Machine Learning Agents for Playing Super Mario Bros

Keywords: Super Mario Bros, Artificial Neural Networks, Backpropogation, Reinforcement Learning, Genetic Algorithms, Q-Learning, Q(lambda), Game Playing, Machine Learning, Naive Bayes Classifier

Jonathan Mullen JONATHAN@OU.EDU
Joshua Southerland JOSHUA.B.SOUTHERLAND-1@OU.EDU

Abstract

This paper explores the application of several Machine Learning techniques such as Artificial Neural Networks, Reinforcement Learning, Naive Bayes Classifiers, and Genetic Algorithms to develop an agent capable of successfully playing Super Mario Bros. The world of Mario presents a partially observable, dynamic, episodic problem, and thus provides an interesting and applicable platform to explore several Machine Learning techniques.

1. Introduction

Nintendo's Mario set the standard for a generation of video games. The linear world of Mario, filled with enemies, obstacles and power-ups, is a template which many others have followed. An agent that could solve the problem of playing Mario successfully could easily be applied to any other similar game such as Mega Man, Sonic, or Kirby.

The problem any agents faces when playing a game like Mario is how to correctly interpret what it sees in the environment and decide on the best action to take. The problem is important because as the numerous variations on Mario shows, there are various interesting incarnations of this problem.

We experimented with several learning solutions to solve the problem of correctly mapping what an agent sees in it's environment (state) to an optimal action. The machine learning aspects of our research involve Artificial Neural Networks (ANN), Genetic Algorithms (GAs) and Reinforcement Learning (RL) with Q-learning. In this paper we will explore the strengths and weaknesses of several agent designs. First we will show the results of an agent that uses an ANN with genetically evolved weights to map discrete values about the agent's environment to one of eight actions. Then we will look at a Q-learning, particularly Q(λ) agents that uses RL to learn which action will yield the greatest reward both immediately and in the discounted future. Lastly, we will show that our best performing agent used a mixture of Q(λ) and a Naive Bayes Classifier.

This work is the result of a Machine Learning(CS4033/5033) class project at the University of Oklahoma, Fall 2009.

2. Problem Definition and Agent Designs

2.1. Task Definition

For the purpose of this paper we will be working with a modified version of the Infinite Mario Bros game by Markus Persson[1], which is an all-Java tribute implementation of Nintendo's Super Mario Bros. The Infinite Mario simulator has the ability to generate random Mario levels of varying difficulty, length and type. The simulator was modified for an IEEE conference on Computational Intelligence and Games to allow for artificial agents to interact with the simulator.

The Mario agent exists in a two dimensional world. While the length of the level can vary, the observable world at any time consists of a 22 x 22 square where Mario is always located in the center. At each time-step the agent is handed an observation of the current world state. The observation is broken down into three main parts.

1. Information about the type of each square
   - FREE SPACE
   - QUESTION BOX
   - BRICK
   - COIN
   - CANNON or FLOWER TUBE
   - STANDABLE BOUNDRY
   - BOUNDARY WHICH YOU CAN JUMP FROM UNDERNEATH
2. Information about what is in each square
   - BULLET
   - GOOMBA
   - WINGED GOOMBA
   - GREEN KOOPA
   - GREEN WINGED KOOPA
   - RED KOOPA
   - RED WINGED KOOPA
   - SHELL
   - ENEMY FLOWER
   - SPIKY WINGED KOOPA
3. Additional information not directly related to a square
   - Mario's position in the world
   - Mario's Mode (i.e isBig, isSmall, is Fire)
   - Number of Coins collected
   - Number of lives left

The agent must then process this observation of the world and choose an action. An action is defined by the boolean on off state of each of the following keys:

- LEFT
- RIGHT
- JUMP
- SPEED

The process of observation, processing and action forms the basis of a single episodic sequence. The agent will repeat this process until one of three terminal states are reached. The three terminal states are:

- Agent Died (i.e ran into enemy, or fell in hole)
- Time Expired
- Completed Level

Designing an agent capable of playing Mario is an important task, since a solution to the general problem of learning how to optimally map states to actions in a partially observable and dynamic environment has wide range of applications. Mario specifically is a very interesting problem since like in most real world problems the state space is so large that most search based solutions would be intractable. As a result a reasonable agent must be able to generalize about its environment in order to overcome the issues associated with such a massive state space.

2.2. Algorithm Definition

2.2.1. Genetic Agent

The genetic agent is essentially an ANN, which takes values computed from the state observation as the inputs and one real output value for each possible action, the max value is considered the optimal action for that state. The ANN is a fully connected Feed Forward Neural Network with two (2) hidden non-linear layers. The number of input nodes is determined by the number of squares in Mario's state representation. The number of output nodes is fixed by the number of actions. The number of hidden nodes is not inherently constrained by the implementation, and can be varied as dictated by performance. The agent chooses an action from the linear output layers by selecting the action with the highest associated output value for a given time step. The algorithm used by the agent for action selection is shown below.

Algorithm 1 Genetic Agent Action Selection

Input: observation $state_t$, neural network net
Initialize $outputs = double[]$, $inputs = double[]$
$inputs$ = processObservation($state_t$)
$outputs = net$.evaluate($inputs$)
output MAX($outputs$)

As you can see the actual action selection process of the agent is fairly simple and straightforward. The performance of the agent is affected by the inputs to the ANN, or in this case the processed state observation, and the weights of the connections in the ANN. The policy of the agent is adjusted by varying the weights between nodes.

During development we observed that at times the agent would get stuck as the result of a bad policy. To help address this issue we developed a variation on the agent described above. This new agent follows an ϵ-greedy strategy in which 3% of the time it is allowed to act off policy choosing a random action.

The weights between nodes are considered to be the genes of the agent. The weights are adjusted through an Evolutionary Strategy (ES) that evolves them via random mutations. The process starts with an initial population of one hundred (100) agents each with an ANN initialized by a random Gaussian value with a mean of zero (0) and a standard deviation of one half (0.5). Once we have initialized the initial population we can begin the evolutionary process of selection, crossover and mutation. The specifics of each step in the process are described in detail below.

- **Selection**
  Each agent in the population is evaluated at the end of a run and given a fitness score based on the agents performance. The fitness function is computed using the following algorithm:

Algorithm 2 Fitness Reward Function

Input: evaluation *eval*
Initialize $Fitness = 0$
$Fitness$ += DistanceTraveled
$Fitness$ += TimeLeft
$Fitness$ += 10*CoinsCollected
$Fitness$ += 50*EnemiesKilled
$Fitness$ += 1000*WonLevel
$Fitness$ += -1000*DiesOnLevel
output $Fitness$

Once each individual in the current generation has been evaluated the population can then be sorted by fitness. The best performing half of the population is considered elite. The other half is discarded making room for new individuals.

- **Crossover and Mutation**
  The half discarded during selection is replaced by copying the genes from the elite population. They are then mutated by adding a small random Gaussian value to each gene. Once mutated the individuals are then combined with the elite to form a new generation.

The fitness function defined previously serves to evaluate the agents, and then by discarding the weaker performing half of the population we introduce an evolutionary imperative. Pseudocode for the evolutionary algorithm is provided below.

Algorithm 3 Genetic Evolution Strategy

Input: Initial Population $initialPop$, $size$
Initialize $nextGeneration$ = new Agent[$size$]
for $i = 0$ **to** $size$ **do**
  $initialPop_i$ = EvaluateFitness($initialPop_i$)
end for
SortByFitness($initialPop$)
for $i = 0$ **to** $size/2$ **do**
  $nextGeneration_i$= $initialPop_i$
end for
for $i = size/2$ **to** $size$ **do**
  $k = size$ - i
  $nextGeneration_i$= Mutate($initialPop_k$)
end for
output $nextGeneration$

2.2.2. Q(λ) Agent

We have chosen to implement a Watkin's Q(λ) agent which should perform well but has been a challenge due to the large statespace and the need to discretize the state representation. There have been many problems associated with the high speed decision making of our agent, which are described in 3.3 Discussion.

Q(λ) modifies regular Q-learning by adding a Temporal Difference(TD) learning component. By using an eligibility trace, the learning algorithm can now update states in the past for actions taken in the present, using the TD update δ. This is very useful for assigning blame to the actual action of jumping off a cliff, rather than the action of essentially do-nothing, the time step before hitting the bottom of the screen and entering a terminal state(death of Mario). This eligibility trace is typically cut off, or "zeroed" when an off-policy action is taken, such as when using ϵ-greedy. Watkin's Q(λ) in particular is often explained as zeroing the eligibility trace whenever a random exploratory action is taken(Sutton and Barto 1998). It is important to note that we are careful to avoid zeroing the eligibility trace whenever this random exploratory action is the same as the greedy action. Later in this paper we will compare Watkin's Q(λ) to McGovern's Q(λ), which is also called McGovern's Q(λ) for never zeroing the eligibility trace[11][12]. We have provided Watkin's Q(λ) pseudocode in Algorithm 4 as a reference.

Algorithm 4 Watkin's Q(λ) as presented by (Sutton and Barto 1998) modified to demonstrate replacing traces

Initialize *Q(s,a)* and *e(s,a)* = 0, for all s,a
Repeat (for each step of the episode)
Take action a, observe r,s'
Choose a$\prime$ from s' using policy derived from Q (e.g., ϵ-greedy)
a* $\leftarrow$ $argmax_b$Q(s$\prime$,b) (if a' ties for the max, then a*$\leftarrow$ a')
$\delta \leftarrow$ r + γQ(s',a*) - Q(s,a)
e(s,a) $\leftarrow$ 1
For all s,a:
Q(s,a) $\leftarrow$ Q(s,a) + $\alpha\delta$e(s,a)
If a' = a*, then e(s,a) $\leftarrow$ $\gamma\lambda$e(s,a)
else e(s,a) $\leftarrow$ 0
s $\leftarrow$ s'; a $\leftarrow$ a'
until s is terminal

Our Q(λ) agent's state representation makes use of a combination of thirty-six grid squares that are approximately the size of Mario, as described earlier. For the Q(λ) results these squares were either: free space, a non spiked enemy, a spiked enemy, a bullet, coin/powerup, or other. Additionaly, the Q(λ) agent can observe ten columns of a "can fall" heuristic, which returns true if every gridsquare in that column down to row twenty-two is freespace. The Q(λ) agent also observed a boolean value of whether or not it may jump, and whether or not it has fire power. Finally, the agent observes a discretized representation of it's own velocity. In the case of the agents playing difficulty 2, only the x velocity was considered, which can be six possibilities{<-30.0, <-10.0, <0.0, <10.0, <30.0}. Where as the difficulty 3 agents represented three posibilities for Mario's x velocity {<-31.0, <10.0, >=10.0} as well as three ranges of Mario's y velocity {<-10.0, <10.0, >=10.0} . The reward function is provided in Algorithm 5.

Algorithm 5 Our Q(λ) Reward Function for Mario

Input: stateInformation *stateInfo*
Initialize *Reward* = 0
Reward += 0.75*PositiveProgressDelta
Reward += -NegativeProgressDelta
Reward += 50*MarioStatusDelta
Reward += 50*EnemyKillsDelta
Reward += 50*PowerupsDelta
Reward += 25*CoinsDelta
Reward += -1000*ifDied
Reward += -0.5
output *Reward*

Given our Q(λ) agent state representations, it is impossible to visit every state in a reasonable amount of cpu time. Our agents will have to learn a good action to perform from many states and hope that after training is over, the unseen states visited in the test set or upon deployment of the agent are forgiving and rare. A Naive Bayes Classifier(NBC) will be tested in order to generalize the agent's knowledge to unseen states. In our application the NBC assumes that each state variable is independent of the max Q value for that state. Given this assumption, we can estimate the probability of the Q(λ) agent normally choosing each action in a new state, if it had experience with that state that was consistent with already visited states. The Q(λ) agent then simply chooses the action with maximum calculated probability, or a non-greedy action based on ϵ. In order to address the issue of getting stuck in unseen states, we intend for the agent to continue using ϵ-greedy after training finishes, with a small ϵ such as 0.005.

3. Experimental Results and Evaluation

In the following sections we will discuss the methodology used to evaluate each agent, and present the results of those tests along with an analysis of the results.

3.1. Genetic Neural Network Agent

3.1.1. Experimental Methodology

To evaluate the effectiveness of the genetic agent and evolutionary strategy an experiment was designed to evaluate if the evolutionary process would actually evolve an agent with greater fitness.

The initial generation was comprised of one hundred (100) randomly initialized agents. The evolutionary process was then run for a set level seed for fifty (50) generations. The ANN used by each agent had five (5) nodes in each of the two (2) hidden layers. Since the environment is stochastic this process was repeated five (5) times to obtain a good sample. The fitness of each generation is determined by the best performing individual in a generation. By plotting each generating against the fitness of that generation you will get a good idea if the population is able to learn and improve its fitness over several generations. Learning would be indicated by a increasing trend in fitness over the course of several generations.

During testing we found the agent would often get stuck at some point in the map due to a bad policy. By allowing the agent to act randomly off-policy a small percentage (ϵ=0.03) of the time we give the agent the ability to overcome some obstacles it might

not otherwise know how to get around get. To test the effects that this behavior would have on training we ran another learning test identical to the one described above, but with the agent able to act off policy.

Since the agent relies on the ANN to interpret the environment and map to an action it seemed logical to test what effects changes in the structure of the ANN have on learning. To test these effects we reran the above tests, but with ten (10) nodes in each of the two (2) hidden layers. This gives us four learning tests comparing the effects of epsilon-greedy and varying number of hidden nodes on the training process.

After testing the different aspects of training we will want to get a sense of how well the genetic algorithm is working an experiment was designed to compare the fitness of our evolved agent against other random and heuristic agents. The experiment consists of running each agent twenty (20) times for a random seeds and on a set difficulty. The fitness of each agent is averaged over all trials then recorded and plotted. The average fitness for each agent over all the trials is a good indicator of how well the agent will perform on a given level of that difficulty.

Then to test the ability of the agent to learn consistently across several difficulties we also ran a longer learning test where we trained a population of randomly initialized agents for two hundred (200) generations at each difficulty between zero (0) and five (5).

3.1.2. Experimental Results

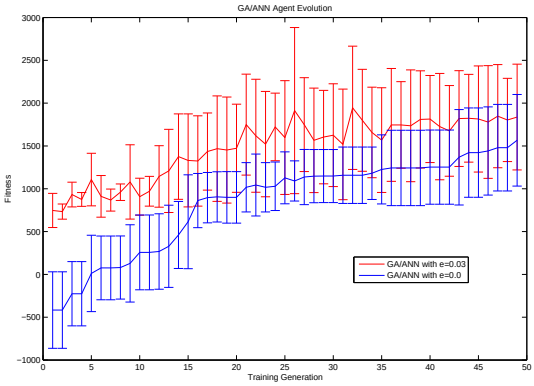

Figure 1. Averaged Results over 5 runs with 5 nodes in each hidden layer

Here we see the results of training with the GA/ANN agent with and without the ϵ-greedy behavior and for both five (5) and ten (10) nodes for each hidden now. Using the unpaired Student's T-test both were statistically significant within a 95% confidence interval.

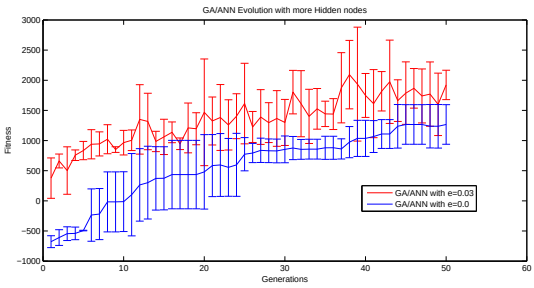

Figure 2. Averaged Results over 5 runs with 10 nodes in each hidden layer

The difference in the number of nodes in each hidden layer did not have a meaningful effect on the training process. There is however a noticeable difference between the using an ϵ-greedy behavior behavior an not. Initially it may seem that using ϵ-greedy was preferred, but while it did consistently achieve a greater fitness the overall increase in fitness was slightly less than the agent that always acted on policy. By allowing the agent to act of policy during training it is able to achieve more fitness, but it does not always have to learn how to overcome an obstacle. Instead it can occasionally rely on the off policy actions to overcome certain obstacles. The best approach perhaps is the train with out the ϵ-greedy behavior and then turn the behavior on during evaluation.

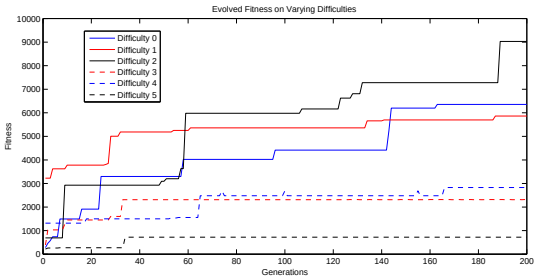

Figure 3. Evolved over increasing difficulty with accumulated knowledge

In Figure 3, the plot for difficulty zero (0) shows, that the first few generations after randomly initializing the population have a fitness value under one thousand (1000), and near zero (0). This is consistent with the random behavior we would expect from a randomly initialized ANN. The agents fitness however quickly improves over the course of the next two hundred (200) generations. By the end of training on seed 456 and difficulty 0 the agent populations best fitness was nearly seven thousand (7000), a more than seven hundred percent (700%) increase it fitness.

Difficulty one (1) shows that as the agent starts to train on a seed of 456 and a difficulty of 1 the populations best fitness is around three thousand (3000), almost half the fitness it had reached by the end of the last training batch, but still more than three hundred percent (300%) better than when the random agent had started the first training batch. This clearly shows that the agent is applying knowledge it learned previously. It should be noted that while it is able to reach nearly the same fitness for a difficulty of one as with a difficulty of zero, the net increase in fitness over the second two hundred (200) generations is less than the net increase over the first two hundred (200) generations.

Difficulty two (2) however bucks this trend of a declining net increase in fitness. The agent starts with a fitness of less than one thousand (1000), yet over the course of the next two hundred (200) generations reaches the highest fitness yet, and also has the greatest net increase in fitness. At first glance this may seem like an abnormality in the data, but we have to remember that the evolution of the agent is governed by random mutations and thus consistency is not to be expected.

This inconsistency is exemplified by difficulty three (3). The agent initially starts with a fitness of less than one thousand (1000). This quickly increases over the first forty (40) generations as it learns to apply knowledge it learned previously to current problem. After this initial rapid increase the agent plateaus near twenty-five hundred (2500). Again one could naively attribute this to the agent reaching its limitations, but again we have to remember the mutations are random, and thus the increases in fitness will also be random. The only long-term trend is that fitness increases over the course of several generations. How much it increases and how often is essentially unpredictable.

Both difficulties four (4) and five (5) continue to show the trend of random increases and long plateaus. However with these higher difficulties we start to see a definite decrease in the max fitness the agent population is able to achieve. This is very evident in the performance at difficulty five (5) where the agent for the first time is unable to achieve a fitness greater than one thousand (1000). It should be noted that this is not a proof of the algorithms limitations, but rather a reminder than evolution is a slow and unpredictable process.

When the trained genetic agent is compared to random and heuristic agents on unseen levels we can see that while it does better than the random agent and the forward heuristic agent it is outperformed by the forward jumping agent. This is mostly a result of the GA/ANN getting stuck on obstacles as the result of a bad policy. While the ϵ-greedy behavior helps in these situations the heuristic of the forward jumping agent clearly does a better job. Perhaps incorporating a similar heuristic and allowing the agent to realize when the on policy actions are no longer allowing it to progress.

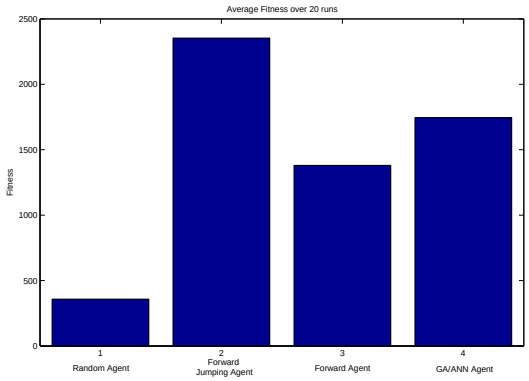

Figure 4. Compare Trained GA/ANN to Random and Heuristic Agents

3.1.3. Discussion of Results

The results from training the GA/ANN clearly show an ability to learn and adapt to the problem at hand. This is evident in the agents ability to increase its fitness over the course of several generations. We can see this supported by the stair step pattern of the fitness over the course of several generations. The problem that becomes clearly evident in the final comparison between the different agents is that the GA/ANN is not learning to generalize about the environment. When it does learn, it is learning just what it needs to increase fitness for a specific level or environment. This knowledge does not always translates to other environments, even if simmilar nature. As the agent trains on different levels it is unlearning a previous niche and learning to function in a new one. This is not to say the agent effectively scraps its knowledge from previous levels, but rather that it only learns to apply its knowledge in a specific way. This is supported by the rapid increase in fitness over the first several generations in the training tests. The rapid increase in the early generations of a new training example shows that the agent has a good general idea of what to do, but requires a few generations to learn how to apply this general knowledge to the specific problem at hand.

3.2. Q(λ) Agent

3.2.1. Experimental Methodology

The Q(λ) agent is attempting to maximize reward over time, so comparing performance in terms of total reward per game make sense. The Mario simulator allows for the random seeds to be specified, which, if all other game variables(length, time limit, difficulty, etc) remain fixed, ultimatley determines which random game is presented to the agent. In order to compare total reward with different levels of training experience, we will reserve the first 10,000 seeds(0-9999) to use as a test set. In the case of the final agent comparisons at difficulty 3, the first 3000 seeds were used in 1000 seed batches to improve the meaning of standard deviations across each learned agent.

3.2.2. Experimental Results

The state visitation is logarithmic, as expected. Our state representation, allows for approximately 1.857 x 10^{27} states at difficulty 2, although some cannot be generated. For example, it is not possible for there to be a block under mario and have the can-fall heuristic return true. The following figure shows an example of the number of unique states visited as number of difficulty 2 training games increases:

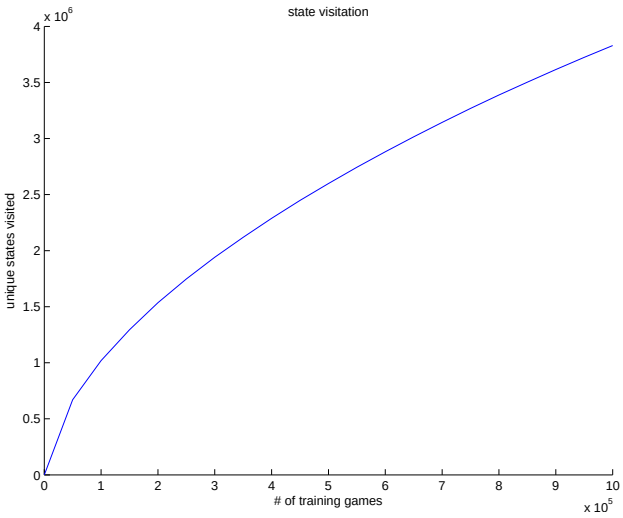

Figure 5. Logarithmic state visitation

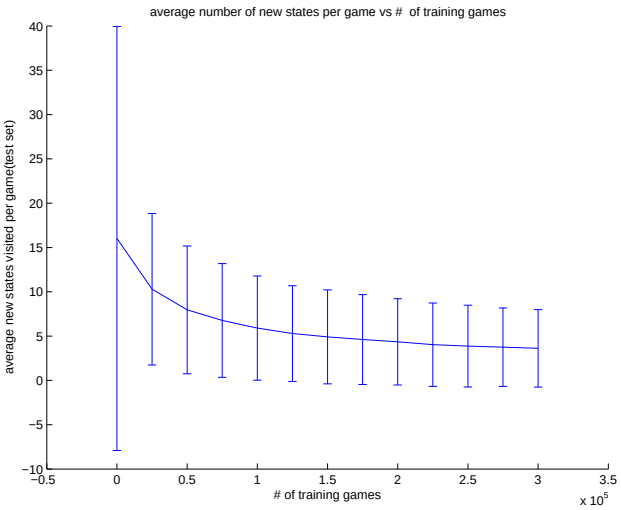

Figure 6. Average of unseen, or "surprise" states seen per game on the difficulty 2 test set with varying knowledge. Zero training games corresponds to a random agent with our selected actions

The total reward per game plot shows that our Q(λ) agent is learning, but learning slows down abruptly by 50,000 training games, which can be seen in Figure 7. Average kills against the test was an important graph to analyze(Figure 9). The results were not quite as expected and show that Mario kills about 3.25 enemies per level. Still, the Q(λ) agent kills many more enemies than both the random agent as well as the forward jumping heuristic, and would likely kill even more if there was less of a penalty for going backwards and spending time to accomplish this subtask.

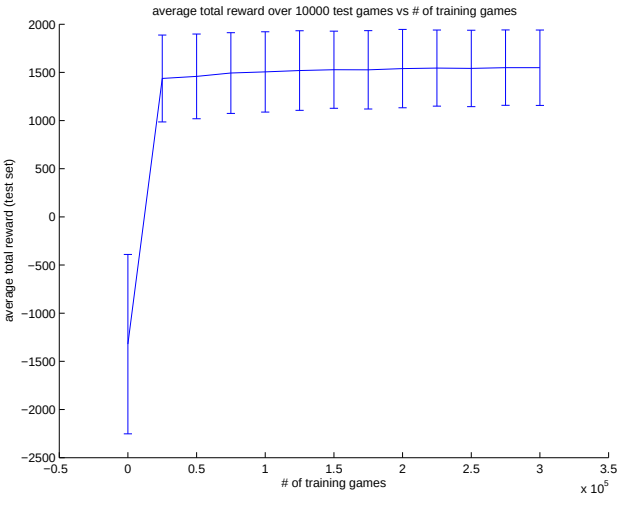

Figure 7. Average total reward per game over the difficulty 2 test set with varying knowledge. Zero training games corresponds to a random agent with our selected actions

Our Q(λ) agents have outperformed both the random and forward jumping heuristic agents on difficulty 3. Although the random agent tends to win some games at easier difficulty levels of length 200, it won 0 out of 225,000 test games and as a result we have not shown it in our agent comparison(Figure 8). The addition of the CPT resulted in better Q(λ) agents for Mario. Using the unpaired Student's T-test we have calculated this result as statistically significant on a 99% confidence interval at all of the levels of experience shown in Figure 8. The result of our agent using McGovern's Q(λ) with a NBC as an slight improvement over the variant using Watkin's Q(λ) instead has been similarly found to be statistically significant on a 95% confidence interval (for example at 60,000 training games). We were not able to find a point at any experience level where Watkin's Q(λ) with NBC outperformed McGovern's Q(λ) with NBC on a 95% confidence interval.

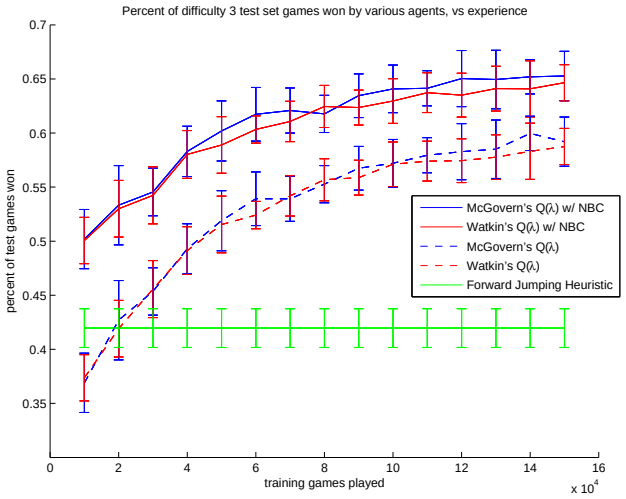

Figure 8. Number of wins against the difficulty 3 test set, with varying knowledge. The random agent had 0 wins out of 225000 attempts and was not shown for that reason

3.2.3. Discussion of Results

Given our state model, state visitation is sparse as we had expected. The number of new states seen each game goes down quickly and after one million training games, the agent visits as few as two new states per game on average. In the mixed Q(λ) and Neural net case this allows for hundreds of thousands of games to be played before saving the Q values becomes enough of a burden to require neural net backpropogation and clearing of the Q hashmap. Another advantage of this limited state is that the Q(λ) agent can store it's knowledge from some moderate amount of training, such as 300,000 games of length 200 on difficulty 2, and still perform well against new levels. An

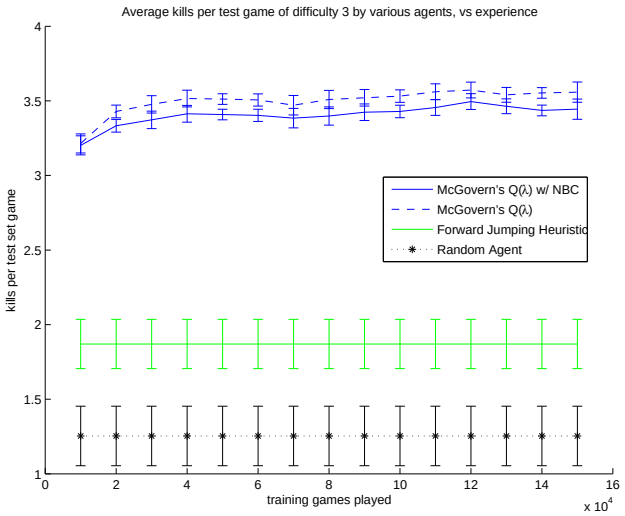

Figure 9. Average kills per difficulty 3 test set game, with varying knowledge.

important note on making the agent successful on new games when it hasn't visited the entire statespace, especially before the neural net can be used, is to leave learning on and/or epsilon greater than zero. This will ensure that when the agent finds these few surprise states, such as stuck up against a tube, that it can find a way to jump over it.

Our Mario agent is normally responsible for choosing 24 actions per game second, which presented challenges for the Q(λ) portion of our agent. Normally with such high-speed decisions, epsilon must be chosen to be very low in order to learn relationships such as "Don't jump off a cliff". If only 12 timesteps(half of a game second) pass from when Mario actually jumps to when he ends in the terminal state of death, with epsilon of even .05, the chance of taking a non-greedy, off-policy action between jumping and death is almost 100 percent. This means that Mario would need to jump into the same hole with exactly the same state information multiple times to finally blame the act of jumping off the cliff. However, having an epsilon of lower than .05 when using Q(λ) can dramatically increase the time it takes to find an optimal solution, when compared to Q-learning, as a sub-optimal solution is found more quickly. This is using optimal in the sense of the best thing to do in some part of the game, since we are no where near learning the optimal policy for all of Mario. We have settled at an α of .0025 and ϵ of .03 when training.

There are many ways to deal with the eligibility traces zeroing during the jump as a result of high-speed decision making. One way would be to switch the learning

algorithm to one which doesn't suffer from frequent eligibility trace zeroing such as our Watkin's Q(λ) implementation. One option is to use Peng's Q(λ), which doesn't make a distinction between on/off policy, but is difficult to implement. Another option is McGovern's Q(λ), which would be easy to implement but increases the cost of updates as we would have to keep an eligiblity trace containing every state seen in the current game. Quick tests have shown an increase of approximately 50 percent in the time required to complete runs. Besides switching learning methods, it is also possible to slow down Mario's decision making by return each chosen action multiple times in a row, such as two. Although this should help with credit assignment, we have informally seen a decrease in performance with this modification.

4. Related Work

To run our simulations we used a version of the Infinite Mario simulator written in Java. The simulator was originally developed by Markus Petersson[1], but was heavily modified by Sergey Karakovskiy and Julian Togelius [1]. Their modifications involved adding the ability to run faster than real-time Agent simulations for a AI competition they sponsored. For our purposes we had to make some further modifications to the simulator itself. Their version of the simulator did not have the ability for the agent to see some crucial game elements such as coins.

Julian Togelius et. al. describe, in [2], their approach to the to the Mario AI Competition. They comment on the problem of many controllers to generalize in this domain, such as controllers that learn to clear levels of difficulty three but fail to clear those with a difficulty of zero[2]. While they present the simulator as a benchmarking system for reinforcement learning, they mostly compare Neural Nets of different complexities. They comment on the need for temporal difference(TD) type consideration in algorithms, which is something we have been trying to work into our algorithm. They still arrive to a number of approximately 100,000 states and claim that value iteration would take too long.

In [3], John Asmuth and Chris Mansley of Rutgers describe using RL in a C++ clone of Mario Bros, titled Mega Mario. They recognize the need for methods to abstract the large state space, which is something we have we have done in our project. Their states, however, contained three components. First they track position and velocity of Mario in a discrete form based on block width. The third piece of information is a block vector, which contains information about what blocks are present in the grid squares around Mario. It was unclear to me how they handle enemies in the state space, or if they ignore them.

In [4] Kennedy and MacNish explore the limitations of Recurrent Neural Networks in solving signal delay problems. In particularly they investigated the relationship between the number of nodes and connections in the network and its ability to adequately model the problem. Previous research shows that when the number of nodes is greater than or equal to the number of delay steps then the RNN is able to learn to solve the problem by pipelining the nodes. This has obvious issues as the number of delay steps increases and or the size of the input grows. They find that when the number of nodes is limited to a point where pipelining cannot solve the problem the ability of the RNN to solve the problem is greatly reduced. Specifically the RNN is able to learn an adequate state representation, but is unable to learn the correct transitions to access the states in the correct order. This in-depth research into the computational abilities of RNNs and how the structure of a network affects its computational abilities is extremely beneficial in the design of our agent. Knowing what problem domains are well suited for RNNs allow us to design our agent in a way that does not require the RNN to learn problems it is not well suited for. The design of our agent calls for a Neural Network to analyze the state and select and object to interact with that has the greatest possibility of reward. Learning how to specifically interact with the chosen object will be handled by a Reinforcement Learning (RL) Agent. This design exploits the Neural Networks strengths in classifying and predicting while delegating the task of planning to the RL agent who is better suited for such a problem domain.

A common real world problem requires the classification of labels from noisy un-segmented data. While RNN can be trained to make independent label classifications, there is an inherent draw back in that the training data must be pre-segmented. According to [5] this is traditionally addressed through the use of hybrid systems. In a hybrid system a HMM is used to structure the training data and the RNN makes a classification. To avoid having to segment the training data given to the RNN and interpret the output signals into a label, or implement a hybrid system the authors of [5] present an approach for training the RNN directly. By training the RNN directly we avoid the need for a hybrid approach while at the same time eliminating the need to segment the training data. The only requirement of their approach is that the networks have a softmax output layer where there is one more output node than there are labels. Together these outputs

define the probabilities of all possible ways of aligning all possible labels with the input sequence. For the purpose of our project we can extend this technique of dealing with unlabeled data with the goal of making it possible for the agent to learn while playing against itself. Doing so allows us to use the infinite level generation provided by the Mario simulator as an infinite source of training data without the need to label it.

Another common problem associated with Neural Networks is over fitting the network to the training set data. This delicate relationship between generalization and over classification has previously been manipulated through methods such as structured risk minimization which adjusts the complexity of the networks classification function to optimize generalization. In [6] an alternative approach which focuses on defining a training algorithm that automatically tunes the classification function by maximizing the margin between training examples and class boundary. The hypothesis that maximizing the margin is equivalent to minimizing the maximum loss as presented in the paper allows for several novel optimizations. The most interesting of which is the ability to remove atypical or meaningless examples from the training data. The result is an equivalent training set which is often much smaller subset of the original training set. The obvious applications of such a method would be for problems which are sensitive to noise in training, and those which are computationally complex to train. Given the literally infinite amount of training data available through the simulator I do not believe such a method would necessarily be beneficial in the training of our agent. If our agent has issues with over fitting or adequate generalization this method would allow us to prune training data in a way that would minimize the error in the networks classification.

In [7] Rohwer examines how different Neural Networks model time. The paper explores the limitations of feed forward networks in modeling problems that require attention to distant temporal contexts. More specifically it examines the usefulness of feed forward networks in solving Markovian problems. The paper also shows that through the use of delay lines a nearly Markovian problems can be converted into Markovian problems. This idea is further extended to show that RNN are needed to solve non-Markovian problems that have a high temporal dependency. The values of the hidden nodes in RNNs are used to store pertinent information from previous inputs. This however does not mean RNNs are a universal solution to temporal problems. The paper shows that current methods such a back propagation while effective in training for Markovian problems has inherent limitations when training non-Markovian problems. Training with back propagation through time can solve this issue in theory by allowing previous errors to influence the current network. Now when we consider our Markovian problem of playing Mario we know with great certainty that a feed forward network trained through back propagation would be capable of solving the problem.

In [8] Franken and Engelbrecht investigate the idea of co-evolution as a way to train neural nets to play games. In their traditional form neural networks lend themselves to supervised learning problems. When applied to games with incomplete state information or games where the state space is too large to effectively search this idea of co-evolution create a chicken and the egg scenario. In such a game a supervised learning agent only knows the correct move at a given time for a solved game. Obviously if the game is solved than the move at any given time is inconsequential in the scheme of things. Co-evolution solves this problem by evolving neural nets that play against each other. This creates a competitive relationship in which there is a quid pro quo between the competing agents. As one agent improves, the other is forced to improve in response. This creates an iterative evolutionary environment. Given our simulators ability to generate infinite Mario levels we can train our agent by having it compete with itself. We can simulate competition by comparing the scores different agents achieve for a given run. Since each agent is evaluated in the same world their relative scores are an effective metric for comparing the abilities of a given agent. This was the the inspiring idea our research into the GA/ANN agent.

Bartz-Beielstein and Konen examine Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES) as an alternative to neuroevolution in [9]. Particularly they evaluate the convergence of different RL algorithms. The CMA-ES algorithms they adapt to the domain of game playing in [9] are subsequently shown to converge faster than Temporal Difference Learning (TDL). The performance of CMA-ES is further explored by evaluating it under several different fitness functions. The different functions vary in how they evaluate the offspring and how they select the offspring to be evaluated. The high rate of convergence is attributed to the instructive nature of failure on RL tasks. These early failures lead to higher levels of success as opponents change. The methodology they use to evaluate and evolve each generation is easily applied to the ideas of co-evolution explored above. If the results they obtained with TicTackToe translate into our problem of playing Mario we can expect to see much faster convergence in far fewer generations

than other methods. Their results show CMA-ES algorithms converged with one third the number of generations needed by TDL. These kind of results warrant further exploration.

5. Future Work

5.1. Genetic Agent Future Work

The current genetic agent despite its capacity to learn over time has some serious shortcomings in that it has difficulty generalizing its knowledge and applying it to new problems and environments. Given enough training it may be able to learn techniques to generalize about the environment, but reaching this point through random mutations would take an unreasonable amount of time. A more realistic and practical application of the genetic evolutionary techniques would be to evolve an existing agent rather than a random agent. By starting with an agent that has the weights of its ANN initially trained from a good intelligent policy, the evolutionary process could focus on learning new and interesting behaviors. This could greatly reduce the number of generations needed to see truly interesting results. By starting with an intelligent agent instead of a randomly initialized agent we have a good base line for the weights of the ANN and no longer need to spend the massive amount of training time evolution requires to converge to a decently intelligent policy.

5.2. Q(λ) Future Work

A major focus of the Q(λ) future work should be to find a representation and training process that allows for learning efficiency in that fewer states and training games are necessary to provide a good performance against test sets. This will mean trying new state representations, new learning parameters, as well as variations on the Q(λ) and NBC agent. As long as tabular RL is being used on this scale, RAM becomes a problem and the agent is foced to stop training early. The NBC has improved the performance to states visited ratio and one idea is to remove from our Q-value hashmap any states that are consistent with the NBC's estimates. Also, Of the Q-values obtained in initial training, we expect many to be boring in that they correspond to rare states that have small min/max Qs. We would like to try culling these states at the ANN or gradient descent training process of our future function approximating RL agents.

5.3. QNet(λ) Agent Future Work

It has become obvious that it is necessary to consider more state information at higher difficulty levels. One important modification is expanding the 6x6 grid to 8x8 in order to handle more flying enemies and bullets. We would also like to represent each square as either a spiked enemy/flower, enemy that can be jumped on or bullet, coins/powerups, free space, question box, or other. These changes will make our statespace large enough that we expect Q(λ) to be infeasible. As an alternative we will use ANNs to store Q-values which should allow for generalization. Our general approach will be to have one neural network for each action and to use those ANNs to load the corresponding Q(s)[a] value of the hashmap when a state missing from the hashmap is encountered. Once learning levels off or the memory requirement is pushing the limits of the system's RAM, we will iterate through the Q hashmap (Qmap) many times while using backpropogation to train the ANNs to our new Q-values. Initial attempts at this agent have failed, despite individually tested ANN and RL agents, and expect this approach to require considerable cpu time.

Algorithm 6 Proposed QNet(λ) Agent

```
comment: Heavily using a hashmap for Q values
which we will refer to as (Qmap)

DO
  Begin playing n games
    if state s has not beed seen
      if first batch of n games
          initialize Qmap(s)[a] = 0 for all a
      else
          load Qmap(s)[a] from ANN_a, with inputs_s
      endif
    endif
    continue as if using Q(λ)

  for j backprop loops
    foreach state s in Qmap
      foreach action a
        backprop ANN_a using:
          stateinfo inputs_s
          output label Qmap(s)[a]
  end
  clear Q hashmap

UNTIL convergence, satisfactory performance obtained, or out of time
```

6. Conclusion

6.1. Genetic Agent Conclusion

From the experiments on the evolution of the genetic agent we can conclude that while random evolution can certainly learn to increase the fitness of an agent for a specific example. The problem however is that evolution through random mutations is a slow process. So slow that it would be unrealistic to train an agent from random to a decent level of intelligence through evolution alone. The technique shows more promise in the application of augmenting an established policy. By using the evolutionary strategies to evolve agents already initialized to an intelligent policy we can bypass the massive amount of time required to evolve from random. Leaving evolution free to evolve the agents policy to perform new and previously unimagined behaviors.

6.2. Q(λ) Conclusion

The Q(λ) agent has been shown to learn on and perform well against levels of difficulty 2. Against unseen levels of difficulty 3, we have shown that our Q(λ) agents can outperform the random and forward jumping heuristic agents. The McGovern's Q(λ) agent that uses a Naive Bayes Classifier to generalize to unseen states has shown the best performance and this result has been shown to be statistically significant. We have informally seen that some performances, such as average number of powerups gained per level, increases when the agent uses a 7x7 grid. We hypothesize this trend to continue with the move to 8x8, at which point function approximation will be necessary.

An representative example of the Watkin's Q(λ) + Naive Bayes Classifier agent described in this paper can be observed at `http://cs.ou.edu/~southerland/mario.ogg`.

7. Bibliography

[1] Karakovskiy, S. and Togelius, J. (workshop presentation) Mario AI Competition @ CIG 2009, CIG 2009 (`http://julian.togelius.com/mariocompetition2009/CIG2009Competition.pdf`)

[2]Julian Togelius, Sergey Karakovskiy, Jan Koutnik and Jrgen Schmidhuber (2009): Super Mario Evolution. To be presented at CIG 2009. (`http://julian.togelius.com/Togelius2009Super.pdf`)

[3] Asmuth, J. and Mansley, C. (abstract) RLMario: A New Reinforcement Learning Domain, 2008 (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.5532&rep=rep1&type=pdf`)

[4] Kennedy, A. and MacNish, C. 2008. An investigation of the state formation and transition limitations for prediction problems in recurrent neural networks. In Proceedings of the Thirty-First Australasian Conference on Computer Science - Volume 74 (Wollongong, Australia, January 01 - 01, 2008). G. Dobbie and B. Mans, Eds. ACSC, vol. 312. Australian Computer Society, Darlinghurst, Australia, 137-145.

[5] Graves, A., Fernndez, S., Gomez, F., and Schmidhuber, J. 2006. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In Proceedings of the 23rd international Conference on Machine Learning (Pittsburgh, Pennsylvania, June 25 - 29, 2006). ICML '06, vol. 148. ACM, New York, NY, 369-376. DOI= `http://doi.acm.org/10.1145/1143844.1143891`

[6] Boser, B. E., Guyon, I. M., and Vapnik, V. N. 1992. A training algorithm for optimal margin classifiers. In Proceedings of the Fifth Annual Workshop on Computational Learning theory (Pittsburgh, Pennsylvania, United States, July 27 - 29, 1992). COLT '92. ACM, New York, NY, 144-152. DOI= `http://doi.acm.org/10.1145/130385.130401`

[7] Rohwer, R. 1994. The time dimension of neural network models. SIGART Bull. 5, 3 (Jul. 1994), 36-44. DOI= `http://doi.acm.org/10.1145/181911.181917`

[8] Franken, N. and Engelbrecht, A. P. 2003. Evolving intelligent game-playing agents. In Proceedings of the 2003 Annual Research Conference of the South African institute of Computer Scientists and information Technologists on Enablement Through Technology (September 17 - 19, 2003). J. Eloff, A. Engelbrecht, P. Kotz, and M. Eloff, Eds. SAICSIT, vol. 47. South African Institute for Computer Scientists and Information Technologists, 102-110.

[9] Konen, W. and Bartz-Beielstein, T. 2009. Reinforcement learning for games: failures and successes. In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers (Montreal, Qubec, Canada, July 08 - 12, 2009). GECCO '09. ACM, New York, NY, 2641-2648. DOI= `http://doi.acm.org/10.1145/1570256.1570375`

[10] Sutton, R. S. and Barto A. G. 1998. Reinforcement Learning: An Introduction. MIT Press. `http://www.cs.ualberta.ca/~sutton/book/the-book.html`

[11] McGovern, Amy and Sutton, Richard S. (1997) Towards a better Q(lambda). Presented at the Fall

1997 Reinforcement Learning Workshop

[12] McGovern, Amy. Personal Communication: Naive Q(lambda) / McGovern's Q(lambda). November 22, 2009.