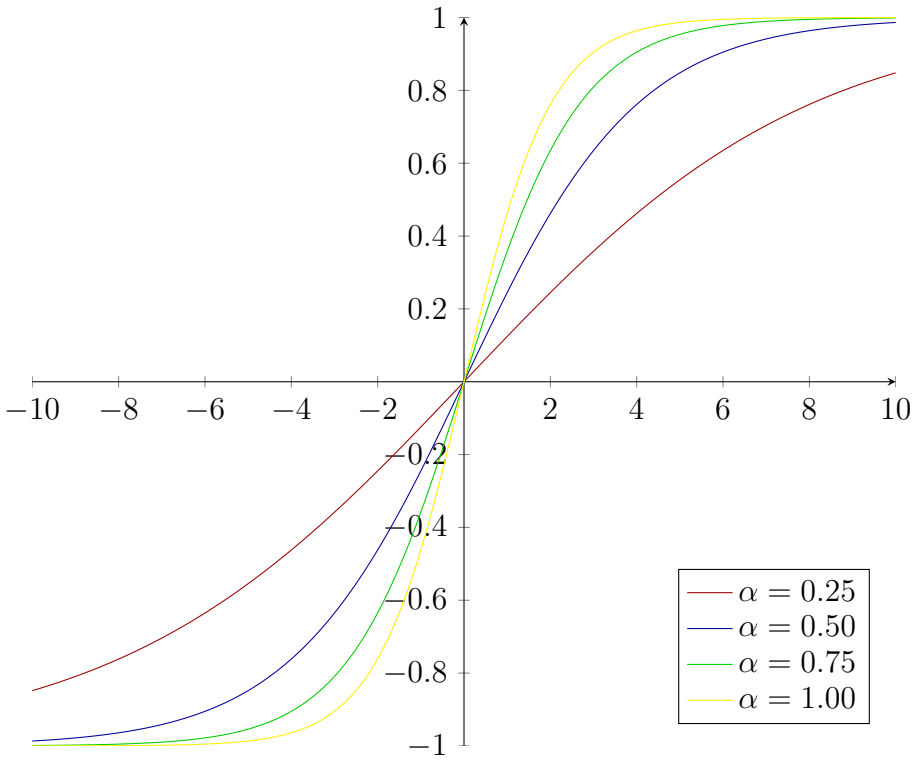
gdzie w_{ij}^k - waga synapsy pomiędzy i-tym neuronem k-tej warstwy a j-tym neuronem warstwy poprzedniej, y_j^{k-1} - wartość sygnału wyjściowego j-tego neuronu warstwy poprzedniej, b - zakłócenia sieci (tzw. bias). Najczęściej we wzorze tym nie uwzględnia się ostatniego czynnika (zakłada się, że sieć nie posiada zakłóceń tj. b = 0). Z kolei sygnał wyjściowy i-tego neuronu to:

$$y_i^k = f(s_i^k) = f(\sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b).$$

Wyróżniamy wiele funkcji aktywacji, jednak najczęściej wykorzystywaną (i wykorzystaną również w tym projekcie) jest funkcja bipolarna liniowa, której wzór wygląda następująco:

$$f(s_i^k) = \frac{2}{1 + e^{-\alpha s_i^k}} - 1 = \frac{1 - e^{-\alpha s_i^k}}{1 + e^{-\alpha s_i^k}}$$

gdzie α jest współczynnikiem korygującym rozpiętość funkcji aktywacji w przestrzeni decyzyjnej. Jej wykres zamieszczono na następnej stronie.



Bipolarna liniowa funkcja aktywacji

Kiedy sztuczna sieć neuronowa jest już odpowiednio zbudowana, należy ją nauczyć tego, czego od niej oczekujemy. Polega to na modyfikowaniu wag synaps w ściśle określony sposób. Istnieje wiele metod nauczania sieci. W naszym przypadku używamy metody heurystycznej: Algorytmu genetycznego (AG).

0.0.1 Algorytm genetyczny

Algorytm genetyczny to heurystyczny algorytm optymalizacji. Działa on na zasadzie symulacji ewolucji darwinowskiej, gdzie słabe jednostki umierają, a silne mnożą się i przekazują swoje geny następnym pokoleniom. W naszym przypadku zwierzętami są sieci neuronowe, które będziemy chcieli zoptymalizować. Na początku działania algorytmu, inicjalizujemy populację, złożoną z losowo wygenerowanych sieci. Takie sieci będą podejmowały losowe decyzje (gdyż mają losowe wagi). Następnie sieci te przechodzą przez test, który ocenia ich sprawność. Każde ze zwierząt otrzymuje ocenę, która oznacza jak dobrze sobie poradziło. Po symulacji pokolenia, przychodzi czas na proces selekcji; Sortujemy sieci względem ich wyników malejąco i usuwamy stałą część najgorszych jednostek. Następnie jednostki, które przetrwały, mnożą się przekazując swoje geny następnej generacji, w naszym przypadku genotypem są wagi połączeń. Schemat tworzenia kolejnej generacji wygląda następująco. Po usunięciu najgorszych jednostek, pozostałe kopiowane są do momentu uzyskna wyjściowej ilośći, a nastepnie wagi utworzonych w ten sposób klonów, mutowane są o losową wartość z przedziału $[-0.2; 0.2]$, przy jednoczesnym zachowaniu wag oryginałów.

Implementacja

Spis metod

`public class MovementScoreRule : MonoBehaviour, IScoringRule`

- `public float GetScore()` - Zwraca wynik agenta, bazujący na dystansie który został przebyty

`public class Run`

- `public string runName = "Run #N"` - Nazwa przebiegu symulacji
- `public event EventHandler<List<AgentResult>> RunComplete` - zdarzenie informujące o zakończeniu przebiegu symulacji
- `public List<GameObject> agents = new List<GameObject>()` - Lista agentów biorąca udział w tym przebiegu
- `public List<AgentResult> results = new List<AgentResult>()` - Lista rezultatów, aktualizowana po zakończeniu przebiegu
- `public static GameObject agentPrefab` - Prefab agenta, który będzie instancjonowany w czasie rozpoczęcia przebiegu
- `public Run(int num_agents)` - Konstruktor przebiegu
  - `int num_agents` - Ilość losowo zainicjalizowanych agentów, która zostanie zainstancjonowana
- `public Run(List<NetworkModel> models)` - Konstruktor przebiegu
  - `List<NetworkModel> models` - Lista modeli, z której będą inicjalizowani agenci
- `public void BeginRun()` - Rozpoczyna przebieg symulacji

`public class AgentResult`

- `public double score` - wynik agenta
- `public NetworkModel model` - model agenta
- `public AgentResult(double score, NetworkModel model)` - konstruktor rezultatu agenta

`public interface IScoringRule`

- `float GetScore()` - Ma zwrócić wynik dla konkretnej zasady oceniania

`public class RunManager : MonoBehaviour` - klasa zarządzająca przebiegami symulacji
`public class CameraController : MonoBehaviour`

- `public Transform target` - Obiekt obserwowany
- `public Vector3 offset` - Oddalenie od obiektu obserwowanego

`public class ModelManager`

- `public List<NetworkModel> Models = new List<NetworkModel>()` - Kolekcja aktualnych modeli
- `public int NumModels { get; }` - Docelowa liczba modeli
- `public double LearningRate { get; }` - Szybkość uczenia
- `public ModelManager(List<NetworkModel> models, double learningRate = 0.1f)` - Konstruktor
  - `List<NetworkModel> models` - ustawia Models
  - `double learningRate` - ustawia LearningRate
- `public void SaveTop(int n)` - Zapisuje pierwsze n modeli (sortowane po score)
- `public void Expand()` - Tworzy nowych agentów, tak długo aż Models.Count osiągnie NumModels

`public class Agent : MonoBehaviour`

- `public NetworkModel network` - sieć neuronowa agenta
- `public static Transform cookieJar` - `Transform` zachęty
- `public Action<Agent> deathCallback` - Metoda, która zostanie wywołana w czasie śmierci agenta
- `public List<double> lastInputs = new List<double>()` - Lista ostatnio zarejstrowanych wartości wejściowych
- `public List<double> lastOutputs = new List<double>()` - Lista ostatnio zarejestrowanych wartości wyjściowych
- `public float ViewArc` - Kąt widzenia w radianach

`public class InputMonitor : MonoBehaviour`

- `public Agent Target` - Referencja do agenta, który ma być obserwowany
- `public Text text` - Referencja do tekstu UI na którym mają zostać wypisane informacje o `Target` agencie

`public static class ScoreCalculator`

- `public static float CalculateScore(GameObject obj)` - Oblicza wynik dla danego agenta

`public class LifetimeScoreRule : MonoBehaviour, IScoringRule`

- `public float GetScore()` - Zwraca wynik czasu życia agenta

`public class Pulse` - Klasa reprezentująca aktywację

- `public double Value { get; set; }` - Zwraca wartość dla tego pulsu

`public class NeuralLayer` - Klasa reprezentująca warstwę neuronów

- `public List<Neuron> Neurons { get; set; }` - Lista neuronów w warstwie
- `public string Name { get; set; }` - Nazwa warstwy
- `public double Weight { get; set; }` - ???
- `public NeuralLayer(int count, double initialWeight, Func<double, double> activation, string name = "")` - Konstruktor warstwy
  - `int count` - Liczba neuronów, która będzie wygenerowana w tej warstwie
  - `double initialWeight` - Początkowa waga połączeń
  - `Func<double, double> activation` - Funkcja aktywacji, która zostanie nadana wszystkim neuronom w tej warstwie
  - `name` - Nazwa warstwy
- `public void Randomize(double lr)` - Wywołuje `Neuron.Randomize(lr)` na każdym neuronie w warstwie
- `public void Forward()` - Wywołuje `Neuron.Fire()` na każdym neuronie w warstwie
- `public override string ToString()` - Konwertuje wszystkie neurony na typ `string` i zwraca `string` zawierający wszystkie neurony

`public class Dendrite`

- `public Pulse InputPulse { get; set; }` - Impuls przychodzący
- `public double SynapticWeight { get; set; }` - Waga tego połączenia
- `public bool Learnable { get; set; }` - Flaga, gdy ustawiona na `true`, to połączenie może się uczyć
- `public Dendrite()` - Konstruktor połącznia
- `public void Randomize(double lr)` - Dodaje wartość w zakresie [`-lr`, `lr`] do wagi
- `public override string ToString()` - Konwertuje wagę połączenia na `string`

`public class Neuron`

- `public List<Dendrite> Dendrites { get; set; }` - Kolekcja połączeń przychodzących
- `public Pulse OutputPulse { get; set; }` - Ostatnio zarejestrowana wartość wychodząca
- `public Func<double, double> Activation` - Fukncja aktywacji
- `public Neuron(Func<double, double> activation)` - Konstruktor

  - `Func<double, double> activation` - Ustawia `Activation`

- `public void Randomize(double lr)` - Wywołuje `Dendrite.Randomize(lr)` na każdym z `Dendrite` w `Dendrites`
- `public void Fire()` - Ustawia `OutputPulse` na wynik funkcji aktywacji
- `public override string ToString()` - Konvertuje wszystkie `Dendrite` w `Dendrites` na typ `string`

`public static class ActivationFunc`

- `public static double Tanh(double x)` - Funkcja tangensa hiperbolicznego $1 - \frac{2}{e^{2x}+1}$
- `public static double Linear(double x)` - Funkcja liniowa $f(x) = x$
- `public static double BinaryStep(double x)` - Funkcja:

$$f(x) = \begin{cases} \text{-1} & \text{if x} < 0 \\ 0 & \text{if x} = 0 \\ 1 & \text{if x} > 0 \end{cases}$$

`public class NetworkModel` - Klasa reprezentująca sieć neuronową

- `public List<NeuralLayer> Layers { get; set; }` - Kolekcja warstw
- `public NetworkModel()` - Konstruktor
- `public NetworkModel DeepCopy()` - Metoda wykonująca kopię głęboką
- `public void AddLayer(NeuralLayer layer)` - Dodaje `layer` do Layers
- `public void Build()` - Buduje sieć, łączy ze sobą kolejne warstwy za pomocą obiektów klasy `Synapse`
- `public void Randomize(double lr)` - Wywołuje `Layer.Randomize` na każdym `Layer` z `Layers`
- `public List<double> Decide(List<double> X)` - Wykonuje sieć dla wektora wejściowego `X` i zwraca wynik w postacie wektora wyjściowego
- `public void Print()` - Wyświetla informacje o warstwach
- `public override string ToString()` - Konwertuje sieć na typ `string`

`public static class JsonService`

- `public static void SaveModelsList(List<NetworkModel> models)` - Zapisuje listę modeli do pliku JSON
- `public static List<NetworkModel> LoadModelsList(TextAsset jsonFile)` - Odczytuje listę modeli z pliku JSON i zwraca listę modeli
  - `TextAsset jsonFile` - Referencja do zasobu tekstowego, zawierającego plik JSON z modelami

Testy

Podczas uczenia sieci ukrytych w agentach, w większośći symulacji można było zaobserwować scenariusz opisywalny w czterech punktach.

1. Agenci pierwszej generacji zachowywali się zupełnie chaotycznie, kilku z nich uciekało z daleka od punktu startowego.

2. Jako że funkcja nagrody optymalizowała jak największy czas życia oraz przebytą odległość, wspomniani wcześniej "uciekinierzy", przekazywali swoje geny następnemu pokoleniu.

3. Przez kilka kolejnych generacji, agenci przebywali coraz większe odległości, by rozbić się o coraz dalszą scianę, przebywając po drodze coraz bardziej krętą trasę.

4. Po kilku lub kilkunastu epokach, można było wyróżnić dwa typy agentów, ąktywni", którzy objęli strategię polegającą na órbitowaniu"pomiędzy przeszkodami. Drugi typ to agenci łeniwi", ich strategia była znacznie bardziej defensywna. Polegała bowiem na staniu w miejsu, lub bardzo wolnemu poruszaniu się w kółko, w celu zminimalizowania ryzyka kolizji ze ścianą.

Pełen kod aplikacji

Kod znajduje się w repozytorium pod adresem: reee jak i również pod spodem.

MovementScoreRule.cs

```
using UnityEngine;


//Calculates score based on total moved distance
public class MovementScoreRule : MonoBehaviour, IScoringRule
{

    float score = 0;
    Vector3 lastPos;

    void Start()
    {
        lastPos = this.transform.position;
    }


    void FixedUpdate()
    {
        score += Vector3.Distance(this.transform.position, lastPos);
    }



    public float GetScore()
    {
        return score;
    }
}
```

Run.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using NeuralNetwork;
using UnityEngine;
using System.Linq;


//Class that represents data gathered from one simulation run
public class Run
{
    public string runName = "Run #N";
    public event EventHandler<List<AgentResult>> RunComplete;
    public List<GameObject> agents = new List<GameObject>();
    public List<AgentResult> results = new List<AgentResult>();

    public static GameObject agentPrefab;

    //Creates a new run with num_agents number of randomly initialized
       agents
    public Run(int num_agents)
    {
        for (int i = 0; i < num_agents; i++)
        {
            agents.Add(CreateNewAgent());
        }
    }

    //Creates a new run with agents initialized with given models
    public Run(List<NetworkModel> models)
    {
        foreach (NetworkModel m in models)
        {
            GameObject a = CreateNewAgent();
            a.GetComponent<Agent>().network = m;
            agents.Add(a);
        }
    }

    //Begins run by activating all agents
    public void BeginRun()
    {
        foreach (GameObject a in agents)
        {
            a.gameObject.SetActive(true);
        }
    }

    //Called when all agents died
    private void EndRun()
    {
        Debug.Log(runName + " ended");
        string resultString = "";
```

```
53          for (int i = 0; i < results.Count; i++)
54          {
55              resultString += "Agent #" + i + " | Score: " + results[i].
                    score;
56          }
57          Debug.Log(resultString);
58          agents.Clear();
59          RunComplete(this, results);
60      }
61
62      private void AgentDied(Agent a)
63      {
64          Debug.Log("Agent Died");
65          results.Insert(0, new AgentResult(ScoreCalculator.CalculateScore
                (a.gameObject), a.network));
66          agents.Remove(a.gameObject);
67          if (agents.Count == 0)
68          {
69              EndRun();
70          }
71      }
72
73
74      //Creates new empty agent, that is inactive
75      private GameObject CreateNewAgent()
76      {
77          GameObject agent = GameObject.Instantiate(agentPrefab);
78          agent.SetActive(false);
79          agent.GetComponent<Agent>().deathCallback = AgentDied;
80          return agent;
81      }
82
83
84      public class AgentResult
85      {
86          public double score;
87          public NetworkModel model;
88
89          public AgentResult(double score, NetworkModel model)
90          {
91              this.score = score;
92              this.model = model.DeepCopy();
93          }
94      }
95  }
```

IScoringRule.cs

```
1 public interface IScoringRule
2 {
3 
4     //get current score from component
5     float GetScore();
6 
7 }
```

RunManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using NeuralNetwork;


///<summary>
/// Class that is responsible for run managment
///</summary>
public class RunManager : MonoBehaviour
{
    //List of all results generated by any agent manager
    List<Run> runs = new List<Run>();
    int run_num = 0;
    [SerializeField]
    GameObject agentPrefab;

    ModelManager modelManager = null;
    // Start is called before the first frame update
    void Start()
    {
        Run.agentPrefab = agentPrefab;
        //For now the behaviour is to indefinetely repeat randomized runs;
        StartNewRun();
    }

    // Update is called once per frame
    void Update()
    {

    }

    void StartNewRun()
    {
     // modelManager = new ModelManager(JsonService.LoadModelsList((TextAsset)Resources.Load("20-01-14_20-23-44")));

        Run r = null;

        if (modelManager == null)
        {
            r = new Run(25);
            modelManager = new ModelManager(r.agents.Select(x => x.GetComponent<Agent>().network).ToList(), 0.05f);
        }
        else
        {
            modelManager.SaveTop(5);//kill all models and save top 5
            modelManager.Expand(); //expand models list to original size
            r = new Run(modelManager.Models);
```

```
51          }
52          r.runName = "Run #" + run_num;
53          r.RunComplete += OnRunEnded;
54          r.BeginRun();
55      }
56
57      void OnRunEnded(object sender, List<Run.AgentResult> results)
58      {
59          //Accept only Run senders
60          if (!(sender is Run r))
61              throw new ArgumentException("Sender is not of the type Run")
                    ;
62
63
64          //Unsubscribe from sender to avoid memory leak
65          r.RunComplete -= OnRunEnded;
66
67          //store run
68          runs.Add(r);
69          List<NetworkModel> models = r.results.OrderBy(x => x.score).
                Select(x => x.model).ToList();
70          Debug.Log(models[0].ToString());
71          StartNewRun();
72      }
73  }
```

CameraController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour
{
    public Transform target;
    public Vector3 offset;

    private Vector3 previousMousePos;
    private Vector3 currRot = new Vector3(0, 0, 0);
    private float rotSpeed = 100f;

    // Start is called before the first frame update
    void Start()
    {
        previousMousePos = Input.mousePosition;
        Cursor.lockState = CursorLockMode.Locked;
    }

    // Update is called once per frame
    void Update()
    {

        if (Input.GetButtonDown("ToggleCursor"))
        {
            if (Cursor.lockState != CursorLockMode.Locked)
            {
                Cursor.lockState = CursorLockMode.Locked;
            }
            else
            {
                Cursor.lockState = CursorLockMode.None;
            }
        }


        if (Input.GetButtonDown("BreakTarget"))
            target = null;

        if (Input.GetButtonDown("BestTarget"))
            target = ChooseBestTarget();


        if (target == null)
        {
            transform.Translate(Vector3.forward * Input.GetAxis("Forward
                "));
            transform.Translate(Vector3.up * Input.GetAxis("Up"));
            transform.Translate(Vector3.right * Input.GetAxis("Right"));

            Quaternion rot = Quaternion.identity;
```

```
53              currRot += new Vector3(-Input.GetAxis("Mouse Y"), Input.
                    GetAxis("Mouse X"), 0f) * Time.deltaTime * rotSpeed;
54              rot.eulerAngles = currRot;
55              transform.rotation = rot;
56          }
57
58          previousMousePos = Input.mousePosition;
59
60          if (target != null)
61              transform.position = target.position - offset;
62          //Check for input
63
64      }
65
66
67
68
69      Transform ChooseBestTarget()
70      {
71          throw new NotImplementedException();
72      }
73
74  }
```

ModelManager.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using System.Linq;
5  using UnityEngine;
6  using NeuralNetwork;
7
8  public class ModelManager
9  {
10     public List<NetworkModel> Models = new List<NetworkModel>();
11     public int NumModels { get; }
12     public double LearningRate { get; }
13
14
15     public ModelManager(List<NetworkModel> models, double learningRate =
           0.1f)
16     {
17         LearningRate = learningRate;
18         Models = models;
19         NumModels = models.Count;
20     }
21
22
23     public void SaveTop(int n)
24     {
25         if (n > Models.Count)
26         {
27             Debug.Log("Provided number is lower than count of models!");
28         }
29         else
30         {
31             Models.RemoveRange(0, n);
32             JsonService.SaveModelsList(Models);
33         }
34     }
35
36     public void Expand()
37     {
38         int n = Models.Count;
39
40         int clonesNeeded = NumModels - n;
41
42         List<NetworkModel> clones = new List<NetworkModel>();
43
44         if (n >= NumModels)
45         {
46             Debug.Log("Models collection is full!");
47         }
48         else
49         {
50             for (int i = 0; i < clonesNeeded; i++)
51             {
52                 clones.Add(Models[i % n].DeepCopy()); // add
```

```
                    randomization
                clones[clones.Count() - 1].Randomize(LearningRate);
            }
        }

        Models = new List<NetworkModel>(Models.Concat(clones));
    }


}
```

Agent.cs

```
1  using System;
2  using NeuralNetwork;
3  using UnityEngine;
4  using System.Collections.Generic;
5  using System.Collections;
6
7  public class Agent : MonoBehaviour
8  {
9      public NetworkModel network;
10
11     //Target cookie jar for every agent that exists (?TODO?: handle null
           case?)
12     public static Transform cookieJar;
13
14
15     //function to call when this agent dies
16     public Action<Agent> deathCallback;
17     public List<double> lastInputs = new List<double>(); //inputs that
           were fed in previous frame (for UI and debugging)
18     public List<double> lastOutputs = new List<double>(); //outputs that
           were outputted in previous frame (for UI and debugging)
19     private float ForceMultiplier = 10.0f;
20
21     //View arc in radians
22     [SerializeField]
23     private float viewArc = 2.0f;
24
25     public float ViewArc
26     {
27         get { return viewArc; }
28         set
29         {
30             viewArc = value;
31             arcStep = viewArc / (float)rayCount; // recalculate arcStep
32         }
33     }
34
35     [SerializeField]
36     //Number of rays that will be cast
37     private int rayCount = 8;
38
39
40     private float arcStep = 0f;
41
42
43     void Awake()
44     {
45         arcStep = viewArc / (float)rayCount;
46
47         cookieJar = GameObject.Find("cookieJar").transform;
48         network = new NetworkModel();
49         network.Layers.Add(new NeuralLayer(1 + rayCount, 0.0,
               ActivationFunc.Linear, "INPUT")); //rayCount + one for
```

```
            CookieJar position
50          network.Layers.Add(new NeuralLayer(11, 0.0, ActivationFunc.
                Linear, "HIDDEN"));
51          network.Layers.Add(new NeuralLayer(2, 0.0, ActivationFunc.Tanh,
                "OUTPUT"));
52          network.Build();
53          network.Randomize(0.5);
54      }
55
56      void Start()
57      {
58
59
60      }
61
62
63      void FixedUpdate()
64      {
65          lastOutputs = network.Decide(GatherInputs());
66          ParseOutput(lastOutputs);
67      }
68
69
70      ///<summary>Parses output of a neural network</summary>
71      ///Activations go as follows:
72      ///[0] - force on X axis
73      ///[1] - force on Z axis
74      private void ParseOutput(List<double> activations)
75      {
76          this.GetComponent<Rigidbody>().AddForce(new Vector3((float)
                activations[0] * ForceMultiplier, 0.0f, (float)activations[1]
                 * ForceMultiplier));
77      }
78
79      //Gathers inputs from enviroment
80      private List<double> GatherInputs()
81      {
82          List<double> results = new List<double>();
83          //1. raycast
84          for (int i = 0; i < rayCount; i++)
85          {
86              float curr_arc = i * arcStep * Mathf.PI;
87              Vector3 dir = new Vector3(Mathf.Cos(curr_arc), 0, Mathf.Sin(
                    curr_arc));
88              dir.Normalize(); //OPTM: Not needed as Cos and Sin are in
                    [-1,1]?
89              RaycastHit hit;
90              if (Physics.Raycast(this.transform.position, dir, out hit,
                    100.0f, 1 << 10))
91              {
92                  results.Add((double)hit.distance / 100.0f);
93                  Debug.DrawRay(transform.position, hit.point - transform.
                        position, Color.black, 0.01f, true);
94              }
95              else
```

```
96                  {
97                      results.Add(1.0f); // if nothing was hit, add max
98                  }
99              }
100
101             //2. get distance from the cookie jar
102             results.Add(Vector3.Distance(this.transform.position, cookieJar.
                    position) / 100.0f);
103             lastInputs = results;
104             return results;
105         }
106
107         public void OnCollisionEnter(Collision c)
108         {
109             //if layer is wall layer
110             if (c.collider.gameObject.layer == 10)
111             {
112                 //die...
113                 this?.deathCallback(this);
114                 Destroy(this.gameObject);
115             }
116         }
117
118         //EDITOR
119         void OnValidate()
120         {
121             //Because Unity does not support property exposing to the
                    Inspector, we use OnValidate (called whenever, whatever
                    changed by the Inspecotr)
122             //And force property to fire.
123             ViewArc = viewArc;
124         }
125
126
127 }
```

InputMonitor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


public class InputMonitor : MonoBehaviour
{


    public Agent Target;
    public Text text;
    // Start is called before the first frame update
    void Start()
    {
        if (Target == null)
            Debug.LogWarning("Target is null, no data will be shown.");
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        if (Target != null)
        {
            string input_data = "";
            string output_data = "";
            for (int i = 0; i < Target.lastInputs.Count; i++)
            {
                input_data += "Input " + i + " --- " + Target.lastInputs
                    [i] + "\n";
            }

            for (int i = 0; i < Target.lastOutputs.Count; i++)
            {
                output_data += "Output " + i + " --- " + Target.
                    lastOutputs[i] + "\n";
            }

            text.text = input_data + output_data;
        }

    }
}
```

ScoreCalculator.cs

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public static class ScoreCalculator
5  {
6
7
8      public static float CalculateScore(GameObject obj)
9      {
10         //Get all components with given interfaces and map them to
               theirs type names
11         IScoringRule[] rules = obj.GetComponents<IScoringRule>();
12         Dictionary<string, IScoringRule> ruleDictionary = new Dictionary
               <string, IScoringRule>();
13
14         foreach(IScoringRule r in rules)
15         {
16             ruleDictionary.Add(r.GetType().ToString(), r);
17         }
18
19
20         return ruleDictionary["MovementScoreRule"].GetScore() + 0.1f*
               ruleDictionary["LifetimeScoreRule"].GetScore();
21     }
22
23
24
25
26 }
```

LifetimeScoreRule.cs

```
using UnityEngine;

//Scoring rule: Add point depending on lifetime
public class LifetimeScoreRule : MonoBehaviour, IScoringRule
{
    float score = 0;

    void FixedUpdate()
    {
        score++;
    }

    public float GetScore()
    {
        return score;
    }
}
```

Pulse.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using UnityEngine;
using Newtonsoft.Json;

namespace NeuralNetwork
{
    [Serializable]
    public class Pulse
    {
        [JsonProperty]
        public double Value { get; set; }
    }
}
```

NeuralLayer.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using UnityEngine;
using Newtonsoft.Json;

namespace NeuralNetwork
{
    [Serializable]
    public class NeuralLayer{
        public List<Neuron> Neurons { get; set; }

        public string Name { get; set; }

        public double Weight { get; set; }

        Func<double, double> Activation;
        public NeuralLayer(int count, double initialWeight, Func<double,
            double> activation, string name = "")
        {
            Activation = activation;
            Neurons = new List<Neuron>();
            for (int i = 0; i < count; i++)
            {
                Neurons.Add(new Neuron(Activation));
            }

            Name = name;
        }


        public void Randomize(double lr)
        {
            foreach (var neuron in Neurons)
            {
                neuron.Randomize(lr);
            }
        }

        public void Forward()
        {
            foreach (var neuron in Neurons)
            {
                neuron.Fire();
            }

        }

        public override string ToString()
```

```
53          {
54              string tmp = "{\n";
55              for (int i = 0; i < Neurons.Count; i++)
56              {
57                  tmp += Neurons[i].ToString();
58                  if (i != Neurons.Count - 1)
59                      tmp += "\n";
60              }
61              tmp+="\n}";
62              return tmp;
63          }
64
65      }
66  }
```

Dendrite.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using UnityEngine;
using Newtonsoft.Json;

namespace NeuralNetwork
{

    [Serializable]
    public class Dendrite
    {
        public Pulse InputPulse { get; set; }

        public double SynapticWeight { get; set; }

        public bool Learnable { get; set; }

        public Dendrite()
        {
            SynapticWeight = 0;
        }

        public void Randomize(double lr)
        {
            float t = (float)lr;
            SynapticWeight += (double)UnityEngine.Random.Range(-t, t);
        }


        public override string ToString()
        {
            return SynapticWeight.ToString();
        }


    }
}
```

Neuron.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using UnityEngine;
using Newtonsoft.Json;

namespace NeuralNetwork
{
    [Serializable]
    public class Neuron
    {
        public List<Dendrite> Dendrites { get; set; }

        public Pulse OutputPulse { get; set; }

        public Func<double, double> Activation;

        public Neuron(Func<double, double> activation)
        {
            Dendrites = new List<Dendrite>();
            OutputPulse = new Pulse();
            Activation = activation;
        }

        public void Randomize(double lr)
        {
            foreach (var dendrite in Dendrites)
            {
                dendrite.Randomize(lr);
            }
        }

        public void Fire()
        {
            OutputPulse.Value = Sum();

            OutputPulse.Value = Activation(OutputPulse.Value);
        }

        private double Sum()
        {
            double computeValue = 0.0f;
            foreach (var d in Dendrites)
            {
                computeValue += d.InputPulse.Value * d.SynapticWeight;
            }

            return computeValue;
        }
```

```
54
55          public override string ToString()
56          {
57              string tmp = "[";
58              for (int i = 0; i < Dendrites.Count; i++)
59              {
60                  tmp += Dendrites[i].ToString();
61                  if (i != Dendrites.Count - 1)
62                      tmp += ", ";
63              }
64              tmp += "]";
65              return tmp;
66          }
67
68
69      }
70  }
```

Activation.cs

```
using UnityEngine;
using System;

namespace NeuralNetwork
{
    public static class ActivationFunc
    {
        public static double Tanh(double x)
        {
            return 1 - (2.0) / (Math.Exp(2 * x) + 1);
        }

        public static double Linear(double x)
        {
            return x;
        }

        public static double BinaryStep(double x)
        {
            if (x > 0)
            {
                return 1;
            }
            if (x == 0)
            {
                return 0;
            }
            if (x < 0)
            {
                return -1;
            }
            return 0;
        }

    }

}
```

NetworkModel.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using UnityEngine;
using Newtonsoft.Json;

namespace NeuralNetwork
{

    [Serializable]
    public class NetworkModel
    {
        public List<NeuralLayer> Layers { get; set; }

        public NetworkModel()
        {
            Layers = new List<NeuralLayer>();
        }

        public NetworkModel DeepCopy()
        {
            using (MemoryStream ms = new MemoryStream())
            {
                BinaryFormatter formatter = new BinaryFormatter();
                formatter.Context = new StreamingContext(
                    StreamingContextStates.Clone);
                formatter.Serialize(ms, this);
                ms.Position = 0;
                return (NetworkModel)formatter.Deserialize(ms);
            }
        }

        public void AddLayer(NeuralLayer layer)
        {
            int dendriteCount = 1;
            if (Layers.Count > 0)
            {
                dendriteCount = Layers[Layers.Count - 1].Neurons.Count;
            }

            foreach (var element in layer.Neurons)
            {
                for (int i = 0; i < dendriteCount; i++)
                {
                    element.Dendrites.Add(new Dendrite());
                }
            }
        }

```

```
53          public void Build()
54          {
55              int i = 0;
56              foreach (var layer in Layers)
57              {
58                  if (i >= Layers.Count - 1)
59                  {
60                      break;
61                  }
62                  var nextLayer = Layers[i + 1];
63                  CreateNetwork(layer, nextLayer);
64                  i++;
65              }
66          }
67
68
69          public void Randomize(double lr)
70          {
71              foreach (var layer in Layers)
72              {
73                  layer.Randomize(lr);
74              }
75          }
76
77          public List<double> Decide(List<double> X)
78          {
79              var inputLayer = Layers[0];
80              List<double> outputs = new List<double>();
81
82              for (int i = 0; i < X.Count; i++)
83              {
84                  inputLayer.Neurons[i].OutputPulse.Value = X[i];
85              }
86              ComputeOutput();
87              foreach (var neuron in Layers.Last().Neurons)
88              {
89                  outputs.Add(neuron.OutputPulse.Value);
90              }
91              return outputs;
92          }
93
94          public void Print()
95          {
96
97              Debug.Log("Name | Neurons");
98
99              foreach (var layer in Layers)
100             {
101                 Debug.Log(layer.Name + " | " + layer.Neurons.Count);
102             }
103         }
104
105         private void CreateNetwork(NeuralLayer connectingFrom,
                NeuralLayer connectingTo)
106         {
```

```
107                 foreach (var to in connectingTo.Neurons)
108                 {
109                     foreach (var from in connectingFrom.Neurons)
110                     {
111                         to.Dendrites.Add(new Dendrite() { InputPulse = from.
                            OutputPulse });
112                     }
113                 }
114             }
115 
116             private void ComputeOutput()
117             {
118                 bool first = true;
119                 foreach (var layer in Layers)
120                 {
121                     if (first)
122                     {
123                         first = false;
124                     }
125                     else
126                     {
127                         layer.Forward();
128                     }
129                 }
130             }
131 
132             public override string ToString()
133             {
134                 string tmp = "";
135                 for (int i = 1; i < Layers.Count; i++)
136                 {
137                     tmp += Layers[i].ToString();
138                     if (i != Layers.Count - 1)
139                         tmp += "\n";
140                 }
141                 tmp+="\n";
142                 return tmp;
143             }
144 
145 
146     }
147 }
```

JsonService.cs

```
using System;
using System.IO;
using System.Text;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using UnityEngine;
using NeuralNetwork;
using Newtonsoft.Json;
public static class JsonService
{
    public static void SaveModelsList(List<NetworkModel> models)
    {
        string timeString = DateTime.Now.ToString("yy-MM-dd_HH-mm-ss");
        timeString = timeString.Replace(' ', '_');

      JsonSerializerSettings settings = new JsonSerializerSettings();
      settings.NullValueHandling = NullValueHandling.Include;
      settings.ReferenceLoopHandling = ReferenceLoopHandling.Serialize;

        string jsonString = JsonConvert.SerializeObject(models);
        string path = Application.dataPath + "/jsonModels/" + timeString
            + ".json";
        File.WriteAllText(path, jsonString);
    }

    public static List<NetworkModel> LoadModelsList(TextAsset jsonFile)
    {
        List<NetworkModel> models = JsonConvert.DeserializeObject<List<
            NetworkModel>>(jsonFile.ToString());
      return models;
    }

}
```