Modern AI for Games - Report on Individual Assignment

Björn Þór Jónsson - bjrr@itu.dk

***Abstract*— Following is a report on using AI methods to control an agent in the game of Ms PacMan. The discussed methods are: Genetic Algorithm with a Behaviour Tree; Artificial Neural Network with Backpropagation; and Monte Carlo Tree Search.**

I. Introduction

THIS report outlines the implementation and results gained from the use of Artificial Intelligence methods to control Ms PacMan in a provided test bed framework [1]. The three methods chosen to implement are based on an Evolutionary Algorithm, a Neural Network with Backpropagation, and Monte Carlo Tree Search.

The following sections describe each implementation. The methods were implemented in Java and the sections will contain references to the Java package names containing the relevant implementation classes.

Plots of the generated data - performance measures an experiments - were made with simple R scripts that can be found in the `plots` directory within the supplied implementation project.

II. Genetic Algorithm with a Behaviour Tree

A. Behaviour Tree with prioritized tactics

A Behaviour Tree framework was implemented - `bjrr.pacman.behaviourtree.ai.*` - and a tree constructed with a root priority selector for three tactics, based on the strategies in the `StarterPacMan` procedural controller supplied with the Ms Pac-Man vs Ghosts League framework. The strategies are, in order of priority:

- Get away from any non-edible ghost that is in close proximity
- Go after the nearest edible ghost
- Go to the nearest pill/power pill

A controller constructing this tree can be seen in `bjrr.pacman.behaviourtree .BehaviourTreePacManController` and an initial sketch of the tree layout can be seen in figure 1.

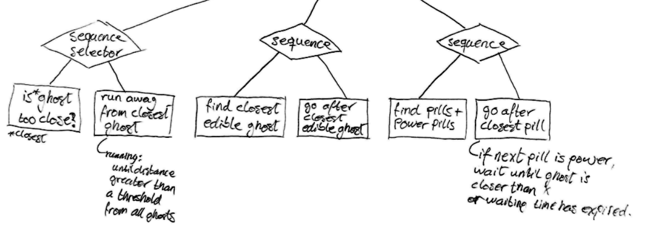

priority selector
sequence selector
sequence
sequence
is* ghost too close?
*closest
run away from closest ghost
running: until distance greater than a threshold from all ghosts
find closest edible ghost
go after closest edible ghost
find pills + Power pills
go after closest pill
if next pill is power, wait until ghost is closer than x or waiting time has expired.

Fig. 1. Sketch of a Behaviour Tree prioritizing the control of Ms PacMan

B. Genetic Evolution of tactical attribute values

Within the tasks of this constructed behaviour tree are three attributes governing tactical decisions by the Ms PacMan agent. Those attributes are stored in a centralized memory - a black board class `bjrr.pacman.behaviourtree.PacManBlackboard` - accessible by all tasks of the tree. The attributes are used in the tasks `IsGhostTooClose` and `GoAfterClosestAvailablePill` and are:

- `MIN_GHOST_DISTANCE` in `IsGhostTooClose` to determine when to run away from a non-edible ghost.
- `MAX_POWER_PILL_DISTANCE` in `GoAfterClosestAvailablePill` used to define how close to a power pill Ms PacMan can get, before she turns away from it.
- `POWER_PILL_WALK_AWAY_DISTANCE` also in `GoAfterClosestAvailablePill` declaring for how long Ms PacMan maintains her chosen course away from a power pill.

The combined use of those attributes results in the tactic where Ms PacMan never eats a power pill free-willingly, but may run away from ghosts towards a power pill, luring them to close proximity when she eats one. By using this tactic there will initially be a short distance to edible ghosts after eating a power pill.

A Genetic Algorithm was used to obtain optimal values for those attributes - `bjrr.pacman.ga.PacManGeneticAlgorithm` - evolving them in successive generations of genes - `bjrr.pacman.ga.PacManGeneForBehaviourTree`.

It was chosen to work with a population of size 500. The Genetic Algorithm's fitness function consists of the score results from one run of a Ms PacMan controller without visuals - `bjrr.pacman.ga .GeneticAlgorithmPacManController`. That controller constructs a Behaviour Tree and populates its central memory with attribute values from the gene being evaluated for fitness.

Several options were considered for the reproduction of the next generation. From the example applications discussed in section 2.4 of [2] the Evolutionary Algorithm in section 2.4.1 (The 8-Queens Problem) was chosen as a model, where the best 2 of random 5 individuals / genes from the current population are chosen to be parents, and their children replace the two worst individuals in the population.

Offspring reproduction from the two chosen parents is done by giving each child an attribute value from either parent with equal chance. The gene of each child is then mutated by giving each attribute a new random value with a

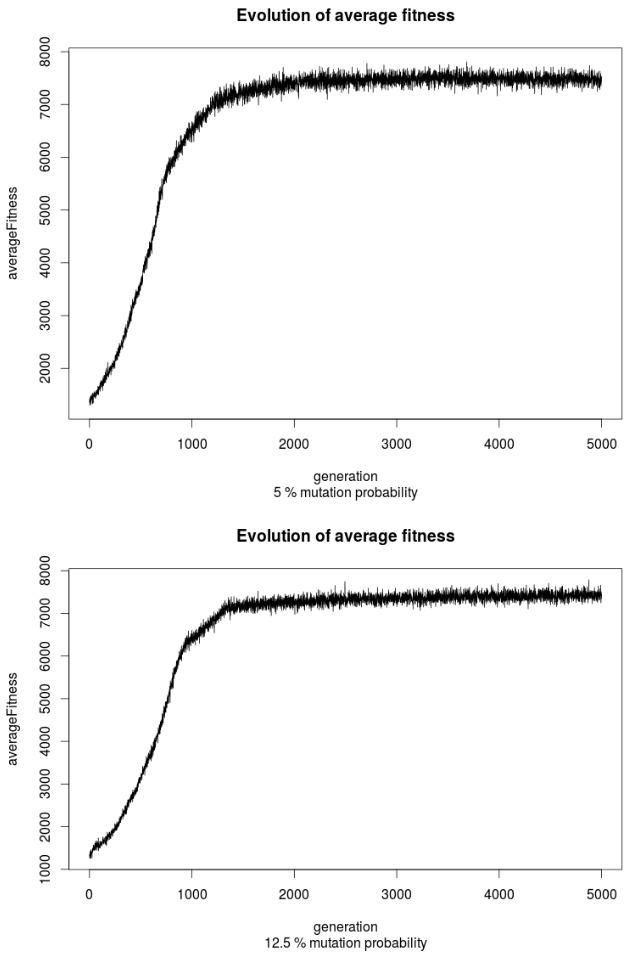


Fig. 2. Evolution of fitness through generations of attribute value genes for a Ms PacMan controller. Each plot is from an evolution with different probabilities of mutation, 5% and 12.5%, and apparently the mutation probabilities do not change much in the progress of the evolution.

predefined probability. The random values are chosen within an upper bound of 100, as most values in the files in data/distance directory of the Ms PacMan framework were observed to be lower than 100. Mutation probabilities of .125 and .05 were tried and the evolution with each can be seen in figure 2.

C. *Performance*

While running the Genetic Algorithm, evolution was set to continue while the difference between the best and worst fitness, within the current generation, was over the score threshold of 100, and while the all time best fitness from all generations and that of this generation was above the same threshold. This goal was not reached after several thousands of generations and the evolution was arbitrarily stopped when the fitness values were observed to be similar between generations.

When the Genetic Algorithm had produced two thousand generations, the best fitness obtained from each generation fluctuated in the range of Ms PacMan scores between 7000 and 8000, as can be seen in figure 2.

III. Neural Network with Backpropagation

A. *Inputs*

In an attempt to explore the benefits of providing different kinds of inputs for training a Neural Network with Backpropagation, new attributes were added to the DataTuple class that was provided in a data recording lab code package - `bjrr.pacman.ann.dataRecording.DataTuple`. Those new attributes are the booleans: `lastDirectionSame`, indicating whether PacMan is about to choose the same move as he did on the previous game tick; `isClosestGhostMovingInSameDirectionAsPacman`, `isClosestGhostEdible`; and the integers: `distanceToClosestPill`, `distanceToClosestPowerPill`, and `distanceToClosestGhost`. The attributes are normalized with the normalization methods provided in the DataTuple class. A sketch of those inputs can be seen in figure 3.

Two input sets were tried for training the Neural Network. The first input set contains only values collected in the original version of the data collection package. The second input set contains the new attributes, outlined above, in a mix with some of the previously provided attributes.

Input set 1 consists of:

- Ms PacMan current level
- Ms PacMan position
- Number of pills left
- Number of Power pills left
- Whether each of the ghosts is edible
- Distances to each of the ghosts

Input set 2 consists of:

- Ms PacMan current level
- Ms PacMan position
- Number of pills left
- Number of Power pills left
- Whether Ms PacMan last direction is same as the chosen one
- Distance to closest pill
- Distance to closest power pill
- Distance to closest ghost
- Whether the closest ghost is edible
- Whether the closest ghost is moving in the same direction as Ms PacMan

B. *Training*

The two input sets were tried for training the Neural Network, and for each of them, two different numbers of hidden neurons were tried. For each of those four combinations, two different learning rate configurations were tried, accounting for a total of eight training sessions. Each training session lasted for more than five hundred thousand iterations / epochs and in figure 5 plots are shown of training epochs up to 550.000 against the rate of error in the output prediction of the network.

For training, a two-layer neural network was constructed (as defined in [3], section 9.2.1), with one hidden layer layer. The number of neurons for the hidden layer was chosen by considering rules of thumb presented in [4], chapter 5:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

The latter two rules of thumb were implemented for the training sessions.

For the last rule, which states *that the number hidden neurons should be less than twice the size of the input layer*, the multiplier 1.7 was used against the number of input neurons to obtain the hidden neuron count. This resulted in a slightly larger number of neurons (20 versus 12) and in somewhat less error rate in the predictions output by the network, as can be seen in figure 5.

Both a fixed learning rate of 0.1, and a learning rate of 1/t (where t is increased on every 100th epoch, as suggested in the labs), were tested for training the neural network. The fixed learning rate gave a substantially better result in network prediction, as can be seen in figure 5 when comparing the plots with thin straight lines (decreasing learning rate) with the pots depicting fluctuating lines (fixed learning rate); the decreasing learning rate results in prediction errors of around and above 25% and the fixed learning rate results in prediction errors below 20%.

Difference in prediction performance between the two input sets is barely observable, though input set 1, which does not contain the custom attributes, performs slightly better, as can be seen when comparing the second and last plots in figure 5.

C. *Performance*

The training data consisted of data recordings from five games of Ms PacMan played by the author of this report, which happens to be quite bad at playing the game. So a controller using a neural network trained with this data could not be expected to perform well.

The actual result of using a trained network in a Ms PacMan controller is even worse than expected, where Ms PacMan starts by choosing sensible moves, but soon gets stuck in a corner without choosing a move leading the agent onto a path out of that corner. The reason for that behaviour has not been investigated further and so is yet unknown.

An embarrassing average score of 255 was obtained from running a Ms PacMan game simulation 100 times with one of the trained networks, and the average score could go as low as 71 for other networks.

D. *Implementation*

The neural network implementation - `bjrr.pacman.ann.feedforwardbackpropagation` - is straight forward with a Java class for the network, and one class each for the layers and neurons. Connections are also defined in their own Java class, whose instances are shared between connecting neurons. A Backpropagation class was created to perform the training iterations. A sketch of the code structure can be seen in figure 4.

The Java class `bjrr.pacman.ann.FeedforwardBackpropagationPacmanTraining` was created to set up the network for training, computing the number of hidden input neurons and managing the learning rate during the training iterations. A controller for trying out the trained networks can be found in `bjrr.pacman.ann.FeedforwardBackpropagationPacmancontroller`.

Before starting this straight forward implementation, I was exposed to a more sophisticated implementation when reading chapters two and five of [4], which introduce the concept of using matrices to represent weights and bias for each neuron, and how matrix arithmetic can be used to update the weights during the backpropagation of errors. In addition to specifying a learning rate, the implementation discussed in chapter five of that book uses the concept of momentum, which scales the learning from the previous iteration before applying it to the current iteration; it “specifies how much of an effect the previous training iteration will have on the current iteration”[4]. It was interesting to learn about that concept, where "Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum"[5].

Supplied with the book is an implementation of a neural network and training with backpropagation (along with other methods of training) [6] and my first inclination was to base my own implementation on that code. Then I realized it would be hard to not just copy paste the relevant code and even more time consuming than doing my own thing from scratch. So even though it was interesting to learn about the use of matrix arithmetic to update the network weights and biases, I decided to skip using matrices and the simple implementation discussed above.

One class I did copy straight from that book code though, and that is `SerialzieObject`, for saving and loading the trained network.

IV. Monte Carlo Tree Search

A. *Implementation*

In preparation for using MCTS in a Ms PacMan controller, the partial tutorial implementation provided in the labs was completed - `bjrr.tutorial.mcts`.

The tutorial implementation was then adapted for use in a Ms PacMan controller - `bjrr.pacman.mcts`. Specifically the methods in `bjrr.pacman.mcts.UCT` for selection (`TreePolicy`), expansion (`Expand`), playouts / simulation (`DefaultPolicy` or `GeneticBehaviourTreePolicy`) and backpropagation (`Backpropagate`) were adapted to use the Ms Pac-Man vs Ghosts League framework methods to obtain the

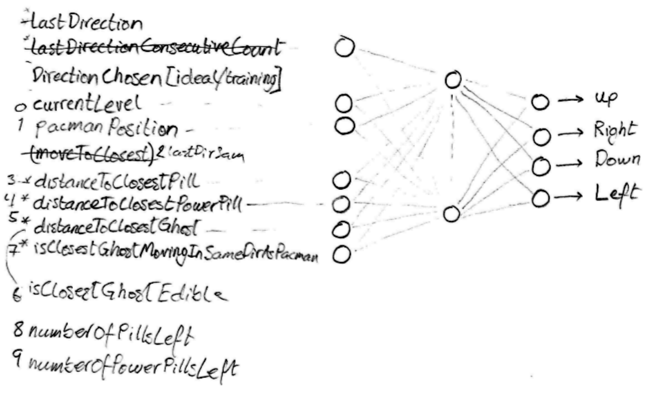


Fig. 3. Sketch of Neural Network inputs and outputs.

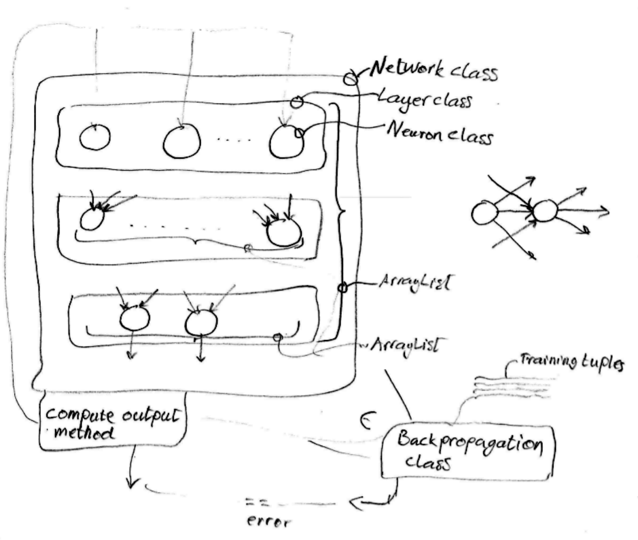


Fig. 4. Sketch of main components in a straightforward Artificial Neural Network implementation.

possible moves at any given state during expansion and to retrieve the (non-normalized) score during simulation for backpropagation.

Instead of having a predefined number of iterations for building the MCTS tree, it was decided to continue iterating while the current time was within the time due provided by the Ms PacMan framework in the getMove call. To avoid beginning a new iteration that would take longer than the remaining available time, the average time taken by previous iterations was calculated. As each iteration proved to take less than one millisecond, that average was deemed to be insufficient for ensuring the MCTS would finish in time to return a move. So one millisecond was simply added to the current time when determining whether the next iteration would finish in time:

```
while( System.currentTimeMillis() + 1 <
timeDue ) {
```

Using those conditions an iteration count from over 100 to little under 700 was obtained for each move request on a five year old Ubuntu desktop with `Intel(R) Core(TM)2`

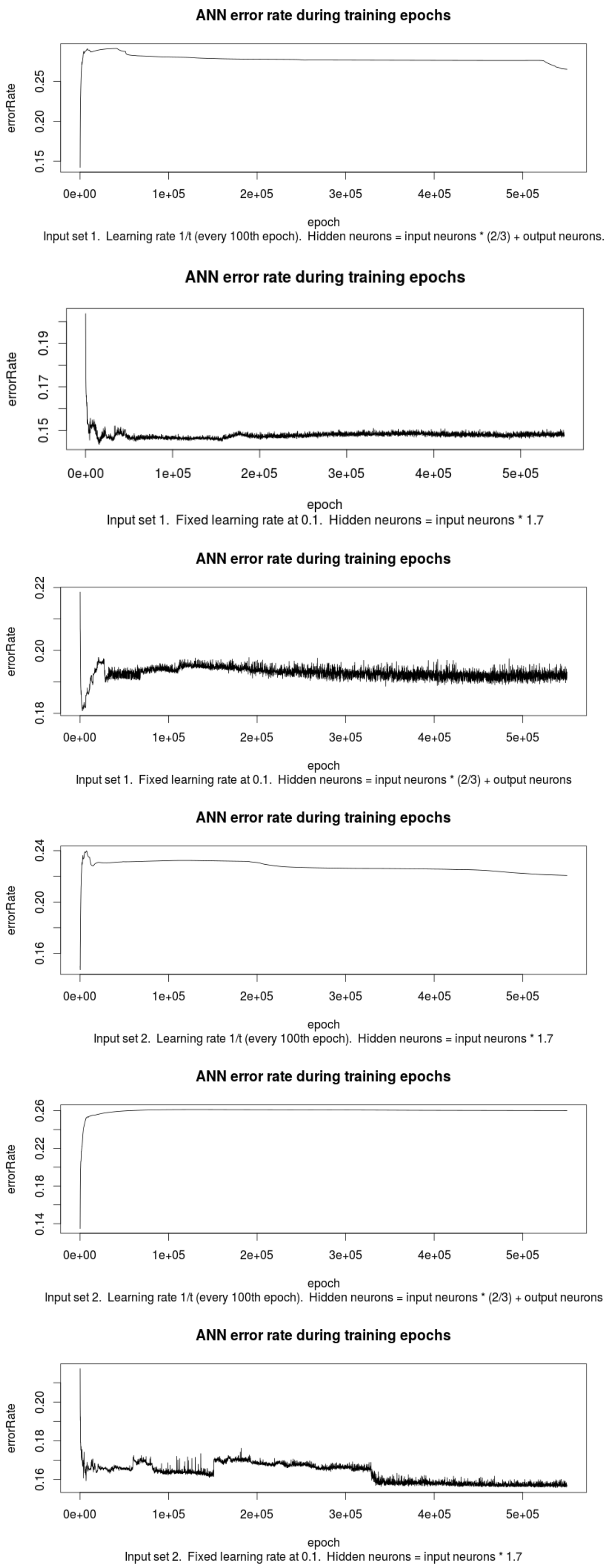


Fig. 5. Error rates, in the predictions of trained neural networks, against 550.000 training iterations. Fixed and decreasing learning rates were tried, with the former giving better results. A higher number of hidden neurons, chosen from two used rules of thumb, gives a slightly better result.

`Quad CPU @ 2.40GHz`. It was interesting to observe that using debug log statements, in the MCTS iteration loop, with `System.out.println` severely skewed those time calculations; no more than one iteration was achieved with a log statement there, resulting in the controller always returning NEUTRAL moves for Ms PacMan.

Instead of applying specific conditions to "determine the short- and long-term safety and reward of a selected path" as is done in the playout strategy of [7], an attempt was made to play the PacMan game until either of it's normal end states are reached - "when either Pac-Man loses all lives, or the 16th level is cleared"[7] - and instead of manually determining PacMan's score according to the games subgoals mentioned in the [7], we'll simply get the score at the current state from the PacMan game framework (`Game.getScore()`). As this may not be feasible within the time frame of 40 milliseconds, for example when the playout strategy is good enough to keep PacMan alive for a long time, we'll instead introduce a limit on the number of iterations of the playout, or simulation.

What limit on iteration count gives the best result was to be obtained by measuring the score from one game play with the `MCTSPacmanController` with a range of iteration limits, and choosing the limit that gave the highest average score. An average of more than one play result for each limit would have been preferable, but as the MCTS controller takes a long time, even in a simulation without any visuals, where it uses all the available time for each tick / game advancement, time only allowed one game play per iteration limit.

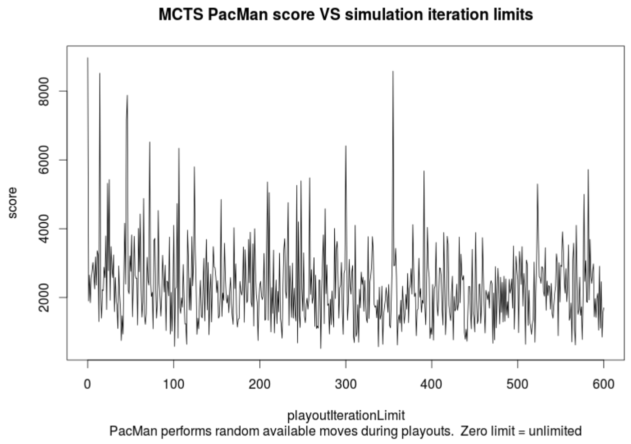
B. Performance

Two different playout strategies (DefaultPolicy) were be tried:

- A strategy where random moves of those available at any given time for PacMan will be tried. Ghost moves will be obtained from the StarterGhosts controller.
- Or a strategy where the behaviour tree controller previously discussed, with genetically evolved attribute values for controlling strategic moves, will be used to obtain PacMan moves at any given time and the ghosts will be controlled as in the previous strategy.

Resulting scores from controlling Ms PacMan with a MCTS, with varying limits on the number of MCTS iterations, can be seen in figure 6. The iteration limits shown range from 0 to 600, where 0 is interpreted as a no limit. The plot is quite noisy, with the scores fluctuating across a wide range between different iteration limits; the curves would probably be much smoother when an average would be taken from several played games for each iteration limit, but as mentioned before, only one game is played for each limit due to time constraints.

With the playout strategy of random movements for Ms PacMan, having no limit (0) on the number of MCTS tree building iterations gives the best result, with one comparable spike between the limits of 300 and 400. A playout strategy using the genetically evolved controller for moves, generally



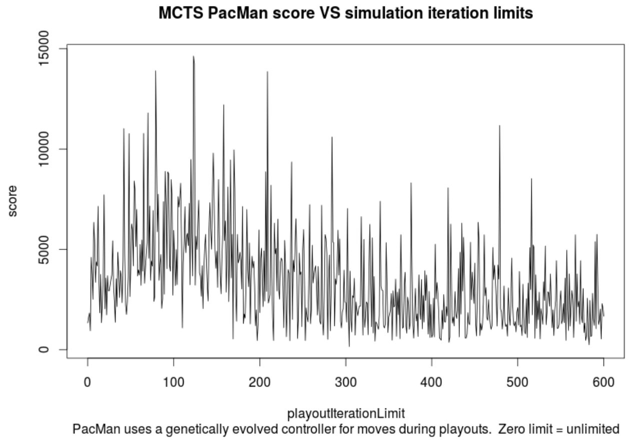


Fig. 6. Limits on MCTS iterations, used for getting moves in a Ms PacMan controller, plotted against the resulting scores.

gives higher scores for MCTS iteration limits around 100, as can be seen on the second plot of figure 6.

V. Conclusions

It has been interesting to try out different Artificial Intelligence techniques for controlling Ms PacMan vs the Ghosts League. The simplest method of genetically evolving values for three strategic attributes, used in Behaviour Tree tasks, seemingly gives the consistently best results of those obtained from the implementations discussed in this report.

References

[1] Ms pac-man vs ghosts league framework. [Online]. Available: http://www.pacman-vs-ghosts.net/software

[2] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. springer, 2003.

[3] J. Han, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[4] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[5] Gradient descent with momentum backpropagation. [Online]. Available: http://www.mathworks.se/help/nnet/ref/traingdm.html

[6] Source code from jeff heaton's books. [Online]. Available: https://code.google.com/p/jeffheaton-bookcode/

[7] T. Pepels and M. H. Winands, "Enhancements for monte-carlo tree search in ms pac-man," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 265–272.