Daniel Lambert

GIMM 110

Individual Game Rhetorical Analysis

20171101

Network Combat.

I knew that I wanted to create a simple game that used machine learning in some way. I understood the challenges associated with the performance of actionscript, so I decided to go with the simplest feed forward neural network I could come up with. I hope to come back to this some time in the future and implement a few more features that increase the users level of control.

I realized right away that the charge game was the simplest core to build on when it comes to inputs that a network would take. The game is essentially a one dimensional playing field. Many different simple and fast boolean values can be passed into the network for training. Additionally, scoring for each network is greatly simplified as the starting scoring system is basically how far away from the enemy castle they are. This immediately gets soldiers moving towards each other.

Performance is the hardest thing to deal with in this project. Developing this in Unity and compiling the scripts into actual machine code via IL2CPP would make this at least a hundred times faster. I had to scrap a few ideas as continuously blitting things to the terrain collision bitmap turned out to have a huge performance penalty.

Since this is primarily supposed to be a code analysis, lets get into it. I will target only ChargeGame.as and SoldierRobot.as as otherwise this document will expand to dozens of pages.

ChargeGame.as

*public static var chargeGame:ChargeGame;*

The one ChargeGame object that is created is treated as a singleton. Other classes can grab a reference to it at any time if they need to query information about the game state.

*allSoldiers = new Array();*

*blueNetworks = new Array();*

*redNetworks = new Array();*

*blueNetworks.push(new Network(2,10));*

*redNetworks.push(new Network(2,10));*

Here we initialize our starting game state and generate a single ten neuron two layer network for each side.

*private function createGuy(dir:Boolean, newGuy:SoldierRobot):void*

createGuy does just that, you pass in the side they are on and a new SoldierRobot to add to the stage. This is a side-effect method that returns nothing.

*private function update(evt:Event):void*

Update is called every frame and handles the majority of the game loop. It calls updateScores and calls Update on every SoldierRobot that is currently a part of the game. It also manages the despawn timer for every SoldierRobot based on the current lifespan limiter, which lengthens as the game progresses so that initial training is rapid. Terrain collision bitmap updates are handled here, but are set to only execute on the first frame for performance reasons. Eventually this will be re-enabled and only perform this action when a flag is set to update it. Update also handled the spawn timers, and the calling of CreateGuy. Red and Blue teams have the same spawn rate, since it is the emergent behavior we are actually interested in. Update also calls TrimNetworkLists.

*public function getColorSample(x:int, y:int):Boolean*

getColorSample is the bitmap collision detection system. Some of the code was found online and then modified for the purpose of returning if a terrain collision is happening or not.

*public function getBestRedNetwork():Network*

*public function getBestBlueNetwork():Network*

The getBestNetwork functions return the highest scoring network stored in our network arrays. These are cloned and mutated when spawning new SoldierRobots.

*public function trimNetworkLists()*

This method simple limits the number of stored networks in the Blue and Red lists. It retains 100 networks in each by default and is primarily there for debug purposes. The same effect could be made by simply storing only the best scoring network at one time. SoldierRobots store their own network until they die, then their “brain” gets added to these lists. Only the worst is removed.

*public function getDistanceRatioFromBlueBase(xa:Number, ya:Number):Number*

*public function getDistanceRatioFromRedBase(xa:Number, ya:Number):Number*

These provide us with an introductory scoring mechanism for the networks.

*public function getAlliesNearbyClose(xa:Number, red:Boolean):Number*

*public function getAlliesNearby(xa:Number, red:Boolean):Number*

*public function getAlliesNearbyFar(xa:Number, red:Boolean):Number*

*public function getEnemiesNearbyClose(xa:Number, red:Boolean):Number*

These methods query the game state and their output is directly used as inputs into a network’s input layer. Many of these can be easily created to provide interesting input into the neural networks.

*public static function degFromRad( p\_radInput:Number ):Number*

*public static function radFromDeg( p\_degInput:Number ):Number*

Simple math helper functions.

SoldierRobot.as

*public var brain:Network;*

The soldier’s brain is where all the magic happens. It controls ALL of their behavior.

*public function SoldierRobot(br:Network,red)*

The only constructor available for a SoldierRobot. It must be passed in a brain, and needs to know what side it is on.

*public function startWalk()*

This method simply restarts the walking animation if the soldier was previously stopped or crouched.

*public function update():void*

Called in the main game loop, this handles updating every soldier.

*if(brain==null)*

*{*

*trace("This guy has no brain!!!!!!");*

*return;*

*}*

It even has some error catching tests. It also handles looping the soldiers animation and flipping the soldiers image if it changes direction. It also tests to see if it is resting on the terrain and generates a boolean so the network knows if it is on the ground or not. It then sets the input values of the soldier’s “brain” and executes brain.tickNetwork().

*var leftBias:Number = brain.getSingleOutput(0);*

*var rightBias:Number = brain.getSingleOutput(1);*

*var shootBias:Number = brain.getSingleOutput(2);*

*var aimAngle:Number = brain.getSingleOutput(3)\*90;*

*var crouchBias:Number = brain.getSingleOutput(4);*

*var normTotal:Number = Math.sqrt(leftBias\*leftBias + rightBias\*rightBias + shootBias\*shootBias + crouchBias\*crouchBias);*

*leftBias/=normTotal+0.00001;*

*rightBias/=normTotal+0.00001;*

*shootBias/=normTotal+0.00001;*

*crouchBias/=normTotal+0.00001;*

Here I normalize a four dimensional vector and add a bit of fudge to prevent the generation of FP NaNs. This allows the network to decide what state it wants to be in on the next frame.

*this.Gun.rotation = aimAngle;*

The soldier’s gun aim is taken directly, unfiltered, from the neural network. This sometimes causes them to swing their gun around like a baton, but since there is no penalty for that, it is likely to find its way into networks that have evolved for hours.

*if(health<0)*

*{*

*trace("Killed by hostile action.");*

*killMeAndSaveMyBrain();*

*}*

This portion checks to see if the soldier is dead, if he is, store his network.

*public function killMeAndSaveMyBrain():void*

This method handles the final scoring and saving of networks, and also handles the despawning of soldiers.

In conclusion, I’m quite likely to expand on the actionscript neural network library into something that I can use on future actionscript projects. Though, it is very likely, I will not be using actionscript for anything other than school. If members of my future group project want to utilize it, it may be expanded upon much sooner.This game only acts a vessel for experimenting with simple feed forward neural networks and perceptrons. Any game more complex than this would likely suffer far worse from performance problems. It has been a few years since I implemented neural networks outside of python and a few years since I’ve run them on anything other than a GPU. It has been an interesting experience so far. It is amazing what emergent behavior can come about with only a few hundred lines of neural network and hill climbing code.