Pacman Game AI using Genetic Algorithm and Q-Learning

Abhishek Gore, 140040011

Samarjeet Sahoo, 150100017

Madhav Goel, 150110017

Shriram S B, 150050099

Abstract

Genetic Algorithms and Reinforcement Learning are useful for finding solutions to problems where there is no clear solution and the situations vary with time, i.e. for probabilistic situations. In this project, we first implemented an AI for the Pacman Game using Neural Networks with Genetic algorithm. To overcome its limitations, we then implemented Reinforcement learning with Q-learning algorithm to train the game.

Description of the Game

Pacman is a game in which the player has to control Pac-Man through a maze to collect food dots while avoiding the ghosts which will be roaming around the maze. Power-dots provide Pac-Man with a temporary ability to eat ghosts and earn bonus points. The ghosts reappear in the center after being eaten. The goal of the game is to consume all the dots in order to win. The ghosts roam the maze, trying to kill Pac-Man. If any of the ghosts touch Pac-Man, he loses the game.

Genetic Algorithms

Genetic algorithms (GAs) are search algorithms based on the mechanics of natural selection and genetics as observed in the biological world. They use both direction ("survival of the fittest") and randomisation to robustly explore the best set of variables.

In GAs, we have a pool or a population of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

To this problem, genetic algorithm is applied by treating each vector of weights of neural network as an individual in a generation. Then, each individual neural network is used to play the game and a performance(score obtained at the end) is associated with it. Top 1/4th individuals are passed on to the next generation(selection step - equivalent to survival of fittest). Then, cross-over step is performed in which two parents are selected and a child is created by crossing them. For each position in the weight vector, one

values out of two from parents is selected at random and passed on to the child(equivalent to cross of two chromosomes). This step is performed till remaining 3/4th of population is obtained. Then, with a small probability, weights of an individual in new generation is multiplied by some random number(mutation step).

Q-Learning Algorithm

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations.

In Q-learning and related algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment. A history of an agent is a sequence of state-action-rewards:

(s0,a0,r1,s1,a1,r2,s2,a2,r3,s3,a3,r4,s4...)

which means that the agent was in state s0 and did action a0, which resulted in it receiving reward r1 and being in state s1; then it did action a1, received reward r2, and ended up in state s2; then it did action a2, received reward r3, and ended up in state s3; and so on. We treat this history of interaction as a sequence of experiences,
where an experience is a tuple (s,a,r,s'),
which means that the agent was in state s, it did action a, it received reward r, and it went into state s'. These experiences will be the data from which the agent can learn what to do. The aim is for the agent to maximize its value, which is usually the discounted reward. Recall that Q*(s,a), where a is an action and s is a state, is the expected value (cumulative discounted reward) of doing a in state s and then following the optimal policy. Q-learning uses temporal differences to estimate the value of Q*(s,a). In Q-learning, the agent maintains a table of Q[S,A], where S is the set of states and A is the set of actions. Q[s,a] represents its current estimate of Q*(s,a). An experience ¡s,a,r,s'¿ provides one data point for the value of Q(s,a). The data point is that the agent received the future value of r γ V(s'), where V(s') = max Q(s',a'); this is the actual current reward plus the discounted estimated future value. This new data point is called a return. The agent can use the temporal difference equation to update its estimate for Q(s,a):

$Q[s,a] := Q[s,a] + \alpha * (r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$

or, equivalently,

$Q[s,a] := (1-\alpha)Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'])$

Approximate Q-Learning Algorithm

Intuitively, we see that humans are able to play the game decently well without playing a huge number of games. That suggests that humans figure out simple heuristic rules to avoid the ghosts and eat food. This would in turn motivate us to posit that ta good decision making algorithm depends only on a few features. Using this, we can greatly reduce the size of Q-matrix and formulate a good algorithm much faster.
Approximate Q-learning assumes the existence of a feature function f(s,a) over state and action pairs, which yields a vector f1(s,a) .. fi(s,a) .. fn(s,a) of feature values. Intuitively, what this means is that now each action value pair is made of different components which can affect the potential rewards. These components

or features are conceptually similar to the features we used in the genetic algorithm. Some features we tried and used in our code include -
1. Checking if the non-scared ghosts (scared ghosts don't matter as they can't eat pacman) are within a small distance (varied)from pacman
2. Checking if the food dots are in the next state
3. Distance of the closest food dot
4. Distance of closest capsule
5. Actual distance of closest ghost

The approximate Q-function takes the following form

$Q(s,a) = \sum_{i=1}^{n} fi(s,a) * wi$

where each weight wi is associated with a particular feature fi(s,a). The update of weight vectors is similar to the updation of the Q-values:

$wi := wi + \alpha * difference * fi(s,a)$

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

Code Details

Neural Network in Genetic Algorithm

1. The **NeuralNet** class in NeuralNet.py contains the implementation for the neural network
2. The **GeneticAgent** class in SearchAgents.py contains the implementation for the agent implementation using genetic algorithm
3. The **getAction** method of the class generates the features given the state of the Game as input
4. The **crossOverAndMutate** method of the class performs the crossing over and mutation over the population in the current generation to generate the children for the new generation
5. The **tournamentSelect** method of the class randomly selects one quarter of the population and the chooses the best individual from it
6. The **updateGen** method of the class creates the population for the subsequent generation

Reinforcement Q-Learning

1. For this project, we have used UC Berkeley assignment on Pacman to get the logic for the game, and we have edited the agent files to implement our own decision making algorithm
2. The game logic asks our agent for a decision given an observation of the code. We pass game state and appropriately extracted features to Q-learning agents
3. We have edited the featureExtractors.py, game.py, pacman.py,ghostAgents.py files to implement our functionality, and written qlearningAgents.py and valueIterationAgents.py ourselves.

Observations

Neural Network in Genetic Algorithm

Genetic algorithm performed well on small mazes. For a population size(number of individuals in a generation) of 100, it trained in 200 generations started winning almost 95% of the time. For medium mazes, for generation size of 200, it trained in 250 generations.

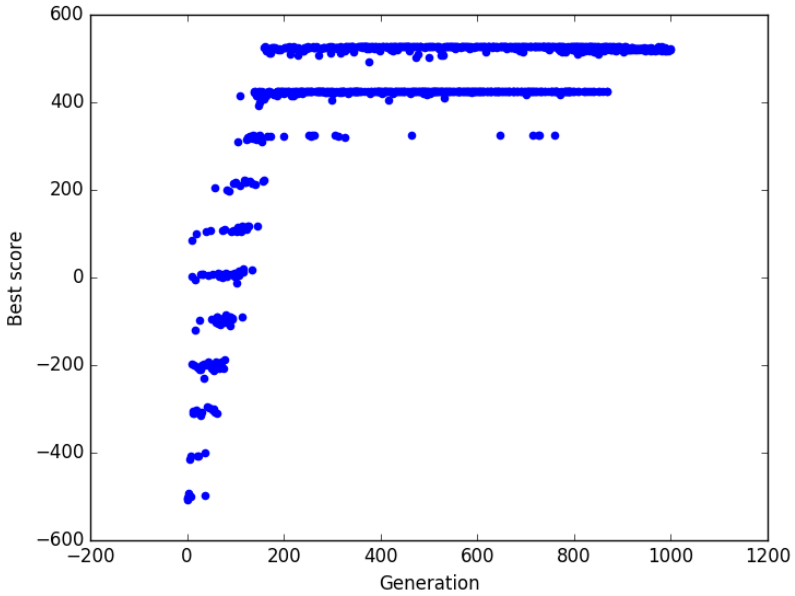

Figure 1: Best Score vs Generation

Figure 1 shows variation of scores with number of generations trained. The average performance increases with the number of generations which is expected.

Figure 2 shows variation of best score of each generation with population size. It is clear from this figure that large population is needed for convergence. This is true because with large population size, we have large variety in a generation.

Figure 3 shows difference in method of crossing used. In "average", average of two parents' weights is weight of child. In "pick random", one of two parents' weights is selected at random and passed to child. In the first one, the child always lies on the line joining the two parents. But in second, the child is close but not necessarily on the line joining them. This gives some randomization and leads to convergence quickly.

In Figure 4, "average of n" says that in each generation, each individual is used to play n times and the average of scores obtained in each play is taken as its performance. Ghosts are random and this helps to avoid individuals which win the game by chance. For example, there might be an individual which might always be picking a predefined path. If the random ghost, takes the same path(not coinciding with this individual) in all 5 times, then this individual will be passed on to next generation in "average of 5" but there is lesser chance that ghost will take same path 10 times and so, this individual will lose more number of times and has lesser chance of getting passed on to next generation. So, in the long run, higher n gives better result as it chooses individuals which perform better in all situations. The graph misleads into thinking that lower n gives faster learning initially. This is not true as the best score is the average over n times played of the best individual and it might have won by chance.

Reinforcement Q-Learning

Exact Q-learning was able to perform very well on small mazes. It is able to train in under 100 episodes, and wins virtually always. In this approach, we use the entire board state as data, to learn the Q-matrix. However, it is unable to learn over larger mazes as the number of cases increase. This is because, the algorithm isn't able to generalize learning over multiple different configurations in the grid (for example, escaping ghosts in one part of the grid vs another) into a neat algorithm. Extending the dimensions greatly increases number of possible states, making the Q-matrix very large. It would theoretically be possible to train it to figure the optimal algorithm to escape from any position, but this would take prohibitively long training times. The training is very sensitive to values of alpha, epsilon and gamma as a lot of variables are to be

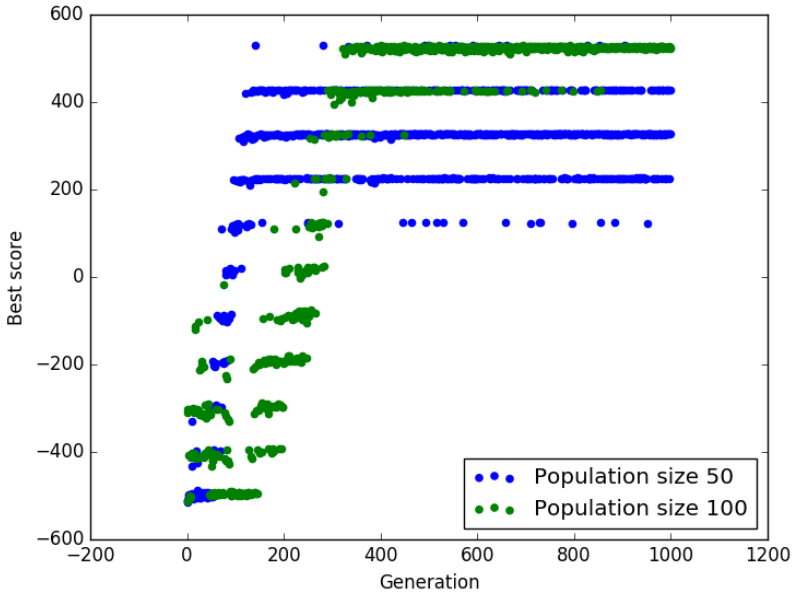

Figure 2: Comparison of Population size

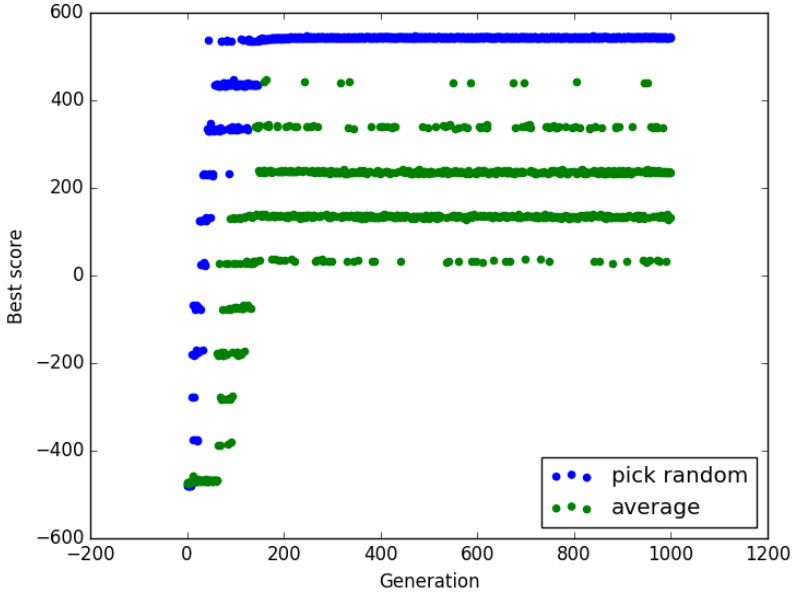

Figure 3: Comparison of crossing - Pick Random vs Average

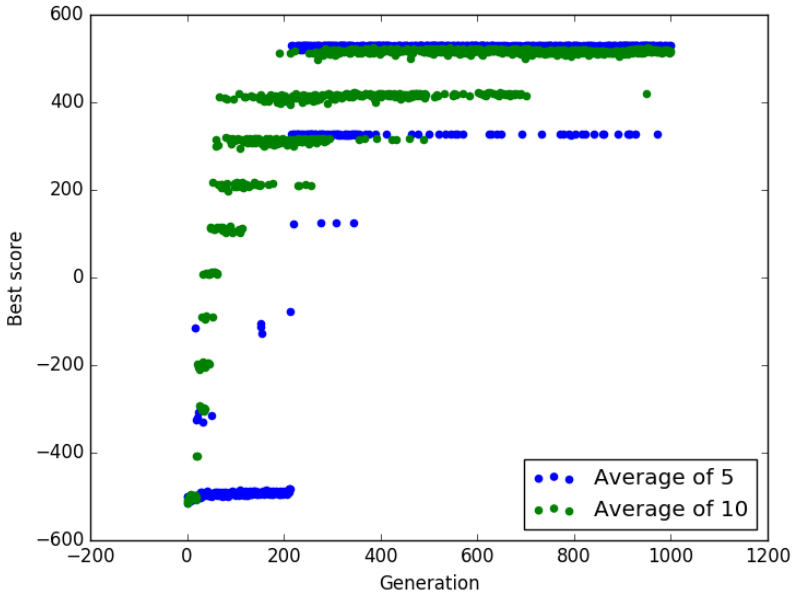

Figure 4: Averaging number comparsion

learnt.

In approximate Q-learning, we are more tolerant of differences in alpha, epsilon and gamma as the number of features to be learned are lesser. Which means, over multiple iterations, it would learn the optimal Q-matrix and be resistant to changes in parameters. In practice, it requires far fewer training episodes to be effectively trained as compared to exact Q-learning.

Alpha decides the learning rate for the algorithm, basically deciding how much weightage should be given to the new information, vs earlier existing info. In this paradigm, it can be thought of as both the learning rate, and th weight decay. Keeping Aplha extremely low means that the algorithm learns very slowly and is unable to converge.On the other hand, keeping alpha very high would make the algorithm sensitive to noise as it disregards earlier learning in favour of current noisy observation. For exact Q-learning, given the number of parameters to be learnt, these considerations are very important in picking optimal value of alpha. For approxiamte Q-learning, given that the heuristic features are already designed in a way to make decision making easier, a wrong decision is very likely to lead to a loss. Hence, we don't see problems with noise on increasing alpha.

Gamma decides how much weight to give already learnt information in the form of already existing Qmax values, vs immediate reward. A low gamma, would mean high emphasis on immediate reward, whereas increasing gamma makes the agent look for long-term rewards. For optimal learning, the ideal gamma must achieve a balance between the 2.

Epsilon decides the classic exploitation vs exploration tradeoff. Higher epsilon means the agent explores the grid much faster, and is important to learn all states, especially at the start. A low epsilon reduces training error, and focuses on the currently estimated best path. Ideal epsilon depends on size of grids, position of walls, etc. This parameter can also be decayed to zero as we become more confident in our guesses. However, for approximate Q-learning, as all parts of the grid don't need to be explored to learn the heuristics, we don't see much variation of score with epsilon.

Features in approximate Q-learning have to be well formulated heuristics. Ideally, a small set of heuristics would be able to perform very well. Approximate Q-learning manages to beat the random ghosts exceptionally well. Against directional ghosts, it is able to play well, but not always win. This is to be expected as even humans using the same heuristics don't always win. Giving the non-scared ghosts in a 1,2, or 3 step radius, allows us to avoid the ghosts very well, with increasing computational complexity, as we increase number of steps. This impact is visible in the game that is played. Giving closest food allows the pacman

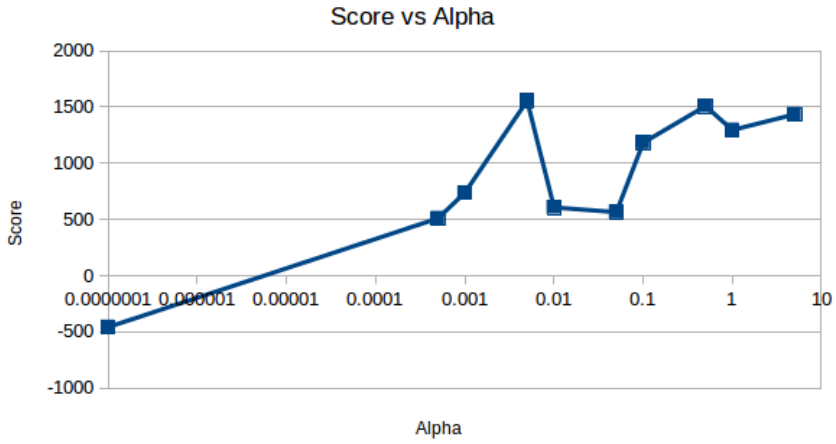

Figure 5: Test Score v/s Alpha

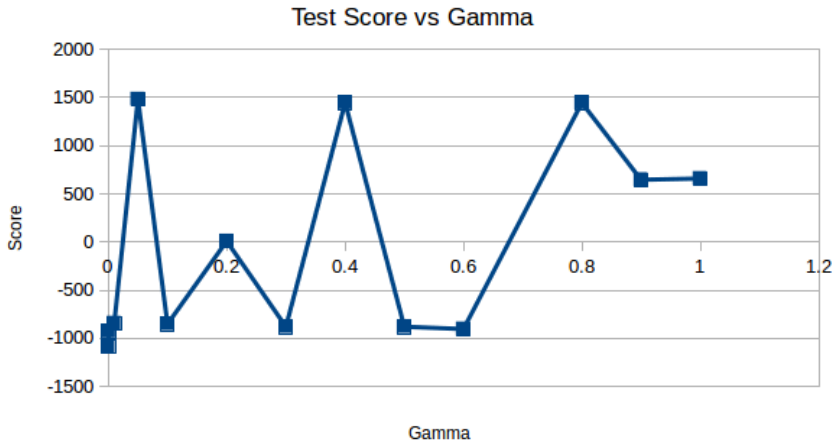

Figure 6: Test Score v/s Gamma

to move towards it and finish the game quickly. Closest capsule is not effective as the game cannot learn from given heuristics that the ghosts get scared on eating it (This is a common problem with Pacman AI). Food particles nearby enables the algorithm to continue eating while it moves. Giving the exact distance of ghosts along each path, can avoid ghosts better. But sometimes leads to stuck states as the Pacman prefers to hide.

Future Work

1. With more hidden layers and changing number of nodes per network in the neural network, we could probably improve the efficiency
2. We could have taken the reward while playing the game itself to update the neural network instead of doing it at the end
3. Similar ideas of genetic algorithm and Reinforcement learning can be applied in other games too

Datasets

Since this project is based on genetic algorithms and reinforcement learning, the algorithm learns by getting output from the game and hence no datasets are required.

References

Pacman Game Source Code
Pacman Reinforcement Q-Learning CS 188 UC Berkeley
Genetic Algorithm - Article
Pac-Man using Genetic Algorithm Report
Playing Atari with Deep Reinforcement Learning
Deep Reinforcement Learning in Pac-man