Neural Networks With Super Mario Bro's

Eoin Murphy 11487358 , Dublin city university

CASE Final year project

eoin.murphy74@mail.dcu.ie

http://blogs.computing.dcu.ie/wordpress/eoinmurphyfinalyear/

Table of contents

Introduction

Throughout my time in DCU I began to have a ever increasing interest in the idea's of neural networks and their possible applications. I have very much attracted to the idea of intelligent machines and decision making processes whiten computers. Before I finished my studies in DCU I wished to get a better understanding of Neural networks which is where the idea for this project arose. As in most problems in AI it is very attractive to use games as way of demonstrating intelligent design as one can argue if a machine can learn to play a game better then any human can there has to be some underlying intelligence in the system, This is why I choose super Mario as the application of my Neural Network.

The following document describes my research into Neural networks and the two machine learning algorithms used in training them and the steps involved in training the networks. Also presented in the Document is the System I developed which allowed me to train the networks as such and facilitate any user regardless of their background to be able to run the system and immediately begin training and experimenting with Neural Networks.

1 Super Mario Bro's

Super Mario Bro's (SMB) is a video game developed in 1985 by Nintendo for the Nintendo Entertainment System (NES). SMB is a 2D platform game which means that a player must navigate a 2 dimensional game space avoiding obstacles and enemy's in order to make it to the end of each level. Available today are ROM files of the original SMB game, a ROM file is a copy of the data written on read-only memory chip in the NES video game cartridge. Using emulators of the NES hardware we can run ROM files and as such play the video game as if it was running on the original NES system, Figure 1.1 shows the SMB ROM file being played in an emulator.

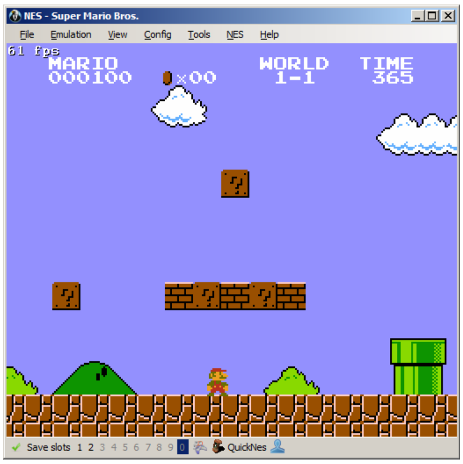


Figure 1.1: Super Mario Bro's running in the BizHawk emulator

2 Neural Networks

A Neural Network is a interconnected group of nodes. they consist of 3 layers of nodes the first being the input layer, secondly the hidden layer (some networks can consist of multiple hidden layers this is known as deep learning) and finally the output layer. The input layer is simply put what the network sees and is passed to the nodes of the hidden layer via weighted connections called synapses. Each node of the hidden layer calculates the sum of all inputs to it times the weight of the connection, this sum is passed through a activation function to determine whether or not the node “fires”. This result is then passed to the output layer and again by summing the weights times the inputs from the hidden to the output, determines what is outputted by the network.

In the case of this project I'm using a type of network type called a fully connected feed forwarding network. Simply put this means all nodes are directly connected to each other and data passed through the network goes in only one direction. So throughout this document when I refer to a neural network I'm referring specifically to this type.

2.1 Inputs

The inputs used by the network is the tile data in a radius around Mario for the given frame in the game. The total inputs is given as: $I=(r\times2+1)^2$ where r is the tiles radius around Mario. These inputs are read directly from the games memory at the RAM address range 0x0500 → 0x069F. The values for the tiles are then encoded where a solid object tile is written as 1, a enemy tile is written as -1 and a tile in which Mario can move freely is written as 0. This input scheme allows us to reduce and increase our inputs by changing the value of r during the experimentation phase of the networks implementation allowing us to determine what size inputs results in the best network

2.2 Outputs

In this neural network application values calculated in the output layer are translated into key presses on the NES controller that are executed on each frame advance. Each output node is mapped to a corresponding key; 'A', 'B', 'DOWN', 'LEFT', 'RIGHT', 'UP'. This means that when a output node 'fires'(its value is greater then a given threshold) the corresponding key is pressed for all outputs and then executed by the emulator.

2.3 Network Architecture

The following diagram depicts the network architecture used in this application.

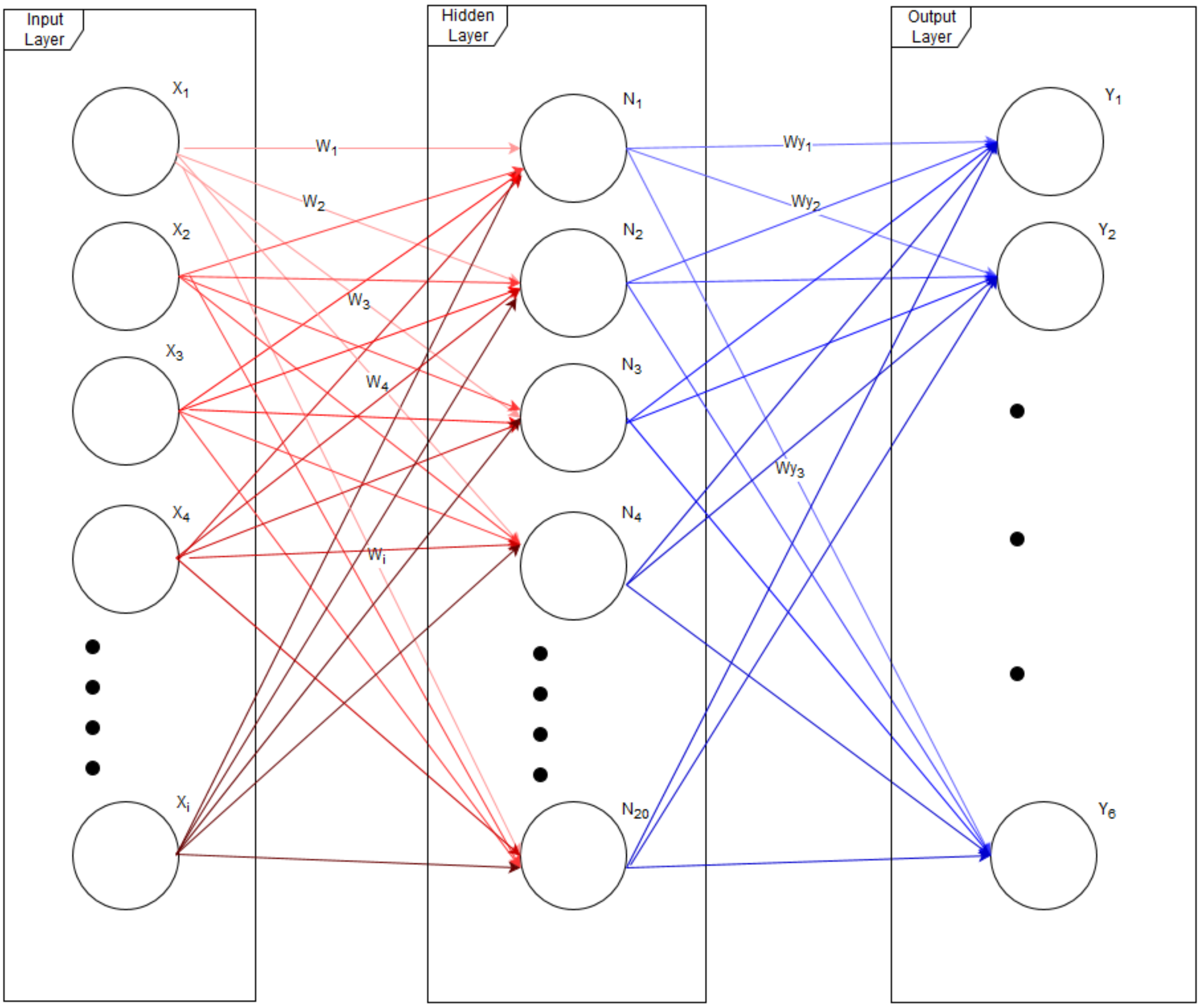


Figure 2.1: architecture of the network used in supervised learning

The number of nodes in the input layer is given by the equation: $\mathrm{I}=(r\times 2+1)^2$ the network uses 20 nodes in the hidden layer and output layer has 6 nodes representing each controller button.

The network operates as follows, Data is passed through the network via the input layer x_i and sent to the hidden layer, the value at each node n_j in the hidden layer is found by summing all inputs x_i to the node times their connection weights weights w_{ij} .

$$n_j=\sum_{i=1}^{I} x_i w_{ij}$$

As we are using back-propagation in this project for training the network we need outputs of the nodes to be a continues output so that we can modify the weights of each connection to gradually reach a desired output, this is known as gradual descent/ascent. With this in mind using the typical step activation function will not work:

$$n_j=\begin{Bmatrix} n_j>t, n_j=1 \\ n_j<t, n_j=0 \end{Bmatrix}$$

where:
t is some threshold

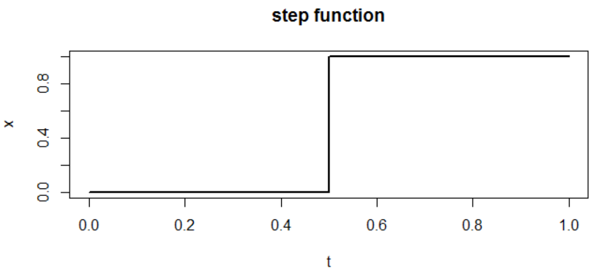


Figure2.2: step activation function

Due to this we use a logistic function to produce smooth output between 0 and 1, for this we use the sigmoid function so our activation function now becomes:

$$n_j=\mathrm{sig}(n_j)=\frac{1}{1+\mathrm{e}^{w\,n_j}}$$

where:
w is a weight which determines the steepness of the curve

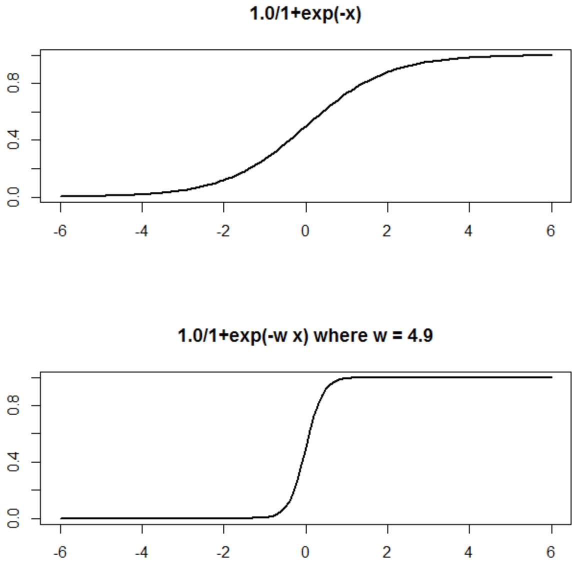


Figure 2.3: sigmoid function and how w affects the steepness sigmoid curve

The same is done to get the values of each node in the output layer y_k however in the output layer as we us a step function on the value y_k in order to translate this to a whether or not the button mapped to that value is pressed.

3 Supervised Learning

Supervised Learning is the process that was used to initially used the train the network to play. It works by showing the network a set of exemplars, Telling the network that given a input x that it should produce the correct output y. I update the weights of the network by using back-propagation where by we start at the last layer and work our way backwards from the output to the input updating the weights using the error calculated by what the network produced for the input as to what it should have produced as shown in the exemplars. The typical algorithm for supervised leaning with propagation used is:

1. Forward pass the inputs x through the network from our exemplars
2. observe the network output y
3. Tell the network the correct output o from the exemplar output
4. Compare y with o giving us some error E
5. using back-propagation with the error E modify the weights W_{ij} so that next time the network receives the same input x it will produce a output y closer to the correct output o
6. Repeat steps 1 – 5 until all exemplars have been show to the network

3.1 Exemplars

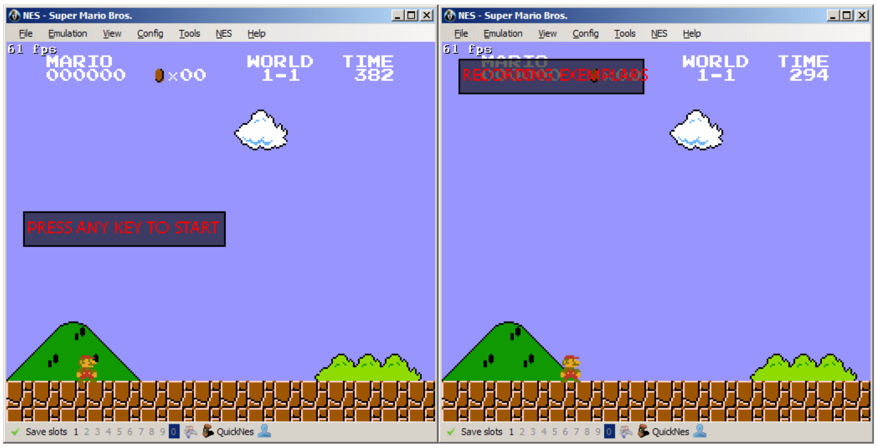
To gather exemplars to train the network the system provides the functionality for the user to specify the number of frames of gameplay they want to record. Once the user chooses the option to record game play the system loads up a save state in the emulator corresponding to the level they want to record and then allows the user to play through the level until they reach the desired number of frames as shown in figure 3.1. As the user is playing the level the tile data around Mario are read from the games RAM though the emulator and stored in a file matched to the buttons the user pressed on the controller read directly from the emulator. Each exemplar is stored in the form:

<Tile data separated by pipe> ; <Button separated by pipe>

This allows us to easily pass the input/output pair to the network in training.

The following is an example of that the exemplars in the exemplar file look like where in this example the radius of the tiles around Mario is two.

Inputs	Outputs
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|-1|0|0|0|0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1	1|0|0|0|1|0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|-1|0|0|0|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1	1|0|0|0|1|



NES - Super Mario Bros.
File Emulation View Config Tools NES Help
61 fps
MARIO 000000
×00
WORLD 1-1
TIME 382
Save slots 1 2 3 4 5 6 7 8 9 0
QuickNes
NES - Super Mario Bros.
File Emulation View Config Tools NES Help
61 fps
MARIO
WORLD 1-1
TIME 294
Save slots 1 2 3 4 5 6 7 8 9 0
QuickNes

PRESS ANY KEY TO START

RECORDING EXEMPLARS

Figure 3.1: When a user selects the option to record exemplars the game loads the level they want to record and prompts them to begin playing

3.2 Back-propagation

When we pass and exemplar input x through the network and get its output y the output will not be the correct output or even close to the correct exemplar output o to measure how close the networks input was to the correct output we calculate the error between the two, we use the squared error:

$$E=\frac{1}{2}(y-o)^2$$

The goal of the back-propagation algorithm is to reduce this error by changing the weights at each layer in the network. As E is a function that is effected by the variables y, o, w, net we need to look at how E changes with respect to these variables. To do this we need to calculate the partial derivatives of E with respect to each of these variables, so we can see how changing the weights w_{jk} effects E or put other wise the partial derivative of the error with respect to the weights. Before I continue I'm just going to explain derivatives and partial derivatives of a function keeping the examples in line with our error function E

Derivatives

Normal derivatives talk about a function with one variable so for example $\mathrm{f}(x)=x^2$ what the derivative of $\mathrm{f}(x)$ denoted by $\frac{df}{dx}$ is asking is what effect of changing the value of x have on the output of the function f .

dx represents how much in the x direction the value changes and df represents the resulting change in the output after you make the change indicated in dx

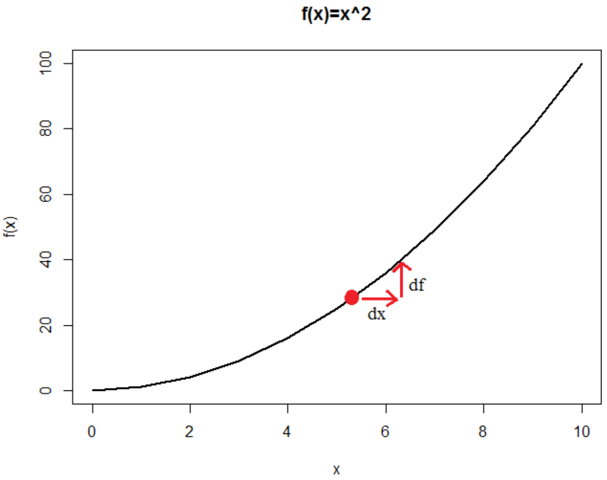


Figure3.2: Graphical visualisation of the derivative of the function $\mathrm{f}(x)=x^2$ at the point 5

Partial Derivatives

Partial derivatives are now looking at multi-variable functions for example $\mathrm{f}(x,y)=\frac{1}{2}(x-y)$ so the partial derivatives of $\mathrm{f}(x,y)$ are denoted as $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ which are asking, same as with single derivatives what effect of changing x have on f or what effect of changing y have f

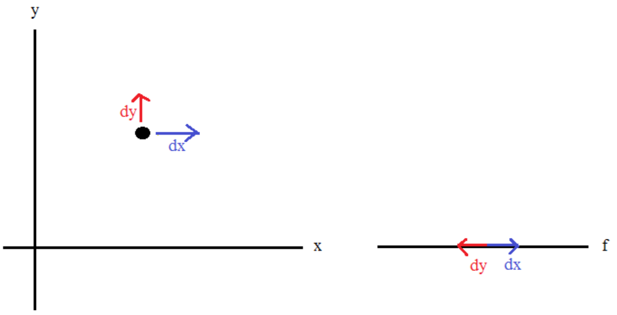


Figure3.3: Graphical visualisation of the partial derivatives of the function $\mathrm{f}(f,x)$

Total Derivative

Total derivative looks to find the over dependency of the function f on the given variable it is done by adding all the indirect dependencies. For example given the function f(t,x,y) the total derivative with respect to t is:

$$\frac{df}{dt}=\frac{\partial f}{\partial t}+\frac{\partial f}{\partial x}\frac{dx}{dt}+\frac{\partial f}{\partial y}\frac{dy}{dt}$$

Chain rule

The rule is used when you are looking to find the derivative of a more complex function like the one we are using for our error function. What the chain rule says is to find the derivative of such function you find the derivative of the sub functions and then you can find the derivative of the whole function. the chain rule is written as follows:

$$\frac{dz}{dy}=\frac{dz}{dy}\cdot\frac{dy}{dx}$$

Where:

z is a function with the variable y, which is itself a function of x.
simply put this looks like the function $\mathrm{z}(\mathrm{x}(y))$

Deriving the Error

With all this now in mind we can now derive our error function $E=\frac{1}{2}(y-o)^2$ in order to update our weights to move us closer to the correct output.

Where:

net_j is the value of the neuron at j

w_{ij} is the weight between the i and j neurons

sig is our activation function

E is our error functionality

y is the networks output

o is the correct exemplar output

We use the chain rule to calculate the partial derivative of E with respect to w_{ij} :

$$\frac{\partial E}{\partial w_{ij}}=\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

As only one variable in calculating the value at a neuron depends on w_{ij} (*as shown in the equation in 2.4*) we can say:

$$\frac{\partial net_i}{\partial w_{ij}}=y_i$$

The derivative of the output of the neuron j with respect to its input is the partial derivative of the activation function since we are using the sigmoid function this is:

$$\frac{\partial y_j}{\partial net_j}=\mathrm{sig}(net_j)(1-\mathrm{sig}(net_j))$$

Now we need to derive for the nets output, if the neuron is in the output this is pretty straight forward as $y_j=y$:

$$\frac{\partial E}{\partial y_i}=\frac{1}{2}2(y-o)=y-o$$

if j is not in the output and is in the hidden layer there is a little bit more to it.
Considering E as a function of the inputs of all the neurons $N=u,v,....w$ receiving input from the neuron j we derive:

$$\frac{\partial E(y_j)}{\partial y_j}=\frac{\partial E(net_u,net_v,...net_w)}{\partial y_j}$$

Now taking the total derivative of this with respect to y_j we get:

$$\frac{\partial E}{\partial y_j}=\sum_{n\in N}(\frac{\partial E}{\partial net_n}\frac{\partial net_n}{\partial y_j})=\sum_{n\in N}(\frac{\partial E}{\partial y_n}\frac{\partial o_n}{\partial net_j}w_{jn})$$

Now putting everything above together we get the partial derivative of the error with respect to the network weight :

$$\frac{\partial E}{\partial w_{ij}}=\delta_j y_j$$

$$\delta_j=\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}$$

$$=\left\{\begin{array}{l}(y_j-o_j)y_j(1-y_j)\ \text{ if } j \text{ is an the output layer}\\ (\sum_{n\in N}\delta_n w_{jn})y_j(1-y_j)\ \text{ if } j\text{is in the hiden layer}\end{array}\right\}$$

now to update the weights we use a learning rate α where $0<\alpha\leq 1$ The purpose of the learning rate is to effect how much of the new weight is applied to the old weight(*for the network that was able to successfully play and finish the level a value of 0.8 was used*) and update as follows:

$$w_{ij}:=w_{ij}-\alpha\frac{\partial E}{\partial w_{ij}}$$

Now that we have a way of being able to update our weights we can write the algorithm used to train our network:

Algorithm 1 Supervised Learning With Back-Propagation Using Recorded Human Gameplay

```
Initialize random network weights between some small value -C and C

for iteration = 1, TRAIN_ITERATIONS do
  forEach $e$ in EXEMPLAR_FILE do
    Split the $e$ on the “;” char to get the inputs $I_i$ and correct
    output $O_k$
    Send the inputs $I_i$ for the exemplar $e$ to the network
    Store the correct exemplar output $O_k$
    Forward-pass $I_i$ and observe the output $y_k$
    Measure the error $E$
    //Back-propagation algorithm
    //update the weights at the hidden → output layer
    for $k=1$ , NUMBER_OUTPUTS, do
       calculate $\delta_k=(y_k-o_k)y_k(1-y_k)$
       calculate $\frac{\partial E}{\partial w_{jk}}=\delta_k y_k$
    end for
    for $j=1$ , NUM_NUERONS_IN_HIDDEN do
      for $k=1$ , NUMBER_OUTPUTS, do
        update the network weights $w_{jk}:=w_{jk}-\alpha\frac{\partial E}{\partial w_{jk}}$
      end for
    end for

    //update the weights at the input → hidden layer
    for $j=1$ , NUM_NUERONS_IN_HIDDEN do
       calculate $\delta_j=(\sum_{n\in N}\delta_n w_{jn})y_j(1-y_j)$
       calculate $\frac{\partial E}{\partial w_{ij}}=\delta_j i_j$
    end for
    for $i=1$ , NUM_INPUTS do
      for $j=1$ , NUM_NUERONS_IN_HIDDEN do
        update the network weights $w_{ij}:=w_{ij}-\alpha\frac{\partial E}{\partial w_{ij}}$
      end for
    end for
    write the results for passing and updating the network with $e$
  end forEach

end for
write the trained networks weights to a XML file to be able to be parsed and
loaded into the system at later time if needed.
```

3.3 Training

3.3.1 Training Round 1

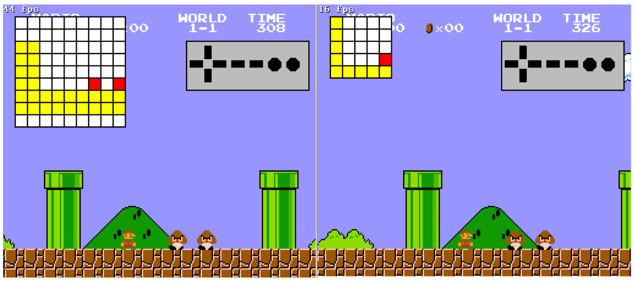


Figure3.4: Visualization of network input schemes with tile radius 4(left) and 2(right). White tiles represent the free space(0) in which Mario can pass through, yellow represents solid tiles(1) in which he cannot pass and red represents enemy's(-1)

I began to train the network using a input scheme of a 4 tile radius around Mario with 20 neurons in the hidden layer. After training it for a set number of iterations of the exemplars very little progress was made. The network was unable to even copy jumping the first enemy in he level as described in the exemplars. Due to this I began to look at the exemplars. A problem was identified that when the user presses a button on the controller to make a action the game will update its tile array in RAM to reflect this immediately and load it along side the button press. As such the button press is mapped to the new RAM tile array and not the state that caused the user to press the button. What this meant for the network is it was reviving the input reflecting a jump over an object but never the actual state that caused the jump in first place. To fix this the function which records the users gameplay as the exemplars was updated so that when it writes an exemplar it takes the current buttons presses and maps it to the previous states tile inputs.

3.3.2 Training Round 2

With this in place the network was retrained and began to show progress it completed the jump over the first enemy and made its way over the first obstacle in the form of the green pipes but was unable to progress further as another pipe lay in its way which was higher then the one previously seen. The exemplar file was examined again to try and detect a reason for the failure. I found that there was a lot of noise in the exemplars that were causing exemplars that resulted in a output that pressed the jump button to be drowned out. The reason for this is that outputs where the jump button was pressed were very rare in the exemplars. Throughout the exemplar file there would exist exemplars with the same inputs but different outputs and those with a jump output were extremely rare compared to those with out so were drowned out during training resulting in them being classified incorrectly. I decided to pre-process the exemplar file before training, To do this I wrote a simple Java program which iterates over the exemplars removing any duplicate exemplars and if an two exemplars existed with the same input but different out puts to choose the exemplar with the higher number of “on” outputs(outputs which had a value of 1).

Algorithm 2 Pre-Process Exemplars

```
Initialize HashMap $H$
forEach e in EXEMPLAR_FILE do
  split exemplar file e on “;” char into input string $i$ and output string $o$
  if $H$ .hasKey( $i$ ) then
    if numberOfOn( $o$ ) > numberOfOn( $H$ .get( $i$ )) then
      $H$ .setValue( $i$ , $o$ )
    endIf
  else
    $H$ .add( $i$ , $o$ )
  endIf
endFor
writeToFile( $H$ )
```

3.3.3 Training Round 3

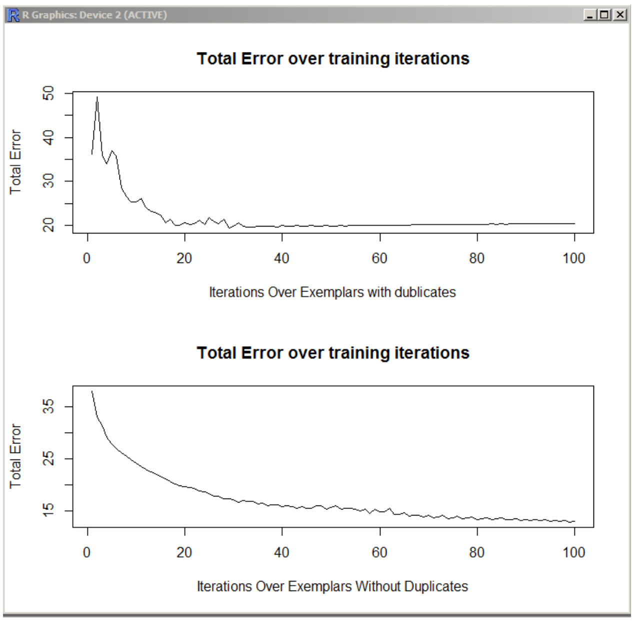


Figure3.5: Training error for network trained on exemplars with duplicates(top) and exemplars without(bottom)

With the pre-processed exemplars the network again made significant progress, the networks error was significantly reduced Figure3.5 shows a graph of the total error over each iteration of the exemplar file. We can see a dramatic effect in performance, With duplicates present we can see over-learning starting to occur after around the 40 iteration mark where the error actually begins to creep back up. We can see without duplicates the error is not only lower but a more stable curve is produced with over-learning not as much of an issue at 100 iterations. With this the network was able to make its way over the obstacles in its path, However now a new problem arose. After training with the pre-processed exemplars the network made it over all obstacles in its path but was unable to make it over pits in the level. After examining the exemplars again to try and see if the cause lay in them, I was unable to find anything significant so I decided to look at the networks configuration.

The current network was using a input tile radius of 4 as such there were 81 single inputs to the network. I began looking to see how the input size effects the networks performance. Trying the network with larger and smaller radius I found the network performance increased the lower the radius became with the lowest being a 2 radius(Figure 3.4 shows the difference between a 2 and 4 radius input scheme) As such a radius of 2 was used for the next round of training.

With the network trained with a input radius of 2 it was able to make It a lot further through the level, more then half way from completing the level. However in the later sections of the level jumping at the correct moment is significantly more important as there is a larger number of enemy's present and hard pit like obstacles to get over. The issue re-arose of the network not classifying inputs with the jump output “on”. To get over this issue I added a selective training procedure to my original training algorithm(Algorithm 1) using the same approach as [1]. When an exemplar is received where the output contains a jump and the network outputs a value less then 0.1 for it add it to a list. At the end of each iteration we replay the list until the exemplars are classified with the correct output. With this the original training algorithm(1) was modified as follows:

Algorithm 3 Supervised Learning With Selective Exemplar Replay

```
Initialize random network weights between some small value -C and C
Initialize storage $S$
for iteration = 1, TRAIN_ITERATIONS do
  forEach $e$ in EXEMPLAR_FILE do
    Split the $e$ on the ";" char to get the inputs $I_i$ and correct
    output $O_k$
    Send the inputs $I_i$ for the exemplar $e$ to the network
    Store the correct exemplar output $O_k$
    Forward-propigate $I_i$ and observe the output $y_k$
    Measure the error $E$
    if $y_{jumpIndex} < 0.1$ then
      add $e$ to $S$
    endIf

    back-Propagate Error as in Algorithm 1
    write the results for passing and updating the network with $e$
  end forEach

  if $S$ is not empty then
    replay exemplars in $S$ until all classified correctly
  endIf

end for
write the trained networks weights to a XML file to be able to be parsed and
loaded into the system at later time if needed.
```

3.3.4 Training Round 4

After training the network with the above selective training procedure in place the network was now 90% from completing the level. I just had to resolve one other issue that I identified in this round of training which was not to do with the network but actually with how the super Mario game registers a jump action by the player. In Super Mario Bro's for the game to register the player pressing the jump key on the controller the key must not be pressed before the Mario hits the ground. So to illustrate this, If a player makes a jump and holds the jump button for the entire jump and when Mario lands on the ground Mario will not jump again until the button has been released and repressed as the jump button was not registered. This typically isn’t an issue cause a player can see they are about to hit the ground on release the button before they hit it and then repress it.

The problem for the network is that it is making decisions at a frame by frame basis so much faster then a human and as such might press the jump button literally a frame before it has registered in memory that Mario is on the ground resulting in the jump output not being executed. The reason why this became a issue is there is an obstacle later in the level that requires the user to jump on top of an obstacle and then immediately jump over a gap if the player just keeps walking the fall into the gap and die in the game. The network was jumping onto the obstacle and then I could see in the run logs it was making the correct the output jumping when it landed but watching the run Mario just kept walking of into the pit. By reading the super Mario ROM map[2] I found some memory locations that’s uses in the game would allow me to right a function that would detect if Mario was failing from the air after jumping and landed on the ground if the jump button was pressed in the frame before he landed to set the jump output to 0 for a single frame allowing the game to execute the networks jump output for the following frame. The ram addresses used to detect and resolve this are described below:

RAM Address	Values	Description
0x001D	0x00 → 0x03	Player "float" state • 0x00 - Standing on solid/else • 0x01 - Airborne by jumping • 0x02 - Airborne by walking of a ledge • 0x03 - Sliding down flagpole

Using this after each frame advance in the emulator I read and store this value in a variable MARIO_STATE, and also before making a frame advance I store the previous state value into a variable PREV_MARIO_STATE this allowed me to create the following function:

ResetJumpOnAirToGround(buttons)

```
//buttons is the array which contains the current buttons to be executed by the emulator
if PREV_MARIO_STATE >= 0x01 and MARIO_STATE == 0x00 and buttons["P1 A"] == true then
      buttons["P1 A"] = false
      joypad.set(buttons)
      emu.frameadvance()
      buttons["P1 A"] = true
end
```

This very simple function allows me resolve this issue.

3.3.5 Training Round 5

Retraining the network with all the previously described fixes to the problems identified the network was retrained and completed the first level in Super Mario Bro's successfully. A video demonstrating the network running and playing the level can be found at [L.1], The file containing the network weights for the network can be found on GitHub at [L.2] if you wish to load the values into the system and run the network for yourself.

3.4 Supervised Learning Conclusions

The training scheme described above shows that the network is capable of learning how to navigate through the level in Super Mario Bro's when shown exemplars from the level. When the trained network was presented with the second level which was completely unseen before, the network was not able to progress far at all due to the dramatic differences between the two levels. As I completed the initial goal of training a network to complete the first level of Mario with supervised learning much earlier as expected I decided to look at a different technique to be able to train the network with out having to show it human gameplay of the level it is looking to complete and have it learning entirely on its own. As such I began to look at Reinforcement learning, more specifically Q-learning.

4 Reinforcement Learning

Reinforcement learning comes from the idea of being able to tell the machine whether or not a action it performed was good or not and the machine being able to learn from its previous experiences what the best action to make is given the current state. This method differs from the standard supervised learning scheme as described above in that correct input/output pairs(exemplars) are never presented and the machine learns completely from its own experiences based on the reward it received for making an action in the current state. This leaning method I found was very attractive to use for this type of application of machine learning as it is the exact same way we humans learn to complete the game. Where we try different moves given what we see on the screen and learn from what gets us further into the level and what does not.

4.1 Q-learning

The technique I used to implement reinforcement learning was Q-learning. Q-learning works where by a function $Q(x,a)$ returns an value which indicates how good it is to execute the given action a when in the state x . Typical applications of q-learning use a lookup table to store all the these Qvalues. The basic algorithm works as follows:

1. get the current state x
2. select the action a which has the highest $Q(x,a)$ value
3. execute a and observe the next state y
4. receive some reward r for transitioning from state $x \rightarrow y$
5. update the value $Q(x,a)$ to reflect the given reward
6. repeat steps $1 \rightarrow 5$ for the next state until the end of the training iterations.

With the size of the state space as defined by my input encoding scheme using a lookup table would be unfeasible and extremely inefficient as the total number of possible states is $3^I where\, I=(r\times2+1)^2$ for even the smallest tile radius of 2 this would be $3^{25}=847288609443$ possible states and the agent may only experience 10% of those states. Creating an array of floats where a float takes up 4 bytes would mean potentially taking up to 3.389 Terabytes of memory! Even if I created some dynamic creation of the array in which we only stored Qvalues that the agent actually experienced. Even if the agent only experiences 1% of the possible states it is still an array using 33.891 Gigabytes of memory! This is where the powerful capabilities of neural networks really shine as we can use a neural network to act as a function approximator for $Q(x,a)$ and be able to work on such a large state space problem.

4.2 Q-learning With Neural Networks

To use a neural network as a function approximator for $Q(x,a)$ we have to change our current architecture so that we are now passing the state,action pair in the inputs and we only have one output neuron representing the value Q:

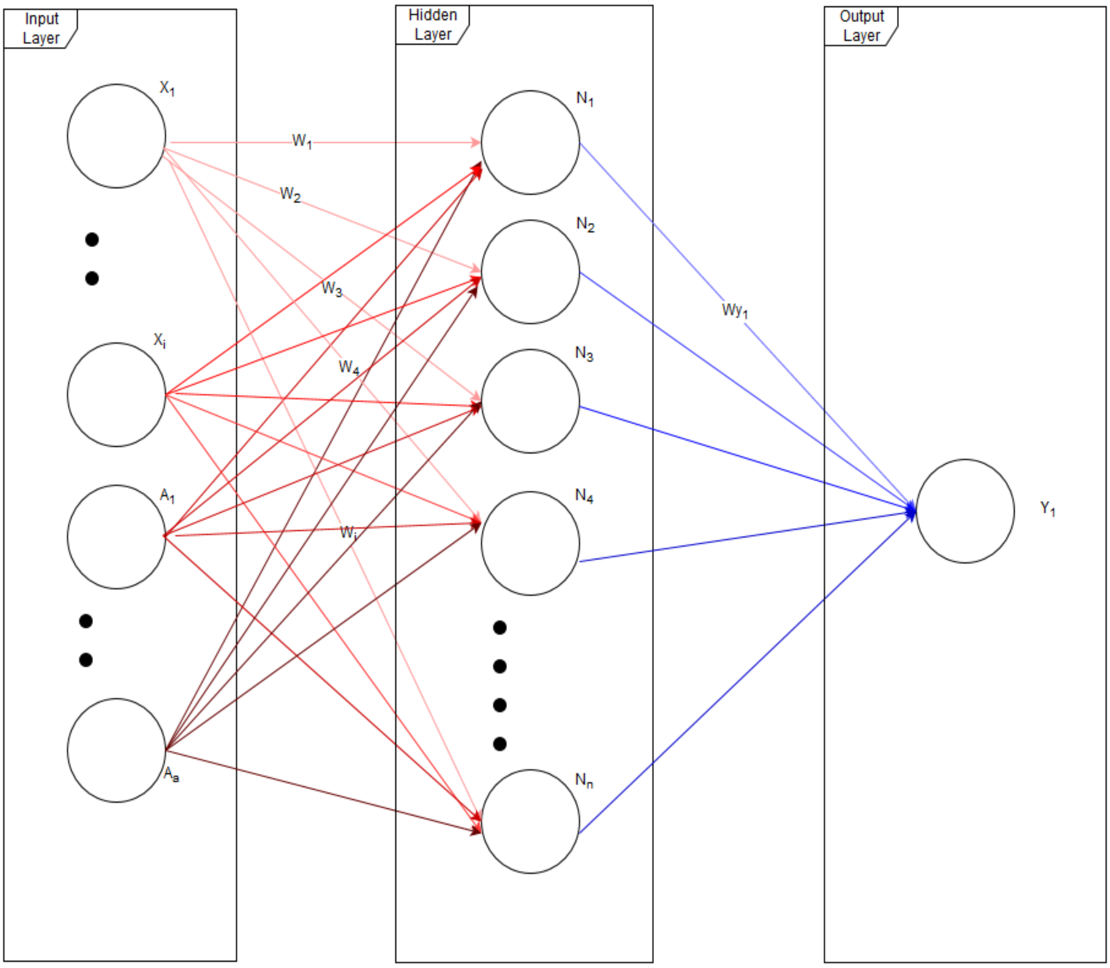


Figure 4.1: Q-learning Neural Network Architecture

As with Q-learning we are learning based off how well an action was based off the received reward for making taking that action. We are now learning from estimates. The output we are moving towards:

$$O_k = r + \gamma \max_{b \in A} Q(y,b)$$

Where:

r is the reward value return from our reward function for making the transition from state x to y

γ is the discount function which determines how much the agent should choose imediate rewards over later ones $0 > \gamma > 1$

$\max_{b \in A} Q(y,b)$ is the higest q value for the next state y

A is all possible action the agent can perform

The way we update the q-values is the same way we would if we were using a lookup table:

$y_k := (1-\alpha) y_k + \alpha O_k$

or

$Q(x,a) := (1-\alpha) Q(x,a) + \alpha(r + \gamma \max_{b \in A} Q(y,b))$

Where:

α is our learning rate $0 < \alpha < 1$

To update the network to get closer to the output closer to the estimate O_k we can again use back-propagation to back-propagate the error $E = \frac{1}{2}(y - O)^2$

4.3 Exploration vs Exploitation

In the algorithm for q-learning in 4.1 we are always selecting the highest Qvalue this makes sense you want to choose the best action. However at the start of training it is better for the agent to explore the environment so that it can get a much better understanding and a broader view of what how to react in the environment. With this it would be best so to have the agent explore(choose not necessary the best action)at the start and as training goes on begin to exploit the better Qvalues.

The way we can achieve this is by using the Boltzmann probability distribution[3]:

$$P(a|x) = \frac{e^{Q(x,a)/T}}{\sum_{b \in A} e^{Q(x,b)/T}}$$

Where T balances the effect of exploration vs exploitation.

4.4 Experiences

As shown in the formulas in 4.1 at each step we are receiving the information that taking action a in state x lead to state y and revived the reward r . As the network would need a lot of updates in order to learn, what we can do to speed up the process is to save these 4 values (Experience) and replay them back to the network a set number of times after each training run. At the begin of the rounds of training I used backwards experience replay in which experiences are represented to the network with the latest experience being first (the array I=is read backwards).

4.5 Rewards

In order to be able to inform the agent on how well the execution of an action was in the given state I had to define a reward function to be able to reward the agent for progressing further in the level and penalising if the action resulted in either no progression closer to finishing the level or resulting in Mario being killed by either an enemy or falling into a pit. The finalised reward scheme is designed to reflect this points is as follows:

```
$r = 0.0$
if Mario hit an enemy then
  return -0.2
if Mario fill in a pit then
  return -0.2
if Mario got closer over an obstacle then
  $r = r + 0.2$
if Mario got closer over an enemy then
  $r = r + 0.2$
```

```
if the previous state $x$ is the same as the new state $y$ then
  $r=r-0.04$
if Mario's x position has increased i.e. Further in the level then
  $r=r+0.1$
if $r$ == 0.0 then
  $r=r-0.02$
return $r$
```

With all of this we can now create our algorithm for leaning with q-learning:

Algorithm 4 Reinforcement Learning with Q-learning

```
Initialize network with inputs $I+Actions$
Initialize random network weights weights between some small value -C and C
Initialize experience storage $E$
Initialize Qvalue tupale storage $S$
for run = 1, TRAIN_RUNS do
  for experience = 1, EXPERIENCES_PER_RUN do
    forEach $a$ in $Actions$ do
      load current state $x$ inputs into network
      load Action inputs with $Action_a=1$
      forward-propagate the inputs
      receive output $Qx$
      store in $S$ the tupale $(Qx, i, a)$
    end ForEach
    using Boltzmann $P(a|x)=\frac{e^{Q/T}}{\sum_{b\in S.A} e^{Q/T}}$ select an action
    execute the controller button mapped to the chosen action and get the new
    state inputs $y$
    receive the reward $r$
    forEach $a$ in $Actions$ do
      load state $y$ inputs into network
      load Action inputs with $Action_a=1$
      forward-propagate the inputs
      receive output $Qy$
      keep the highest $Qy$ value
    end ForEach
    //if terminal state
    if marioHitEnemy() or marioFellInPit() then
      $O_k=r$
    else then
      $O_k=r+\gamma Qy$
    end If
    calculate $Qx:=(1-\alpha)Qx+\alpha O_k$
    back-propagate $E=\frac{1}{2}Qx-O_k$
    store (x,a,y,r) in $E$
  end For
  backwards replay Experiences $E$
end For
```

4.6 Q-learning Training

4.6.1 Training Round 1

In the first round of training very little progress was made the reason being with the configuration of passing the current action to be evaluated to the network in the inputs the network was having an extremely difficult time in separating the actions from the inputs and as such updating the network for one action was nearly identical to updating for all actions so updating for one action might as well have been the same as not updating. To fix this I began to look ways to best separate the actions,inputs to the network. As in[4] I have such a small action space 6 buttons I can create six individual networks each representing each of the different actions that way updating for one action would never effect any other action and the only inputs to the networks is the state input. I then modified algorithm 4 to facilitate multiple action networks:

Algorithm 5 Reinforcement Learning With Q-learning Using Multiple Action Networks

```
Initialize Action networks $ActionNet_A$
Initialize random network weights weights between some small value -C and C for all $ActionNets_A$
Initialize experience storage $E$
Initialize Qvalue tupale storage $S$
for run = 1, TRAIN_RUNS do
  for experience = 1, EXPERIENCES_PER_RUN do
    forEach $a$ in $Actions$ do
      load current state $x$ inputs into $ActionNet_a$
      forward-propagate the inputs
      receive output $Qx$
      store in $S$ the tupale $(Qx, i, a)$
    end ForEach
    using Boltzmann $P(a|x) = \frac{e^{Q/T}}{\sum_{b \in S.A} e^{Q/T}}$ select an action
    set the button on the controller input in mapped to the chosen action to
    true and set all others to false
    execute the controller input
    state inputs $y$
    receive the reward $r$
    forEach $a$ in $Actions$ do
      load state $y$ inputs into $ActionNet_a$
      forward-propagate the inputs
      receive output $Qy$
      keep the highest $Qy$ value
    end ForEach
    //if terminal state
    if marioHitEnemy() or marioFellInPit() then
      $O_k = r$
    else then
      $O_k = r + \gamma Qy$
    end If
    calculate $Qx := (1-\alpha) Qx + \alpha O_k$
    back-propagate $E = \frac{1}{2} Qx - O_k$ for $ActionNet_a$
    store (x,a,y,r) in $E$
  end For
  backwards replay Experiences $E$
end For
```

4.6.2 Training Round 2

With the multiple action nets I began to see progress and Mario was beginning to makes it way through the level reasonably well. However a problem arose when Mario reached the first of the highest obstacles in the level. To get over objects of this highest the player must actually hold the jump button for the entire hight of the jump and then at its peak press the directional button that will place Mario on top of the object at the same time. Its actually not possible just hold the jump button and then release it and press the directional button at the peak as for what ever reason in the games design the player will not have enough height to make it. As with the current algorithm we are only executing one button at a time this means Mario was never able to reach the top of the obstacle and thus get over it. To resolve this I added to new actions to the implementation, Action LEFT,B and RIGHT ,A. This meant adding another two networks and resulted in the action space looking as shown in figure 4.2.

Q(x,"A")
Q(x,"B")
Q(x,"UP")
Q(x,"LEFT")
Q(x,"RIGHT")
Q(x,"DOWN")
Q(x,"LEFT , A")
Q(x,"RIGHT A")

Figure 4.2 Q-learning Action Space

4.6.3 Training Round 3

Training the network with the additional actions added the network was able to make it over the higher obstacles and would make very good progress through the level and was showing very strong signs of completing it. However it in the early stages of my reward scheme I was rewarding the agent very highly for getting closer over and object and penalising heavily for getting hit by an enemy. What this actually caused the agent to learn was the best thing for him to do was to jump in front of the highest obstacles and then once at the peak instead of jumping on top it to the right it would jump back to the left and continue doing this over and over as it would always give him the highest possible reward values. A video showing this happening can be found at [L.3] and happens at the 1:08 mark. With this reduced how much I was rewarding for getting closer over an enemy and object and penalising for hitting an enemy or falling into a pit to the values shown in 4.4. I also added into my functions detecting if Mario is closer over an enemy or object that it would only return true if Mario was actually facing what he was actually trying to get over. Along with this I added to the reward scheme so that the reward scheme would be able grow/shrink if an action accomplished multiple things such as getting closer over a object and increasing his x position. I did not modify the punishments for hitting an enemy or failing into a pit as these are terminal states and result in the level having to be start from the beginning.

4.6.4 Training Round 4

With the new reward scheme in place Mario was no longer getting stuck and learning this incorrect strategy and was able to progress through the level to the point where he was now able to complete the level completely by learning from rewards and his previous experiences, however this now presented a new issue. The new issue was the problem with Forgetting in Neural Networks was now apparent . As the Network was progressing into the later stages of the level it was still continuously updating its weights to reflect the experiences it was encountering how ever in doing so this would also effect the weights related to the inputs from the earlier stages in the level, Thus in some cases completely overwriting them resulting in the network choosing the incorrect action when placed back at the beginning of the level after completing it. Even though it previously was able to make the correct actions. That has lead me to now re-evaluate my replay strategy in order to hopefully minimize this effect. At the time of writing this document is still have not been able to effectively come up with a replay strategy to resolve this. I have gotten very close by reducing how many experiences I show the network at each run and this has shown strong improvement and the finished trained network is able to complete most of the early stages of the level but unable to make it to the later stages as it either falls into a pit or hits an enemy.

5 Developed System To Facilitate Using Neural Networks With Super Mario Bro's

In order to be able to effectively build and train the neural networks used I designed and wrote a system that would allow me easily and effectively modify network configurations to make training and trying new experiments as easy and painless as possible. Along with this the system developed would produce multiple files in the form of run logs and training logs. This files were designed to aid in the debugging process of the network by being able to see what was happing at each step of the networks stages to get a good understanding of what was happening and where problems were arising.

The development philosophy I was following with the design of the system was so that anyone regardless of their background in neural networks could easily load the main Lua script into BizHawk and begin to train and build there own network capable of playing Mario. The following section describes the systems overall architecture and each of the functions that make up it.

5.1 System Architecture

The majority of the system is written in the scripting language Lua and 2 functions were written using Java. A breakdown of the files that make up the system are as follows:

- SMB_Neural_Net.lua
  - This script provides the main functionality of the system the functions included in this script are:
    - Read Tile values in radius Around Mario (2.2)
    - Record Exemplars (3.1)
    - Parse and load Network weight values from XML file
    - System configuration file
    - Log Network operations during runtime for evaluation
    - User Interface
    - Exploit the network
    - Train Network with Q-learning(4)
- Train_network_supervised.lua
  - This script trains the network using supervised learning using the exemplars file currently selected for us in the configuration file. The functions included in this script are:

        - Parse and load existing network weight values for retraining
        - Log Network training operations for evaluation
        - Use selective training procedure (3.3.3)
- ReduceInputs.java
    - This Java class takes 2 arguments the location of the exemplar file for processing and the radius of the inputs you wish to reduce it to. The usage of this file is to reduce a exemplar file that was recorded with a high radius down to a specified radius.
- PreProcess_Exemplars.java
    - This Java class removes any duplicate exemplars in the file passed as its argument and if two exemplars are found with with same input but different outputs the one with the higher number of “on” outputs is selected. This is previously discussed in 3.3.2.

Along with these files the network

5.1.1 Functions

This section will covers the each of the main functions that make up the system and how the were implemented to facilitate the training of the previously described networks.

Parse And Load XML Network Weights

After finishing training the networks weights are written to an XML file with the following data structure:

Supervised trained network	Q-Learning trained network:

<ExemplarFileName_TrainIterations>
  <IL_HL_Weights>
    < i_I ></ i_I >
    <iT></iT>
  </IL_HL_Weights>
  <HL_OL_Weights>
    < j_J ></ j_J >
    <jT></jT>
  </HL_OL_Weights>
</ExemplarFileName_TrainIterations>
	<Q-Learning_NETWORKS>
	  < ACTION_NET_a >
	    <IL_HL_Weights>
	      < i_I ></ i_I >
	      <iT></iT>
	    </IL_HL_Weights>
	    <HL_OL_Weights>
	      < j_J ></ j_J >
	      <jT></jT>
	    </HL_OL_Weights>
	  </ ACTION_NET_a >
	</Q-Learning_NETWORKS>

Tag Descriptions:

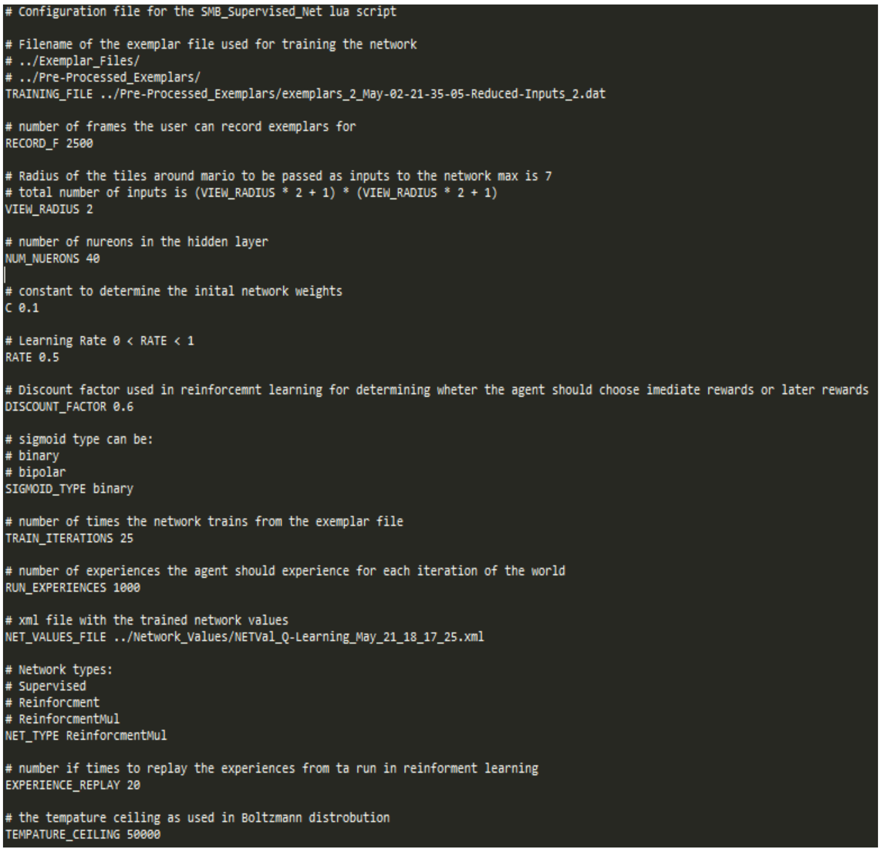
<IL_HL_Weights>	Weights for the synapses from the Input layer to the Hidden layer
<HL_OL_Weights>	Weights for the synapses from the Hidden layer to the Output layer
$< i_I >$	Input node i_I : stores the weights w_{ij}
$< j_J >$	Hidden node j_J : stores the weights w_{jk}
<iT>	The thresholds for the Hidden layer nodes j_J
<jT>	The thresholds for the output layer nodes k_K
$<$ ACTION_NET$_a$ $>$	Represents each individual action network

weight values are separated by the ' | ' char.

Parsing this file is made possible with Lua's patterns functionality, Patterns in Lua are regular expressions the only difference is that patterns are not not as powerful as they do not allow of the “and” and “or” operators found in normal regular expressions. The nice thing how ever is that in patterns you can specify which data point in the matched pattern you want to extract by encapsulating it in brackets. This is why in my XML data structure I include the index of the input node in the tag as I can extract this value and use it when loading the values into the array structure I use for my networks.

System Configuration File

In order to make the system as flexible as possible and to make the process of trying new network settings as easy as possible I decided early on to encapsulate all the main system and network variables into a configuration file. This meant that I could easily build a network with some configuration and if I needed to move to another computer running the system I could load some previously saved configuration and be able to work on the same network environment. It also meant changing network values did not involve having to make modifications to the code base in order to change something in the network. This function again makes use of Lua's pattern functionality and utilises Lua's table data structure for storing the values in a very nice data structure. Figure 5.1 shows the variables contained in the configuration file and its structure.



```
# Configuration file for the SMB_Supervised_Net lua script

# Filename of the exemplar file used for training the network
# ../Exemplar_Files/
# ../Pre-Processed_Exemplars/
TRAINING_FILE ../Pre-Processed_Exemplars/exemplars_2_May-02-21-35-05-Reduced-Inputs_2.dat

# number of frames the user can record exemplars for
RECORD_F 2500

# Radius of the tiles around mario to be passed as inputs to the network max is 7
# total number of inputs is (VIEW_RADIUS * 2 + 1) * (VIEW_RADIUS * 2 + 1)
VIEW_RADIUS 2

# number of nureons in the hidden layer
NUM_NUERONS 40

# constant to determine the inital network weights
C 0.1

# Learning Rate 0 < RATE < 1
RATE 0.5

# Discount factor used in reinforcemnt learning for determining wheter the agent should choose imediate rewards or later rewards
DISCOUNT_FACTOR 0.6

# sigmoid type can be:
# binary
# bipolar
SIGMOID_TYPE binary

# number of times the network trains from the exemplar file
TRAIN_ITERATIONS 25

# number of experiences the agent should experience for each iteration of the world
RUN_EXPERIENCES 1000

# xml file with the trained network values
NET_VALUES_FILE ../Network_Values/NETVal_Q-Learning_May_21_18_17_25.xml

# Network types:
# Supervised
# Reinforcment
# ReinforcmentMul
NET_TYPE ReinforcmentMul

# number if times to replay the experiences from ta run in reinforment learning
EXPERIENCE_REPLAY 20

# the tempature ceiling as used in Boltzmann distrobution
TEMPATURE_CEILING 50000
```

Figure5.1: System Configuration File

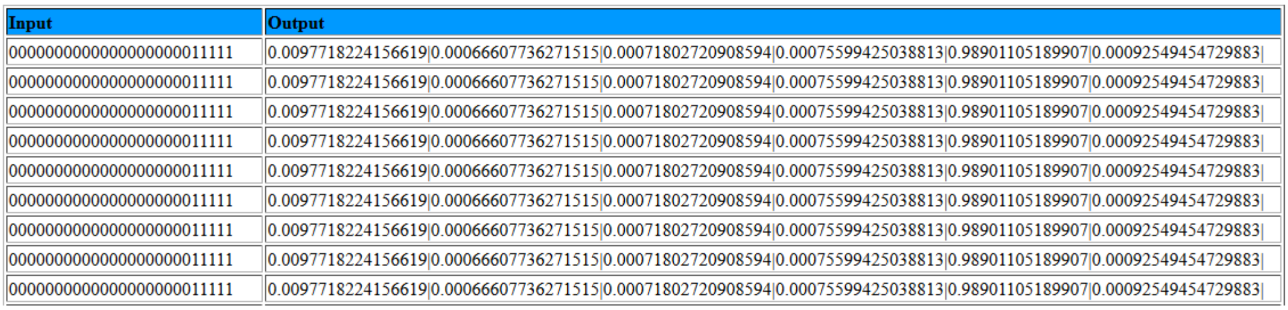
Making use of Lua's table structure, data is easily accessible and accessed by passing the configuration variables name as the index for example to access the value for the number of neurons in the hidden layer you can access it simply with config[“NUM_NUERONS”] this structure meant adding new variables to the file was efficient and required very little effort.

Log Network operations

Logging network operations consists of 2 types of operations those during run time and those during training.

Runtime Logs:

When the user exploits the system the networks input and network outputs are written to an XML file, the reason for using XML is that a XSL style-sheet can be applied to the XML files so that they are easily readable by users.

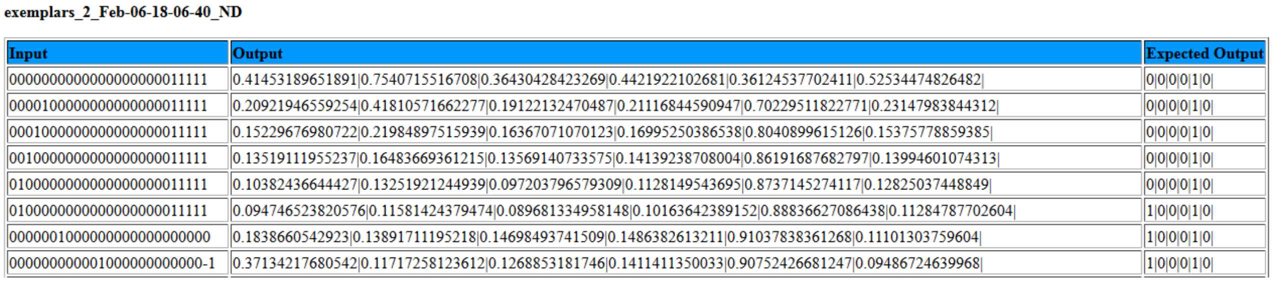


Input	Output
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|
0000000000000000000011111	0.0097718224156619|0.00066607736271515|0.00071802720908594|0.00075599425038813|0.98901105189907|0.00092549454729883|

Figure 5.2 Runtime log

Supervised Learning logs:

During training of the network with supervised learning the user can specificy if they wish to log training. This file consists of the exemplar input, Network Outputs and the correct Exemplar Outputs.



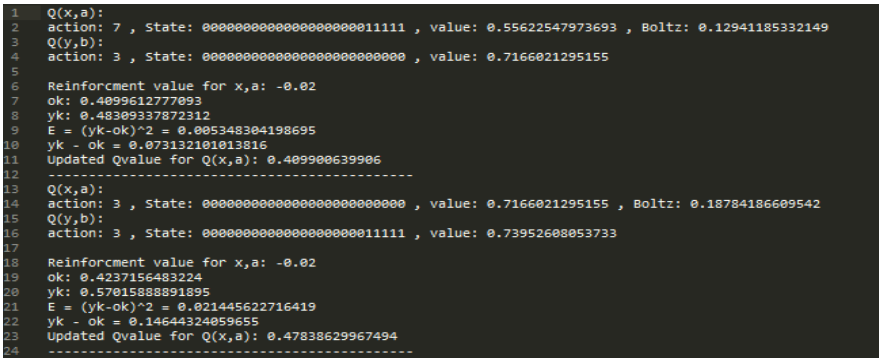
exemplars_2_Feb-06-18-06-40_ND

Input	Output	Expected Output
0000000000000000000011111	0.41453189651891|0.7540715516708|0.36430428423269|0.4421922102681|0.36124537702411|0.52534474826482|	0|0|0|0|1|0|
0000100000000000000011111	0.20921946559254|0.41810571662277|0.19122132470487|0.21116844590947|0.70229511822771|0.23147983844312|	0|0|0|0|1|0|
0001000000000000000011111	0.15229676980722|0.21984897515939|0.16367071070123|0.16995250386538|0.8040899615126|0.15375778859385|	0|0|0|0|1|0|
0010000000000000000011111	0.13519111955237|0.16483669361215|0.13569140733575|0.14139238708004|0.86191687682797|0.13994601074313|	0|0|0|0|1|0|
0100000000000000000011111	0.10382436644427|0.13251921244939|0.097203796579309|0.1128149543695|0.8737145274117|0.12825037448849|	0|0|0|0|1|0|
0100000000000000000011111	0.094746523820576|0.11581424379474|0.089681334958148|0.10163642389152|0.88836627086438|0.11284787702604|	1|0|0|0|1|0|
0000001000000000000000000	0.1838660542923|0.13891711195218|0.14698493741509|0.1486382613211|0.91037838361268|0.11101303759604|	1|0|0|0|1|0|
000000000000100000000000-1	0.37134217680542|0.11717258123612|0.1268853181746|0.1411411350033|0.90752426681247|0.09486724639968|	1|0|0|0|1|0|

Figure 5.3 Supervised Learning log

Q-learning logs:

When the network is trained with Q-learning data for each state transition and network update is written to a file that can easily be examined.



```
Q(x,a):
action: 7 , State: 0000000000000000000011111 , value: 0.55622547973693 , Boltz: 0.12941185332149
Q(y,b):
action: 3 , State: 0000000000000000000000000 , value: 0.7166021295155

Reinforcment value for x,a: -0.02
ok: 0.4099612777093
yk: 0.48309337872312
E = (yk-ok)^2 = 0.005348304198695
yk - ok = 0.073132101013816
Updated Qvalue for Q(x,a): 0.409900639906
-------------------------------------------
Q(x,a):
action: 3 , State: 0000000000000000000000000 , value: 0.7166021295155 , Boltz: 0.18784186609542
Q(y,b):
action: 3 , State: 0000000000000000000011111 , value: 0.73952608053733

Reinforcment value for x,a: -0.02
ok: 0.4237156483224
yk: 0.57015888891895
E = (yk-ok)^2 = 0.021445622716419
yk - ok = 0.14644324059655
Updated Qvalue for Q(x,a): 0.47838629967494
-------------------------------------------
```

Figure 5.4 Q-Learning log

User Interface

The system provides simple user interface allowing user to access each of the systems functionality (Figure 5.5) selecting functionality by pressing the cosponsoring key on the number pad. Along with this during the networks exploitation it provides a visual representation of the inputs the network is receiving and the outputs on the NES controller that is currently being executed(Figure 5.6) Users can also view important currently loaded configuration variables by accessing the settings menu which also provides the functionality to reload the configuration file updating the system with the new values(Figure 5.7).

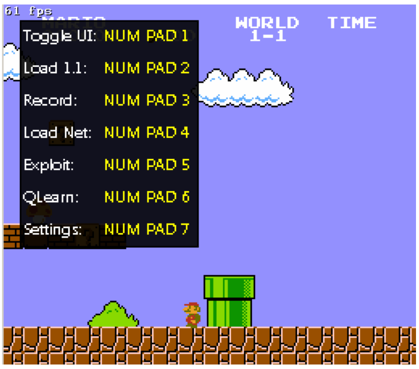


Figure5.5: User Interface

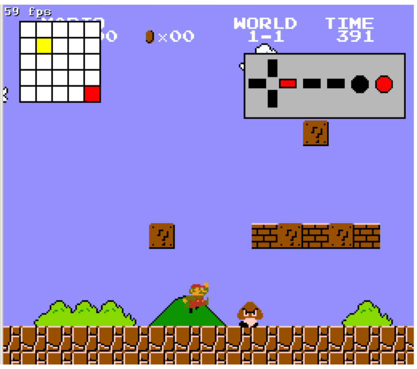
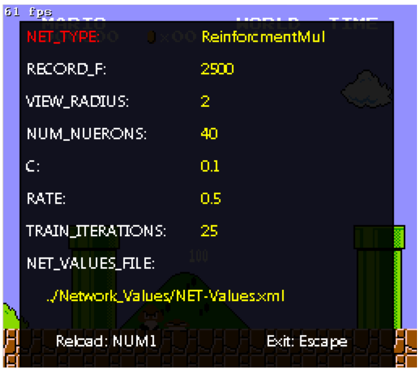


Figure5.6 Network Exploitation View



61 fps

NET_TYPE: ReinforcmentMul

RECORD_F: 2500

VIEW_RADIUS: 2

NUM_NUERONS: 40

C: 0.1

RATE: 0.5

TRAIN_ITERATIONS: 25

NET_VALUES_FILE:

../Network_Values/NET-Values.xml

Reload: NUM1 Exit: Escape

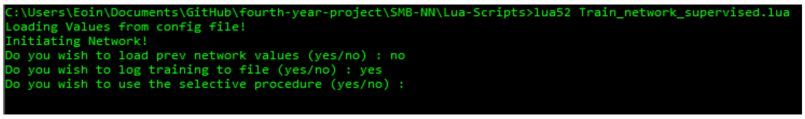
Figure5.7: Settings menu

Exploit Network

The system provides the capabilities to exploit the currently loaded network whiten in the system. Based on the network type that loaded and specified in the system configuration the system will exploit the network in one of two ways for each of the different network types. The first being for networks trained with supervised learning where the network with forward-propagate the current tile inputs and receive the network outputs based on a set threshold if the output at each node is greater then the threshold the button on the controller is pressed. For networks trained on reinforcement learning the network will forward-propagate the outputs through each of the action networks. The button corresponding to the action network with outputs the highest Qvalue is pressed.

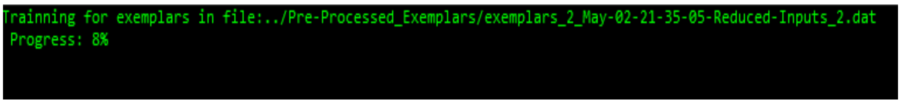
Train_Network_Supervised.lua

Training the network with supervised learning is separated from the emulator in order to increase performance and reduce the computation required to do so and as such increase the speed of training. When the user runs the script the exemplar file and the networks configuration is read from the configuration file. The user is presented with 3 options before training commences(Figure 5.8) theses are; Load previous Network from file stored in the configuration file, Log the training operations and use the selective training procedure. Once training commences the script will continuously output the progress of training allowing the user to judge how long is left to complete training the network(Figure 5.9).



```
C:\Users\Eoin\Documents\GitHub\fourth-year-project\SMB-NN\Lua-Scripts>lua52 Train_network_supervised.lua
Loading Values from config file!
Initiating Network!
Do you wish to load prev network values (yes/no) : no
Do you wish to log training to file (yes/no) : yes
Do you wish to use the selective procedure (yes/no) :
```

Figure5.8: Supervised learning Training options



```
Trainning for exemplars in file:../Pre-Processed_Exemplars/exemplars_2_May-02-21-35-05-Reduced-Inputs_2.dat
 Progress: 8%
```

Figure5.9: Training Progress display

6. Future Work

Visual Representation Of Trained Networks

In the future development of this project I hope to develop a way of graphical representing the trained network similar to the graphs depicting the network architecture in Figure 2.1 and Figure 4.1 I had initially wanted to do this by parsing the XML file containing the network network value and displaying it in this form where the lines representing the weights would have their thickness vary depending on the value of the weight. Thick lines would represent a high weight value and small lines would represent a low weight value. This was the reason I chose XML as creating such functionality is relatively simple by navigating through the XML elements with a language such as Javascript. I feel this feature would be extremely useful in debugging as it would allow users to be able to identify which inputs are having the highest effect in the network based on their weights which would be easily identifiable in such a representation.

Different Network Architectures

Another area I wish to expand on with this project is making use of different network architectures rather then just the FNN architecture used in this project I would like to look at Recurrent Neural Network's(RNN) and deep learning architectures.

References

[1] Murray F. Alan, Applications of Neural Networks, University of Edinburgh, 1995, page 40 (3)

[2] romhacking, Super Mario Bros RAM Map, [ONLINE] Available at: http://datacrystal.romhacking.net/wiki/Super_Mario_Bros.:RAM_map [Last Accessed 21 May 2016]

[3] Dr Humphry's Mark, Boltzmann "soft max" distribution, [ONLINE] Available at: http://computing.dcu.ie/~humphrys/Notes/AI/boltzmann.html [Last Accessed 21 May 2016]

[4] Dr Humphry's Mark, Action Selection Methods Using Reinforcement Learning, Cambridge 1997, 4.3.2

Links

[L.1] https://www.youtube.com/watch?v=4n8xiN9e5gM

[L.2]https://raw.githubusercontent.com/Eoin-Mur/SMB-Neural-Network/master/SMB-NN/Network_Values/SuperviesedNetwork_Used_in_Video.xml

[L.3] https://www.youtube.com/watch?v=gIhxU2Ub5us