First applied deep learning

In this applied exercice, we will build a maze in unity and will try to teach deep learning agents to go through this maze without touching the walls, helped by a checkpoitn system. Instead of using backpropagation to make the network learn, we will select the most perfomant networks for one iteration, clone them and slightly mutate their weights. Hence, this lesson can be use as an introduction to genetic algorithms.

A labyrinth

Create a ground with a simple cube in unity.

Create a cube and name it Wall. Create a new tag named "Wall" and set the wall object with it. Add a box collider component and set it as trigger. Make this object a prefab and then use wall objects to create a small labyrinth.

A simple agent

First, we need to create the agent. It will be able to rotate, accelerate and brake on a 2d plane. It will be able to detect walls through 5 raycasts : front, left, diagonal left, right and diagonal right. We will start by creating a player controllable and agent, and later change the player's inputs to a neural network controlled inputs.

Create a cube, a texture and a material for it. Set the box collider as a trigger. Add a Character controller component to the object. Create a script component called Controller.

Here are the class variables:

```
public class Controller : MonoBehaviour
{
    private CharacterController unityController;
    private Vector3 agentPosition;

    // Outputs
    private Vector3 moveDir = Vector3.zero;
    public float currentVelocity = 0f;

    // Input
    public float distForward = 0f;
    public float distLeft = 0f;
    public float distDiagLeft = 0f;
    public float distRight = 0f;
    public float distDiagRight = 0f;

    // Parameters
    [SerializeField] private float rotationSpeed = 300f;
    [SerializeField] private float speed = 0.5f;
    [SerializeField] private float minVelocity = 0.0f;
    [SerializeField] private float maxVelocity = 3.0f;
    [SerializeField] private float accelerationRate = 0.2f;
    [SerializeField] private float decelerationRate = 0.8f;
```

```
[SerializeField] public float maxViewDistance = 30f;

// Become inactive if hits a wall
public bool isActive = true;
```

We will output the veiw variables in this paragraph, even if it will only be used in the learning part.

The start and update function will be such:

```
// Start is called before the first frame update
void Start()
{
    unityController = GetComponent<CharacterController>();
    lastPosition = transform.position;
}

// Update is called once per frame
void Update()
{
    if (!isActive) return;

    // Player control (test)
    if (Input.GetKey(KeyCode.UpArrow))
    {
        currentVelocity += (accelerationRate * Time.deltaTime);
    }
    else
    {
        currentVelocity -= (decelerationRate * Time.deltaTime);
    }
    // Agent movement
    currentVelocity =
        Mathf.Clamp(currentVelocity, minVelocity, maxVelocity);

    moveDir = new Vector3(0, 0, currentVelocity);
    moveDir *= speed;
    moveDir = transform.TransformDirection(moveDir);
    unityController.Move(moveDir);

    // Player rotation
    float rotationAngle =
        Input.GetAxis("Horizontal") * rotationSpeed * Time.deltaTime;

    transform.Rotate(0, rotationAngle, 0);


    // Agent vision
    RaycastVision();
}
```

The last functions will serve for the interactions.

```
private void RaycastVision()
{
    agentPosition = transform.position;
    Vector3 forwardDir = transform.forward;
    Vector3 rightDir = transform.right;
    Vector3 leftDir = rightDir * -1;
    Vector3 rightDiagDir = Vector3.Normalize(forwardDir + rightDir);
    Vector3 leftDiagDir = Vector3.Normalize(forwardDir + leftDir);

    Ray forwardRay = new Ray(agentPosition, forwardDir);
    Ray leftRay = new Ray(agentPosition, leftDir);
    Ray leftDiagRay = new Ray(agentPosition, leftDiagDir);
    Ray rightRay = new Ray(agentPosition, rightDir);
    Ray rightDiagRay = new Ray(agentPosition, rightDiagDir);

    RaycastHit hit;
    if (Physics.Raycast(forwardRay, out hit, maxViewDistance)
    && hit.transform.CompareTag("Wall"))
    {
        distForward = hit.distance;
    }

    if (Physics.Raycast(leftRay, out hit, maxViewDistance)
    && hit.transform.CompareTag("Wall"))
    {
        distLeft = hit.distance;
    }

    if (Physics.Raycast(leftDiagRay, out hit, maxViewDistance)
    && hit.transform.CompareTag("Wall"))
    {
        distDiagLeft = hit.distance;
    }

    if (Physics.Raycast(rightRay, out hit, maxViewDistance)
    && hit.transform.CompareTag("Wall"))
    {
        distRight = hit.distance;
    }

    if (Physics.Raycast(rightDiagRay, out hit, maxViewDistance)
    && hit.transform.CompareTag("Wall"))
    {
        distDiagRight = hit.distance;
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Wall"))
    {
        lastPosition = transform.position;
        isActive = false;
    }
}
```

Set this agent as a prefab (drag it to the content window). We will use the prefab later to generate multiple agents.

You can test your agent movement with keyboard or controller.

A neural netwok class

We will now create a class to handle a neural network. This class will act as the "brain" of the agent. We want to be able to copy it and mutate it, in order to implement the genetic iterative learning.

Here are the class variable. Note the network class is comparable, in order to sort netwoks by fitness.

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

public class NeuralNetwork : IComparable<NeuralNetwork>
{
    private int[] layers;

    // Neuron matrix
    private float[][] neurons;

    // Weight matrices
    private float[][][] weights;

    // Network fitness
    public float Fitness { get; set; }
```

The layers array give the number of neurons for each layer. The neuron matrix will hold the neurons values for each layers. The weight three dimensional array holds each layer's weight matrice.

We will create two constructors: one to initialize neurons and weights with the layer data, one that will copy a given neural network.

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

    public NeuralNetwork(NeuralNetwork copy)
    {
        layers = new int[copy.layers.Length];
```

```
        for (int i = 0; i < copy.layers.Length; i++)
        {
            layers[i] = copy.layers[i];
        }
        InitNeurons();
        InitWeights();
        CopyWeights(copy.weights);
    }
```

The InitWeight function will randomly initialize data in weight matrices. The other functions just create the necessary data structures.

```
    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();
    }

    private void InitWeights()
    {
        List<float[][]> weightsList = new List<float[][]>();

        // For all neurons with inputs (note i starts at 1)
        for (int i = 1; i < layers.Length; i++)
        {
            List<float[]> layerWeightsList = new List<float[]>();
            int nbNeuronsInPreviousLayer = layers[i - 1];

            // For all neurons in this layer
            for (int j = 0; j < neurons[i].Length; j++)
            {
                float[] neuronsWeights =
                    new float[nbNeuronsInPreviousLayer];
                // Init weights randomly between previous and next layer
                for (int k = 0; k < nbNeuronsInPreviousLayer; k++)
                {
                    neuronsWeights[k] = Random.Range(-1f, 1f);
                }
                layerWeightsList.Add(neuronsWeights);
            }
            weightsList.Add(layerWeightsList.ToArray());
        }
        weights = weightsList.ToArray();
    }

    private void CopyWeights(float[][][] copyWeights)
    {
        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] = copyWeights[i][j][k];
                }
            }
        }
    }
```

The FeedForward function will take inputs, make data go forward and finally return the output:

```
    public float[] FeedForward(float[] inputs)
    {
        // Add inputs in input neurons' matrix
        for (int i = 1; i < inputs.Length; i++)
        {
            neurons[0][i] = inputs[i];
        }

        // For each layer and each neuron in the layer
        for (int i = 1; i < layers.Length; i++)
        {
            for (int j = 0; j < neurons[i].Length; j++)
            {
                // Compute the value for this neuron,
                // then apply activation function
                float val = 0f;
                for (int k = 0; k < neurons[i - 1].Length; k++)
                {
                    val += weights[i - 1][j][k] * neurons[i - 1][k];
                }
                // Activation function is hyperbolic tangent
                neurons[i][j] = (float)Math.Tanh(val);
            }
        }

        // Return output layer
        return neurons[neurons.Length - 1];
    }
```

The Mutate function will give a chance for each weight to take back a randow value.

```
    public void Mutate(float condition)
    {
        // Mutate some weights if a value
        // between 0 and 100 is inferior to condition
        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++)
```

```
                {
                    float weight = weights[i][j][k];
                    float randNum = UnityEngine.Random.Range(0f, 100f);
                    if (randNum <= condition)
                    {
                        weight = UnityEngine.Random.Range(-1f, 1f);
                    }

                    weights[i][j][k] = weight;
                }
            }
        }
    }
```

Other functions are needed for utility:

```
    public int CompareTo(NeuralNetwork other)
    {
        if (other == null) return 1;

        if (Fitness > other.Fitness) return 1;
        if (Fitness < other.Fitness) return -1;
        return 0;
    }

    public void AddFitness(float fitnessP)
    {
        Fitness += fitnessP;
    }
}
```

Checkpoints

Before using the neural network with the agents, we want to incentive agents to move in a special direction. We thus create a Checkpoint prefab that will improve fitness when the agents will go through.

Create a cube with a texture and a material. Set the box collider to trigger. Add a Checkpoint tag and set the physics layer to Ignore Raycast (next to tag). Add a Checkpoint script on the object.

```
public class Checkpoint : MonoBehaviour
{
    public float RewardMultiplier = 1.0f;
}
```

Create different checkpoints with increasing RewardMultipliers.

Updating the controller to use the neural network

Here is the final Controller class, with the previous code in comments.

```
public class Controller : MonoBehaviour
```

```
{

    private CharacterController unityController;
    private Vector3 agentPosition;

    // Outputs
    private Vector3 moveDir = Vector3.zero;
    public float currentVelocity = 0f;

    // Input
    public float distForward = 0f;
    public float distLeft = 0f;
    public float distDiagLeft = 0f;
    public float distRight = 0f;
    public float distDiagRight = 0f;

    // Parameters
    [SerializeField] private float rotationSpeed = 300f;
    [SerializeField] private float speed = 0.5f;
    [SerializeField] private float minVelocity = 0.0f;
    [SerializeField] private float maxVelocity = 3.0f;
    [SerializeField] private float accelerationRate = 0.2f;
    [SerializeField] private float decelerationRate = 0.8f;
    [SerializeField] public float maxViewDistance = 30f;


    // Become inactive if hits a wall
    public bool isActive = true;


    // Learning
    public float fitness = 0f;
    private Vector3 lastPosition;
    private float distanceTraveled = 0f;
    [SerializeField] private float fitnessTimeDecreaseRate = 10f;
    [SerializeField] private float fitnessCheckpointIncreaseRate = 100f;

    // Neural network output: acceleration and orientation
    public float[] outputs;


    // Start is called before the first frame update
    void Start()
    {
        unityController = GetComponent<CharacterController>();
        lastPosition = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        if (!isActive) return;
        if (outputs.Length > 0)
        {
            /*
```

```
            // Player control (test)
            if (Input.GetKey(KeyCode.UpArrow))
            {
               currentVelocity += (accelerationRate * Time.deltaTime);
            }
            else
            {
               currentVelocity -= (decelerationRate * Time.deltaTime);
            }
            */
            // Neural linear movement
            currentVelocity += (accelerationRate * Time.deltaTime)
                * outputs[0];

            // Agent movement
            currentVelocity = Mathf.Clamp(currentVelocity, minVelocity,
                                                           maxVelocity);

            moveDir = new Vector3(0, 0, currentVelocity);
            //moveDir *= speed;
            moveDir = transform.TransformDirection(moveDir);
            unityController.Move(moveDir);

            /*
            // Player rotation
            float rotationAngle = Input.GetAxis("Horizontal") * rotationSpee
                                  * Time.deltaTime;
            */
            // Agent rotation
            float rotationAngle = outputs[1] * rotationSpeed
                                  * Time.deltaTime;

            transform.Rotate(0, rotationAngle, 0);
        }

        // Agent vision
        RaycastVision();

        // Fitness management: increase with traveled distance
        // but decrease with time
        agentPosition = transform.position;
        distanceTraveled += Vector3.Distance(agentPosition, lastPosition);
        lastPosition = agentPosition;
        if (distanceTraveled > 0.2f)
        {
            fitness += distanceTraveled / 100;
        }
        fitness -= Time.deltaTime * fitnessTimeDecreaseRate;
    }

    private void RaycastVision()
    {
        agentPosition = transform.position;
        Vector3 forwardDir = transform.forward;
        Vector3 rightDir = transform.right;
```

```
        Vector3 leftDir = rightDir * -1;
        Vector3 rightDiagDir = Vector3.Normalize(forwardDir + rightDir);
        Vector3 leftDiagDir = Vector3.Normalize(forwardDir + leftDir);

        Ray forwardRay = new Ray(agentPosition, forwardDir);
        Ray leftRay = new Ray(agentPosition, leftDir);
        Ray leftDiagRay = new Ray(agentPosition, leftDiagDir);
        Ray rightRay = new Ray(agentPosition, rightDir);
        Ray rightDiagRay = new Ray(agentPosition, rightDiagDir);

        RaycastHit hit;
        if (Physics.Raycast(forwardRay, out hit, maxViewDistance)
        && hit.transform.CompareTag("Wall"))
        {
            distForward = hit.distance;
        }

        if (Physics.Raycast(leftRay, out hit, maxViewDistance)
        && hit.transform.CompareTag("Wall"))
        {
            distLeft = hit.distance;
        }

        if (Physics.Raycast(leftDiagRay, out hit, maxViewDistance)
        && hit.transform.CompareTag("Wall"))
        {
            distDiagLeft = hit.distance;
        }

        if (Physics.Raycast(rightRay, out hit, maxViewDistance)
        && hit.transform.CompareTag("Wall"))
        {
            distRight = hit.distance;
        }

        if (Physics.Raycast(rightDiagRay, out hit, maxViewDistance)
        && hit.transform.CompareTag("Wall"))
        {
            distDiagRight = hit.distance;
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Wall"))
        {
            lastPosition = transform.position;
            fitness -= fitnessCheckpointIncreaseRate;
            isActive = false;
        }

        else if (other.gameObject.CompareTag("Checkpoint") && isActive)
        {
            fitness += fitnessCheckpointIncreaseRate
            * other.gameObject.GetComponent<Checkpoint>().RewardMultiplier;
```

```
            }

        }
    }
```

A manager to link agents and networks

Now, create an empty object in unity and add a Manager script on it.

```
public class Manager : MonoBehaviour
{
    [SerializeField] private GameObject agentPrefab;

    [SerializeField] private Vector3 startPosition =
        new Vector3(20f, 0, -20);
    [SerializeField] private int populationSize = 100;
    [SerializeField] private float timeLimit = 15f;
    private int generationCount = 0;
    private bool isTraining = false;

    private List<NeuralNetwork> currentGeneration;
    private List<NeuralNetwork> nextGeneration = new List<NeuralNetwork>();
    private List<GameObject> agents = null;

    // 6 inputs, 2 outputs, 3 layers
    private int[] layers = new[] { 6, 8, 10, 6, 2 };

    private float generationFitnessMean = 0f;


    private void CloseTimer()
    {
        generationCount++;
        isTraining = false;
    }
```

The CloseTimer function is used to interupt learning.

The Update function will handle both the training generation switch and the agent move using the neural network. We allow to interrupt training (not bug proof though).

```
    // Update is called once per frame
    void Update()
    {
        if (!isTraining)
        {
            // When training is about to start
            if (generationCount == 0)
            {
                InitAgentNetworks();
                CreateAgents();
                Invoke("CloseTimer", timeLimit);
```

```
            isTraining = true;
            Debug.Log($"Generation {generationCount}");

        }
        // When training is finished
        else
        {
            Debug.Log($"Generation {generationCount}");
            generationFitnessMean = 0;
            for (int i = 0; i < populationSize; i++)
            {
                Controller controller =
                    agents[i].GetComponent<Controller>();
                float fitness = controller.fitness;
                currentGeneration[i].Fitness = fitness;
                generationFitnessMean +=
                    currentGeneration[i].Fitness;
            }

            // Compute generation mean
            generationFitnessMean /= populationSize;
            Debug.Log($"Generation fitness:{generationFitnessMean}");

            // Sort from best to worst fitness
            currentGeneration.Sort();
            currentGeneration.Reverse();

            // Next generation: duplicates the best fifth
            // of current generation
            nextGeneration.Clear();
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < populationSize / 5; j++)
                {
                    var network =
                        new NeuralNetwork(currentGeneration[j]);
                    // More mutations for second,
                    // third, fourth and fifth group
                    switch (i)
                    {
                        case 1:
                        case 2:
                            network.Mutate(0.5f);
                            break;
                        case 3:
                            network.Mutate(2f);
                            break;
                        case 4:
                            network.Mutate(9f);
                            break;
                    }
                    nextGeneration.Add(network);
                }
            }
```

```
                // Start new generation after timer
                currentGeneration =
                    nextGeneration.ConvertAll(network => new NeuralNetwork(r
                Invoke("CloseTimer", timeLimit);
                CreateAgents();
                isTraining = true;
            }
        }

        // Feedforward
        // Transfer info from the controller to the
        // associated network
        for (int i = 0; i < populationSize; i++)
        {
            Controller controller = agents[i].GetComponent<Controller>();
            float vel = controller.currentVelocity
                / controller.maxViewDistance;
            float distForward = controller.distForward
                / controller.maxViewDistance;
            float distLeft = controller.distLeft
                / controller.maxViewDistance;
            float distDiagLeft = controller.distDiagLeft
                / controller.maxViewDistance;
            float distRight = controller.distRight
                / controller.maxViewDistance;
            float distDiagRight = controller.distDiagRight
                / controller.maxViewDistance;

            float[] inputs = { vel, distForward,
                distLeft, distDiagLeft, distRight, distDiagRight};
            var outputs = currentGeneration[i].FeedForward(inputs);
            controller.outputs = outputs;
        }
        // If you need to go quickly to the next gen
        if (Input.GetKeyDown("space"))
        {
            CloseTimer();
            CreateAgents();
        }
    }
```

Other functions are creation and destruction utilities.

```
    private void InitAgentNetworks()
    {
        currentGeneration = new List<NeuralNetwork>();
        for (int i = 0; i < populationSize; i++)
        {
            NeuralNetwork net = new NeuralNetwork(layers);
            net.Mutate(0.5f);
            currentGeneration.Add(net);
        }
```

```
        }

        private void CreateAgents()
        {
            if (agents != null)
            {
                // Destroy previous agents
                for (int i = agents.Count - 1; i >= 0; i--)
                {
                    Destroy(agents[i]);
                }
                agents.Clear();
            }

            // New generation
            agents = new List<GameObject>();
            for (int i = 0; i < populationSize; i++)
            {
                GameObject agent = Instantiate(agentPrefab,
                    startPosition, Quaternion.identity);
                agents.Add(agent);
            }
        }
    }
```

Test!

You should now be able to test your agents. Even with this a simple model, you have a huge variety of training variables to change: position and values of checkpoints, ways of selecting agents, input variables, fit increase or decrease value and parameters, complexity of labyrinth. You can even change the labyrinth ingame when enough training has occured in order to check your agents are really smart.