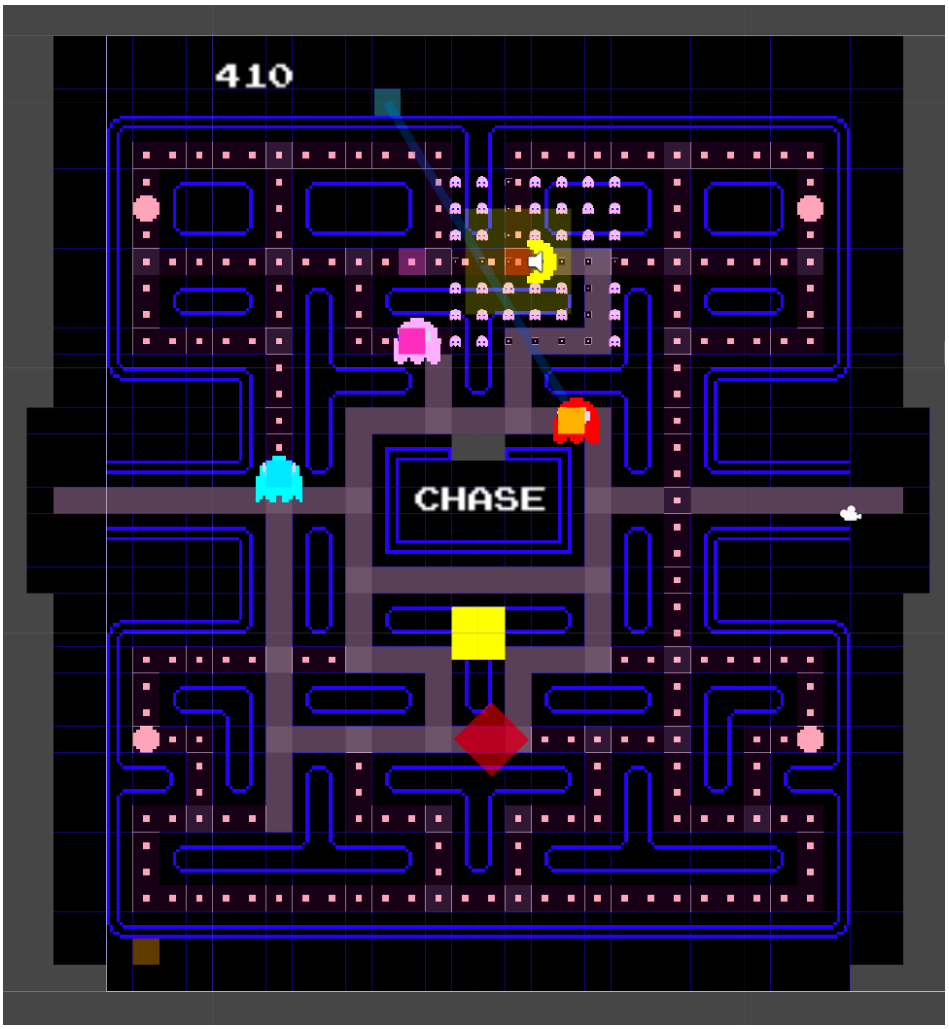
Matthew Duddington 33435289 *Luke Sanderson 33497213*

Mathematics and Graphics for Computer Games 2

Assignment 4
May 2017

Investigating Neural Networks for Autonomous Pac-Man



Contents

Project Overview

Repository and Video Links

github.com/MatthewDuddington/MG2_02_MachineLearning
youtu.be/Ol0_9Hj_92E

Aims

Our initial intention with our aims for this project was to set a task that would be sufficiently simple that we would be able to achieve it straightforwardly and then be able to iterate additional illustrative features and functions on top of. These aims are reflected in the list below, however, during the research and implementation, we gradually discovered that the challenges presented by our deceptively seemingly straightforward list of tasks was much more complex than we initially understood.

- Research a practical understanding of the general structure of and approaches to Neural Networks.
  - Identify an approach or multiple approaches that would be suitable for a specific simple task.
  - Locate existing framework code from which it might be possible to adapt an approach
- Create a simple set of interconnected classes that demonstrate machine learning techniques in practice.
  - Create a basic neural network that can 'learn' how to perform an overall task.
  - Have evolutionary stratification of the agents created in this process.
  - Use specific metrics to evaluate performance of each generation of agents.
- Contextualise the project within a games environment.
  - Use the Unity engine and C# as a basis for prototyping.
  - Using Pac-Man as a learning environment, the network will learn the optimal approach to playing.
    - Build a functioning replica of classic PacMan.
    - Develop information hooks for the Neural Network to perceive the game from.

Of these aims, the functionally faithful implementation of classic PacMan performs well as a working test environment from which various types of input data could be measured. Likewise, we certainly increased our understanding of the variety, scope and common challenges and limitations faced by different types of common neural networks. However, as we sought to solve each interconnected challenge, the size of the research space gradually became overwhelming to us. Attempting to identify a 'suitable approach' lead to many differing opinions between sources; this is perhaps reflective of the growth and age of the field itself.

Consequently, our practical application of the neural network grew out of multiple re-configurations, resulting in a set of unsuccessful implementations. While some elements operate correctly, in so far as we have a PacMan that will traverse a part of the maze and can observe an incremental development of saved network variables that are recording reactions to the training data, the behaviour is not consistent with a useful model of play nor did it notably improve with more training. AI controlled PacMan will usually gravitate into a particular corner of the map, ignore the ghosts or constantly attempt an invalid direction.

Despite this we believe we have reached the conclusion of this project with an intuition for what direction we would need to take to resolve some of these issues - both in terms of tasks that were outside the time scope for the project and areas of further research that would be necessary. Likewise, our awareness of more ideal structures for this type of code has developed significantly. Whilst this project's final codebase is somewhat haphazard in appearance, this is reflective of and as a consequence of a relatively rapid incrementation of identified needs and approaches through the course of the project.

Team

Broadly the teamwork split our task into two overlapping focuses. Matthew would handle the initial research and feedback on this to Luke in order for a common plan of approach to be formed. Simultaneously, Luke would begin from the game framework implementation and then evolve this to fit the needs of the common approach once it had been identified.

From there we would share discussions over how to approach each section of the development of the components and how they should interact so that we would have a collaborative overview of the whole system, but the primary responsibility for implementing each component would be divided by concentration. Adaptation of the existing framework for Matthew would be biased toward identifying possibilities within the research and interpreting / adapting the mathematics; for Luke this would be biased toward creating and adapting the scripts. Whiteboard discussion and diagramming played a significant role in our communication (see Process and Learning section below).

- Matthew Duddington

Researching the different machine learning and, more specifically the focus on, neural networks was a highly engaging yet very daunting process. Whilst we had previously scratch the surface of what this field held, the sheer quantity of research and competing viewpoints was staggering. Being able to draw a line through the process to define what was relevant and in-scope, was a considerable early challenge - and one which would define many of our later decisions as we progressed.

A significant challenge to us was that, in order to test our understanding an ability to apply the techniques we discussed, we first needed a base game. Completing the functioning prototype, with enough points of contact to extract the necessary inputs and return outputs, proved to be a non trivial joint task for us. This meant that for a considerable period of the planning process, we had to trust our instincts about what would work and what we were sufficiently uncertain of that we chose to find another approach.

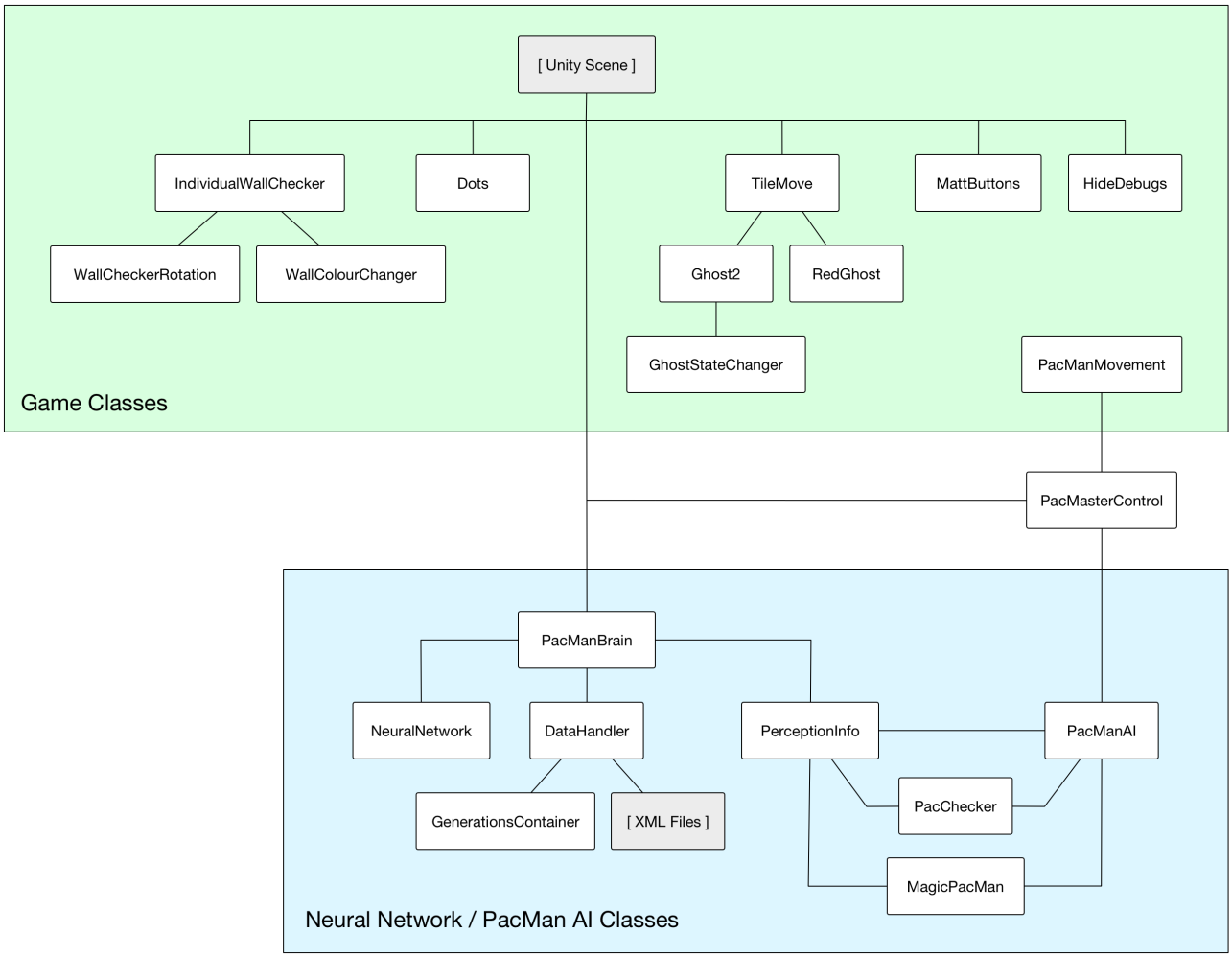
- Luke Sanderson

Namco's Pac-Man (1980) is a fantastic example of excellent AI that follows a very simple set of rules to create a dynamic playfield for the player. We decided that this would be an excellent base to build our neural networking system on as there is a smorgasbord of articles and interviews on the game's creation and its lively AI. Since every ghost follows a certain 'pattern', we challenged ourselves to create an AI that could play the game perfectly. Plus, it's Pac-Man, who doesn't like Pac-Man?

My main contribution was building the game as faithfully as possible to the original, with necessary adjustments to allow our AI to learn and grow. I took care in using the same sprites and sound from the original – even the resolution of the game is the same. The ghosts all have the same AI as the original too, targeting the player in various ways and even retaining the time-based chase-flee system found in the original.

Since Pac-Man uses a tile-based system in order to check for collisions and walls, I thought it would be good for our AI to learn in the same manner. Each moveable object (essentially just Pac-Man and the ghosts) essentially moves precisely between tiles, meaning we can check for other objects at a position rather than using Unity's built in collision system – although I still used that for collisions with dots and power pellets.

High Level Structure



General overview of class structure and relatedness

Class: ***PacManBrain.cs***
Main implementer: Matthew Duddington

The controller for setting up and maintaining the neural network at the start and in-between application loads. The brain is the main front facing interface for Luke's game classes to talk to, alongside the PerceptionInfo class. The brain triggers requests to and gathers information from PerceptionInfo in order to populate the network with the required inputs.

Class: ***NeuralNetwork.cs***
Main implementer: Matthew Duddington

This is code adapted from James McCaffrey's Build 2013 presentation on implementing a neural network for the Iris data set in Visual Studio. I have modified certain functionality to fit our project's differing network structure and discarded some functions that were obsolete to our needs. Likewise, I introduced new measures such as the NextBiggestIndex to cope with smoothing out invalid outputs during the supervised training use, and the alternative methods for instantiating and setting up new networks during scene reloading and the importing / exporting of data to the XML data handler.

Class: ***DataHandler.cs***
Main implementer: Matthew Duddington

Handles the serialisation and de-serialisation of XML data when saving and loading learned neural network values. Storing this data outside the game instance enables the values to be preserved between application instances and also scene re-loads (scene re-loading is the method Luke utilised for resetting the game map). The PacManBrain calls the AddGenerationToSavedData and LoadGenerationData passing in the data path, based on the brain mode and user ID, and returning or receiving the relevant NetworkData instance.

Contains the 'NetworkData' structure class which defines the XML elements and attributes.

Class: ***GenerationsContainer.cs***
Main implementer: Matthew Duddington

Structure class for storing and accessing through the DataHandler functions the instances of NetworkData that are currently in memory.

Class: ***PerceptionInfo.cs***
Main implementer: Matthew Duddington

Contains the results of the observations being made by the PacChecker and IndividualWallChecker classes. Also performs some calculations and re-formatting of the data for use in the inputs of the neural network.

Class: ***PacChecker.cs***
Main implementers: Luke Sanderson / Matthew Duddington

This class finds and stores useful information to be ultimately used in the neural networking's deciding process – everything from the closest (and most desirable) dot's position, the other ghost positions, potential free spaces to move to and Pac-Man's local wall-checkers are calculated here.

Class: ***PacManAI.cs***
Main implementer: Luke Sanderson

Functions in a similar manner to the ghost AI class (Ghost2.cs).

Class: ***MagicPacMan.cs***
Main implementer: Luke Sanderson

Functions in a similar manner to the PacManAI class, with some notable differences. Essentially a visual result of the fastest route to the destination tile.

This 'magic' Pac-Man has no collisions and moves at the fastest rate possible to the destination tile. It keeps a record of how many tiles it passes through to reach it, and passes this value back to PacManAI, which is rewarded/punished depending on how many tiles it takes to reach the destination itself.

Class: ***PacManMovement.cs***
Main implementer: Luke Sanderson

Handles all movement, states and score when the player is playing. Use the arrow keys to move around the maze and collect dots.

It moves by checking if the tile ahead of it is clear, then by moving towards it. This means that if Pac-Man is moving right and the player is holding down, Pac-Man will only move down once a space is free below him – a common technique in the original game.

Class: ***PacMasterControl.cs***
Main implementer: Luke Sanderson

The player can press Space at any time to change between player and AI.

Class: ***TileMove.cs***
Main implementer: Luke Sanderson

Does the brunt of the work for every moving object in the game. Checks potential collisions, valid moves and finally moves towards valid tiles.

Class: ***Ghost2.cs***
Main implementer: Luke Sanderson

Main class for each ghost's movement and state.

Each ghost attempts to move to its 'Target Tile'. It moves forward until it hits an intersection (seen in the video as a white block at possible intersections) and determines which route distance is the shortest to its destination (other than backwards), then moves in that direction.

The red ghost primarily goes straight for Pac-Man's position, the pink ghost aims 2 blocks in front of Pac-Man, the orange ghost aims for Pac-Man unless it's < 8 spaces closer, where it retreats to the bottom-left corner. The blue ghost is the trickiest of all – get the red ghost's position, get a Vector2 towards Pac-Man's position, and then double it. All of these AI function the same as the original game, except I removed a bug from the original where the pink ghost would aim up AND left of Pac-Man, rather than just up, when Pac-Man is facing up.

The ghosts follow these rules when they are in a 'Chase' state – when they are in a 'Corner' state, they flee to their respective corner. GhostStateChanger.cs manages the ghosts' changing states.

When Pac-Man eats a power pellet, they become scared and pick a random destination to move towards at every intersection.

Class: ***GhostStateChanger.cs***
Main implementer: Luke Sanderson

A timer that changes the ghost states every few seconds.

Class: ***IndividualWallChecker.cs***
Main implementer: Luke Sanderson

Script used for each of the 'checkers' around Pac-Man. Returns true if hitting a wall.

Class: ***WallCheckerRotation.cs***
Main implementer: Luke Sanderson

Since Pac-Man can rotate when moving in different directions, this script ensures that the checkers stay the same rotation.

Class: ***WallColourChanger.cs***
Main implementer: Luke Sanderson

Cosmetic script for changing the colours of the walls on the fly.

Class: ***HideDebugs.cs***
Main implementer: Luke Sanderson

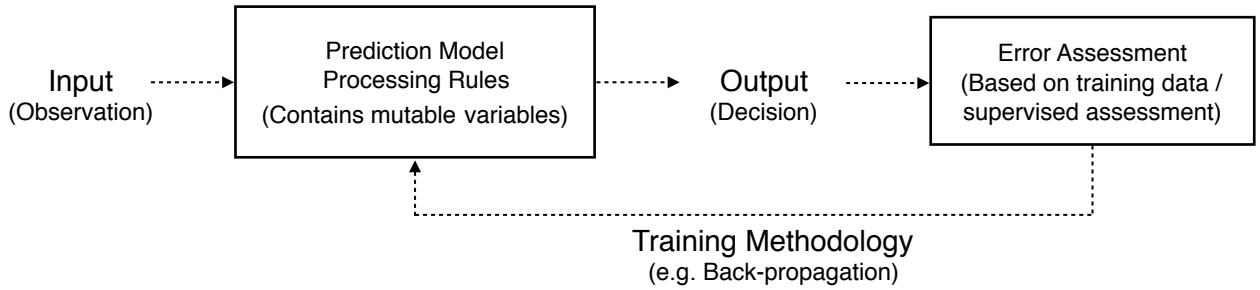
Shows/hides various debugging visuals, namely the various 'destinations' for Pac-Man and the ghosts.

Process and Learning

Matthew Duddington

Background Understanding of Neural Networks

At a base level neural networks make decisions by receiving input, processing that through the nodal layers of a prediction model to generate outputs. Furthermore, if still in training, those outputs are checked for a degree of error and the model is refined through an training methodology based on the error assessment.



The prediction model and its connections, represent the approach to the data flow that the model's architect thinks will lead to the desired output answer. Sometimes known as *hyper-parameters*, the different parts of this approach (*Rohrer, B. 2016 & 2017; McCaffrey, J. 2013*) will usually include:

- Input normalising methodology
- Number of layers / *depth* of the network
- Type of those layers (e.g. *summation*, *pooling*, *convolution*)
- Number of nodes in each layer
- Choice of activation function (e.g. *logistic sigmoid*, *softmax*)
- Layer type-specific parameters (e.g. *stride*, *window size*)

During training, mutable variables (*weights* and *biases*) within each layer (*nodes*, *functions* and *connections*) of the prediction model are modified by the system to refine itself as it seeks a sufficiently ideal solution.

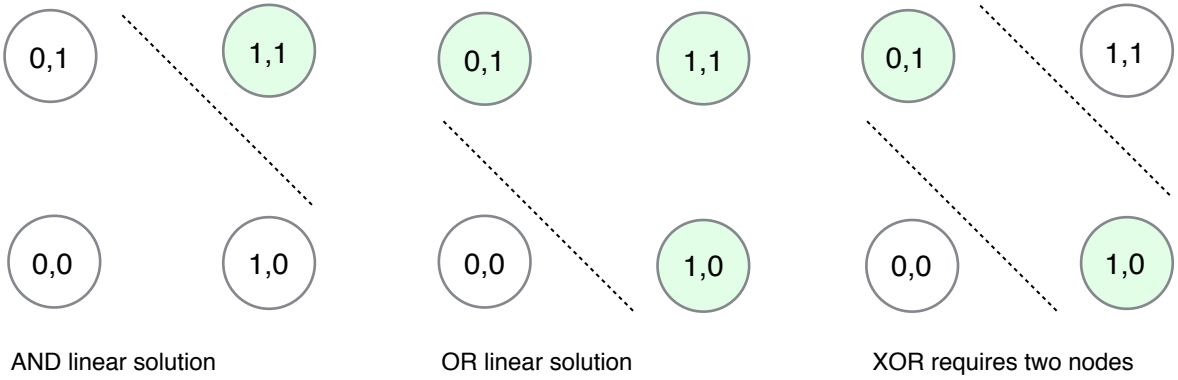
With *supervised* trained networks, labelled data with known, desired answers or instances of exemplar user input are presented to the network. By comparing the network's predictions with the known answers, based on the same inputs, an error rate can be calculated. The training methodology can then refine the pattern of weights, so that the network's structure iteratively settles towards predicting answers that agree with the training data. Furthermore, once a network has a sufficiently reliable pattern for a particular type of input, it can then be used to predict or *classify* new, unlabelled instances of that type of input.

When training a network, further hyper-parameters are required to control the behaviour of the chosen training methodology. One of the most synonymously associated with neural networks is *back-propagation*, where the hyper-parameters used are the *learning rate* and *momentum*. This was the method we utilised.

Learning rate is required in order to preserve traces of the data that has already been tested. Without this effect to moderate / dampen the change of values, each new training example inputted would simply cause the feedback to entirely replace the existing model with the best fit for the most recent data. Likewise, by moderating the rate of change, the impact is reduced for noisy outlier and erroneous data (*Rashid, T. 2016*).

New variable = Old variable - (Learning rate x Error difference)

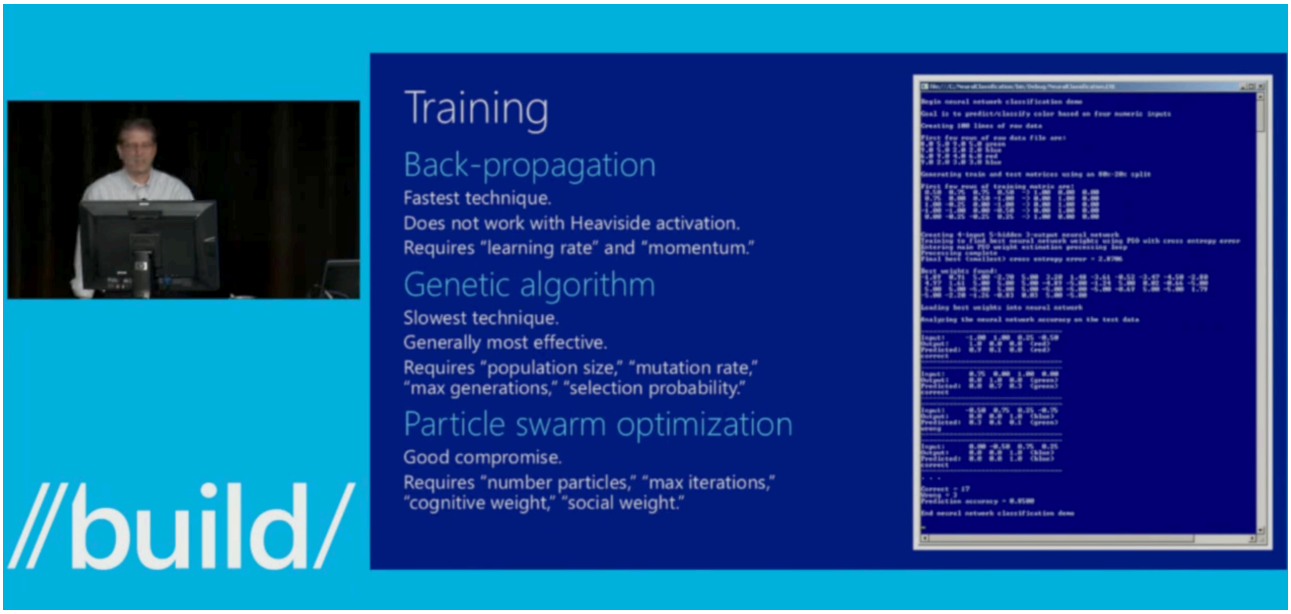
When a network receives multiple inputs for comparison, a single, linear classifier is able to determine the outcome of basic logical statements AND and OR. Yet, in order to answer more complex questions, the ability to ask advanced logical statements, such as XOR, is required. As a single linear line cannot adequately split the data, advanced logic requires combining the predictions from more than one node. (*Rashid, T. 2016*)



As more nodes are stacked the network will begin to resemble an abstraction of the biological structure for neurones. The various input signals are summed, processed through a function and then a resulting signal is fired along each output connection, scaled based on that connections weight and the node's bias. Commonly, those weights and biases are the mutable element of the network and, thus, where the result of training is consolidated.

Assuming training data is readily available, determining what the error rate is, is a relatively trivial task - simply calculating the difference between the expected result and prediction outputs from the network. However, the amount of error attributable to each internal variable, and thus how much to modify those variables individually, would be almost impossible in any network of useful complexity. This is one of the areas in which the most significant progress has been made in neural network machine learning recently.

During his presentation at Build, James McCaffrey (*2013*) describes three main training methods available:



Training Methodologies (McCaffrey, J. 2013)

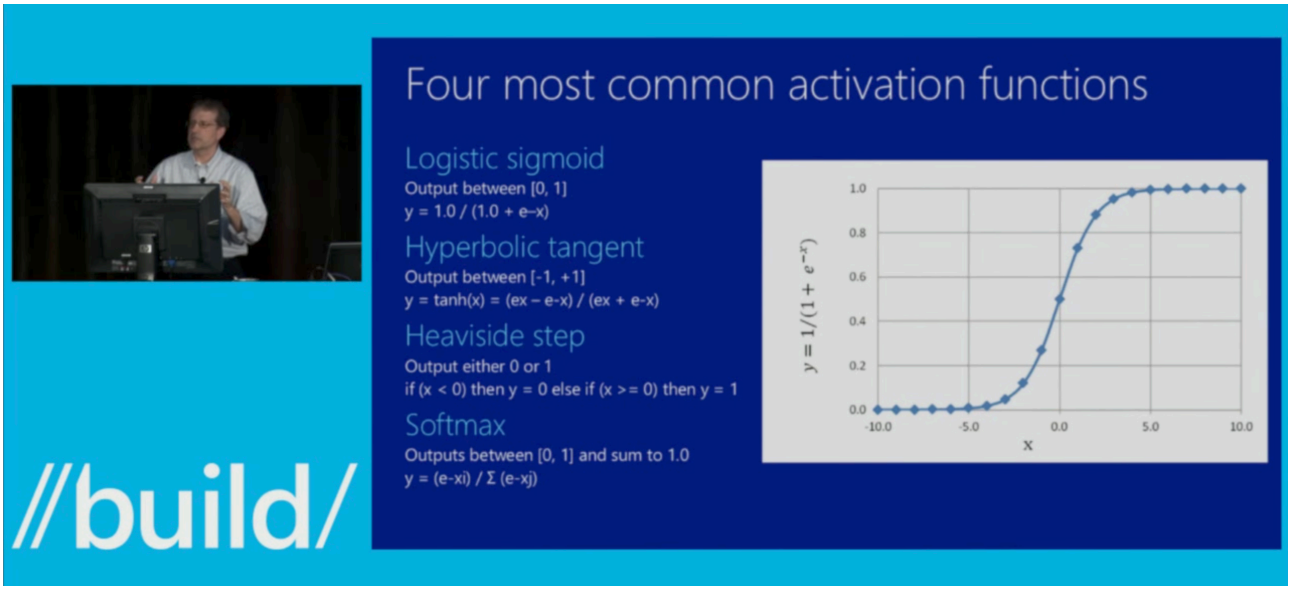
As I will discuss later, a genetic algorithm would have been the most optimistic candidate for our project, however, back propagation was the method eventually used.

The significant development mentioned, was an insight to do with back propagation, that calculating the proportion of the error to assign to any specific weight could be derived through considering the gradient of the error at each step along the chain back to the original value being determined for adjustment. Essentially, by determining which direction to 'nudge' the weight in order to move down the gradient at each step, the whole chain can be updated piece by piece. While this doesn't give the 'perfect' result that performing the much more computationally costly (and frankly almost scarily complex looking equation) would, the benefits mean that 'good enough' is perfectly fine in every case - so long as the design of the network is considered in relation to this (*Rashid, T. 2016*).

One caveat to the training aspect of neural networks is the concept of *over fitting*. If a network is trained too heavily with a particular type of data, then the patterns the network will learn to look for become too tailored to the specific shape of the training data and it passes a point where it would recognise similar but not before seen examples that should otherwise be recognised as valid. For example, a fruit recognition network, if shown too many oranges, might stop recognising apples as valid, because it has weighted itself too much towards orange-like features and has weakened connection weights that represented its more general picture of fruit.

This leads to the conclusion that a healthy network should be trained with a variety of sources that match a spectrum of the type of data attempting to be identified and that it should be trained to the point of good stable performance and then allowed to settle. With care given to the balance of fresh data entering the system that has been flagged to be used for further training.

Similarly to the variety of training types, there is a similar variety of *activation functions*. These are an element applied a the transition from the hidden layer to the outputs / succeeding hidden layer. Unlike the training methods, the activation function is not likely to be an architectural choice by preference, but rather one driven by the nature of the data being processed and in what form it needs to continue on through the network's flow. Commonly Logical Sigmoid is the assumed default, but this is not always the most appropriate (*McCaffrey, J. 2013*).



Common Activation Functions (McCaffrey, J. 2013)

A phrase that frequently goes hand in hand with recent popular discussion of neural networks is the concept of Deep Neural Networks. *Depth* commonly appears to refer to the number of hidden layers utilised within the network. Why this is useful can, perhaps, be most easily understood in the domain of convolution networks, because they deal with spacial / visual information. In these cases each layer can be thought of as looking at different levels of detail on the *features* within the input data (*Rohrer, B. 2016*). The initial layers are more literal and focused on the minutia, whereas deeper layers have successively combined earlier features into more humanly 'recognisable' elements.

In an English language processor the layers might be:
Letters —> Sounds / Short Words —> Words —> Long words / Word combos —> Short phrases —>
Long phrases / Short sentences —> Sentences —> Paragraphs —> Passages …

Similarly, if analysing for faces:
Contrast / Colour differences —> Edges / Directions / Density —> Eye / Nose / Mouth / Wrinkle —>
A face / Portion of face —> Feminine face / Old face / Happy face —> Alex's face …

One point to note with depth is that more layers does not necessarily mean that the network will perform any better. In fact this is just another case of a delicate variable that must be balanced to find the 'perfect storm' which results in the desired output. McCaffrey (*2013*) points towards *Covers Theorem* as evidence that, in fact, any problem that can be solved with multiple hidden layers can ultimately be solved with just one, so long as there is the right number of neurones.

Irrespective of the myriad of techniques and structures available, one of the most important parts of architecting a neural network is determining how the input data should be represented. Consideration has to be paid to finding a way that the network can process. For example, with a convolution network searching for numbers or letters in a photo, the normalised, pixel luminance values of the scanned image could form the input. (*Rohrer, B. 2016*) The granularity of the inputs direction has a direct and very clear impact on the number of inputs. For example, at a 1-to-1 ratio, a 28px square scan would require 784 input nodes.

This data has to be properly normalised (in the general sense), so that different data is within proportion to each other. Because, in practice, all nodes are potentially connected to all other nodes, representing £1000 against and age of 35 and a boolean binary value as the abstracts 1000, 35 and 1, the 1000 would could easily drown out the signal from the binary value and even the age. Therefore, a better approach is to say 1.0, 3.5 (or 0.35) and 1.0. Which crushes the values down to a more comparable relation.

Our Approach

Probably the single greatest challenge we faced during the development of our neural network, was finding a way to determine how right or wrong an one particular decision that the network made was.

With temporally discreet data, such as a photo of some text, there is a single clear decision that can be made as to whether a letter has been correctly identified. However, the problem we found ourselves facing was that, the decision our PacMan AI had to make would only reveal itself as a good or poor decision many frames later - potentially after more decisions would have been made. How then to split apart the influence of each individual decision? We had encountered the *Credit Assignment Problem* (*Karpathy, A. 2016*).

We reached four possible options:

1) Have the network predict a wider reaching temporal decision to allow for a delay in further decision making and giving an opportunity for a meaningful outcome to be inferred.
2) Record training data live, through user input, and consider this as the frame by frame 'truth'.
3) Forgo a neural network entirely, in favour of *Q-Learning*, which can stack multiple decisions together.
4) Use a genetic evolution method of training to evaluate performance based on runs.

Originally, our intention was to follow the 4th option and utilise a *NEAT* architecture (NeuroEvolution of Augmenting Topologies), which would enable us to effectively ignore the temporal element (*Stanley, K. & Miikkukainen, R. 2002*). Initially, this appeared like a promising lead, after finding a few sources who had previously applied this to PacMan (*Schrum, J. 2014; DeLisle, R. K. 2017*) as well as a Unity adapted NEAT library (*Jallov, D. 2014*).

NEAT enables the architect to remove two significant complications from their decision making process:

- The internal prediction model structure.
  Determining the number of nodes, the type of their layer and order they appear is time consuming and errs towards overcomplexity. The genetic approach is to begin with the simplest structures (essentially just inputs connected to outputs) and gradually introduce additional nodes and layers via evolution. Only those modifications that prove to have a high *fitness* value survive to be *cloned* or become a *crossover parent* for the following generation.
- Determining the error rate of individual decisions made by the network.
  While a functionally similar value has to be calculated, the *fitness*, crucially this value is determined at the conclusion of the whole simulation rather than on a moment to moment basis. This means a measure of overall success determines the value of the network's DNA rather than any one connection within it.

While these two points fit our problem space very well, the scale of the architectures for the genetic approach became difficult for us to unpick, working as we were from the base level of understanding that we held at that stage. Likewise, in order to get a good result from the NEAT process, we foresaw a need to be able to test running many iterative steps through the game from an early stage. As simply setting up the PacMan game world with the necessary capabilities was proving to be a non-trivial task, our solution for the network needed to be setup without the possibility of in depth testing before a late stage.

At the time, the prudent option appeared to be to pursue one of our other proposed options. Retrospectively, this was perhaps the wrong call, as the level of success of the practical component would have almost certainly been significantly better than the result we now have at the end of this project - however, this assumes we would have eventually understood and been able to implement a NEAT system.

Moving to Q-learning looked like the next best option, however, by removing the neural network aspect it felt like this might defeat the point of what we had set out to achieve.

Q-learning works by storing [*state, action*] experiences, attaching a reward (or punishment) based on that action and a record of the state reached as a result. (s, a, r, s') (*state*, *action*, *reward*, *new-state*)

Normally this can, more or less, perfectly locate optimal solutions. However, because pure Q-learning requires all possible states to be mapped out, it is unsuitable for large state domains - such as a full map of PacMan, where just one dot being present or absent would define an entirely different state to a map that is otherwise identical.

This can be partially solved through *Approximate Q-Learning*, where the perfect representation states are replaced by recording a collection of individual feature states (*Abbeel, P. 2014*).

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Q-learning with linear Q-functions:

transition $= (s, a, r, s')$

difference $= \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s,a)$

$Q(s,a) \leftarrow Q(s,a) + \alpha$ [difference] Exact Q's

$w_i \leftarrow w_i + \alpha$ [difference] $f_i(s,a)$ Approximate Q's

Evolution of Q-Learning to Approximate Q-Learning functions. (Abbeel, P. 2014)

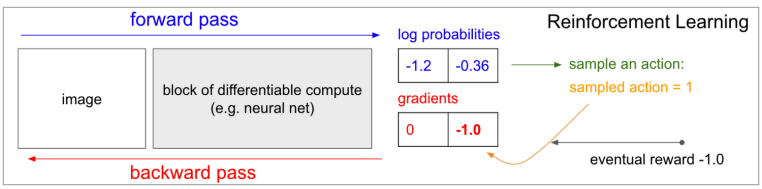
The big advantage this method presents is that while the validity of actions are measured on a per event basis, the actual realisation of their validity can be stacked as a chain, because each 'new-state' is simply the outcome of the following action - state pair. Therefore, when the game eventually reaches a conclusion, whether good or bad. This can be propagated back through the whole system.

Better than even approximate Q-learning, is the use of reinforcement learning and policy gradients, which perform similarly, in that they are able to distribute the result of a later point in time back amongst earlier stages (*Karpathy, A. 2016*).

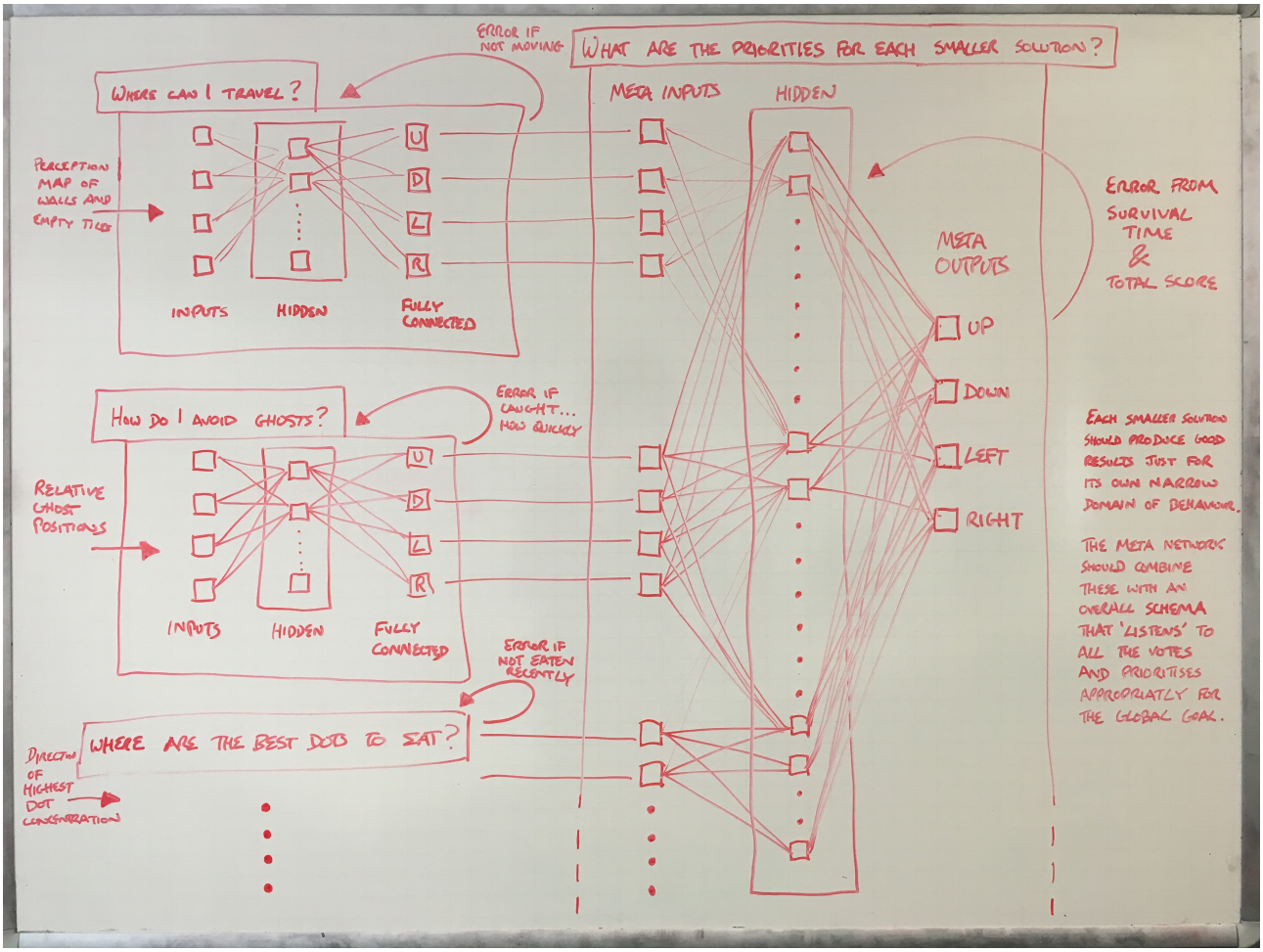
While the general concept of this methodology was appealing, I we were unable to sufficiently parse the mechanics of how it would work for our own game. Similarly, I was unable to find a sufficient number of example sources that explained it in a way that we could adapt reliably without being able to apply it immediately to the game environment. So unfortunately, again, this method had to be passed over as we sought a simpler implementation option.

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}$$

Deriving Policy Gradients and RL. (Karpathy, A. 2016)



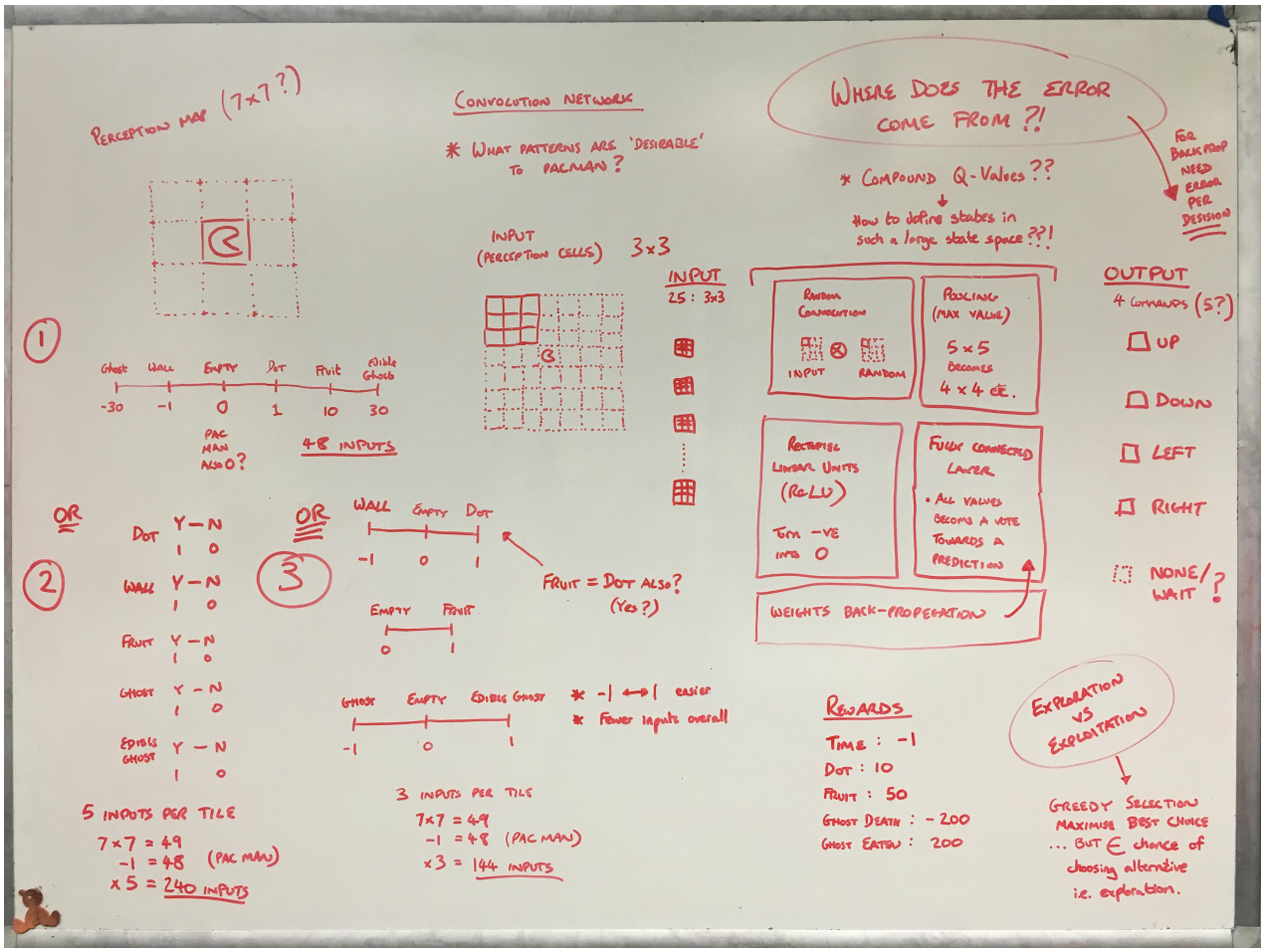
During one of our whiteboard discussions regarding the complexity of the different aspects of gameplay that the error would need to encapsulate, concurrently, both Luke and I had a similar insight: what if we were to break down the decision making process into smaller sub questions, each with their own, considerably more simplified error determination. This lead to our, only briefly entertained, 'Compound Decision Architecture':

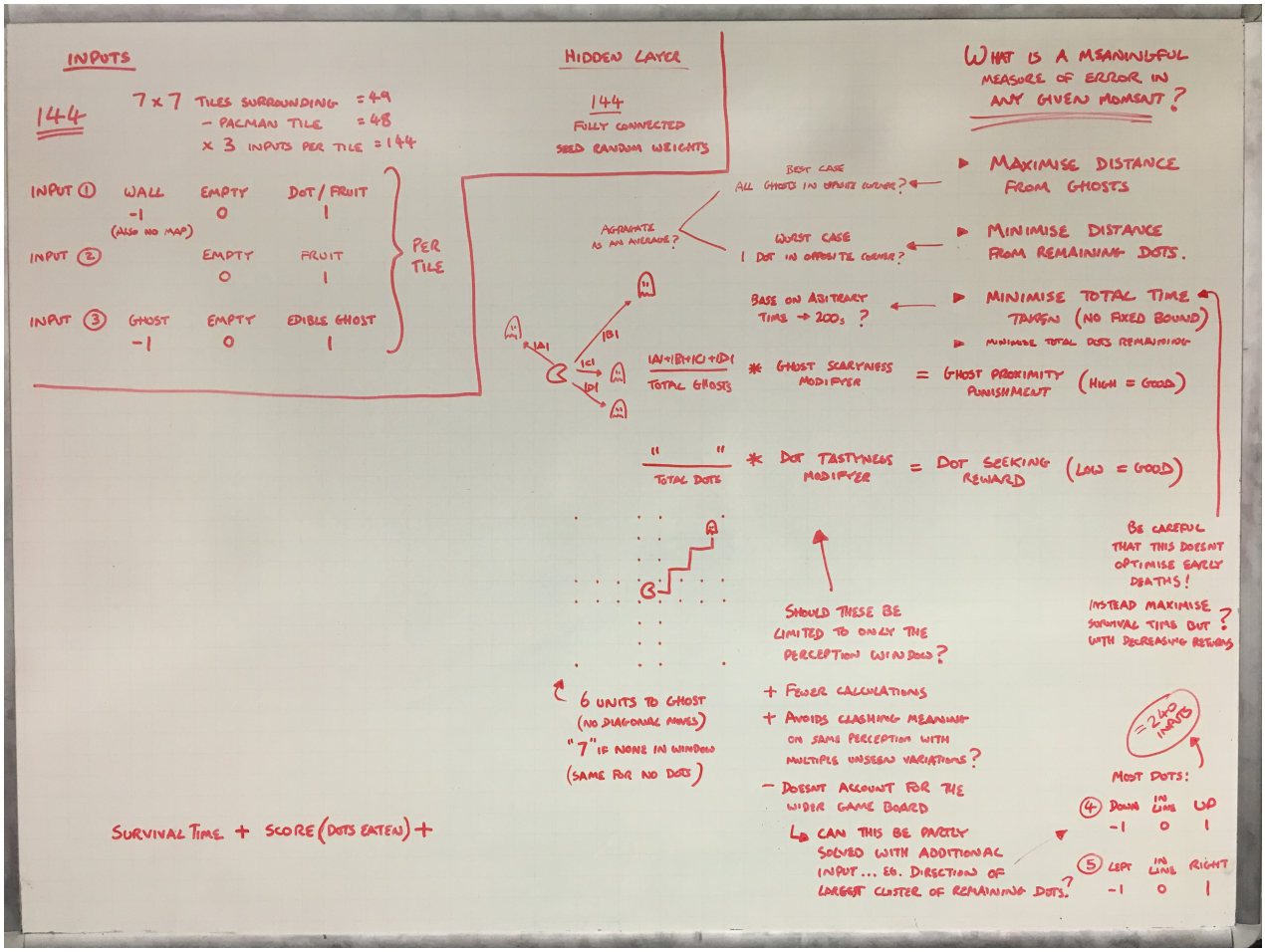


The concept sounded good in theory: smaller and simpler networks all contributing a small amount to a final decision later. However, there were too many concerns that more in depth consideration raised. While each individual mini network would be simpler, the practical complexity of the compound, meta network would be several orders of magnitude higher. It instantly would introduce lot of additional computation, particularly with the assumption that our first attempts at defining a class structure would be sub-optimal. With that would come both a longer tick time and also far more places that we would need to search while debugging if something went wrong.

Furthermore, it wouldn't ultimately remove the main problem; we would still need to find ways of evaluating each of these questions in real time, even if they were individually easier to measure in isolation.

Eventually we settled on attempting to apply both techniques 1 and 2 from the earlier list. While, we knew these would likely lead to sub-optimal behaviours from the AI, we felt both were more within our achievable scope. The following page displays some of the resulting thoughts from this period of discussion.

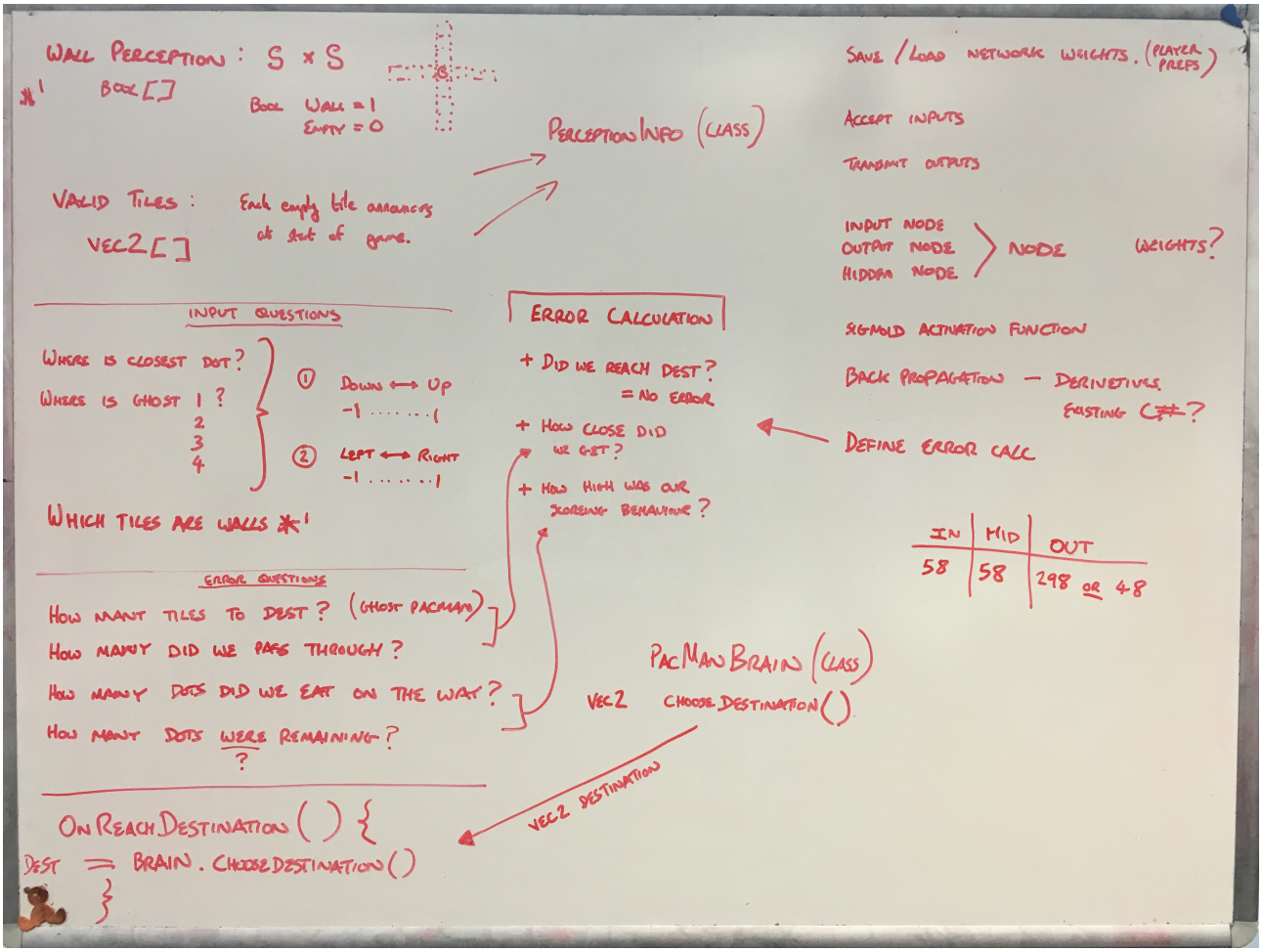

PERCEPTION MAP (7x7?)
CONVOLUTION NETWORK
* WHAT PATTERNS ARE 'DESIRABLE' TO PACMAN?
WHERE DOES THE ERROR COME FROM?!
* COMPOUND Q-VALUES??
How to define states in such a large state space??!
FOR BACKPROP NEED ERROR PER DECISION
INPUT (PERCEPTION CELLS) 3x3
INPUT 25 : 3x3
Ghost Wall Empty Dot Fruit Edible Ghost
-30 -1 0 1 10 30
PAC MAN ALSO?
48 INPUTS
OR
Dot Y-N 1 0
Wall Y-N 1 0
Fruit Y-N 1 0
Ghost Y-N 1 0
Edible Ghost Y-N 1 0
5 INPUTS PER TILE
7x7 = 49
-1 = 48 (PAC MAN)
x5 = 240 INPUTS
OR
Wall Empty Dot
-1 0 1
FRUIT = DOT ALSO? (YES?)
Empty Fruit
0 1
Ghost Empty Edible Ghost
-1 0 1
* -1 ↔ 1 easier
* Fewer inputs overall
3 INPUTS PER TILE
7x7 = 49
-1 = 48 (PAC MAN)
x3 = 144 INPUTS
Random Convolution
INPUT RANDOM
Pooling (Max Value)
5x5 Becomes 4x4 ish.
Rectified Linear Units (ReLU)
Turn -ve into 0
Fully connected Layer
• All values become a vote towards a prediction
WEIGHTS BACK-PROPEGATION
OUTPUT
4 Commands (5?)
UP
DOWN
LEFT
RIGHT
NONE/WAIT?
REWARDS
Time : -1
Dot : 10
Fruit : 50
Ghost Death : -200
Ghost Eaten : 200
EXPLORATION VS EXPLOITATION
GREEDY SELECTION MAXIMISE BEST CHOICE ... BUT ε chance of choosing alternative i.e. exploration.


INPUTS
144
7x7 TILES SURROUNDING = 49
- PACMAN TILE = 48
x 3 INPUTS PER TILE = 144
INPUT ① WALL EMPTY DOT / FRUIT
-1 (ALSO NO MAP) 0 1
INPUT ② EMPTY FRUIT
0 1
INPUT ③ GHOST EMPTY EDIBLE GHOST
-1 0 1
PER TILE
HIDDEN LAYER
144
FULLY CONNECTED
SEED RANDOM WEIGHTS
WHAT IS A MEANINGFUL MEASURE OF ERROR IN ANY GIVEN MOMENT?
BEST CASE ALL GHOSTS IN OPPOSITE CORNER?
WORST CASE 1 DOT IN OPPOSITE CORNER?
AGGREGATE AS AN AVERAGE?
▸ MAXIMISE DISTANCE FROM GHOSTS
▸ MINIMISE DISTANCE FROM REMAINING DOTS.
BASE ON ARBITRARY TIME → 200s ?
▸ MINIMISE TOTAL TIME TAKEN (NO FIXED BOUND)
▸ MINIMISE TOTAL DOTS REMAINING
|a|
|b|
|c|
|d|
VARIABILITY WITH TOTAL GHOSTS
* GHOST SCARYNESS MODIFYER = GHOST PROXIMITY PUNISHMENT (HIGH = GOOD)
" " TOTAL DOTS
* DOT TASTYNESS MODIFYER = DOT SEEKING REWARD (LOW = GOOD)
BE CAREFUL THAT THIS DOESNT OPTIMISE EARLY DEATHS!
INSTEAD MAXIMISE SURVIVAL TIME BUT WITH DECREASING RETURNS
SHOULD THESE BE LIMITED TO ONLY THE PERCEPTION WINDOW?
6 UNITS TO GHOST (NO DIAGONAL MOVES)
"7" IF NONE IN WINDOW (SAME FOR NO DOTS)
+ FEWER CALCULATIONS
+ AVOIDS CLASHING MEANING ON SAME PERCEPTION WITH MULTIPLE UNSEEN VARIATIONS?
- DOESNT ACCOUNT FOR THE WIDER GAME BOARD
↳ CAN THIS BE PARTLY SOLVED WITH ADDITIONAL INPUT ... EG. DIRECTION OF LARGEST CLUSTER OF REMAINING DOTS?
= 2 NO INPUTS
MOST DOTS:
④ DOWN IN LINE UP
-1 0 1
⑤ LEFT IN LINE RIGHT
-1 0 1
SURVIVAL TIME + SCORE (DOTS EATEN) +

The eventual design incorporated a local window view, that we borrowed from the convolution concept. This perceives the walls immediately surrounding PacMan but ignores the rest of the map, in the same way a naive player focuses on the immediate decision making. Each wall tile in the 48 surrounding PacMan is input as a 1 and each traversable empty tile as a 1.

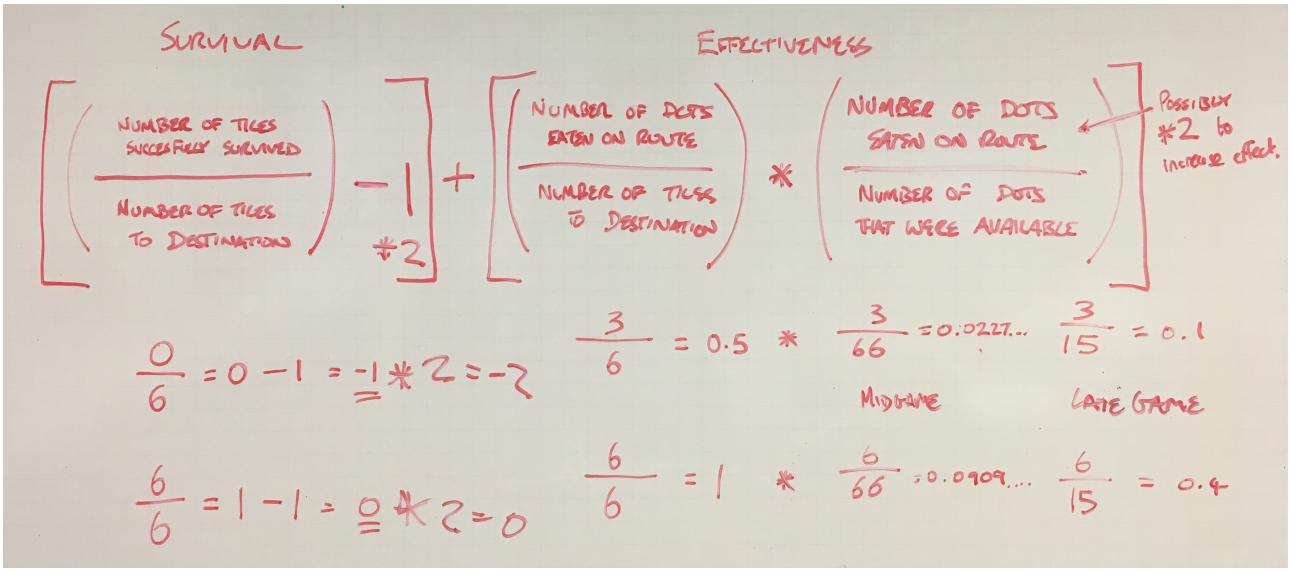
We also brought in more global knowledge of the map with features such as those found in approximate Q-learning. We provide a two input representation of the distance and position of each ghost and the closest food dot to PacMan. These representations take a value between -1 and 1. Down and Left are represented at the negative end and Up and Right at the positive end. The point delivered is as a proportion of the total maximum possible map width or high difference if PacMan and the object were at extreme opposite edges.

The output would be a destination marker within that same window. The time it took for PacMan to reach that marker via the pathfinding AI, would be the temporal unit over which the decision would be evaluated. This was intended to give a short enough period to have a close tie to the decision itself, yet be long enough that something meaningful could be inferred about the decision.



Example Rejected Perception Criteria:

- Which junction are we at on the map?
- Which sector in relation to PacMan has the most dots left? L, R, U or D
- How many squares away is each colour of ghost?
- Where are the power fruit in relation to PacMan?
- Are we under the effect of a power fruit?



The error ratio was calculated when reaching the destination, i.e. the resolution of the decision, through a mixture of evaluating:

- The survival rate.
  In the case that PacMan is killed by a ghost on the way to the destination - we wanted to measure how close / bad the decision was to travel in that direction.

- The scoring effectiveness.
  How good of a path did PacMan choose to take in terms of progression towards completing the map - how many dots were eaten as a total of the distance traveled via that route. This was also modified by an awareness of how many dots were available in total to that the significance of an effective path was weighted more towards end game decision making, where it would be less likely to happen upon a good collection of dots by chance.

Of our two network derived AI methods, this one performs the required behaviour to some small degree. However, there are a significant number of poor behaviours exhibited in its results, based on its functionality after the amount of training we were able to apply.

Frequently PacMan will get stuck in a corner - partly due to the pathfinding system, but largely due to the weights becoming over rewarded for the initial exploration direction. Because there is a lack of punishment for waisting energy traversing the same path or, conversely, rewarding encouragement for exploring new paths and gathering dots early on, our training error function performs inadequately to train significant behaviour at this time.

The second approach, of gathering data from a human player, takes the same perception values, but assesses the error of the outputs by the binary input from the player. A score of 1 is located to the output associated with the same key the player is pressing and a 0 for all other outputs. While the player is pressing no keys, the weights are not updated.

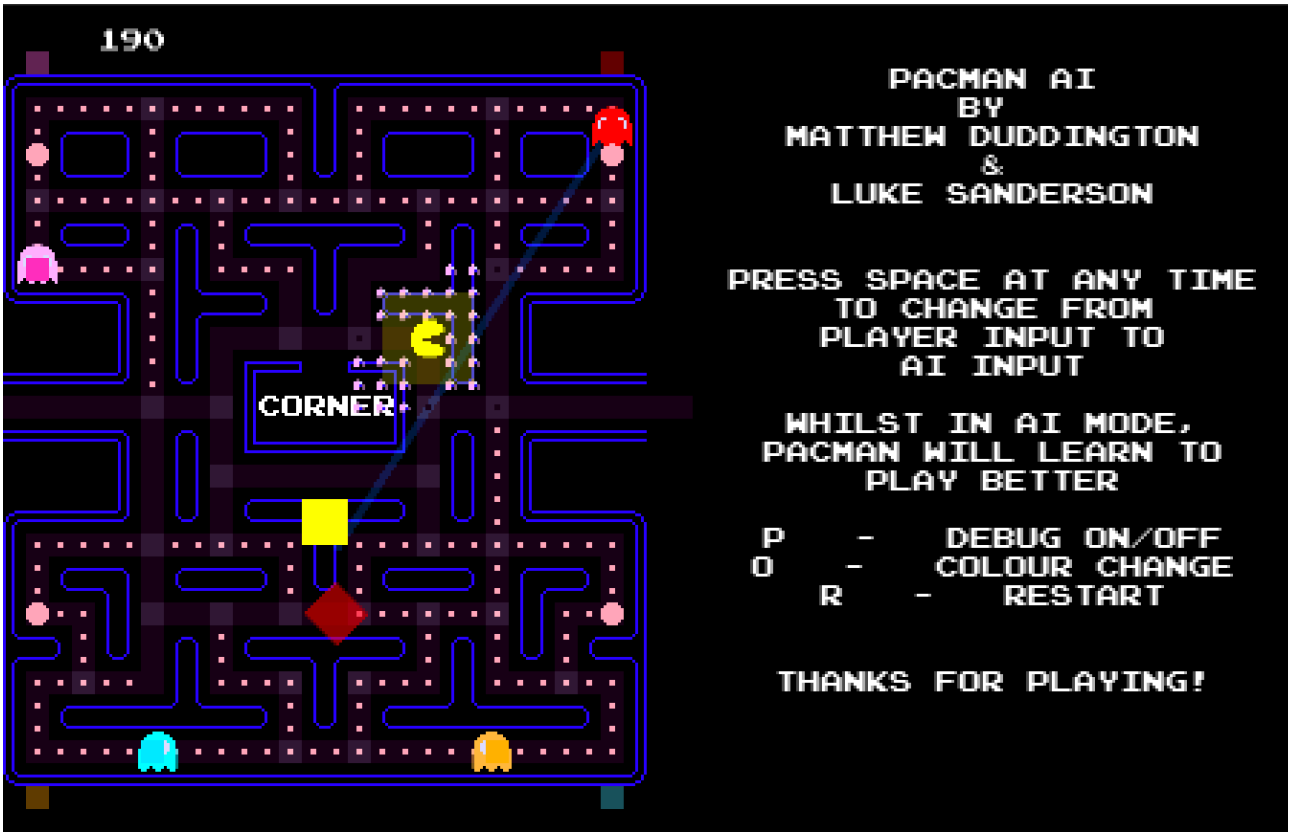
This lead to the appearance of more variably distributed weights in the saved data, however, the system for applying that data back into the game was flawed and the rate of training the data meant that in almost all instances the network would pick its favoured direction and continually try to perform it.

The final piece of the puzzle was storing saved data between training runs and instances of the application. This was achieved through serialising the data into an XML format.

Luke Sanderson

Approach to Integrating with the Neural Network

After implementing the ghost's AI it became clear that we could take a similar approach to how the AI version of Pac-Man would move – namely setting a destination and attempting to reach it as fast as possible. We considered the potential moves a player might want to make at a certain time, and have the AI learn from this experience – for example, if the player is being chased by a ghost, he values his own life over collecting dots. To this end, we had a class that constantly checked many variables in the game, from remaining dots to each ghost's proximity, and passed this information along to the PacManBrain class, where it can assign the possibility of going to certain tiles in different states.



Bibliography

Abbeel, P. (2014) *Approximate Q-Learning*. From 'CS 188: Artificial Intelligence', *Lecture 11: Reinforcement Learning II* (slides by Klein, D and Abbeel, P.). University of California, Berkeley. (accessed May 2017) Available online: youtu.be/yNeSFbE1jdY?t=31m2s

Amir, Ofra, Ece Kamar, Andrey Kolobov, and Barbara J. Grosz. 2016. *Interactive teaching strategies for agent training*. In Proceedings of the Twenty-Fifth international joint conference on Artificial Intelligence (IJCAI'16), New York, NY, July 9-15, 2016: 804-811. (accessed April 2017) Available online: nrs.harvard.edu/urn-3:HUL.InstRepos:29399785

Bling, S. (2015) *MarI/O - Machine Learning for Video Games*. (accessed March 2017) Available online: youtube.com/watch?v=qv6UVOQ0F44

Bling, S. (2015b) *MarI/O source code*. (accessed March 2017) Available online: pastebin.com/ZZmSNaHX

Bodenlos, S. et al. (2013) *Learning how to play Pac-Man*. (accessed April 2017) Available online: ml.informatik.uni-freiburg.de/research/games/pacman

Chiarandini, M. (2011) *Games and Adversarial Search*. (accessed April 2017) Available online: imada.sdu.dk/~marco/DM828/Slides/dm828-lec17.pdf

Dunn, F. and Parberry, I. (2011) *3D Math Primer for Graphics and Game Development*. 2nd ed. CRC Press.

Gow, J. (2017) *The Future of Game AI*. (accessed March 2017) Available online [login restricted]: learn.gold.ac.uk/mod/resource/view.php?id=414838

Jallov, D. (2014) *UnityNEAT: Port of SharpNEAT*. (accessed April 2017) Available online: github.com/lordjesus/UnityNEAT

Karpathy, A. (2016) *Deep Reinforcement Learning: Pong from Pixels*. (accessed May 2017) Available online: karpathy.github.io/2016/05/31/rl/

DeLisle, R. (2017) *AIPac*. (accessed April 2017) Available online: sourceforge.net/projects/aipac/

Lee, S. et. al. (2016) *Predicting Resource Location in Game Maps Using Deep Convolutional Neural Networks*. (accessed April 2017) Available online: julian.togelius.com/Lee2016Predicting.pdf

Manning, D. (2013) *Ms. Pac-Man Tutorial 1: Ghost Behaviour*. (accessed April 2017) Available online: youtube.com/watch?v=sQK7PmR8kpQ

Mazur, M. (2015) *A Step by Step Backpropagation Example*. (accessed May 2017) Available online: mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example

McCaffrey, J. (2013) *Developing neural networks using Visual Studio*. (accessed April 2017) Available online: youtube.com/watch?v=-zT1Zi_ukSk

Mnih, V. et al. (2013) *Playing Atari with Deep Reinforcement Learning*. (accessed April 2017) Available online: arxiv.org/pdf/1312.5602.pdf

Pittman, J. (2017). *The Pac-Man Dossier*. Gamasutra.com (accessed May 2017) Available online: gamasutra.com/view/feature/132330/the_pacman_dossier.php

Rashid, T. (2016) *A Gentle Introduction to Neural Networks and making your own with Python*. (accessed April 2017) Available online: youtube.com/watch?v=2sevic5Vy4E

Raval, S. (2017) *Intro to Deep Learning*. (accessed April 2017) Available online: youtube.com/playlist?list=PL2-dafEMk2A7YdKv4XfKpfbTH5z6rEEj3

Rohrer, B. (2016) *How Convolutional Neural Networks work*. (accessed April 2017) Available online: youtube.com/watch?v=FmpDIaiMIeA

Rohrer, B. (2017) *How Deep Neural Networks Work*. (accessed April 2017) Available online: youtube.com/watch?v=ILsA4nyG7I0

Schrum, J. (2014) *Multimodal Behaviour in One Life Ms. Pac-Man*. (accessed April 2017) Available online: nn.cs.utexas.edu/?blended-pm

Schrum, J. (2014b) *MM-NEAT*. (accessed April 2017) Available online: nn.cs.utexas.edu/?mm-neat

Stanley, K. & Miikkukainen, R. (2002) *Evolving Neural Networks through Augmenting Topologies*. (accessed March 2017) Available online: nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

Xenfinity (2016) *How To Save Your Game In Unity with XML*. (accessed May 2017) Available online: youtube.com/watch?v=Y8Di-Q6qpU4