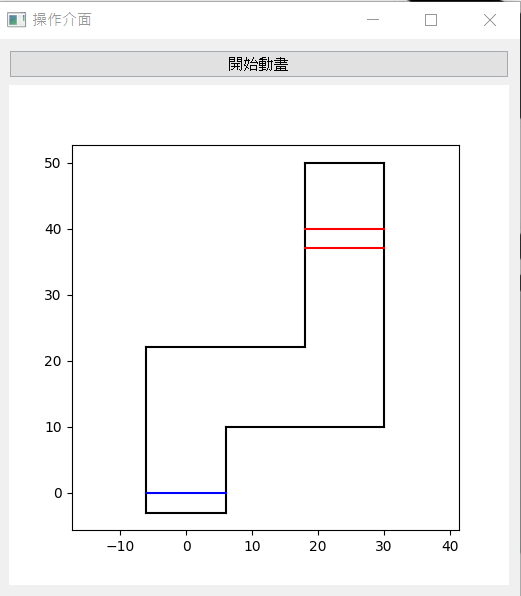
1. **程式介面說明：**

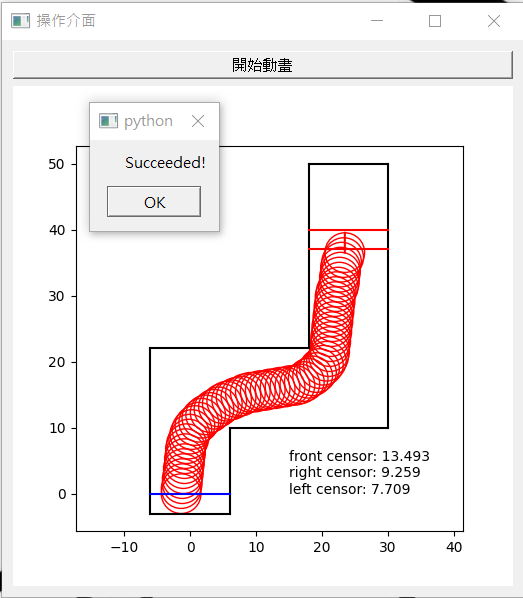
* 初始畫面

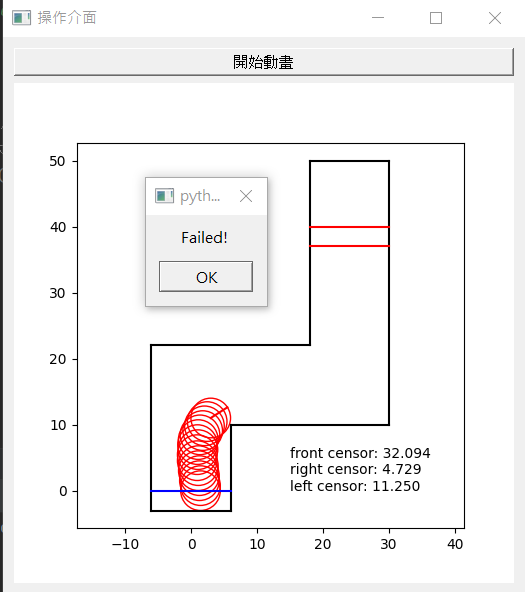


開始按鈕：按下去動畫便會自動訓練程式，之後執行動畫。

顯示動畫執行過程的畫布

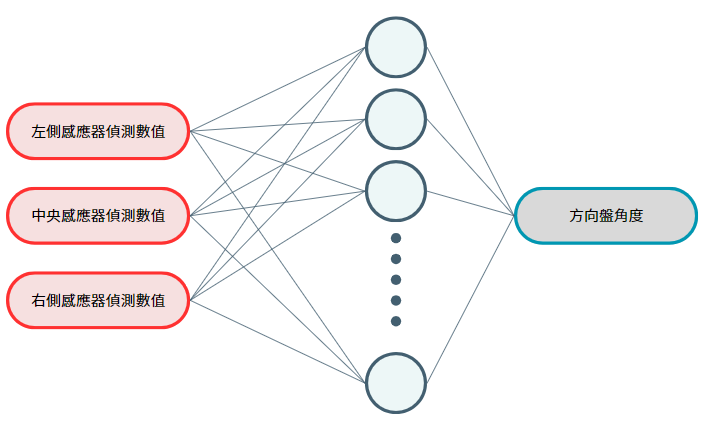
* 動畫執行介面

 (成功畫面)

 (失敗畫面)

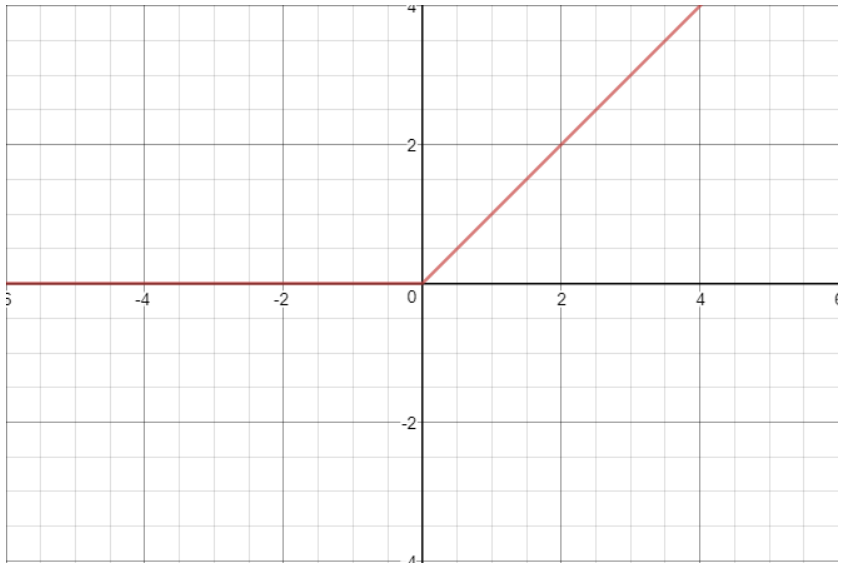
1. PSO實做細節：
2. 自走車方向盤控制：

我建立了一個三層的MLP，其中包含：一個輸入層可以輸入三個感測器偵測的的距離，一個隱藏層，一個輸出層可以輸出當下的方向盤角度。



其中隱藏層的激活函數採用的是ReLu函數

(ReLu的函數)



而輸出層是修正後的tanh函數，使我們輸出的方向盤可以限縮於給定的範圍內



1. Particle:

每個particle包含以下的幾個元素

1. MLP中的權重函數 (視為PSO中每個粒子的x vector，即為當前位置)
2. 每次訓練時，自走車的軌跡 (用於計算適應度)
3. 前一次的訓練速度、當次的訓練速度

(這裡因為避免宣告過於多的playground在particle裡面造成運算和記憶體空間吃緊，所以particle會接收外面的playground後，把它reset後拿來訓練)

1. 適應度函數：

這裡主要是透過

score(自走車的移動軌跡)=

(100 if 這個粒子的權重可以使自走車到達終點 else 0) +

∗ (最後自走車停止時的終點x值) +

∗ (最後自走車停止時的終點y值) −​

∗ (自走車的路徑長度)

我將​, , 分別設為 0.8, 1, 0.5。

主要的想法是因為地圖是個倒z字型，因此最終自走車停止位置y值越大，就越接近終點。

同時，為了避免一開始就撞牆或是往左的結果影響，故設定為最終停止位置的x值越大也越好。但此時也需要考量過於往右方靠近，使得路線增加不必要的長度，因此減少其權重。

最後是為了讓自走車走一個最短的距離而設定，又為了避免一開始就撞牆的結果影響，所以權重設的特別小。

1. 流程



1. **分析：**

這次我將原先的程式再加入了兩個class：PSO和Particle。

本來是想要用原先的playground就直接來進行PSO的，但是我擔心初始化的時候，建立過多的particle會使我需要宣告過多的playground繼而導致整個記憶體資源吃緊。

因此，我額外建立了PSO用來主要負責模型的訓練、適應度的判斷以及取得最佳的權重。而Particle的部分主要用來儲存在MLP中的各項權重、速度等資料。同時更新權重以及進行在playground上的模擬。

這樣只需要將playground傳給Particle，其內部便會將其初始化後進行自走車的移動。最後再將移動的軌跡輸入至PSO中的適應度函數中進行判斷，就可以知道我們的權重的好壞了。而最終訓練好的權重在取出來後，就可以用以執行最後自走車的結果了。

這次可以感覺到PSO算法的方便和強大了。整體的模型不會到太難理解，同時只要粒子的數量夠多而且適應度函數建立的夠漂亮，有準確的對到需求上，就可以得出一個還不錯的結果。而且配合MLP，它可以讓自走車以一個很漂亮的距離和轉彎角度到達終點。

但其實我也可以理解為甚麼有人會不太喜歡這種derivative-free類型的算法的。

裡面的不確定性的確有一點點的高，而且結果和過程有時候也會有些不太踏實。有時候訓練完的權重拿來跑，會出現一些不知所以的軌跡，比如說：在彎道的地方很剛好的切在轉角處，或是會繞一大圈才往終點前進。

雖然說增加粒子數、訓練次數、隱藏層的神經元數量的話就可以解決這樣的問題了。但這個題目是屬於比較簡單的類型，如果出現了更加複雜的題目的話，就很難確定需要訓練的次數、粒子數和神經元的數量了。

模糊函數和q-learning都會有一個感覺可以收斂到的方向，而且每次出來的結果都大致相同。但PSO卻有時會有差異很大的結果，感覺不太像是把它完全收斂，而是將錯誤的機率降至很低很低。

但實際結果很令人驚艷就是了。畢竟一個完全隨機的矩陣最後居然可以跑出一個比q-learning和模糊函數還好不少的結果。果然還是結果決定一切阿。