Playing Mario using advanced AI techniques

Lars Dahl Jørgensen
ldjo@itu.dk

Thomas Willer Sandberg
twsa@itu.dk

***Abstract*— In this paper two approaches using different advanced AI techniques to develop two different AI Mario agents for the classic platform game Super Mario Bros will be proposed. Programming an AI Controller using for instance a Final State Machine (FSM) is not a problem, and has proven to work with almost every game on the market. The problem we are trying to solve here in this paper is to use advanced and more intelligent AI techniques, which are capable of learning to solve a specific problem by itself. To do this we are going to use Genetic Algorithms and Reinforcement Learning to evolve an intelligent Super Mario agent.**

I. Introduction

When working with learning algorithms it is an obvious choice to use computer games as the development platform. In this paper two such learning algorithms will be used for developing an artificial intelligent agent capable of playing a game of Super Mario. Super Mario is a two-dimensional platform game in which the main objective is to traverse each level from left to right, and in this process avoiding various dangers such as enemies and gaps on the ground. The challenge of this problem is not only making an agent for Super Mario that know how to act in different situations but also to translate the game state into a format, that an agent can read and process, and then act accordantly.

The Mario game used in this paper is a Java version of the original Super Mario Bros for Nintendo Entertainment System that allows attachment of an AI agent for controlling Mario. The agent is given access to a simple state representation of the game as input, and the agent should be able to determine which action (left, right, jump etc) to take in a given situation. In an initial approach to this problem an Artificial Neural Network (NN) solution was proposed, but this solution was quickly extended with two more advanced learning approaches using Genetic Algorithm (GA) and Reinforcement Learning.

II. General state representation

In this section the state representation used in the two approaches is presented. The state of the game is represented as a grid containing a numerical representation of the graphical interface. Each container in the grid represents a block on the screen and Mario is always placed in the coordinate of 11,11 of the 22 x and 22 y coordinates. To determine the state of the game the grid should be traversed and an action should be taken accordantly. For example to see what is just in front of Mario the agent will perform a lookup in the block 11,12. The output of the agent is the action represented as an array of Boolean values assigning which buttons the agent wishes to use. The 5 available buttons are left, right, duck, jump and shoot/run.

To make inputs to the agent we ensured that every part of the state representation could be normalized to values between 0 and 1. We chose to give the agent the same insight to the game, as we as humans would have while playing: What are the upcoming dangers, and how close are they. Using this line of thinking, we developed a list of method that can interpret the current game state into values between 0 and 1.

A. Enemy Inputs

We use the following 4 methods when looking for enemies: *enemyInFrontDistance*, *enemyBehindDistance*, *enemyAboveDistance* and *enemyBelowDistance*. The methodology of all four methods is the same. The methods returns the distance to enemy objects within 3 blocks in the direction in question. The agent's vision can be seen in Figure 1. The distance to an enemy can be 0,1,2 or 3 blocks. For example the distance-value will be 0 if no enemy is within the range and 3 if the enemy is right next to Mario. These values are normalized into 0 | 0.33 | 0.66 | 0.99 respectively. The higher the number, the more danger Mario is in. These inputs are obvious to use, since they give Mario information about which directions that are safe to go.

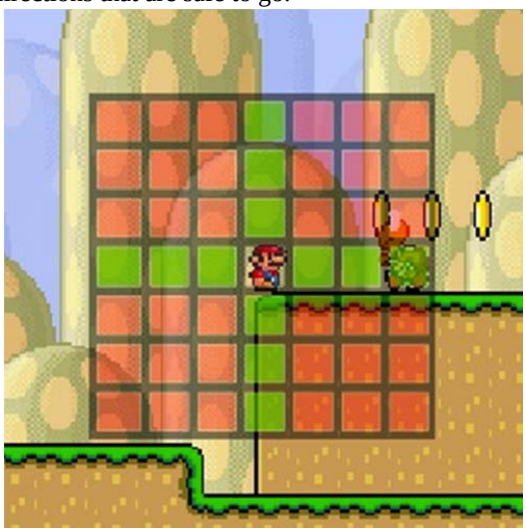

Figure 1: The vision of the agent. Visible blocks are marked with green. Red areas are blind spots.

B. Enviromental inputs

To make Mario able to traverse the length of the map, he has to know about potential obstacles and then hopefully learn to overcome them by jumping over them. These inputs consists of *obstacleDistance*, *obstacleHeight*, *obstacleBelowDistance*, *gapDistance*, *sizeOfGap*

and *flowerPotState*. *ObstacleDistance*, *obstacleBelowDistance* and *gapDistance* returns a value between 1 and 0 telling how far the next obstacle is away from Mario's position. *ObstacleHeight* and *sizeOfGap* represents the size of the upcoming obstacle. *FlowerPotState* also returns a value between 1 and 0 but this value represents if there is a flower pot in front of Mario and if the flower is peeking out of the pot, and therefore can be dangerous. In Figure 2 the method of detecting gaps is visualized.

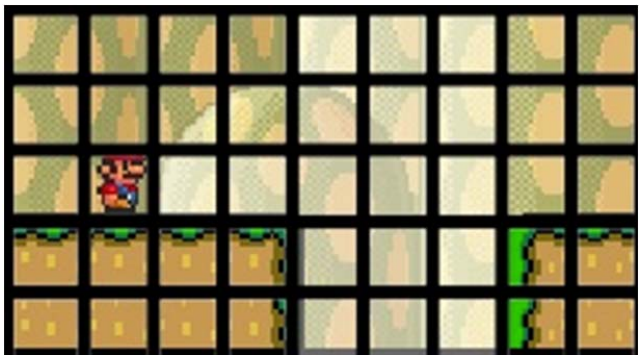

Figure 2: The agent's method of detecting gaps. In this case the both gapDistance and gapSize has a value of 3

C. Input about Mario's status

There exists a rule in Super Mario, that determines of Mario can jump in a certain state. If Mario is already in the air, he is not allowed to jump a second time. Furthermore he is not allowed to jump if the jump-key is already pressed. This rule has to be sent to the learning algorithm to let it know when it can use the jump method. This is why we have the input method canMarioJump, which returns 1 or 0 depending on the current state.

III. Genectic Algorithm Mario agent

A genetic algorithm (GA) is a fairly simple simulation of evolution on a computer. The purpose of evolution is to generate a population of individuals (chromosomes) with as high fitness as possible. All the chromosomes contain some values (also called genes), which can solve a specific problem. The goal for the GA is then to optimize the values, which is done by selecting the chromosomes, which solve the problem best, i.e. the ones with the best fitness values, and remove the worst chromosomes from the population[1]. This optimization is an iterative process, where the GA tries to increase the quality of the values for each iteration until a feasible or optimal solution is found[2].

A. Success criteria of Genectic Algorithm agent

One of the primary goals of this paper is to determine if it is possible to evolve an AI agent that is capable of learning the necessary skills to play the game Super Mario Bros by using GA. To succeed, the evolutionary trained Mario agent should at least be able to learn how to avoid the simplest obstacles and enemies, enabling it to complete a specific level with full life, after it had been trained on it.

Another important criterion for success would be that the best GA trained agent should be better and more intelligent than the Forward Jumping Agent and the Random Agent from the games source code[3]. By intelligent we mean that the agent should not just jump and shoot all the time, without actually analyzing the environment before making a move. The agent should for instance be able to analyze a state where it stands in front of an enemy, flower pot, obstacle etc., and thus learn what action that would be best to perform in this particular state. The GA trained agent would then be more intelligent than both the Forward Jumping Agent and the Random Agent, because they cannot analyze the environment they are in, but instead just jump all the time, or take random actions.

Of course it would be a big plus if the best GA trained agent could learn to generalize so much, that it could complete on any level at any difficulty.

It is not a goal to evolve an agent that could be as good as or better than the A* Agent[4] because this agent continually looks several steps ahead into the future, whereas our GA trained agent will only be able to see and analyze one state at a time.

B. Description of approach

The goal for the GA is to optimize all the values (genes) of the chromosomes in the population. In this paper these values represents the weight values from an Artificial Neural Network (NN). This NN then tells an agent what action to perform in different states. The weights in the NN will be optimized with respect to how an agent performs in a level. So firstly we need an NN and a GA. We have used our own already implemented version of an NN and a GA from the Ludo Assignment. The source code is available on the CD under

"SourceCode/GA/marioai/src/competition/cig/dahlsandberg"

where the GA Agent that will be used in this paper is also available. We have of course modified the source code in order for it to work with the Super Mario Bros game.

But before the NN can learn anything it needs to have an idea of the environment it's in. Therefore we need to define the state space, which will be used as inputs for the NN.

[1] Cf. Charles Darwin's theory on "survival of the fittest" from "The Origin of Species" (1859).

[2] Inspiration from chapter 7 Evolutionary computation, see reference [1].

[3] See folder ch.idsia.ai.agents.ai

[4] The winner of the Super Mario Competition 2009 (see http://tinyurl.com/MarioAstar)

C. State representation for the GA

For the GA approach the NN will have 14 inputs representing the agent's vision. 4 of the inputs are already explained in section II.A. For the GA approach we have also added 2 new enemy inputs: `enemyUpRightDistance` and `enemyDownRightDistan-ce.` These 2 new inputs extend the agent's vision, so that it is able to see enemies that are up to 3 blocks away in the upright and downright direction. That could for instance be a flying enemy or an enemy walking on a platform that is in a diagonal angle from the agent.

But there are still some limitations in these inputs for enemies, though, because if the enemy is walking in a horizontal line on a platform, the agent will only be able to see the enemy while it is in his diagonal angle, so it still has some blind spots in his vision. Another limitation is that the agent cannot tell the difference between enemies who can be killed by shooting and jumping, enemies that can only be killed by shooting, or enemies who cannot be killed by either shooting or jumping. We have made 9 inputs to show the agent if there were those different types of enemies around him, but we have chosen not to use them, because they confused the agent so much it performed worse with them than without them. The reason for this could be that the state space becomes too big for the agent to take advantage of the extra information.

We have therefore chosen not to train the agent on levels higher than difficulty 2, because otherwise it would be overloaded by information about all the enemies that suddenly cannot be killed by shooting them or jumping on them.

The NN also has some environmental inputs like `obstacleDistance`, `obstacleHeight`, `obstacle-BelowDistance`, `obstacleAboveDistance`, `gap`distance, `sizeOfGap`. The `obstacleAboveDistance` is the only input that is not already explained in section II.B and it tells the agent if there is an obstacle above him. In this way the agent has a possibility to learn that his jumping height will be reduced if there is an obstacle above him. The last input is `canMarioJump` and is explained in section II.C.

Those inputs should be enough for the NN to make the agent see obstacles like hills, flowerpots, gaps and the simplest enemies early enough to give the agent a change to make an action before it is too late.

The NN's hidden layer contains 20 neurons. We have chosen this amount of neurons based on what we could see the NN would learn quickest by. We have tried with different numbers of neurons, e.g. 25, 30 and 40, but it seemed like 20 neurons was the number that performed best.

The NN has five outputs which are represented by the five control keys: backward, forward, down/duck, up/jump and speed/shoot.

So the primary goal for the NN is to analyze the state space it gets from the inputs, and return the best combination of actions it has learned for this specific state by the GA.

D. Genetic Algorithm parameters

One of the first things that need to be decided is how big the population size should be, and we have decided on a population of 100 chromosomes. We have also tried with 50, 200 and 500, but 100 seemed to perform best.

1) Genetic Algorithm selection strategy

Below a walkthrough of the selection strategy our GA is displayed which will be used to decide which chromosomes should live, which should be thrown away, crossed over or mutated.

1. A number of chromosomes, equal to the population size, are created.
2. If a data file with weights to load does not already exist:
    a. Each chromosome will then be initialized with all the random weights from different newly created NN's.

    Else: The first 20 chromosomes will be initialized with the weights from the loaded data file, and the rest will be initialized with all the random weights from different newly created NN's.
3. All the chromosomes will be saved into the population.
4. All the weight values from all the chromosomes in the population are copied back to each of the represented links in each NN (equivalent to the population size).
5. All the NN's are then matched with the same amount of *GATrainedNNMarioAgent's*.
6. All these agents will now in turn play a fixed number of Mario levels one time each and the GA will then rank their performance, based on how well they play (fitness value). How the fitness value is calculated will be further explained in the section III.D.2)
7. The entire population will then be sorted, so that the ones that performed best will be in top and the worst in the bottom of the population.
8. Now a repopulation will be run including the following states:
    a. The 20 best performing agents will be saved. We know that crossover and mutation operations can harm or destroy chromosomes in a population[5]. Therefore we have chosen to let a number of the best chromosomes survive after each repopulation, so that we make sure not to delete any potentially optimal agents. Of course if there are better chromosomes in the next repopulation these will survive instead of the ones from the previous population.
    b. A copy of the 90 best agents from the population will be shuffled and run through a crossover operation (the shuffle makes it random which parent chromosomes will be crossed over with each other). This operation randomly chooses a crossover point where two parent chromosomes are split up. The first or second part of the first chromosome will randomly be chosen to match the other chromosomes opposite part. This will then create one new offspring, where the other

[5] Cf. chapter 7.3 Genetic algorithms, see reference [1].

two parts left are thrown away (see Figure 3). So the 90 best agents will be reduced to 45 in the new population. The new population now contains 65 chromosomes.

Figure 3: Random one point crossover

c. A copy of the 60 best performing agents from the population will run through both a crossover and a Gaussian mutation operation. This operation adds random numbers from a Gaussian distribution to a randomly picked number of weight values (genes) for all the 60 chromosomes. According to Negnevitsky[6] mutation can lead to very good results, but usually a mutation can do more harm than good. But why use it then, because it can help avoiding local optimum. With use of only crossover operations, there is a bigger risk that the population gets stuck on a local optimum. This is why we are using both mutation and crossover, and at the same time keep some of the best agents after each repopulation. Below in Figure 4 you will see how mutation works. The new population now contains 95 chromosomes.

Figure 4: An example of how our Gaussian mutation operation works

d. A copy of the 5 best agents from the population will be run through an operation called *MutateCrossOverWithRandomWeights*. This is an extreme operation that starts with flipping all weights[7] from all five chromosomes. Then every weight will run through Gaussian mutation, and five new random chromosomes are created. Those new chromosomes will then run through the crossover operation together with the existing five chromosomes. Afterwards we should have five new chromosomes, which have nothing to do with their parents, and will thus make sure that we do not just end up with agents that will only be optimal locally.

e. All these chromosomes will then form the new population.

9. Use the newly generated population and continue the algorithm from step 4 until a certain number of iterations.

[6] See chapter 7.3 Genetic algorithms for more information, see reference [1].

[7] Which means that all negative weights become positive and all negative weights become positive.

2) Definition of the fitness function

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 specific variables in the fitness is:

1. How far an agent progress in a level, which is measured in the games own distance units, where the standard maximum is around 4100-4500.
2. How much time the agent has left, if the agent gets through a level. The standard maximum start time is 200 seconds.
3. How many kills the agent gets in a level. The maximum number of kills pr. level varies a lot from level to level.
4. Which mode Mario is in when Mario is dead, wins or when the time has run out. Mario has 3 different modes:
    a. When Mario is small and cannot shoot/fire.
    b. When Mario is big and cannot shoot/fire.
    c. When Mario is big and can shoot/fire. This is the start mode for Mario in every level.

The fitness function has been tested with a lot of different variations containing the above-mentioned variables. At first we only tested with the distance passed, but this often resulted in too little variation between the different agents, had the problem that the agent's fitness was higher if it jumped into a gap, than if it got stuck before the gap and the agent got the same fitness for winning a level with full life (Mario mode 2) or with the lowest life (Mario mode 0). Therefore we tried to add some new variables to the fitness function. The first one was the Mario Mode, because with this information it would be possible for the GA to analyze and meet the agent's who managed to win with full life, rather than those who had lost one or two lives on the way. We also added "time left" and a kill score to separate the elite agents from each other. To ensure that the new variables did not affect the result too much, the distance travelled was multiplied by a constant for instance 10. In this way distance traveled would still be the main focus of the fitness function, but the elite would now rarely get the exact the same fitness.

To decrease the risk of having an agent that was very fast, killed many enemies but did not make it to the end of the level, we decided to split the level up in groups. Those agents that managed to traverse more than 4100 distance units (the elite) and those that didn't. The elite would be the only ones who got extra bonus from kills and time left, and the others would only be rated on their distance traveled and their Mario mode.

E. Evolution strategy

At first our strategy was to run the GA on all the chromosomes in the population with the lowest difficulty value (which is 0) and the same random seed until one of the agents had traveled more than 4100 distance units. When this happened, a flag was set, indicating that an agent in the

population had completed the current level, or that it had at least been very close to completing it. A new random seed would then be activated for everyone in the population. Only when an agent in the population had completed 10 different random levels in a row (no misses), the difficulty would be increased. Going from the lowest difficulty to difficulty 1 means that there will be other types of enemies and more of them, and more difficult terrain including gaps with and without stairs.

It seems like it is fairly simple to evolve an NN agent trained by GA to complete 10 different levels of difficulty 0. It only takes around 50 repopulations to do this, and have been tested in several attempts, with the same result. But we found out that the trained agents were not very good at generalizing, when the GA only trained them on one level at a time. The problem is that when the GA let go of one level and proceeded to the next level there is a big risk that it forgets a lot of what it learned from the last level, and instead focuses on how to optimize for this specific level.

Thus we needed to add more levels pr. run. Of course this increased the computational time. To compensate for this we changed the time limit to get through a level to half the standard time, so the GA would not wait too much on agents who got stuck. Another problem in only using one long level at a time, instead of many small levels, is that if the best performing agent get stuck somewhere or fall into a gap, all the other agents in the population will probably also get stuck in the same place or earlier for many repopulations. Many different levels in the same GA run will instead result in the agents' maximizing their fitness in the other levels, and thereby hopefully learning how to solve the problem with falling into a specific gap with fewer repopulations. Of course there is also a negative effect, when using many levels at the same time in the fitness. Sometimes an improvement of the fitness in two different levels can result in lower fitness in another level. But it is still more generalized than when we only trained on one level at a time.

F. Experiments

To test how well some of the different evolved GA trained agents performed, a competition was held between the GA trained agents and the random, forward jumping and A* agents. The competition was held on basis of the same 15 levels that some of the GA agents were trained on. All the GA agents had a population size of 100 and used an NN with 14 inputs, 20 hidden neurons and 5 outputs.

The "GAOnlyTrain1LevelAtOnce" agent is the only agent that was only trained on one level at a time. This agent has been trained with 50 repopulations on difficulty 0, 1 and 2 with seed 1 and with 150 repopulations on difficulty 2, seed 2. This GA agent can be seen in action on YouTube[8]. The "GATrained800" agent is one of the 3 GA agents that have been trained on the 15 levels with 500 repopulations and with 300 repopulations on 10 of the 15 levels (it is derived from the GATrained600 agent). The calculation time has been rather long for the GA and has run for 1,050,000 levels taken over 24 hours to calculate. On youtube a video can be seen showing the agent playing on one of the trained levels[9] and another video showing the agent play a random level[10], both on difficulty 2.

As it can be seen in one of the videos, this agent manages to complete one of the levels it had been trained on with full life. The other video shows that it also sometimes manage to complete levels it haven't been trained on, but not without losing one or more lives. This means that this agent has been able to learn and generalize its knowledge from the other trained levels to actually be able to complete a random none trained level.

The GATrainedWithDistance agent, which can also be seen in the two tables below, is a derived species from the GATrained600 agent, but have been trained further but with a fitness only including the distance it has travelled.

The following tables show how well the best GA trained agents perform against the random, forward and A* agents. There are two types of competition. The first one shows how well they perform in a long level, where the agents have half time and full life (meaning 3 lives) except for the last column showing the agents performance with only 1 life. The second table shows almost the same thing, but the agents are now tested on the 15 levels in normal length and time.

The fitness score is based on the same variables used for training the GA agents, and the fitness include punishments for jumping into gaps.

The distance score only shows how far the agents progress.

Long level, half time (full life)					
Player type	**Fitness**	**Average score pr. level**	**Dist-ance total**	**Aver-age distance pr. level**	**Distance (Only 1 life)**
Random Agent	190,679	12,712	14,910	994	9,920
GAOnly Train1Level AtOnce	229,610	15,307	27,940	1,863	16,680
ForwardJumpingAgent	386,068	25,738	44,048	2,937	23,522
GATrained With Distance	610,857	40,724	63,992	4,266	33,249
GATrained 600	859,691	57,313	41,240	2,749	22,413
GATrained 800	895,502	59,700	39,102	2,607	22,414
AStarAgent	6,765,196	451,013	228,016	15,201	87,248

Table 1: Competition type 1.

[8] http://tinyurl.com/MarioGA1

[9] http://tinyurl.com/MarioGA2

[10] http://tinyurl.com/MarioGA3

Standard level length and time (full life)					
Player type	**Fitness**	**Average score pr. level**	**Distance total**	**Average distance pr. level**	**Distance (Only 1 life)**
Random Agent	181,063	12,071	16,915	1,128	9,260
GAOnly Train1 LevelAt Once	389,221	25,948	26,889	1,793	16,680
Forward Jumping Agent	450,903	30,060	40,360	2,691	23,522
GA Trained With Distance	687,092	45,806	48,943	3,263	25,879
GA Trained 600	818,471	54,565	39,876	2,658	20,003
GA Trained 800	873,652	58,243	36,438	2,429	20,003
AStar Agent	1,971,758	131,451	64,576	4,305	44,440

Table 2: Competition type 2.

The two tables show several interesting things. One thing is that all the GA trained agents that have been trained on the 15 levels are outperforming both the GA agent that has only been training on one level at a time, but they also outperform both the forward jumping agent and the random agent. Of course none of them can outperform the A* agent, but that has never been a goal. The A performance is only presented in the table, because this represents what would be about the optimal score possible.

Another interesting thing showed by the two tables is that even when three of the GA trained agents outperforms the forward jumping and random agents, there is only one of the GA trained agents that has actually got a distance score higher than them, namely the *GATrainedWithDistance*. A reason for this could be that for the last 200 repopulations the GA has only been focusing on how far the agent progressed in the 15 levels, but there is no guaranty that it will complete with more than 1 life left. At the same time had the other two GA agents used the fitness function that also looks at how many life the agent has left, how many kills the player had and how much time the agent had left.

Another reason why the forward jumping agent gets more distance points than two of the GA trained agents could also be that the GA agents get a punishment of negative 5000 fitness points for falling into a gap. This sometimes results in the agent getting stuck on stairs just before a gap, because it is afraid of it, and has not found a better way to get over it. While the GA agent sometimes stops just a few steps before a gap, e.g. behind a stair, the jumping agent will often just jump into the gap or be lucky enough to jump over it. In both situations it will get more distance points than the GA agent.

But it always depends on what the agent's goal is. If you want to have an agent to just get as far as possible, then maybe the distance is enough for the fitness, but if you also want to have an agent that can get through without dying, you will need more variables in the fitness, and it is also more difficult to control, because you then will have 2 or more variables that need to be configured the right way to suits your goal best.

But maybe the fitness could have been tweaked a little more, so it didn't give too much preferential treatment for the fact that an agent had full life all the time compared to getting further in the level.

Another solution could for instance be to only train with the lowest life possible, so the agent would die every time it hit an enemy etc. A problem would then be that the Agent would never learn to shoot (only possible with full life). But when the agent learns to shoot it is also a problem, when it then misses a life, because the state looks completely equal no matter how many lives the agent has left and therefore the agent will think that it is shooting, but instead the *Run* action is activated.

IV. Reinforcement Learning Mario agent

Reinforcement Learning (RL) is a technique used for leaning based on experience. In its core it is based on simple trial and error. A random action is taken and the action is then either rewarded or punished according to a set of rules defining what is good behavior and what is bad. The reason for choosing RL as method for the agent is that Mario is a game that is hard to define a finite solution for. It is very complicated to make a set of finite rules for Mario to follow, since the state space is rather large and the fact that the game generates a random level for each iteration of the game makes the numbers of states even bigger. With RL you do not have to give the agent a set of rules or a finite solution to the game, you only have to define what a good action in the game is and what is not. In the game of Mario it is possible to give a reward or a punishment for every frame of the game. This makes RL powerful for this kind of problem, compared e.g. to a game of chess, where it is only possible to give a reward when the game has finished.

We have chosen to make the RL Mario agent using a Q-learning algorithm. The Q-learning technique was chosen because of the rather large state-action space of Mario. When using an NN for Q-table storage we get both the advantage of the generalization abilities of an NN and the ability to store the vast amount of states presented in the game. The source code and a short description of how to run it can be found in the CD in the folder "SourceCode/RL/src/ch/idsia/ai/agents/rl"

A. Success criteria of reinforcement learning agent

Due to the complexity of the state space of the game a limited success criteria was chosen. The goal for the RL agent was to be able to travel the longest distance possible on a simple random level in the game including enemies, obstacles and gaps but excluding advanced enemies as canons and tubes/flowerpots. As with the genetic algorithm approach we would like to see a more intelligent behavior than the forward

jumping agent supplied with the game. This agent is a static programmed agent, that keeps the forward and speed button pressed and jumps every time it is possible. This player actually scores quite well but is not intelligent at all.

B. Neural network Q-storage

When using Q-learning with an ordinary Q-table the memory limitations are quite significant. According to [2] we could get a working result with our 32.000 different states with and 12 actions per state if we used offline learning. But since we do not know the problem size on beforehand it does not make sense to train a RL agent using offline leaning. This makes the ordinary Q-table to an unavailable option.

According to [3] a Q-table is ruled out from the start since we have too many inputs to the agent. This is the reason why we chose to use a NN as storage for the RL algorithm. One of the major downsides of RL algorithm is that it cannot generalize. If the agent learns, that it is good to kill an enemy in one state it will not assume that killing an enemy is good in another state. This is where the generalizing abilities of NN's come in handy. If the NN is told that killing an enemy is good in several states the NN can assume that it is a good thing to do even when it visits a state that it has never seen before. But using a NN does also have an unfavorable impact on the Q-learning technique.

The Q-value that is given to the NN to represent at state-action pair is not going to be the same value when you request that same value later. This is because the values itself is not stored, but instead the value just influences the current Q-value in a positive or negative direction. Furthermore the Q-value for a state-action pair changes even when it is not updated. This is because the back propagation function of the NN will affect every output when error correcting just one. This is why a Q-table based on a NN will not always return the correct Q-value. If two Q-values are very close there is a big risk that the suboptimal action will be taken. In our approach 12 NNs are used, each representing an action in the game.

C. State representation and actions for the RL

For the RL approach only a selection of the earlier described inputs were used. The 8 inputs that were chosen will be able to give the agent enough information to cover the limited success criteria of the RL agent. The following 8 inputs are covering enough information about the game state for the agent to able to respond to obstacles, simple enemies and gaps of all sizes: enemyBehindDistance, enemyInFrontDistance, enemyAboveDistance, obstacleHeight, obstacleDistance, gapdistance, sizeOfGap and canMarioJump.

As motioned earlier in the latter section we chose to use 8 inputs for the agent to represent the current state. The 8 inputs cover the information needed for the agent to take account for obstacles and enemies: The output of the agent is the action that the Mario-agent should use in this specific state. We chose to filter out actions that are not essential for the game and then create a NN for each action. This resulted in the 12 actions seen in Figure 5 each represented by its own NN. The NN has 8 inputs for each of the state variables and a single output being the Q-value for the action taken in the inputted state. Even though the 5 keys used in the game would create 32 different key combinations we chose to narrow those actions down to only the key combinations that are required to complete the success criteria of the agent. This means that the duck-button is filtered out as well as impossible combinations like using left and right buttons at the same time. This results in the 12 combinations in Figure 5.

Left	Right	Jump	Speed	Represented by NN #
				1
			O	2
		O		3
		O	O	4
	O			5
	O		O	6
	O	O		7
	O	O	O	8
O				9
O			O	10
O		O		11
O		O	O	12

Figure 5: The 12 actions represented as 12 neural networks

D. Algorithm Parameters

The RL algorithm can basically be split up into three parts: exploration strategy, reinforcement function and learning rule.

1) Exploration strategy

The exploration strategy determines how the agent chooses its next action. The agent must make a trade off between exploration, choosing a random action every time, and exploitation, using the actions with the highest Q-value. With pure exploitation the agent risks getting stuck. In the game we have seen this happen plenty of times when the agent chooses to run forward even when running into an obstacle, because this has the highest Q-value. If the exploration rate is too low, the agent will never leave this state. If the exploration rate is too high on the other hand, the agent will not experience any progress. The reason for this is, that the agent will not put the knowledge that it has acquired to any use, and will just keep on taking random moves.

When training the RL agent we chose the start off with a high exploration rate of 90%. This ensures us that the agent will get a wide knowledge of the game states, and will experience a great amount of both rewards and punishments. This will give the agent a general behaviour regarding to the rewards specified. The agent will not do any good, because it does only use what it has learned 10% of the steps. After

having trained 5.000 games with the high exploration rate we change the settings to an even distribution between exploration and exploitation. This will ensure a progress in the game as the agent uses what is has learned 50% of the steps and tries new and random actions the rest of the steps. This will make the agent behave intelligently to the environment but also explore the possibility, that the action with the best Q-value not necessarily is the best action to take. Using this setting we train the agent for yet 5.000 games. As the last training set we use an exploration rate of only 1%. This will make the agent use primarily the knowledge it has acquired when choosing the best action.

When training with a low exploration rate the knowledge that the agent has acquired will be amplified since Q-learning is also applied when taking an action from the Q-storage. This will ensure that correct stored values for good actions will maintain a high Q-value in the Q-storage, and actions that have been given unmerited high Q-values for a bad action will be lowered. When training has finished the agent is used with an exploration rate of 0% to ensure that the agent will only use what is has learned. When the agent is running in real time Q-learning is also disabled since the agent is not learning anything new.

When the agent performs a lookup in the Q-storage it activates each of the 12 NNs for their Q-value using the current state space. As mentioned earlier each network represents an action. The 12 actions now each have a Q-value and the one with the highest value is chosen.

2) Reinforcement function

The reinforcement function is the action of rewarding the agent for taking good actions and punishing it for bad actions. After the agent has chosen an action, either by random or by a lookup in the Q-storage, the action is carried out and a reward or punishment is given. This means that every single step in the game (24 steps per second in run time) an action is carried out and reinforcement is applied. Since we are using a NN as Q-storage we are not able to use negative values as punishment. Instead we are using values between 0.0 and 1.0 and the agent will chose the action with the highest Q-value. So instead of actually punishing the agent when a bad action is taken, no reward is given.

In the game the main objective is to get from the starting position to the end of the level before the time runs out without getting killed. For this reason the reinforcement function is based primarily on rewarding the agent for moving Mario from left to right. To make sure that the agent kept a general speed the reward given for travelling forward depends on the distance progressed in each step. Furthermore a greater reward is given if an actions leads to a point in the level where Mario has not been before. This ensures that the agent will always try to move Mario further ahead and not backtrack on distance that has already been covered. Using this reward the agent will naturally learn to cope with obstacles. When the agent has learned to travel from left to right it will at some point run into an obstacle. This is a state that it has not seen before, so there will be no actual Q-value to give a correct action in this state. If the agent tries to continue running into the wall he will not progress in the level, meaning that no reward will be given. At some point, depending on the exploration rate, a random actions telling Mario to jump will get Mario past the obstacle and a reward will once again be given. So without directly rewarding the agent for climbing an obstacle, the algorithm has learned to do so, because of the missing reward.

A general problem with teaching a Mario agent to play the game was seen in both the GA- and the RL approach. When training was completed, the agent had figured out, that the best way to travel through the level in the fastest way without being killed was to just run and jump through the entire level. Even though this solution works rather well, as seen with the forward jumping agent, it is not showing any intelligent behaviour to the environment. This is why we in RL chose to give the agent an extra reward if it chose to run on the ground when on a straight level section rather than running and jumping. This makes the agent play a more realistic game of Mario, and furthermore it makes the inputs to the agent much more efficient, since Mario's vision is rather limited.

To make sure that the agent can complete the level without getting killed by enemies or falling into a gap, the agent need to be punished for action leading to these states. In the reinforcement function it is made sure that no reward is given if an enemy bumps into Mario from the front or behind. Furthermore rewards are withheld if Mario trips over the edge to a gap or jumps directly into it.

3) Learning rule

The learning rule handles the action of storing the knowledge into the Q-storage. The Q-storage keeps an estimated value of each state-action set, and the learning rule defines how this value is updated. The Q-learning rule is a bit different when using an NN for storage compared to textbook Q-learning. According to [4] the NN handles the learning rate of the algorithm. Furthermore the NN handles the increments on the old Q-value so this variable can be removed from the equation as well. This means that the Q-learning rule will look like this after having the mentioned values removed:

$$Q(s,a) = reward + (gamma * max(Q(s',a')))$$

The discount rate (gamma) in this calculation controls how much an action's Q-value is affected by the Q-values of the states that it leads to. With a low discount rate the Q-value is defined primarily by the reward given to the state action pair. This would make the algorithm incapable of learning sequences of actions since the action is only evaluated on the action itself and not by upcoming actions. With a high discount rate the Q-value of the superseding actions would have an equal effect on the Q-value as the current state. In the game it is important to have a high discount rate. When the agent triggers a jump it is important that the jump is not rewarded if it makes Mario collide with an enemy above. With a high discount rate, the reward given by jumping forward is positive, but the Q-value for this action is lowered because

Mario was killed in the next step.

E. Performance measure of the algorithm

To test the performance of the algorithm the agent was set to train for 10.000 games using the parameters defined in the previous sections. The performance measure was based on the distance that the agent was able to travel without dying. The mode of Mario was set to 1 to make sure that Mario could not shoot the enemies but instead was forced to avoid them. This will however entail that Mario only has two hit points out of three. The result of this training session shows a minor linear evolution over time. As seen in Figure 6 the result are generally scattered over a rather large area, meaning that it's hard to see the evolution of the agent. During the first 2.000 games a clear improvement can be seen, but from here on the evolution can only be seen as the minor gradient on the linear tendency. The reason for this limited evolution can very well be a result of the agent not being able to generalize over the vast amount of randomized levels that it is exposed to. Even though the evolution is only minor, the trained RL agent is actually playing the game quite well. Obstacles can be traversed, and most enemies and gaps are avoided. This performance can be seen in the game trailer placed in the video folder on the enclosed CD or via the link on the introduction.

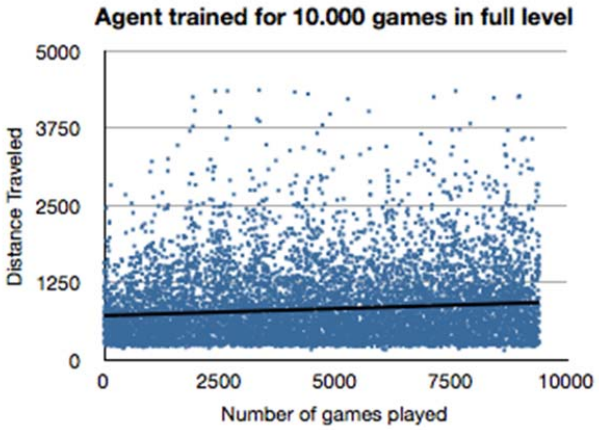

Figure 6: The evolution of the agents score on a full level

F. Experiments

To substantiate the theory of the missing ability to generalize mentioned in the latter chapter a batch of test on simpler problems were carried out. The agent was set to train on a straight level without any obstacles or gaps. The only objects the agent should learn to avoid is enemies. This test was performed to make sure that the agent was able to learn to avoid enemies. After a few hundred games the agent converged at max distance completing the level every time without dying. This proves the agents ability to learn a smaller defined problem, as avoiding taking hits from enemies. The same kind of test was performed with obstacles only-scenarios with the same results. This substantiates the claim that the RL agent has problems generalizing on the large state space of the random levels. The results of these tests can be seen in Figure 7 and Figure 8.

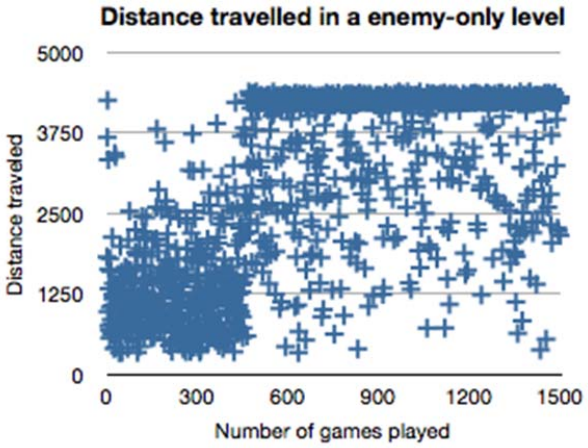

Figure 7: The result of training in an enemy-only level

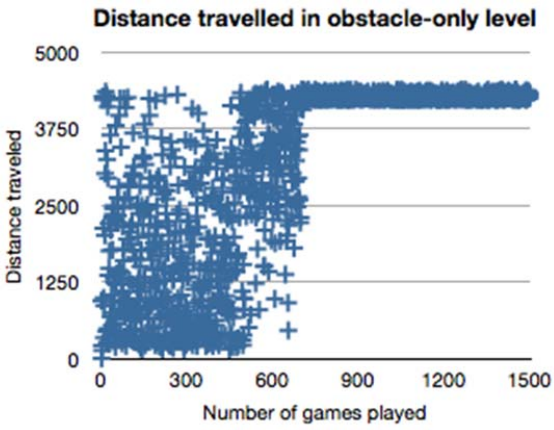

Figure 8: The result of training in an obstacle-only level

V. Conclusion

We will now conclude on the two approaches used in this paper.

GA is a good approach to efficiently find the optimal solution to suit a particular type of levels or problem in general. But GA can have problems with generalization, if you do not handle the circumstances correctly. For instance if you train an agent to play one level at a time with a lot of flower pots, many gaps or many enemies, then the GA trained agent will relatively quickly get good at solving each of these types of challenges separately, but there is a great risk in training the agent with a lot of the same challenges: the agent will very easily forget all the other obstacles or enemies it has already learned to handle, because these may have been sorted out to optimize the new kind of challenges it has been exposed to.

Since the selection process where chromosomes are combined with each other is completely automated and random, you cannot directly do anything about the problem with generalization. But if one chooses to present all the agents in a population with several levels at a time, we have proved in this paper that the GA will be able to train agents to be more generalized, than when they were only trained in one level at a time.

The ultimate solution would then be to create a number of levels with all the different types of challenges existing in the Mario universe and also some levels mixed with all these challenges, and then train a population on all of them in the same training set, running maybe 500 repopulations. The input should maybe also be reconsidered, because the ones we are using for the GA are not optimal for the more difficult levels, with huge amounts of different enemies. But if the inputs were corrected and the agents were trained with the mentioned training, it would likely result in a good all round generalized Mario agent, though of course it would take a lot of computer calculation time to do so.

The RL Mario agent had success with learning to play on random generated levels with a full blown combination of enemies, gaps and obstacles. Even so the algorithm never converged at maximum performance because the limited vision makes the agent jump into gaps and enemies from time to time.

Experiments performed showed that the agent was fully capable of performing RL on simpler levels and converged at a state where the agent traveled the complete length of the level every time without dying. When the simple scenarios were combined to a full blown Super Mario Bros level the agent had serious problems generalizing on the problem, and had a very alternating performance in every random level.

A very plausible reason for this problem would be the rather limited view of the agent combined with a limited temporal difference. To keep the state space small the agent's view was limited to straight-line sight and only a very limited distance of sight. This results in Mario not being able to see gaps and enemies more that 3 blocks away, and not being able to see enemies that have a diagonal angle on Mario. This means, that when Mario chooses to make a large jump to escape from an enemy or an obstacle, he can risk jumping straight into another enemy or a gap that is positioned outside his line of sight.

This could be solved with a bigger temporal difference making the agent apply the Q-learning to several steps in the history when performing an action. In this way the Mario agent will be able to learn much longer sequences of actions leading to success. Another part of the RL algorithm that could solve this problem is the reinforcement function. We had a hard time defining the rewards and punishments to be given to the agent to traverse the gaps in the level and it is clear to see in the game play videos, that it is here that the agent has the biggest problems telling us that a more precise definition of when the agent should be punished and rewarded in the handling of gaps is indeed needed to give the agent a better chance of generalizing on the random levels.

REFERENCES

[1] M. Negnevitsky, "Artificial Intelligence (A Guide to Intelligent Systems), 2nd ed., Addison Wesley, 2005
[2] I. Millington, "Artificial Intelligence For Games", Morgan Kaufmann Publishers, 2006
[3] J. Manslow, "Using Reinforcement Learning to Solve AI Control Problems", pp 591-601
[4] K, Støy, "Adaptive Control Systems for Autonomous Robots", University of Aarhus, 1999