Brief description:

For this project, I wanted to research explainable AI methods and apply them to a gamified world. To do this, I set about creating a simple platforming game, where the player can perform two actions: jump, and move. The player must navigate around the enemy and collect a coin spawned in a random location. I have a basic neural net, with inputs of “distance to enemy”, “distance to coin”, “enemy direction”, and “enemy: distance to wall”. As the game runs, the neural net will learn to optimize its strategy, and the importance of each node in the net will be calculated. The neural net will be displayed in real time, with the nodes scaled in size by their importance, allowing the user to understand why the AI made crucial decisions.

How to use:

Unzip the Unity file, and open the project in the unity hub. To run the game, open “SampleScene” if it is not already out. Set scale to 1.5 in the game window. After that, you can simply run the play button at the top of the screen.

While the neural nets run, hit “v” on the keyboard to see a real-time display of the neural net, and hit “c” to cancel this mode and go back to observing the AI players.

Explainability is initially done equally for both outputs. When the network is displayed, tap the up arrow once to show explainability for the “move” output, and tap the up arrow again to show explainability for the “jump” output. Tapping the up arrow once more will reset it to show explainability equally for both outputs.

Structure of each part:

Game:

The game world puts the player in a walled off area. The player can move and jump, and their goal is to collect coins and jump on the enemies. The coin and enemy are spawned randomly in the area; when you pick up a coin or jump on an enemy, a new one is spawned in a random location (but not near the player). The enemy moves forward slowly, and switches direction when they hit a wall. If they touch the player, the player dies. The player has a limited time to collect coins and kill enemies, so they are incentivized to perform these actions quickly to secure the most points.

A red and blue rectangles

Description automatically generated

The enemy is the red rectangle, the player is the grey rectangle, and the coin is the yellow circle. They are trapped inside the area.

Neural Nets:

A manager oversees the neural nets. It creates 50 unique game worlds at the start, each with their own neural net, and lets them play until the player dies, or the timer completes, whatever happens first. The neural nets have four inputs, displayed in order from top to bottom:

1. Player horizontal distance to enemy
2. Player’s horizontal distance to coin
3. Enemy’s movement direction
4. Enemy’s distance from the wall

These inputs are all normalized. The first two are very important early on; if the player is close to an enemy or a coin, they should jump. The latter two inputs are less important, but still hold some information; they tell if an enemy is moving towards the player or not, and if they are about to switch direction from hitting the wall. The goal of this project is to see these importances reflected in the displayed neural net (more on that in the results section). The outputs are move and jump, displayed in order from top to bottom.

Since there is no way to know the proper action to take at each step, the neural nets cannot backpropagate; instead, I used reinforcement learning to help the neural nets learn. The neural nets get rewarded when they jump on enemies (5) or collect coins (10), and get a penalty when they die to the enemy (-10). Since there is a timer, the neural nets are incentivized to get rewards quickly. Once the timer completes, the neural nets are sorted in ascending order by their points. The bottom 20% of neural nets are replaced with new neural nets, the middle 60% have their weights slightly mutated, and the top 20% of neural nets are kept exactly as they are. Then, the worlds reset, and the neural nets continue on the new generation.

Display:

The display is created by spawning a ‘node’ Gameobject (circle) for each node in every layer. The input and output nodes are placed on the left and right sides of the screen respectively, and then the horizontal distance between them is calculated and divided by the amount of hidden layers. This distance is added to the nodes of each layer, multiplied by their layer number, to make sure each layer is equidistant from each from the other. The same practice is done within each layer to separate the nodes; find distance between top and bottom of the screen, divide by node count in the layer, and then place nodes using this interval.

Once the nodes are placed, ‘weight’ Gameobjects (line) are spawned in. Since I am using a fully connected network, every node was connected to every node in the following layer. I spawned these weight objects in the middle of the two nodes, and then rotated them by the angle between the two nodes. Once they were in place, I grabbed the actual weight value from the neural net, then colored the lines based on the value. A weight near 1 will get a solid red color, a weight near 0 will get a greyish color, and a weight near -1 will get a solid blue color.

Explainable AI:

To make each node explainable, I had to figure out the relevance of each node for a given output. To do this, I decided to perform a variation of Layer-Wise Relevance Propagation, explained in [this paper](https://iphome.hhi.de/samek/pdf/MonXAI19.pdf). For my implementation, I took the following steps:

1. Give relevance scores to the last layer. If you are observing the relevance related to a single output, then give the output an relevance of 1 and all other outputs an relevance of 0. If you are observing the relevance related to all outputs, give each output a relevance of 1/output\_count. These relevancies can be stored as a vector, with the first item representing the first output, and so on.
2. Given n inputs, A = [A1, A2, A3, …, An], and a single output, B = k, the importance of a node can be calculated by computing:

Aabs = k|

for i=0 to len(A)-1:

if B ≥ 0:

Importancei = Ai / Aabs

else:

Importancei = Ai / -Aabs

Where Importancei is the scaled importance of the ith input. Applying this for every node, you get a vector of scaled importance values for a given output. Then, this can be applied for the other outputs, to get a matrix of scaled importance values.

1. You can then multiply the scaled importance matrix by the transpose of the relevance vector, to further scale the importances by how relevant the output is.
2. Now that the importances are fully scaled, you can gather the sum of importances for a given input. Once this is done for every input, you can apply a normalization over the entire layer of these node importances, to get a relevance score for each individual node.
3. Steps 2-4 can now be repeated a layer back, with the new relevance scores used as the output nodes of the current layer. This can continue until all layers have their relevance scores.

Once relevance has been computed for every layer, I scaled the nodes in size based on their contributions. If the relevance was positive, I used the formula 1+3\*(relevance) for size, which would get a size between 1 and 4. If the relevance was negative, I used the formula 1 + (3\*(relevance) /4), which would get a size between 0.25 and 1. Doing this allowed the relevance of each node to be clearly visible, and showed in real time the adaptations the neural nets would make as the inputs changed.

Results:

After running a few simulations, I checked on the explainability of each node in a few different situations. As expected, the first two inputs (enemy distance and coin distance) often played a big role in the decisions of the neural net, either being very big and supporting the decision, or being very small and going against it. The other two inputs(enemy direction and enemy distance from wall) were often an average size, indicating that they didn’t play a big role one way or the other. As time went on, the neural nets started to learn various strategies; one such strategy was going for a coin when the enemy was moving away, and running away when the enemy was moving towards the player. This particular strategy had a much higher importance on the direction the enemy was moving, showing that the neural net had learned how to use the information on enemy direction.

I changed the simulation to give a higher score to coins and a lower score to killing enemies, and noticed a slightly higher importance on the second input. When I changed it again to give a higher score to killing enemies and a lower score to coins, I noticed a substantially higher importance in the first input. This showed that the importance mechanic was working as intended. I theorize the difference in importance gains is due to it being easier to kill enemies than collect coins.

Interestingly, throughout my trials I noticed a dominant strategy taking place; some neural nets would run to the corner, and repetitively jump. Since they jumped fast, they often survived the enemy, and by being in the corner they ensured the enemy would get hit as the player landed. Notably, these neural nets often had meaningless importance values on each node, and the explainability was lost. This happened because the strategy was not related to any specific input; if any input forced the net to continually jump in the corner, it would take the credit for keeping the player alive.

Result pictures:

A computer graphics of a network

Description automatically generated

Both outputs are equally relevant, and it can be seen that the network has a big node on input 0 (enemy is close, so the movement is highly affected by it).

A screenshot of a computer

Description automatically generated

Output 7 (movement) is relevant, and it can be seen that input 0 is somewhat important (enemy is somewhat close) while the other parameters are not very important.

A computer screen shot of a network

Description automatically generated

In this simulation, I increased the value of the coin, so it gave more points. For the output movement, the player still valued Input 0 a lot.

A computer screen shot of a network

Description automatically generated

However, when running the same simulation with a higher relevance for the jumping output, it became clear that input 1 (coin distance) had a lot of importance, especially as the player approached the coin.

A network of yellow circles and red lines

Description automatically generated

When I gave extra points for defeating an enemy, the neural net stopped caring about coins and only jumped when the enemy was right next to it.

Updateable parameters:

There are several parameters that you can update to change how the simulation runs.

**Time:**

Recommended values: 1 through 5

Where: Go to Edit🡪Project Settings🡪Time🡪Timescale to update this value.

Description: Increases the speed at which the game can run.

**Population:**

Recommended value: 50 (choose a multiple of 5)

Where: Manager GameObject🡪(Manager)

Description: Adjusts the number of neural nets spawned each generation.

**Layers:**

Recommended value: <10 (for visibility on layers and nodes)

Where: Manager GameObject🡪(Manager)

Description: The top number is layer count, and the numbers in the list are the node count for each layer. Adjust these to modify the hidden layers, just make sure that the first layer has 4 (inputs) and the last layer has 2 (outputs)

**TimeForGeneration:**

Recommended value: 35

Where: Manager GameObject🡪(Manager)

Description: This is the number of seconds each generation will survive. Increase this to give the neural nets a longer time to survive, shrink it to make them value getting rewards quicker.

**MoveSpeed:**

Recommended value: 15

Where: NNet 🡪Player🡪(PlayerMove)

Description: This controls the player’s top speed.

**JumpSpeed:**

Recommended value: 75

Where: NNet🡪Player🡪(PlayerMove)

Description: This controls the player’s jump height.

**HumanMove:**

Recommended value: False

Where: NNet🡪Player🡪(PlayerMove)

Description: This is a Boolean value; if true, a human can move the player with the arrow keys, and the neural nets will be turned off. If false, the neural nets will have exclusive access to the player’s movements.

**DeathScore:**

Recommended value: -10

Where: NNet🡪Player🡪TriggerCol🡪(PlayerCol)

Description: This is the added penalty if a neural net gets defeated by an enemy.

**CoinScore:**

Recommended value: 10

Where: NNet🡪Player🡪TriggerCol🡪(PlayerCol)

Description: This is an added reward if a neural net collects a coin.

**KillEnemyScore:**

Recommended value: 5

Where: NNet🡪Player🡪CheckGround🡪(DetectBelow)

Description: This is an added reward if a neural net jumps on an enemy and kills it.

**EnemySize:**

Recommended value: 2 (No larger than 4, otherwise the player cannot jump over the enemy.

Where: NNet🡪Enemy🡪(Transform)🡪\*Scale, Y\*

Description: This adapts enemy height, making it easier or harder for the player to jump over the enemy.

**MoveSpeed (Enemy):**

Recommended value: 10

Where: NNet🡪Enemy🡪(EnemyMove)

Description: This controls the Enemy’s speed.