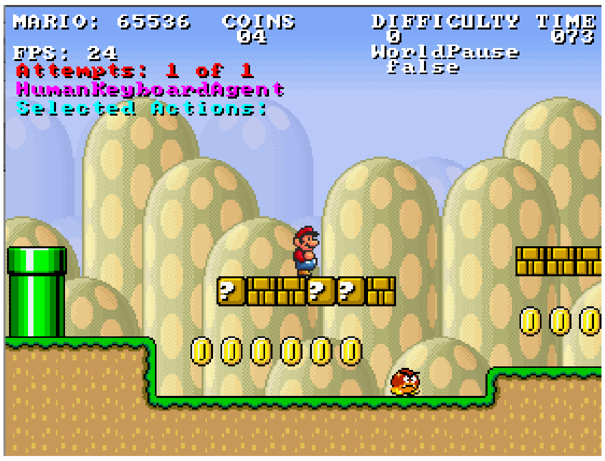
GOLDSMITHS UNIVERSITY OF LONDON

Evolution of a neural network using genetic algorithm

Coursework 1

MSc in Computer Games and Entertainment

Artificial Intelligence for games



Andrea Castegnaro

Abstract

Nowadays one of the most challenging topic in AI development is building intelligent agent where intelligent stands for the common held assumption of mimic a player behaviour. Studies on different techniques showed that the best result for such a purpose can be achieved by using an artificial neural network (ANN) with the weights adjusted by an evolutionary algorithm. In this case a genetic algorithm (GA) will be used. In this development we will try to give just the basic information to the agent based on the current game screen field of view. Each game will be then scored with general parameters and the evolution will then consist on maximising the fitness function built this way. A framework called Mario AI benchmark will be used for the project. It is Java based and has been object of several competitions during the past years.

Contents

List of Figures

Abbreviations

ANN	**A**rtificial **N**eural **N**etwork
GN	**G**enetic **A**gorithm

Chapter 1

Mario AI Benchmark

1.1 Introduction

The Mario AI benchmark is based on Markus Persson s Infinite Mario Bros, which is a public domain clone of Nintendo s classic platform game Super Mario Bros[1].
It is a Java based clone of the famous Super Mario World game and a series of API that made it very suitable to be used for building learning controls. The framework has been also the object of a series of competition over the past years[2]. Important features to be mentioned are also the *Automatic level generation* making it infinite playable, the possibility of changing difficult and customizing the level in terms of dead paths, gaps, enemies type, coins, etc.

1.2 API high-level description

In order to provide the necessary information to the agent playing Mario a representation of the level environment is provide by the use of a set of API built in inside the framework.
In general the information is divided into level scene information and enemies information[3]. Both of them are matrix with a customizable dimension and detail of information (called *zScene* within the framework). The level scene information matrix contains all the different elements that appear in the screen (the screen is divided into cells which are mapped in the level scene matrix) in a concrete moment during the game, representing

the different elements with numbers. The enemy information matrix works in the same way, representing the different enemies as numbers.
For this project a vision grid 8x8 has been used to detect the information from the environment. Fig.1.1 just shows how a centered matrix is (in the image is 4x7).

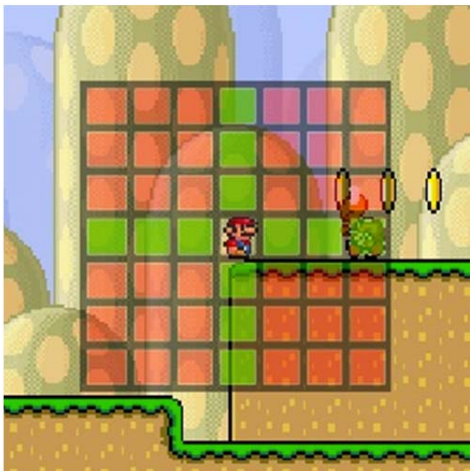


FIGURE 1.1: Mario AI environment representation

The grid size has been chosen by having in mind the the idea of building a human like agent and so no overloading of information is provided to it[4].
In particular the information that we used from the enemy are the following:

- Enemy in front/back/above/below/up-right/down-right: calculates a weighted distance from goomba/troopa enemy. 0,33 in case it is 3 cells distant from Mario, 0.66 in case of 2 cells and 1.0 in case the enemy is on the next cell from Mario.
- Obstacle in front/above/below: calculates a weighted distance from an obstacle. The weight is calculated as in the previous case in terms of distant cells.
- Height of obstacle in front: in case of having an obstacle in front the height is weighted. If 3 cells tall returns 1.0, if 2 returns 0.66, if 1 returns 0.33.
- Mario is able to Jump : returns 1.0 if Mario can jump otherwise 0.0.
- Mario is stuck : returning 1.0 if Mario is not moving anymore otherwise 0.0.

As regard the calculation for each level score the benchmark provides you a plenty of information, such as coins gained, bricked blocks, hidden block, enemy killed by fire,

stomp, turtle and so on.
The information used in our case are the following:

- Distance passed.
- Enemy killed with fire.
- Enemy killed by stomping.
- Mario in a gap: detects if Mario is ended in a gap.
- Mario mode: if small, big or in fire mode
- Mario status: used to detect if Mario has completed the stage.

1.3 Utilization

The benchmark provides the user the possibility of adding a single-threaded Java application that can inherit from an interface called *Agent*.
Each time step, which corresponds to 40 milliseconds of simulated time (an update frequency of 25 fps), the controller receives a description of the environment, and outputs an action representing the joystick inputs[5].
Environment information are provided to the agent after the call of *integrateObservation* method which populates two double array called *enemies, levelscene*.
To access them the calls are:

```
public int getReceptiveFieldCellValue(unt x, int y);
public int getEnemiesCellValue(int x, int y)
```

An example snippet code for above enemy is given at the following:

```
private float enemyInFrontDistance()
{
    int distance = 3;
    float fraction = 1.0f/distance;
    float retWeightedDistance;
    for(int i = 1; i < 4; i++ )
    {
        int value = getEnemiesCellValue(marioEgoRow, marioEgoCol + i);
        if(value == 80)
        {    //Goomba/Troopa //93 is spike koopa
             retWeightedDistance = distance * fraction;
             return retWeightedDistance;
        }
        distance--;
    }
    return 0;
}
```

At every frame the function *getAction* is called from the current agent to decide the output (representing the joystick buttons). In this function we will use the neural network to have the input given the of the best produced artificial neural network that will be described in the following chapter.

As regarding the information for the fitness score this must be took from the class *Environment* which contains the *EvaluationInfo* with all the basic information at every integration step. A little customization has been necessary to get also other information needed for the evaluation.

Chapter 2

Artificial Neural Network and Genetic Algorithm

2.1 Neural Networks and Genetic Algorithms overview

ANN were invented in the spirit of a biological metaphor for neural networks of the human brain (reference to code of nature)[6].
In computing it consists of many small units called neurons interconnected between them. Given an input set the neural network produces an output and usually this is represented in real numbers. Every connection has a weight indicating the strength of the connection as well as a bias representing the base value from which applying the strength. The mathematical feature of being treated like a function is also know as being feed-forward: no loops are allowed between the nodes[7].
Specifically the inputs to the neuron are summed and evaluated against an *activation function* which generates the final output. The net is structured in layers. Input layer, hidden layer and output layer is the basic structure but nothing forbid to have one or more hidden layers.
Neural network per se are not enough to describe a determinate the best solution of a certain problem. A mechanism to train them is necessary to have meaningful behaviours given a certain input. The most common one is back propagation which is based on correcting the weights and bias of the ANN based on the errors between the current input and the expected one. In the scope of this project NN has been combined

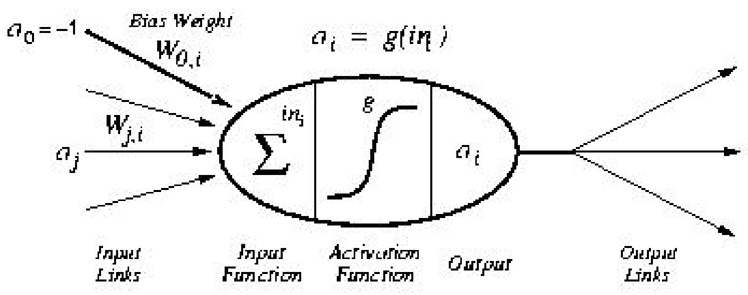


FIGURE 2.1: Neuron output calculation

with another technique called Genetic Algorithm. Both togehter are usually referred as neuro-evolution[8].
GA are based on the process of natural selection, this means they take the fundamental properties of natural selection. This process can be divided in steps:

- Initialization: a random population of desired size is generated. Each member is usually referred as chromosome
- Evaluation: each chromosome of the population is then evaluated and "fitness" score is assigned to them
- Selection: select the best elements given the fitness
- Crossover: combining the genes of two chromosome to form one. See Fig.2.2 for a look of how it works.

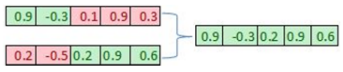


FIGURE 2.2: Schematic representation of crossover opeariotn

- Mutation: mutation of some of the gene of the chromosome. See Fig.2.3

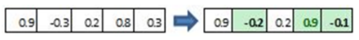


FIGURE 2.3: Schematic representation of mutation opeariotn

- Iterate all of the process

2.2 Algorithm strategy

In this chapter a detailed description of the algorithm used is provided.
The general idea implemented consist of having a neural net topology that never changes and what are adjusted are the values of the weights. Given this is pretty straightforward that the genes of each chromosome are exactly the weight of the net.
The neural net will have as inputs the environment info described in the previous chapter and as outputs the keys for Mario movement, namely LEFT, RIGHT, DOWN, JUMP, SHOOT/DASH.

2.2.1 Genetic Algorithm strategy

The algorithm proceed as follows:

- An initial random population is created.
- In turn each gene of a member of the population is copied into the correspondent link in the neural net and an agent is so created.
- These agents plays a certain number of levels and a fitness score is assigned to them.
- The population is being sorted by the score.
- Re-population starts:
  - A percentage of the best are saved and inserted in the next population
  - A percentage is shuffled and crossover operation is performed. Given two chromosome a random split point inside the chromosome is choose. The first or second part of the first chromosome (also this is random) will be chosen and combined with te second or first part of the second chromosome.
  - A percentage of the population will goes into an operation called *MultiMutateCrossOver* which consist an adjustment of a random number of genes using a Gaussian distribution before performing the crossover operation.

  - Since the population never change its size a final step is performed using a *MultiCrossoverWithRandomWeights* operation where the best chromosome are crossover with completely random one.
  - A new population with the same size of the previous one has been established.

2.2.2 Fitness strategy

One of the most important parameters for a GA is the fitness function. The fitness function, which is used to evaluate the performance of the different NN agents in the population, is based on variables that contain information about how a Mario agent has progressed in a level.
The basic score is given by distance travelled by the agent multiplied by a factor 10. In this implementation the goal is trying to have an agent that can play well in different levels. Considering time therefore did not provide good result (next chapter will discuss this in detail).
If the agent plays well on a certain level, namely surpass half level, it can unlock a series of bonus. These bonuses are:

- Enemies killed with fire x 100 points
- Enemies kill with stomp x 50 points
- Mario mode: fire consist of a 5000 bonus points, big consist of a 2500 bonus points
- Mario winning level: 10000.

2.3 High level implementation description

The implementation consist in building all the structure for the ANN as well as for the Genetic Algorithm trainer.
Additional classes with the solely scope of being an interface with the benchmark has been created as well.

2.3.1 ANN

To describe the implementation a list of the classes created will be given. To write the code for this implementation I have used as reference ”The nature of code”[8].
All the ANN classes are inside the package *competition.cig.andreacastegnaro.ga_an.ann.*

- Neuron: this class handle the output on the single neuron. It will need to track all the input connections coming from previous layer and output to the next layer. Activation function choose is the *Sigmoid* which is pretty standard.
- Connection: this class is a simple abstraction of an input link towards a single neuron. A connection can be a bias or a weight.
- NeuronLayer: this class is meant to be used as an helper for the neural network for handling operations. A layer has n neurons and n outputs and can be connected to another layer allowing a depth inside the structure. The layer se neuron class for activating and set all the output values.
- NeuralNetwork: this class is a wrapper for the Neuron layers. It has inputs, outputs structures and handle the propagation of the inputs towards the output during *ComputeNet()* step. It has also functions to retrieve all the weights from the net and this is necessary to map to a chromosome and viceversa.

2.3.2 Genetic algorithm trainer

The GA consist of three classes:

- Chromosome: output the fitness function value and will stored the weights as genes. It implements clone and comparable interface for sorting.
- GeneticAlgorithm: this class contains all the information of the population and will perform the operation of re-population. It will also manage the connection between a chromosome and the correspondent neural net.
- GeneticAlgorithmTrainer: this class contains the main function. It handle the strategy described in the previous chapter.

Some snippet code have been put in appendix A.

Chapter 3

Result and discussion

3.1 General result

As already declared in the abstract the purpose of this project was having a Mario that could play a several type of different levels having a style that could be similar to a human behaviour. That is the main reason for having a relatively small matrix centred on Mario for extracting environment information and so no intention of giving more than the visualizes screen information. Each GANN agent has been trained on a set of 15 different level with the basic difficult. One of the main improvement has been provided when the mutation has been implemented. In fact using only crossover operation brought rapidly to the g only crossover was

3.2 Table and discussions

Several configuration regarding net topology as well as fitness have been tried. Having a deep neural net increase the general performance and the best configuration has been the one with 30 hidden neurons in one layer. In order to try more combinations the code has not been run for more than 150 epochs. Details with granularity of the epoch have not been stored but in general the rule that if after 30 epochs nothing was changed then a local maximum has been considered to be reached.
With regards to the fitness the best strategy has been the one described in the previous chapter.Giving a fitness reward for time left by the agent was useless and resulted only

in having agents that tried to rush towards the end without shooting fire/stomping on enemies or giving the impression of analysing the situation and generally also provided the lower scores.

Optimal population has been found being 100: the best 20 were saved to the next population, the best 80 were combined with crossover, the best 60 were combined with crossover after mutation and a remaining 10 was created from the crossover of the best 10 with chromosome with random weights. Somehow when increasing the population apart form having a slower time to train it there was no notably performance increment. To train the best agent it took me 12 hours and the result was completing 13 over 15 total levels.

In the following table a report with some different configuration is shown. The table reports just the best population after the trials of adjusting fitness score, epochs and chromosome size. When reading the score value values keep in mind the fitness strategy described in the previous chapter.

Topology	Epochs	Training time(h)	Chromosome size	Score
13,20,5	250	5	100	425237
13,20,5	250	8	500	358631
13,20,5	250	11	1000	189565
13,50,5	250	9	100	265741
13,30,5	250	5	100	658774
13,15,10,5	250	12	100	768129

Of course this value must not be took as absolute since randomization process is involved in it so some fluctuations from the values could be possible.

Finally the result of this project can be found at the following Youtube links:

https://youtu.be/7BlDY9G9aQ8?list=PLIgw136Wn6mzUpNfvyUbitXYrdyu3KW1U

Appendix A

Appendix Title Here

GenericAlgorithmTrainer

```
private void trainPopulation()
{
    double score = 0;
    int indexOfNet = 0;

    for(int g = 0; g < epochs; g++)
    {
        if(g > 0)
            ga.GetPopulationFromNetSet();

        for(int p = 0; p < ga.GetPopulation().size(); p++)
        {
            Agent controller = new MarioAgent_GA_NN(ga.GetNeuralNetworks().get(indexOfNet));
            for(int i = 0; i <15; i++)
                score += PlaySingleNet(controller, i, false);

            ga.GetPopulation().get(indexOfNet).SetFitness(score);

            indexOfNet++;
            score = 0;
        }
    indexOfNet = 0;
    ga.Repopulate();

    ...

    }
}
```

GenericAlgorithm

```
public void Repopulate()
{
        SortPopulationAfterFitness();

        List<Chromosome> newPopulation = new ArrayList<Chromosome>();

        ...

        newPopulation.addAll(newPopPart1);
        newPopulation.addAll(MultiCrossOver(newPopPart2,true));
        newPopulation.addAll(MultiMutateCrossOver(newPopPart21,true));

        int populationLeft = population.size() - newPopulation.size();
        if(populationLeft>0)
        {

                ...

                newPopulation.addAll(MultiCrossOverWithNewRandom(newPopPart3, true));
        }

        ...

        copyPopulationToNet();
}
```

NeuralNetwork

```
public void ComputeNet()
{
        float tempouts[] = null;

        for (int i = 1; i < layers.length; i++)
        {
                tempouts = new float[neuronsEachLayer[i]];
                tempouts = layers[i-1].GetOutputs();
                //Get output from previous layer and setting as inputs of the current one
                layers[i].SetLayerInput(tempouts);
                layers[i].CalculateLayerOutput();

                //The last layer is the net output
                if(i == layers.length - 1)
                {
                        outputs = layers[i].GetOutputs();
                }
        }
}
```

Bibliography

[1] Sergey Karakovskiy Julian Togelius, Noor Shaker and Georgios N. Yannakakis. The mario ai benchmark and competition. 2009. URL http://julian.togelius.com/Togelius2010The.pdf.

[2] Sergey Karakovskiy Julian Togelius, Noor Shaker and Georgios N. Yannakakis. The mario ai championship 2009-2012. 2013. URL http://julian.togelius.com/Togelius2013The.pdf.

[3] Shaker Noor Julian Togelius. Mario ai framework. 2009. URL http://www.marioai.org-.

[4] Togelius Julian Yannakakis Georgios N. Ortega Juan, Shaker Noor. Imitating human playing styes in super mario bros. 2012. URL http://noorshaker.com/docs/imitating.pdf.

[5] Jan Koutnik Julian Togelius, Sergey Karakovskiy and Jurgen Schmidhuber. Super mario evolution. 2010. URL http://julian.togelius.com/Togelius2009Super.pdf.

[6] Artificial neural network. URL http://en.wikipedia.org/wiki/Artificial_neural_network.

[7] Steven Rabin. Game ai pro. *Section 30*, 2013.

[8] The nature of code. *Chapter 9-10*, http://natureofcode.com/book/.

[9] Matt Fisher Justin Johnson, Mike Roberts. Learning to play 2d video games. 2012. URL http://cs229.stanford.edu/proj2012/JohnsonRobertsFisher-LearningToPlay2DVideoGames.pdf.

[10] Peter Ross. Neural network: an introduction. http://www.itu.dk/courses/MVAI/E2008/notes.pdf.