Systemy Sztucznej Inteligencji

Dokumentacja projektu Unity Neural Network Car

Artur Bednarczyk, Dawid Grajewski, grupa A

26 maja 2018

Część I

Opis programu

Unity Neural Network Car to symulacja samochodu poruszającego się po torze. Jednak nie tylko użytkownik może kontrolować pojazd, a również sztuczna inteligencja. Implementacja zaawansowanej technologii pozwoliła na „nauczenie" samochodu w jaki sposób przejechać przez cały tor i nie rozbić się na najbliższym zakręcie.

Instrukcja obsługi

Uruchamiamy aplikację i możemy obserwować ruch pojazdu.

Dodatkowe informacje

Dodatkowe informacje, czyli jakie?

Część II

Opis działania

Sieć neuronowa

Sieć neuronowa jest strukturą inspirowaną budową naturalnych neuronów, łączących je synaps oraz układów nerwowych. Wykorzystana w projekcie sieć jest wielowarstwową siecią jednokierunkową korzystającą z algorytmu propagacji wstecznej.

Budowa sieci

Wykorzystana sieć składa się z trzech głównych warstw.

- Warstwa wejścia.
- Warstwy ukryte.
- Warstwa wyjścia.

Warstwa wejścia Każdy neuron tej warstwy przekazuje do warstwy ukrytej początkowe wartości.

Warstwy ukryte Tutaj dzieje się wszystko co najważniejsze. Neurony mają przypisane wagi, które początkowo są losowe. W ramach uczenia się, wagi są dostosowywane tak, aby rezultat końcowy był jak najbliżej spodziewanego.

Warstwa wyjścia To tutaj nauczona już sieć daje nam wynik.

Uczenie sieci

Uczenie sieci jest realizowane poprzez podanie jej zestawu danych wejściowych oraz spodziewanych wyników. W tym projekcie wykorzystano metody takie jak:

- Propagacja Wsteczna
- Biases
- Momentum
- Współczynnik uczenia

Początkowo wagi przy neuronach są losowe. Każdy z neuronów posiada swój blok sumujący, gdzie oblicz wartość sygnału:

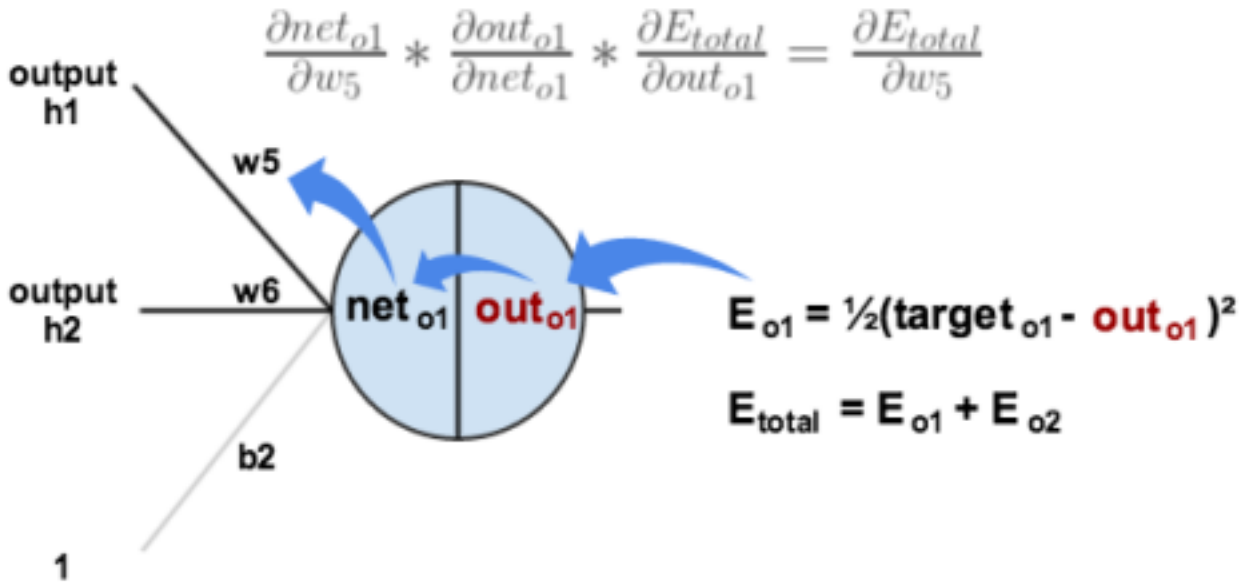
$$\sum_{i=1}^{n} w_i \cdot x_i - p$$

Gdzie n to liczba wejść, w to waga, x to wartość wejściowa oraz p to wartość progowa.

Propagacja wsteczna Po każdej serii danych wejściowych, wynik jest sprawdzany z spodziewanym wynikiem. Następnie błąd jest liczony wzorem:

$$E = \sum \frac{1}{2}(target - out)^2$$

gdzie E to błąd, *target* to spodziewane wyniki, *out* to otrzymane wyniki. Celem propagacji wstecznej jest zminimalizowanie błędu. Wykorzystujemy do tego regułę łańcuchową, która pozwoli obliczyć pochodne funkcji złożonych.



Następnie, wyznaczamy jak bardzo wynik o_1 zmienia się w odniesieniu do wejścia oraz do wag.

Biases tutaj trochę o tym...

Momentum tutaj trochę o tym...

Współczynnik uczenia tutaj trochę o tym...

Funkcja Aktywacji

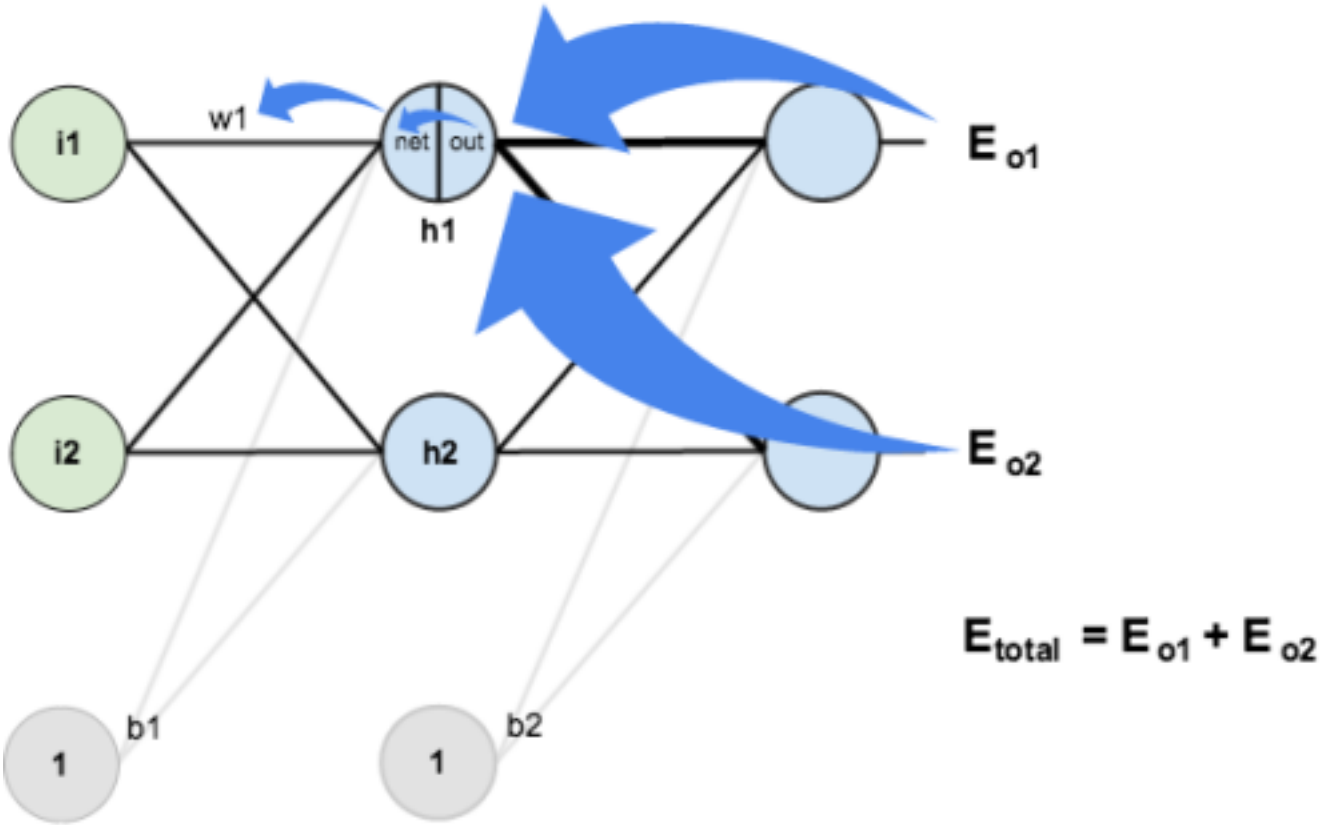
Wartość wyjścia neuronów jest oblicznaa za pomocą funkcji aktywacji. W tym projekcie została użyta funkcja sigmoidalna:

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

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



Algorytm

mile widziany pseudokod z użyciem biblioteki LaTeX

pseudo kod czego naszego AI tutaj sie da; o ile da rade te wszystkie rzeczy w coś krótkiego zwinąć, bo inaczej jest w tym sens, skoro cały kod będzie niżej?

Jak to nie będzie zbyt ogromne, to mogę zrobić ładny(jak się uda) schemat blokowy.

Bazy danych

Należy pokazać przykładowe dane, które były wykorzystywane podczas uczenia klasyfikatorów.
Czyli te 120 wierszy czy ile tego tam mamy? Czy wystarczy mu pokazać, że mamy dane w takiej formie i wkleić kilka wierszy tego i dać trzy kropki, o takie "...".

Implementacja

Opis, zasada i działanie programu ze względu na podział na pliki, nastepnie funkcje programu wraz ze szczegółowym opisem działania

`proszę zwrócić uwagę na wychodzenie poza obszar kartki.`

`No spoko, ja tu dopiljnuje, żeby wyglądało po ludzku.`

Testy

Tutaj powinna pojawić się analiza uzyskanych wyników oraz wykresy/pomiary.
No lekko, tutaj wleci super wykres nakodzony w pytonie.

Pełen kod aplikacji

tutaj wklejamy pełen kod.
Dalej nie rozumiem sensu tego punktu. ale niech będzie. Pewnie wszystko będzie ważyło trochę zbyt dużo, żeby to wrzucić na platformę xD

Neuron

```
1  public class Neuron
2  {
3    #region -- Properties --
4    public Guid Id { get; set; }
5    public List<Synapse> InputSynapses { get; set; }
6    public List<Synapse> OutputSynapses { get; set; }
7    public double Bias { get; set; }
8    public double BiasDelta { get; set; }
9    public double Gradient { get; set; }
10   public double Value { get; set; }
11   #endregion
12
13   #region -- Constructors --
14   public Neuron()
15   {
16     Id = Guid.NewGuid();
17     InputSynapses = new List<Synapse>();
18     OutputSynapses = new List<Synapse>();
19     Bias = Network.GetRandom();
20   }
21
22   public Neuron(IEnumerable<Neuron> inputNeurons) : this()
23   {
24     foreach (var inputNeuron in inputNeurons)
25     {
26       var synapse = new Synapse(inputNeuron, this);
27       inputNeuron.OutputSynapses.Add(synapse);
28       InputSynapses.Add(synapse);
29     }
30   }
31   #endregion
32
33   #region -- Values & Weights --
34   public virtual double CalculateValue()
35   {
36     return Value = Sigmoid.Output(InputSynapses.Sum(a => a.
         Weight * a.InputNeuron.Value) + Bias);
```

```
37    }
38
39    public double CalculateError(double target)
40    {
41      return target - Value;
42    }
43
44    public double CalculateGradient(double? target = null)
45    {
46      if (target == null)
47        return Gradient = OutputSynapses.Sum(a => a.OutputNeuron.
            Gradient * a.Weight) * Sigmoid.Derivative(Value);
48
49      return Gradient = CalculateError(target.Value) * Sigmoid.
          Derivative(Value);
50    }
51
52    public void UpdateWeights(double learnRate, double momentum)
53    {
54      var prevDelta = BiasDelta;
55      BiasDelta = learnRate * Gradient;
56      Bias += BiasDelta + momentum * prevDelta;
57
58      foreach (var synapse in InputSynapses)
59      {
60        prevDelta = synapse.WeightDelta;
61        synapse.WeightDelta = learnRate * Gradient * synapse.
            InputNeuron.Value;
62        synapse.Weight += synapse.WeightDelta + momentum *
            prevDelta;
63      }
64    }
65    #endregion
66  }
```

Synapse

```
1   public class Synapse
2   {
3     #region -- Properties --
4     public Guid Id { get; set; }
5     public Neuron InputNeuron { get; set; }
6     public Neuron OutputNeuron { get; set; }
7     public double Weight { get; set; }
8     public double WeightDelta { get; set; }
9     #endregion
```

```
10
11      #region -- Constructor --
12      public Synapse() { }
13
14      public Synapse(Neuron inputNeuron, Neuron outputNeuron)
15      {
16        Id = Guid.NewGuid();
17        InputNeuron = inputNeuron;
18        OutputNeuron = outputNeuron;
19        Weight = Network.GetRandom();
20      }
21      #endregion
22    }
```

Network

```
1     public class Network
2     {
3       #region -- Properties --
4       public double LearnRate { get; set; }
5       public double Momentum { get; set; }
6       public List<Neuron> InputLayer { get; set; }
7       public List<List<Neuron>> HiddenLayers { get; set; }
8       public List<Neuron> OutputLayer { get; set; }
9           public bool ShowIterationError { get; set; }
10      #endregion
11
12      #region -- Globals --
13      private static readonly Random Random = new Random();
14      #endregion
15
16      #region -- Constructor --
17      public Network()
18      {
19        LearnRate = 0;
20        Momentum = 0;
21        InputLayer = new List<Neuron>();
22        HiddenLayers = new List<List<Neuron>>();
23        OutputLayer = new List<Neuron>();
24      }
25
26          public Network(int inputSize, int[] hiddenSizes, int
               outputSize, double? learnRate = null, double?
               momentum = null, bool showIterationError = false)
27      {
28              ShowIterationError = showIterationError;
```

```
29        LearnRate = learnRate ?? .4;
30        Momentum = momentum ?? .9;
31        InputLayer = new List<Neuron>();
32        HiddenLayers = new List<List<Neuron>>();
33        OutputLayer = new List<Neuron>();
34
35        for (var i = 0; i < inputSize; i++)
36          InputLayer.Add(new Neuron());
37
38        var firstHiddenLayer = new List<Neuron>();
39        for (var i = 0; i < hiddenSizes[0]; i++)
40          firstHiddenLayer.Add(new Neuron(InputLayer));
41
42        HiddenLayers.Add(firstHiddenLayer);
43
44        for (var i = 1; i < hiddenSizes.Length; i++)
45        {
46          var hiddenLayer = new List<Neuron>();
47          for (var j = 0; j < hiddenSizes[i]; j++)
48            hiddenLayer.Add(new Neuron(HiddenLayers[i - 1]));
49          HiddenLayers.Add(hiddenLayer);
50        }
51
52        for (var i = 0; i < outputSize; i++)
53          OutputLayer.Add(new Neuron(HiddenLayers.Last()));
54      }
55      #endregion
56
57      #region -- Training --
58      public void Train(List<DataSet> dataSets, int numEpochs)
59      {
60        for (var i = 0; i < numEpochs; i++)
61        {
62          foreach (var dataSet in dataSets)
63          {
64            ForwardPropagate(dataSet.Values);
65            BackPropagate(dataSet.Targets);
66          }
67        }
68      }
69
70      public void Train(List<DataSet> dataSets, double
           minimumError)
71      {
72        var error = 1.0;
```

```
73        var numEpochs = 0;
74
75        while (error > minimumError && numEpochs < int.MaxValue)
76        {
77          var errors = new List<double>();
78          foreach (var dataSet in dataSets)
79          {
80            ForwardPropagate(dataSet.Values);
81            BackPropagate(dataSet.Targets);
82            errors.Add(CalculateError(dataSet.Targets));
83          }
84          error = errors.Average();
85                  numEpochs++;
86
87                  if(ShowIterationError)
88                      Console.WriteLine(error);
89        }
90      }
91
92      private void ForwardPropagate(params double[] inputs)
93      {
94        var i = 0;
95        InputLayer.ForEach(a => a.Value = inputs[i++]);
96        HiddenLayers.ForEach(a => a.ForEach(b => b.CalculateValue
            ()));
97        OutputLayer.ForEach(a => a.CalculateValue());
98      }
99
100     private void BackPropagate(params double[] targets)
101     {
102       var i = 0;
103       OutputLayer.ForEach(a => a.CalculateGradient(targets[i++])
            );
104       HiddenLayers.Reverse();
105       HiddenLayers.ForEach(a => a.ForEach(b => b.
            CalculateGradient()));
106       HiddenLayers.ForEach(a => a.ForEach(b => b.UpdateWeights(
            LearnRate, Momentum)));
107       HiddenLayers.Reverse();
108       OutputLayer.ForEach(a => a.UpdateWeights(LearnRate,
            Momentum));
109     }
110
111     public double[] Compute(params double[] inputs)
112     {
```

```
113         ForwardPropagate(inputs);
114         return OutputLayer.Select(a => a.Value).ToArray();
115       }
116
117       private double CalculateError(params double[] targets)
118       {
119         var i = 0;
120         return OutputLayer.Sum(a => Math.Abs(a.CalculateError(
              targets[i++])));
121       }
122       #endregion
123
124       #region -- Helpers --
125       public static double GetRandom()
126       {
127         return 2 * Random.NextDouble() - 1;
128       }
129       #endregion
130     }
131
132     #region -- Enum --
133     public enum TrainingType
134     {
135       Epoch,
136       MinimumError
137     }
138     #endregion
```

Sigmoid

```
1   public static class Sigmoid
2   {
3     public static double Output(double x)
4     {
5       return x < -45.0 ? 0.0 : x > 45.0 ? 1.0 : 1.0 / (1.0 +
          Math.Exp(-x));
6     }
7
8     public static double Derivative(double x)
9     {
10      return x * (1 - x);
11    }
12  }
```

Dataset

```
1   public class DataSet
2   {
3     #region -- Properties --
4     public double[] Values { get; set; }
5     public double[] Targets { get; set; }
6     #endregion
7
8     #region -- Constructor --
9     public DataSet(double[] values, double[] targets)
10    {
11      Values = values;
12      Targets = targets;
13    }
14    #endregion
15  }
```

ExportHelper

```
1   public static class ExportHelper
2   {
3         public static void ExportNetwork(Network network, string
              filename, ISerializer serializer)
4         {
5             var dn = GetHelperNetwork(network);
6
7             serializer.Serialize(dn, filename);
8         }
9
10    private static HelperNetwork GetHelperNetwork(Network
         network)
11    {
12      var hn = new HelperNetwork
13      {
14        LearnRate = network.LearnRate,
15        Momentum = network.Momentum
16      };
17
18      //Input Layer
19      foreach (var n in network.InputLayer)
20      {
21        var neuron = new HelperNeuron
22        {
23          Id = n.Id,
24          Bias = n.Bias,
25          BiasDelta = n.BiasDelta,
26          Gradient = n.Gradient,
```

```
27            Value = n.Value
28          };
29
30          hn.InputLayer.Add(neuron);
31
32          foreach (var synapse in n.OutputSynapses)
33          {
34            var syn = new HelperSynapse
35            {
36              Id = synapse.Id,
37              OutputNeuronId = synapse.OutputNeuron.Id,
38              InputNeuronId = synapse.InputNeuron.Id,
39              Weight = synapse.Weight,
40              WeightDelta = synapse.WeightDelta
41            };
42
43            hn.Synapses.Add(syn);
44          }
45        }
46
47        //Hidden Layer
48        foreach (var l in network.HiddenLayers)
49        {
50          var layer = new List<HelperNeuron>();
51
52          foreach (var n in l)
53          {
54            var neuron = new HelperNeuron
55            {
56              Id = n.Id,
57              Bias = n.Bias,
58              BiasDelta = n.BiasDelta,
59              Gradient = n.Gradient,
60              Value = n.Value
61            };
62
63            layer.Add(neuron);
64
65            foreach (var synapse in n.OutputSynapses)
66            {
67              var syn = new HelperSynapse
68              {
69                Id = synapse.Id,
70                OutputNeuronId = synapse.OutputNeuron.Id,
71                InputNeuronId = synapse.InputNeuron.Id,
```

```
72                Weight = synapse.Weight,
73                WeightDelta = synapse.WeightDelta
74              };
75
76              hn.Synapses.Add(syn);
77            }
78          }
79
80          hn.HiddenLayers.Add(layer);
81        }
82
83        //Output Layer
84        foreach (var n in network.OutputLayer)
85        {
86          var neuron = new HelperNeuron
87          {
88            Id = n.Id,
89            Bias = n.Bias,
90            BiasDelta = n.BiasDelta,
91            Gradient = n.Gradient,
92            Value = n.Value
93          };
94
95          hn.OutputLayer.Add(neuron);
96
97          foreach (var synapse in n.OutputSynapses)
98          {
99            var syn = new HelperSynapse
100           {
101             Id = synapse.Id,
102             OutputNeuronId = synapse.OutputNeuron.Id,
103             InputNeuronId = synapse.InputNeuron.Id,
104             Weight = synapse.Weight,
105             WeightDelta = synapse.WeightDelta
106           };
107
108           hn.Synapses.Add(syn);
109         }
110       }
111
112       return hn;
113     }
114   }
```

ImportHelper

```
1   public static class ImportHelper
2   {
3         public static Network ImportNetwork(string filename,
              ISerializer serializer)
4         {
5             var dn = GetHelperNetwork(filename, serializer);
6             if (dn == null) return null;
7
8             var network = new Network();
9             var allNeurons = new List<Neuron>();
10
11            network.LearnRate = dn.LearnRate;
12            network.Momentum = dn.Momentum;
13
14            //Input Layer
15            foreach (var n in dn.InputLayer)
16            {
17                var neuron = new Neuron
18                {
19                    Id = n.Id,
20                    Bias = n.Bias,
21                    BiasDelta = n.BiasDelta,
22                    Gradient = n.Gradient,
23                    Value = n.Value
24                };
25
26                network.InputLayer.Add(neuron);
27                allNeurons.Add(neuron);
28            }
29
30            //Hidden Layers
31            foreach (var layer in dn.HiddenLayers)
32            {
33                var neurons = new List<Neuron>();
34                foreach (var n in layer)
35                {
36                    var neuron = new Neuron
37                    {
38                        Id = n.Id,
39                        Bias = n.Bias,
40                        BiasDelta = n.BiasDelta,
41                        Gradient = n.Gradient,
42                        Value = n.Value
43                    };
44
```

```
45                      neurons.Add(neuron);
46                      allNeurons.Add(neuron);
47                  }
48
49                  network.HiddenLayers.Add(neurons);
50              }
51
52              //Export Layer
53              foreach (var n in dn.OutputLayer)
54              {
55                  var neuron = new Neuron
56                  {
57                      Id = n.Id,
58                      Bias = n.Bias,
59                      BiasDelta = n.BiasDelta,
60                      Gradient = n.Gradient,
61                      Value = n.Value
62                  };
63
64                  network.OutputLayer.Add(neuron);
65                  allNeurons.Add(neuron);
66              }
67
68              //Synapses
69              foreach (var syn in dn.Synapses)
70              {
71                  var synapse = new Synapse { Id = syn.Id };
72                  var inputNeuron = allNeurons.First(x => x.Id ==
                        syn.InputNeuronId);
73                  var outputNeuron = allNeurons.First(x => x.Id ==
                        syn.OutputNeuronId);
74                  synapse.InputNeuron = inputNeuron;
75                  synapse.OutputNeuron = outputNeuron;
76                  synapse.Weight = syn.Weight;
77                  synapse.WeightDelta = syn.WeightDelta;
78
79                  inputNeuron.OutputSynapses.Add(synapse);
80                  outputNeuron.InputSynapses.Add(synapse);
81              }
82
83              return network;
84          }
85
86          private static HelperNetwork GetHelperNetwork(string
                filename, ISerializer serializer)
```

```
87          {
88              return serializer.Deserialize(filename);
89          }
90
91      }
```

Helpers

```
1   public class HelperNetwork
2   {
3     public double LearnRate { get; set; }
4     public double Momentum { get; set; }
5     public List<HelperNeuron> InputLayer { get; set; }
6     public List<List<HelperNeuron>> HiddenLayers { get; set; }
7     public List<HelperNeuron> OutputLayer { get; set; }
8     public List<HelperSynapse> Synapses { get; set; }
9
10    public HelperNetwork()
11    {
12      InputLayer = new List<HelperNeuron>();
13      HiddenLayers = new List<List<HelperNeuron>>();
14      OutputLayer = new List<HelperNeuron>();
15      Synapses = new List<HelperSynapse>();
16    }
17  }
18
19  public class HelperNeuron
20  {
21    public Guid Id { get; set; }
22    public double Bias { get; set; }
23    public double BiasDelta { get; set; }
24    public double Gradient { get; set; }
25    public double Value { get; set; }
26  }
27
28  public class HelperSynapse
29  {
30    public Guid Id { get; set; }
31    public Guid OutputNeuronId { get; set; }
32    public Guid InputNeuronId { get; set; }
33    public double Weight { get; set; }
34    public double WeightDelta { get; set; }
35  }
```

Serializer

```
1   public interface ISerializer
2 {
3     void Serialize(HelperNetwork network, string filename);
4     HelperNetwork Deserialize(string filename);
5 }
6
7 public class XmlSerializer : ISerializer
8 {
9     public HelperNetwork Deserialize(string filename)
10    {
11        using (var reader = new StreamReader(filename))
12        {
13            var serializer = new System.Xml.Serialization.
                  XmlSerializer(typeof(HelperNetwork));
14            return serializer.Deserialize(reader) as
                  HelperNetwork;
15        }
16    }
17
18    public void Serialize(HelperNetwork network, string filename
          )
19    {
20        using (var writer = new StreamWriter(filename))
21        {
22            var serializer = new System.Xml.Serialization.
                  XmlSerializer(network.GetType());
23            serializer.Serialize(writer, network);
24        }
25    }
26 }
```

CarController

```
1   public abstract class CarController : MonoBehaviour
2 {
3     [Header("Steering")]
4     public int MovementSpeed;
5     public int RotationSpeed;
6
7     [Header("Sensors")]
8     public float SensorLength;
9     public float SensorAngle;
10
11    public float DistanceToLeftWall { get; private set; }
12    public float DistanceToRightWall { get; private set; }
13
```

```
14
15    // Update is called once per frame
16    protected virtual void Update () {
17          getDistanceToWalls();
18      }
19
20      public void MoveForward()
21      {
22          transform.position += transform.forward * MovementSpeed
               * Time.deltaTime;
23      }
24
25      public void TurnLeft()
26      {
27          transform.Rotate(Vector3.down * Time.deltaTime *
               RotationSpeed);
28      }
29
30      public void TurnRight()
31      {
32          transform.Rotate(Vector3.up * Time.deltaTime *
               RotationSpeed);
33      }
34
35      private void getDistanceToWalls()
36      {
37          RaycastHit leftHit;
38          RaycastHit rightHit;
39          Vector3 leftSensorStartPosition = transform.position;
40          Vector3 rightSensorStartPosition = transform.position;
41
42          if (Physics.Raycast(leftSensorStartPosition, Quaternion.
               AngleAxis(-SensorAngle, transform.up) * transform.
               forward, out leftHit, SensorLength))
43          {
44              DistanceToLeftWall = Vector3.Distance(
                   leftSensorStartPosition, leftHit.point);
45              Debug.DrawLine(leftSensorStartPosition, leftHit.
                   point);
46          }
47
48          if (Physics.Raycast(rightSensorStartPosition, Quaternion
               .AngleAxis(SensorAngle, transform.up) * transform.
               forward, out rightHit, SensorLength))
49          {
```

```
50              DistanceToRightWall = Vector3.Distance(
                    rightSensorStartPosition, rightHit.point);
51              Debug.DrawLine(rightSensorStartPosition, rightHit.
                    point);
52          }
53      }
54
55      void OnTriggerEnter(Collider other)
56      {
57          SceneManager.LoadScene(SceneManager.GetActiveScene().
                name);
58      }
59  }
```

AutonomicCarController

```
1   public class AutonomicCarController : CarController
2   {
3       [Header("Neural network holder")]
4       // Odwolanie do obiektu z siecia neuronowa
5       public BrainController Brain;
6
7       private const double turnRightCondition = 2f / 3f;
8       private const double turnLeftCondition = 1f / 3f;
9
10      // Update is called once per frame
11      protected override void Update ()
12      {
13          base.Update();
14          MoveForward();
15          steer();
16    }
17
18      private void steer()
19      {
20          neuralNetworkSteer();
21      }
22
23      private void simpleSteer()
24      {
25          var difference = DistanceToRightWall -
                DistanceToLeftWall;
26          makeDecision(difference, 1, -1);
27      }
28
29      private void neuralNetworkSteer()
```

```
30      {
31          var networkOutput = Brain.Compute(DistanceToLeftWall,
               DistanceToRightWall);
32          makeDecision(networkOutput, turnRightCondition,
               turnLeftCondition);
33      }
34
35      private void makeDecision(double value, double
           turnRightCondition, double turnLeftCondition)
36      {
37          if (value > turnRightCondition)
38              TurnRight();
39          else if (value < turnLeftCondition)
40              TurnLeft();
41      }
42  }
```

ManualCarController

```
1   public class ManualCarController : CarController
2   {
3       [Header("Steering")]
4       public KeyCode UpKey = KeyCode.W;
5       public KeyCode RightKey = KeyCode.D;
6       public KeyCode LeftKey = KeyCode.A;
7
8       [Header("Learning Data")]
9       public bool CollectLearningData;
10      public double CollectiongDataInterval;
11      public string LearnignDataFileName;
12
13      private double elapsedTimeSinceLastCollection = 0;
14      private const char learningFileDelimiter = ';';
15
16
17      // Update is called once per frame
18      protected override void Update ()
19      {
20          base.Update();
21
22          move();
23
24          if (CollectLearningData)
25          {
26              elapsedTimeSinceLastCollection += Time.deltaTime;
27
```

```
28              if (elapsedTimeSinceLastCollection >=
                    CollectiongDataInterval)
29              {
30                  collectLearningdata();
31                  elapsedTimeSinceLastCollection = 0;
32              }
33          }
34      }
35
36      private void move()
37      {
38          if (Input.GetKey(UpKey))
39          {
40              MoveForward();
41          }
42
43          if (Input.GetKey(RightKey))
44          {
45              TurnRight();
46          }
47          if (Input.GetKey(LeftKey))
48          {
49              TurnLeft();
50          }
51      }
52
53      private void collectLearningdata()
54      {
55          using (var writer = File.AppendText(LearnignDataFileName
                ))
56          {
57              var expectedOutput = Sigmoid.Output(
                    DistanceToRightWall - DistanceToLeftWall);
58              Debug.Log(expectedOutput);
59              writer.WriteLine(string.Format("{0}{3}{1}{3}{2}",
                    DistanceToLeftWall, DistanceToRightWall,
                    expectedOutput, learningFileDelimiter));
60              Debug.Log(DistanceToLeftWall + " " +
                    DistanceToRightWall + " " + expectedOutput);
61          }
62      }
63  }
```

BrainController

```
1  public class BrainController : MonoBehaviour
```

```
2 {
3     [Header("Learned")]
4     public string LearnedNetworkFileName;
5 
6     [Header("Learning")]
7     public bool TrainOnInit;
8     public string LearningFileName;
9     public double MaxError;
10 
11    private const char _learningFileDelimiter = ';';
12 
13    public NeuralNetwork.NetworkModels.Network NeuralNetwork {
          get; private set; }
14 
15  // Use this for initialization
16  void Start ()
17    {
18        if (TrainOnInit)
19            Train();
20        else
21            Load();
22    }
23 
24    /// <summary>
25    /// Wyciaga z sieci wynik na podstawie odleglosci od scian
26    /// </summary>
27    /// <param name="distanceToLeftWall"></param>
28    /// <param name="distanceToRightWall"></param>
29    /// <returns></returns>
30    public double Compute(double distanceToLeftWall, double
          distanceToRightWall)
31    {
32        return NeuralNetwork.Compute(new double[] {
              distanceToLeftWall, distanceToRightWall })[0];
33    }
34 
35    public void Train()
36    {
37        NeuralNetwork.Train(getDatasets(), MaxError);
38    }
39 
40    public void Save()
41    {
42        ExportHelper.ExportNetwork(NeuralNetwork,
              LearnedNetworkFileName, new XmlSerializer());
```

```
43      }
44
45      public void Load()
46      {
47          NeuralNetwork = ImportHelper.ImportNetwork(
                LearnedNetworkFileName, new XmlSerializer());
48      }
49
50      public void TestNetwork()
51      {
52          double l = 14;
53          double r = 14;
54          Debug.Log (NeuralNetwork.Compute(new double[] { l, r })
                [0] );
55
56          l = 20;
57          r = 10;
58          Debug.Log(NeuralNetwork.Compute(new double[] { l, r })
                [0]);
59
60          l = 10.72468;
61          r = 42.37657;
62          Debug.Log(NeuralNetwork.Compute(new double[] { l, r })
                [0]);
63      }
64
65      private List<DataSet> getDatasets()
66      {
67          var result = new List<DataSet>();
68
69          using (var reader = new StreamReader(LearningFileName))
70          {
71              while(!reader.EndOfStream)
72              {
73                  var elements = reader.ReadLine().Split(
                        _learningFileDelimiter);
74                  var dataset = new DataSet(new double[] { double.
                        Parse(elements[0]), double.Parse(elements[1])
                         }, new double[] { double.Parse(elements[2])
                        });
75                  result.Add(dataset);
76              }
77          }
78
79          return result;
```

```
80          }
81  }
```