Application of Artificial Intelligence to Games

Authors:
Nikita Devyataykin, Milo Donnelly, Mateusz Staniszewski, Jean-Francois Ton

Supervisor:
Dr Ben Calderhead

August 2016

Abstract

The first computer took 3 years to construct and arrived in 1946 occupying around 1800ft and weighing 50 tons. In 1986 IBM introduced Deep Blue - a computer that managed to beat Kasparov in Chess a game with over 288 billion different possible positions. Since then capabilities and use of algorithms have increased with exponential speed, with the most recent breakthrough being AlphaGO - a computer designed by Deep Mind that managed to beat a world master Sedol in GO a game googol times more complex than chess. In this project we seek to understand the possibilities of designing algorithms that use artificial intelligence and their applications to various games.

Key Words: Artificial Inteligence, Machine Learning, Reinforcement Learning, MiniMax, Alphe-Beta Pruning, Game Theory, Neaural Networks

Contents

1 Artificial Intelligence

Artificial Intelligence (AI) is a branch of Computer Science and the study of man-made computational devices that act in a manner that could be classified as “intelligent”. Quoting Kurzweil AI is the “art of creating machines that perform functions that require intelligence when performed by people”. The beginning of this field can be tracked back to 1950 where Alan Turing proposed that a machine has the ability to be taught like a child. Publishing the paper “Computer Machinery and Intelligence” A. Turing argued that is a machine could pass the “Turing test” we would have the right to classify the computer as intelligent [1] – the test was passed the first time in 2014 by machine called “Eugene” simulating a 13 year old boy. In present day AI has several applications including medical diagnosis, stock trading, robot control, voice recognition and text translation.

1.1 Neural Networks

Neural Networks is mans attempt of simulating the brain electronically. Our brains are made of 100 billion interconnected neurons, which communicate via electrochemical signals. Attempting to mimic the way neurons work in our brain programmers introduced the artificial neural network (ANN) [2]– “a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs” [8]. Majority of ANNs contain a form of ”learning rule” allowing them to learn by example, as do their biological counterparts; a child learns to recognize a specific animal by observing other examples of that animal. Present ANNs can similarly recognize characters from observing a large amount of handwritten letters, transforming out handwriting into text[9].

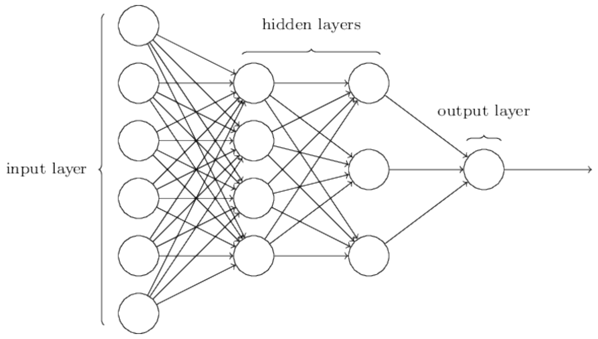


Figure 1. Artifial Neural Networks representation

1.2 Machine Learning

When algorithms and general automation are being discussed we often mention how great it is that a machine can do a boring repetitive job. Machine learning extends these skills and allows the Machine to perform better with experience. This area of science has been around for quite sometime but the power of computers has only recently allowed us to see its implementation in daily life. Starting with your emails filtering spam, your smart-phones voice recognition or text prediction, to the nascent self-driving cars. In general, through iteration of a few examples, computers understand how to optimise and improve models for the future, without any human intervention. Many researchers believe it is also the next step forward for developing a human like artificial intelligence (AI).

2 Game Theory

We will provide a basic introduction to Game Theory which will serve the following purposes:

- Provide a classification system for games.
- Allow us to find a tailored approach to different classes of games.

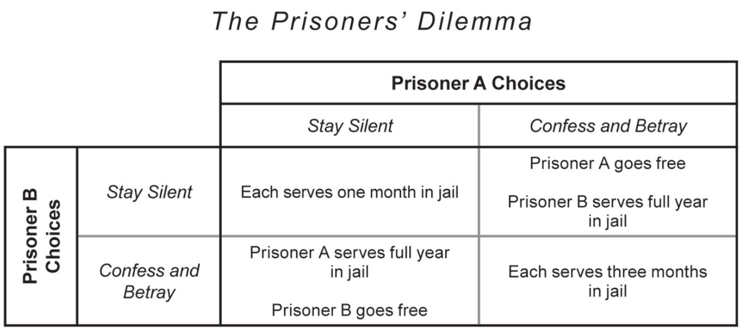
2.1 Zero-Sum Game

An important distinction for us will be that between zero-sum and non-zero-sum games. A zero-sum game is one in which the relative gain of any one player is equal to the relative losses of all other players. The 'net gain' is zero. A simple bet between two people is an example of a zero-sum game: if Player A and Player B both bet £1 on opposite outcomes of a coin-toss then, whatever the realisation, one player will gain £1 and the other will lose £1. This is an important category in game theory since, in a zero-sum game, it is in none of the players' interest to cooperate with any other player. (This links to another distinction between cooperative and non-cooperative games).

Although not a focus of this report, it is to be noted that the birth of Game Theory is attributed to the study of zero-sum games by Von Neumann and his theorem that there exists a mixed-strategy equilibrium in a 2 persom zero-sum game of finite moves. That is to say that each player can follow a (possibly different) strategy in order to minimise their maximum loss. This equilibrium is the same as the Nash equilibrium, from which either player will reduce their gain with any change of strategy.

2.2 Non Zero-Sum Game

A game where a players loss (or gain) does not necessarily result in the opponents gain (or loss). This can be pictured the following way: the sum of winnings and losses of all players do not add up zero. Scenarios of win-win situations are possible, a good example would be the "Prisoners Dilemma".



The Prisoners' Dilemma

		Prisoner A Choices	
		Stay Silent	*Confess and Betray*
Prisoner B Choices	*Stay Silent*	Each serves one month in jail	Prisoner A goes free Prisoner B serves full year in jail
	Confess and Betray	Prisoner A serves full year in jail Prisoner B goes free	Each serves three months in jail

Figure 2. Choices in prisoners Dilemma.

2.3 Other Classes of Games

A **simultaneous** game is one in which all players make their 'move' at the same time, without previous knowledge of any other players' move. On the other hand, a **sequential** game is one in which the current player has some (possibly restricted) knowledge of the choices and moves of the previous players. A sub-classification of sequential games is that of games of **perfect information**, in which the current player has complete, unrestricted knowledge of the previous players' moves, and games of **imperfect information**, in which their knowledge is limited. Tic-Tac-Toe is an example of a sequential game of perfect information. It is easy to see how we must approach these games differently in the context of machine learning.

Combinatorial games are those in which the number of branches of possible states in the 'game tree' becomes very large after a few moves. Chess and Go are both combinatorial games. Such games pose a problem when it comes to strategising with machine learning since it is unfeasible to assess each possible game state for its *reward* (it's capability to lead to a final state of good *value*)

There also exist **solved** games in which a player can guarantee at least a draw if they follow a set 'perfect play' strategy. Games such as Connect4 and Hex are solved games. When using machine learning, one can simply 'teach' the computer how to play perfectly for these solved games. However, the situation with **unsolved** games, such as Chess, is much more complicated and we have to call upon other, more involved methods (reinforcement learning, deep neural networks etc).

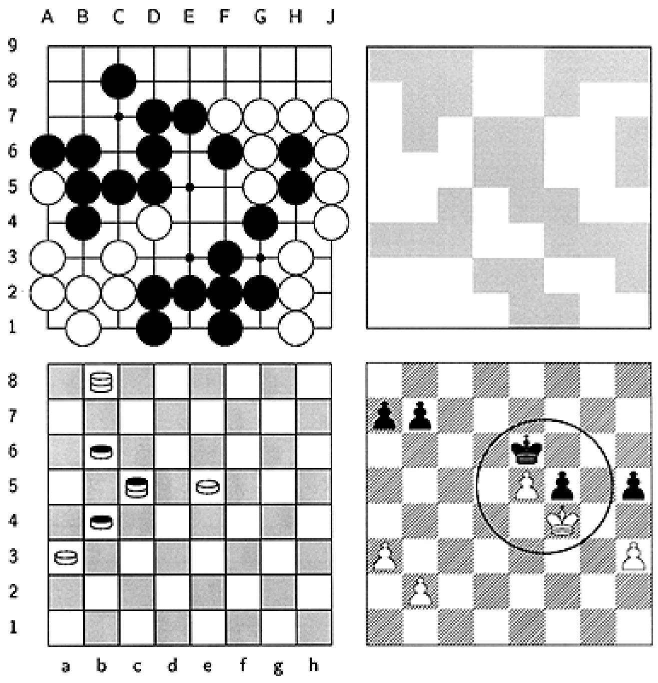


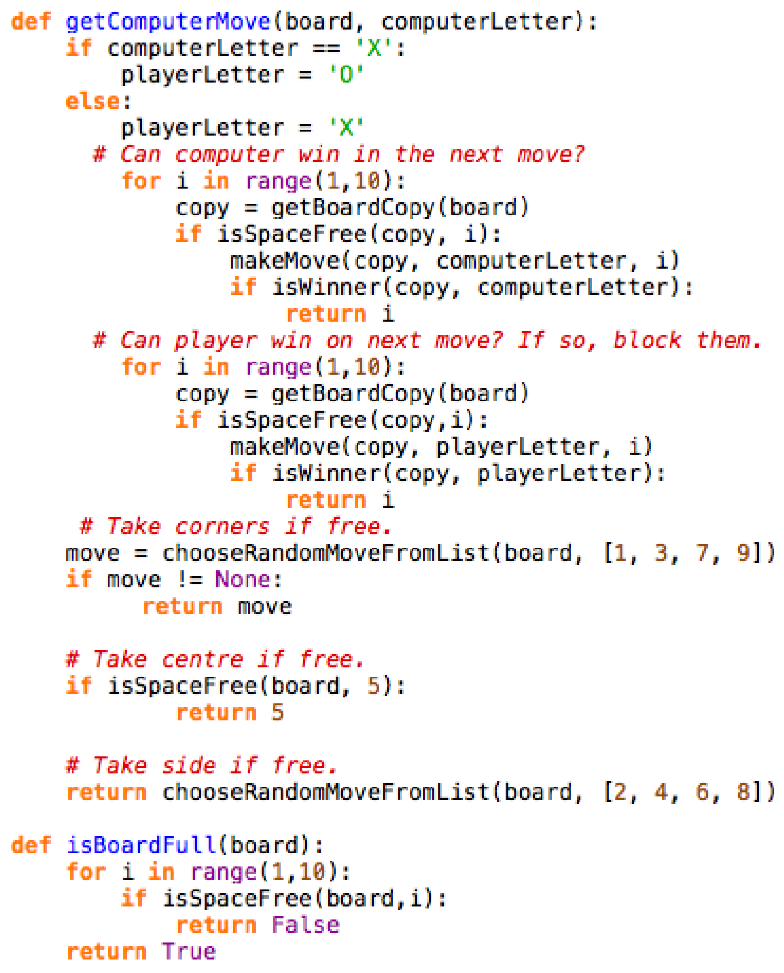
Figure 3. Go and Chess board games and their respective visual representation in the algorithm.

3 Application to Discrete Games

Definition: A Discrete game has finite number of players, moves and possible outcomes. The player chooses his move from a finite set of pure strategies which in theory (may be hard to calculate) have a defined outcome.

3.1 Tic-Tac-Toe

Tic-tac-toe is one of the simplest games that just requires pen and paper, however the small 3x3 board has 765 essential different positions and 26830 different possible games up to rotations and reflections. Players soon understand that ”perfect” play can always result in a draw, here our motivation was to introduce an algorithm that will mimic ”perfect” play and be unbeatable by any player. Playing the game several times you understand that the most strategic starting points is either the centre or the corners, providing this information to the computer we set up the following algorithm which had almost perfect draw convergence rate, with only some minor flaws when the opponent creates a fork movement.



```
def getComputerMove(board, computerLetter):
    if computerLetter == 'X':
        playerLetter = 'O'
    else:
        playerLetter = 'X'
      # Can computer win in the next move?
        for i in range(1,10):
            copy = getBoardCopy(board)
            if isSpaceFree(copy, i):
                makeMove(copy, computerLetter, i)
                if isWinner(copy, computerLetter):
                    return i
      # Can player win on next move? If so, block them.
        for i in range(1,10):
            copy = getBoardCopy(board)
            if isSpaceFree(copy,i):
                makeMove(copy, playerLetter, i)
                if isWinner(copy, playerLetter):
                    return i
     # Take corners if free.
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:
          return move

    # Take centre if free.
    if isSpaceFree(board, 5):
            return 5

    # Take side if free.
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def isBoardFull(board):
    for i in range(1,10):
        if isSpaceFree(board,i):
            return False
    return True
```

In order to avoid the fork cases we could add more instructions to our code, however the aim with machine learning is that we don't do all the work for the computer - we want there to be a learning process. Pursuing this goal, we now introduce the MiniMax algorithm in the next section which avoids the pitfall encountered by the elementary approach.

3.1.1 MiniMax algorthim

Definition: Minimax (sometimes MinMax or MM[*]) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.

Intuitively, given a zero-sum game, we can establish a decision tree (illustrated below for tic-tac-toe), where each layer represents a one players possible moves. Each layer represents a players in alternating fashion. Note that, we will always have to consider a game from one players perspective in order to have some sort of reference. In addition the nodes will be evaluated according to this player gains or losses, i.e. if Player A wins the value of the current state is +1, if Player A ties the value is 0 and if Player A looses the value is -1. Therefore, given that we are in a zero-sum game setting, it is clear that Player A will always try to maximize his nodes(taking the decision that will go to a state with maximal value) and Player B will want to minimize on his nodes (taking the decision to minimize the value).

But how do we set up such a tree so that we can use the MiniMax algorithm?

The way we do it is by propagating the values from the last layer to the second last layer and then from the second last layer to the third last layer and so on. This can easily be done, as the last layer represents the final state of the game, i.e. we know if we won, lost or tied. Before we go into the example of illustrating how one can implement and code up these ideas for a game of tic-tac-toe, we need to introduce the concept of "Breath First Search" in order to navigate efficiently inside the decision tree.

Definition: Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[*]) and explores the neighbor nodes first, before moving to the next level neighbors.

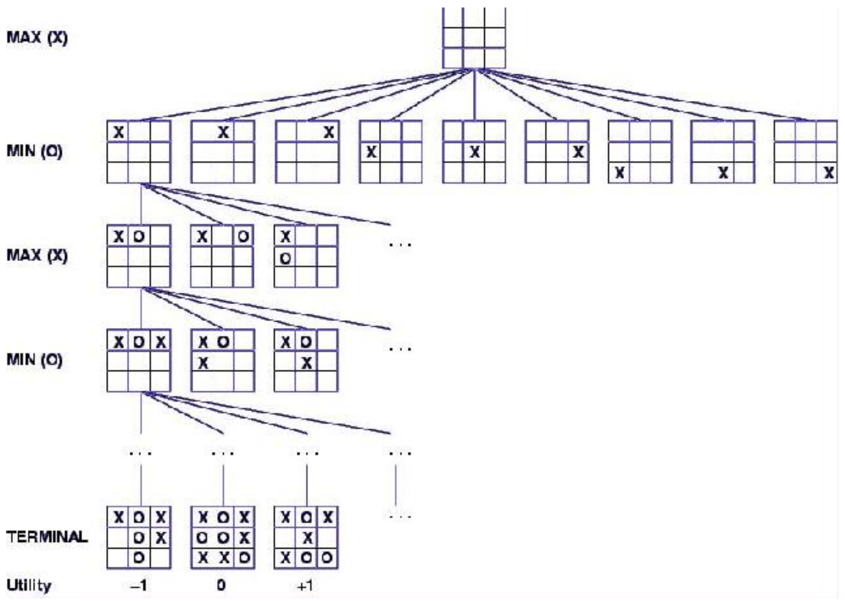
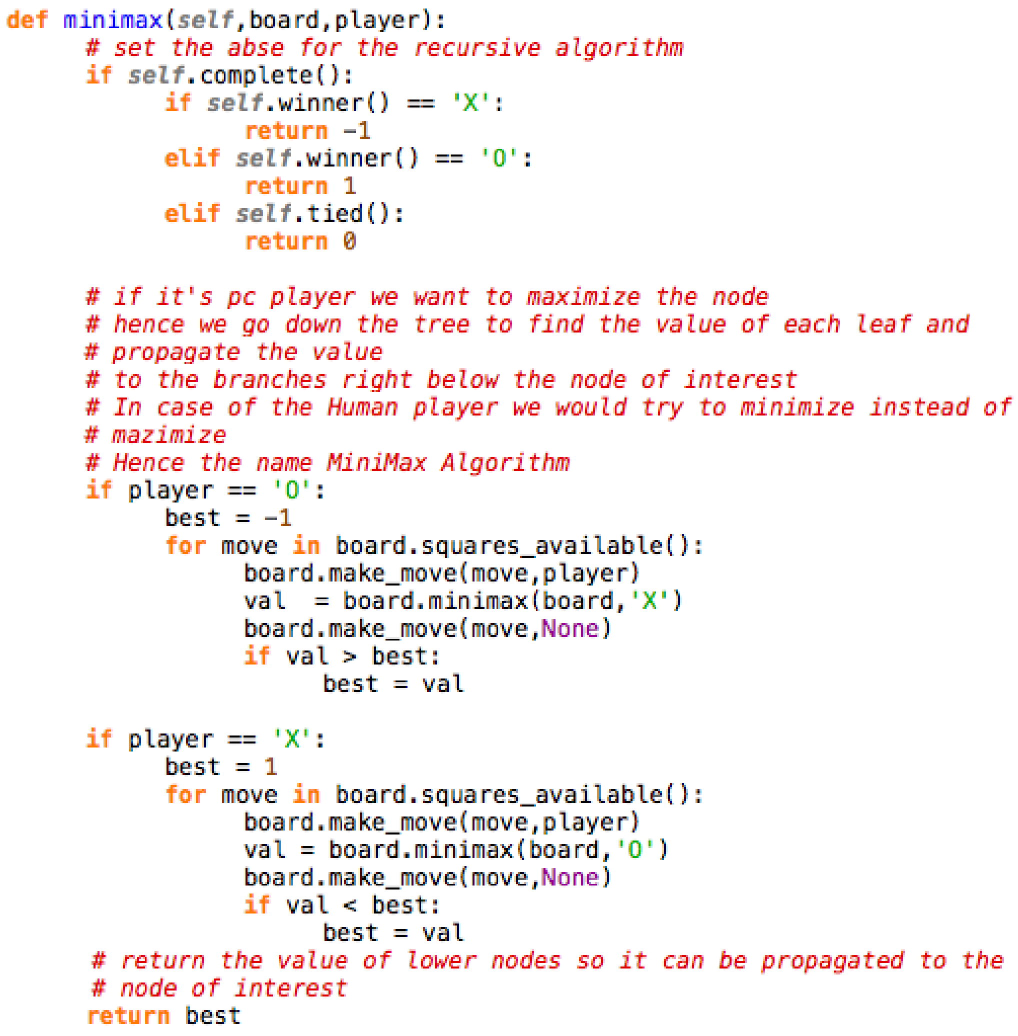


Figure 4. The tree for different scenarios in tic-tac-toe game used in minimax algorithm.

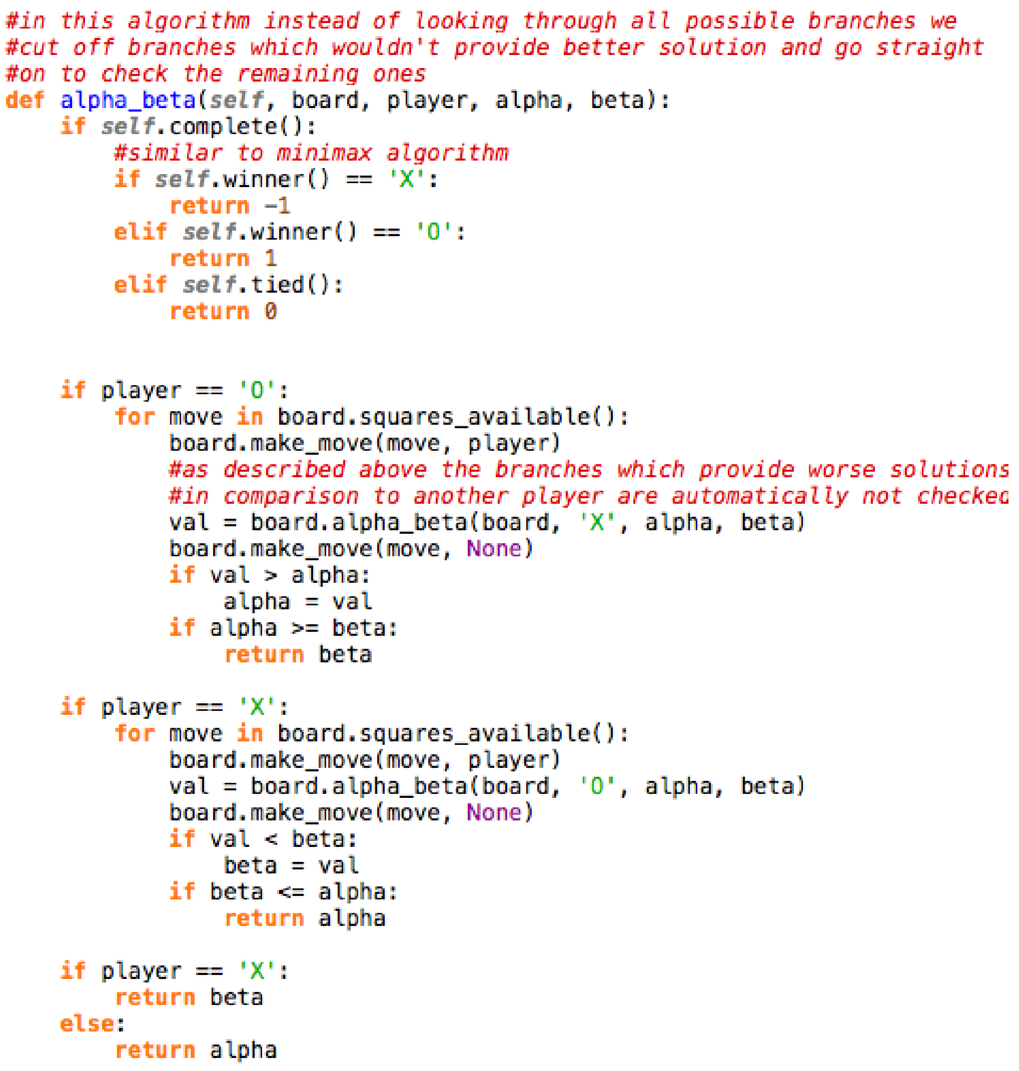
Hence this algorithm will allow us to navigate through the decision tree layer by layer i.e. from one player to the other Having gone through these concepts, we can now implement the Mini-Max algorithm to create an AI that will never lose a game of Tic-tac-Toe.



```
def minimax(self,board,player):
      # set the abse for the recursive algorithm
      if self.complete():
            if self.winner() == 'X':
                  return -1
            elif self.winner() == 'O':
                  return 1
            elif self.tied():
                  return 0

      # if it's pc player we want to maximize the node
      # hence we go down the tree to find the value of each leaf and
      # propagate the value
      # to the branches right below the node of interest
      # In case of the Human player we would try to minimize instead of
      # mazimize
      # Hence the name MiniMax Algorithm
      if player == 'O':
            best = -1
            for move in board.squares_available():
                  board.make_move(move,player)
                  val  = board.minimax(board,'X')
                  board.make_move(move,None)
                  if val > best:
                        best = val

      if player == 'X':
            best = 1
            for move in board.squares_available():
                  board.make_move(move,player)
                  val = board.minimax(board,'O')
                  board.make_move(move,None)
                  if val < best:
                        best = val
      # return the value of lower nodes so it can be propagated to the
      # node of interest
      return best
```

3.1.2 Alpha-Beta-Pruning

The MiniMax algorithm essentially explores parts of the tree that are unnecessary increasing the operation time; we can avoid this by introducing alpha beta pruning. The Alpha Beta algorithm is a significant enhancement of MiniMax. It maintains two values alpha and beta, representing the minimum and maximum possible score by the player. Once a significantly good result is found (a maximum) , the tree search can be terminated as we know we can not find a value above the max node [6].



```python
#in this algorithm instead of looking through all possible branches we
#cut off branches which wouldn't provide better solution and go straight
#on to check the remaining ones
def alpha_beta(self, board, player, alpha, beta):
    if self.complete():
        #similar to minimax algorithm
        if self.winner() == 'X':
            return -1
        elif self.winner() == 'O':
            return 1
        elif self.tied():
            return 0


    if player == 'O':
        for move in board.squares_available():
            board.make_move(move, player)
            #as described above the branches which provide worse solutions
            #in comparison to another player are automatically not checkec
            val = board.alpha_beta(board, 'X', alpha, beta)
            board.make_move(move, None)
            if val > alpha:
                alpha = val
            if alpha >= beta:
                return beta

    if player == 'X':
        for move in board.squares_available():
            board.make_move(move, player)
            val = board.alpha_beta(board, 'O', alpha, beta)
            board.make_move(move, None)
            if val < beta:
                beta = val
            if beta <= alpha:
                return alpha

    if player == 'X':
        return beta
    else:
        return alpha
```

MiniMax poses some restrictions to its application:

- Exponential computation time in large search trees in games such as GO
- Restrictions in Non Zero Sum games

Hence we decided to introduce the Reinforcement Algorithm. Even though it does not give an optimal solution for tic-tac-toe and in many simulations loses to our MiniMax algorithm, it is a good introduction to the field which has numerous other possibilities.

3.1.3 Reinforcement Learning

Reinforcement learning determines the ideal behavior given the environment, in order to maximize performance. It would be too costly to use supervised learning in instances when there's a huge amount of possibilities and cases for the next move, and there exist instances where your move can not be rated until the next move follows, for example in Chess. The most commonly used methods here are trial and error search and delayed reward.

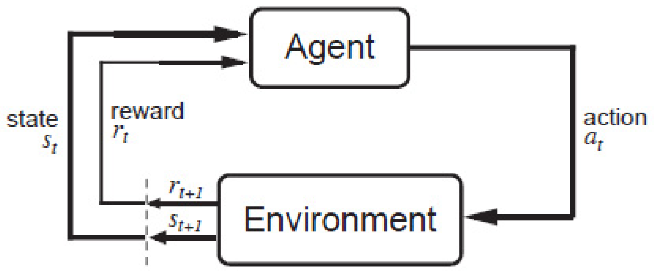


Figure 5.Reinforcement learning, reward scheme as implemented in the algorithm.

The machine can learn the behavior once and for all or update itself as it continues with more operations and understand that certain good actions in the past were rewarding. In this case we are not directly teaching the algorithm but rather telling it when it has made a good decision and where it went wrong, acting as a ”critic”. The ”critic” only comments on what has happened in the past and never informs about anything in advance. In the end the algorithm attempts to analyze the situation by assessing each individual action and checking which of those lead us to the maximum number of good comments, to store in the system and recall them later on. [5].

3.2 Connect 4

Connect Four (Four in a row) is a somewhat similar but a more advanced version of tic-tac-toe. It is played on a 6x7 board and each player chooses a column and places its piece on bottom most cell. The idea is to obtain 4 pieces in a row horizontally, vertically or diagonally. There are over 4 trillion unique ways to fill a board. In case of "perfect" play from both players, the starting player can win every game in 21 moves.

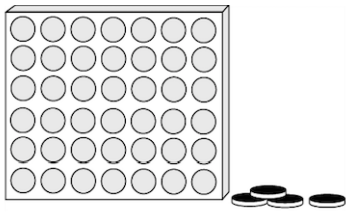
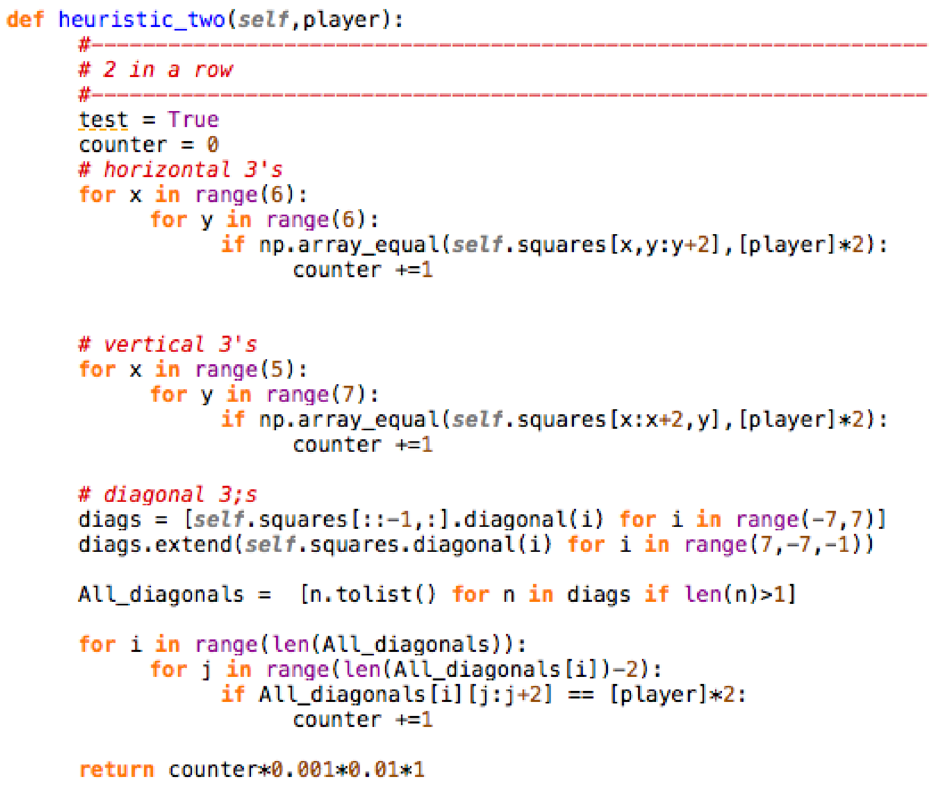


Figure 6. Connect 4 game representation.

The Connect Four game cannot be simply takled by using Alpha-Beta Pruning as was the case with tic-tac-toe. Due to number of possibilities we needed to implement Heuristic Functions which saves the number of computations needed.

3.2.1 Heuristic Function

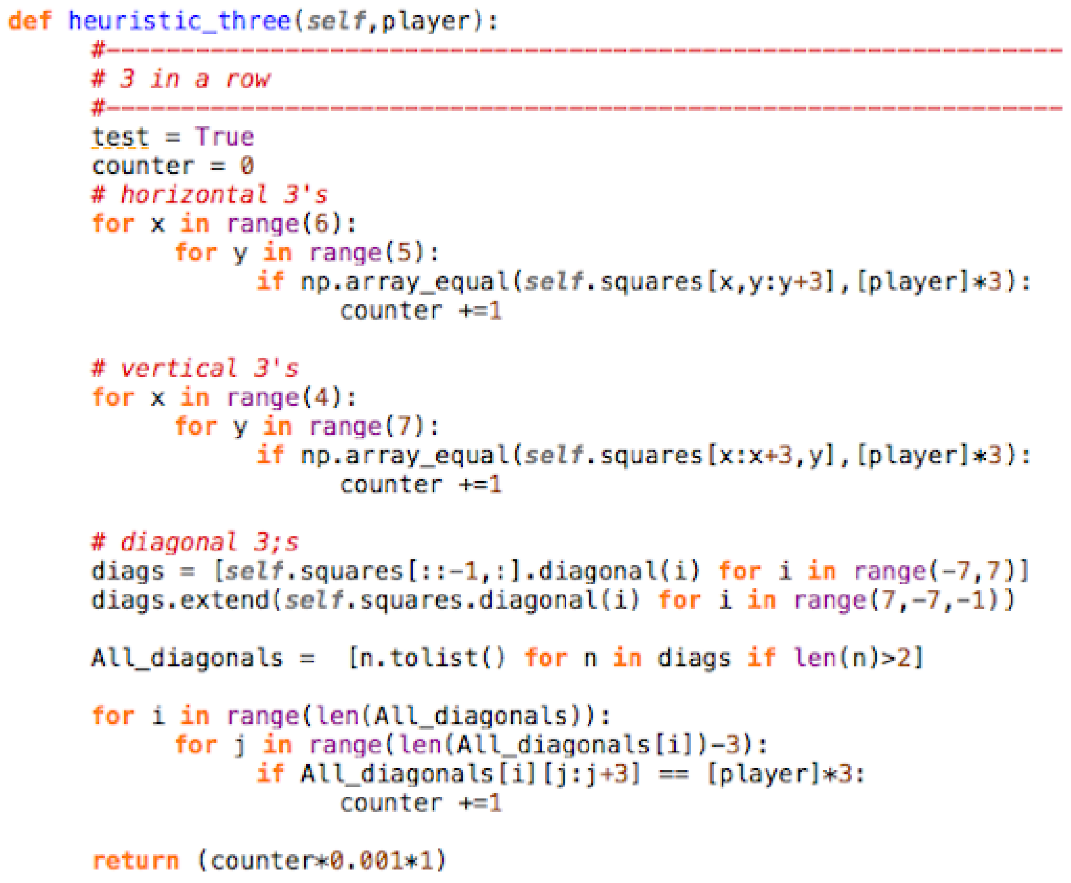
Used to avoid computations that progress in exponential time. If we imagine our search to be a large tree of possible outcomes, the heuristic function allows us to consider this function to a desired depth and by sacrificing completeness it reduces the computational time. It might not always be the best solution however the function does guarantee that it finds a good solution in reasonable time [7].



```
def heuristic_two(self,player):
    #-------------------------------------------------------------------
    # 2 in a row
    #-------------------------------------------------------------------
    test = True
    counter = 0
    # horizontal 3's
    for x in range(6):
        for y in range(6):
            if np.array_equal(self.squares[x,y:y+2],[player]*2):
                counter +=1


    # vertical 3's
    for x in range(5):
        for y in range(7):
            if np.array_equal(self.squares[x:x+2,y],[player]*2):
                counter +=1

    # diagonal 3;s
    diags = [self.squares[::-1,:].diagonal(i) for i in range(-7,7)]
    diags.extend(self.squares.diagonal(i) for i in range(7,-7,-1))

    All_diagonals =  [n.tolist() for n in diags if len(n)>1]

    for i in range(len(All_diagonals)):
        for j in range(len(All_diagonals[i])-2):
            if All_diagonals[i][j:j+2] == [player]*2:
                counter +=1

    return counter*0.001*0.01*1
```

3.2.2 Alpha-Beta Pruning Algorithm

Here we have computed a Connect 4 algorithm with a **heuristic function of debt 1-3** (the larger the depth the harder it is for the opponent to beat the computer).



```
def heuristic_three(self,player):
    #-----------------------------------------------------------------
    # 3 in a row
    #-----------------------------------------------------------------
    test = True
    counter = 0
    # horizontal 3's
    for x in range(6):
        for y in range(5):
            if np.array_equal(self.squares[x,y:y+3],[player]*3):
                counter +=1

    # vertical 3's
    for x in range(4):
        for y in range(7):
            if np.array_equal(self.squares[x:x+3,y],[player]*3):
                counter +=1

    # diagonal 3;s
    diags = [self.squares[::-1,:].diagonal(i) for i in range(-7,7)]
    diags.extend(self.squares.diagonal(i) for i in range(7,-7,-1))

    All_diagonals =  [n.tolist() for n in diags if len(n)>2]

    for i in range(len(All_diagonals)):
        for j in range(len(All_diagonals[i])-3):
            if All_diagonals[i][j:j+3] == [player]*3:
                counter +=1

    return (counter*0.001*1)
```

4 Application to Continuous Games

Definition: A Continuous game is an extension of the notion of discrete games, allowing the player to choose his move from a set of pure strategies $[a, b]$ or $[c, \infty)$.

4.1 Reinforcement Learning

4.1.1 MDP

Before we start with actual reinforcement learning and it's application to the game of 'Pacman' and Crawler(see later), we need to introduce the idea of an **Markov Decision Process** (MDP). Contrary to a deterministic search problems like the Tic-Tac-Toe Problem for example, where each move is free from randomness, i.e. when you put your 'O' in the top left corner the 'O' is going to be there 100% of the time. There is no randomness in the outcome of the action. Hence, in cases where there is randomness, i.e. non-deterministic search problems like 'Pacman' for example, where the movements of the ghosts are unpredictable. In these cases we can no longer use our MiniMax algorithm as there is now a element of chance involved.

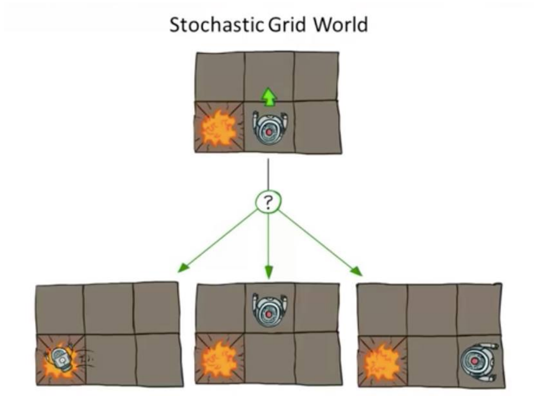


Figure 7. GridWorld game visual representation with different scenarios.

The way we define a MDP is as follows:

1. A set of state s ∈ S
2. A set of actions a ∈ A
3. A transition function $T(s, a, s')$, which represents the probability to move to state s' from s under the action a
4. A reward function $R(s, a, s')$, which represents the reward of moving to state s' from s under the action a
5. A start state and maybe a terminal state(end state)

The following picture will illustrate the the idea of a MDP. Note : the triangles represent decision nodes whereas the circles represent chance nodes, i.e. an action a from state s might not take the desired action to land on s'.

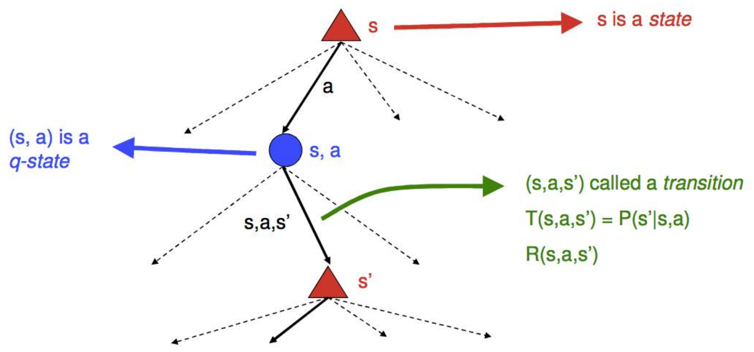


Figure 8.Markov Decision Process represenation with associated functions.

Example : HIGH - LOW CARD GAME Rules:

1. There are 3 types of cards 2,3,4
2. There are twice as many 2's in the infinite deck
3. After a card has been shown you have to say either 'high' or 'low'
4. If you are right you get the points on the card that will be shown
5. In case of a tie, no gains and the game ends when you are wrong
6. First card that is revealed is 3

MDP setting:

1. States : 2,3,4, finished
2. Actions : 'high' and 'low'
3. Transition Model : T(s,a,s'):
   - T(4,'low',4) = 1/4
   - T(4,'low',3) = 1/4
   - T(4,'low',2) = 1/2
   - T(4,'low',finished) = 0
   - T(4,'high',4) = 0
   - T(4,'high',3) = 0
   - T(4,'high',2) = 0
   - ...
4. Rewards : R(s,a,s'): The value of s' if s $\neq$ s'
5. Start state : 3

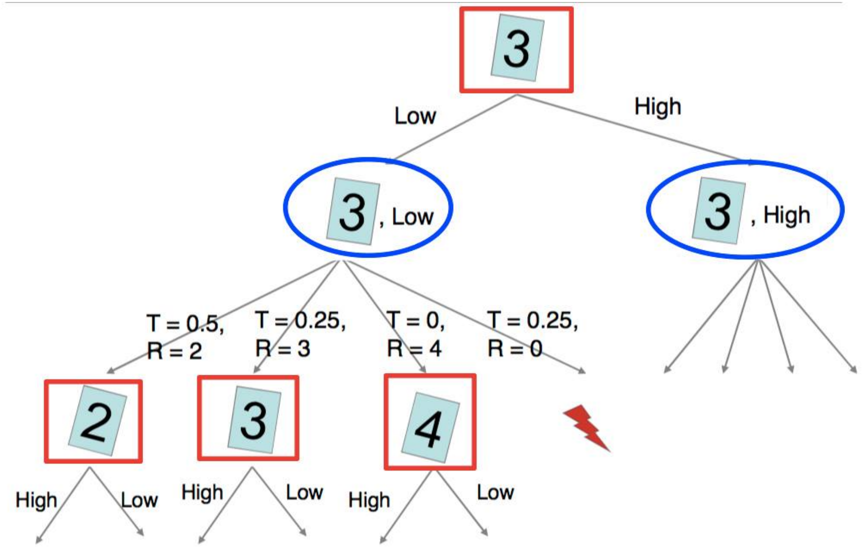


Figure 9. Associated decision tree for Markov Process.

Hence, it become clear that we can not use the conventional method of search trees anymore, not even 'Expectimax search' would help, which takes into account randomness, by choosing the action that has highest expected rewards on the chance nodes, i.e. circles in the graphs. Expectimax would break in this case as in the above example,the game could potentially go on forever. MDP's simplify this problem,as they only take the states from the set S into account and assign a policy over them.

Next, in order to solve these MDP's, we need the notion of policies. In deterministic problem, we were able to pre-plan the whole game ahead of time as there was no randomness in the outcome of the actions in the game. Now however, this is no longer possible. Hence we need a policy π ,which tells us what action to take in each possible situation of the game. We denote the optimal policy, i.e.the policy that will return the highest utility to the player π^*. In reinforcement learning this is going to be key, as we are basically going to train our agent to find the optimal policy over loads of training iterations of the game.

Lastly, we need to introduce the idea of discounting rewards. Question : Imagine you can get 1 million pounds right here, right now, or you can get 1 pound over the next 1 million days. Which option would you prefer? Obviously, the first one is more reasonable, but for the current setting both options would be valued the same. Hence, we introduce a γ which is going to act as a discount factor, i.e later rewards are going to be worth less than more recent ones. More intuition on this topic will be provided when we introduce the Bellman equations.

4.1.2 Value Iteration and Bellman equation

First of all we need to introduce some notation:

1. $V^*(s)$ = expected utility starting from s and acting optimally
2. $Q^*(s,a)$ = expected utility starting from s and taking action a and thereafter action optimally. The q-states represent an intermediate state between state s and landing state s' (see picture above)
3. $\pi^*(s)$ = optimal action from state s
4. $V^*_k(s)$ = expected utility starting from state s and acting optimally considering only the next k time steps. i.e. as k $\longrightarrow \infty$ we obtain the optimal value

Thus, using these notation we can finally formalize the Value iteration algorithm/ Bellman equation:

$$V^*(s) = max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a) + \gamma V^*(s)]$$

Hence :

$$V^*(s) = max_a \sum_{s'} T(s,a,s')[R(s,a) + \gamma V^*(s)]$$

Intuitively $V^*(s)$ is nothing but maximum over all the actions from state s of the the weighted average of the potential rewards of that action. In addition the future rewards are being discounted by a factor of γ

This recursive algorithm hence allows us to update the value of each of the state in the environment, given that we know the transition and the reward function, which in itself is a huge restriction. We are going to solve in the next part. Before that, there we still need to introduce policy into these equations, as they are going to be crucial in the reinforcement learning. Another method of updating values of each state is called policy iteration.

So let's first define the value of a state s under the policy π to be:

$$V^\pi(s) = \sum_{s'} T(s,\pi,s')[R(s,\pi) + \gamma V^\pi(s)]$$

And the way we evaluate these values of each of these states is by using policy evaluations which is defined by :

$$V^\pi(s)_0 = 0, \text{bottom of the tree(leaf node)}$$

$$V^\pi_{k+1}(s) = \sum_{s'} T(s,\pi,s')[R(s,\pi) + \gamma V^\pi_k(s)]$$

As one can see, we are recursively update until we reach the state s. This is also a linear system and can hence be solved quite easily.

Policy iteration is very similar to value iteration. The only difference is that we freeze the policy, iterate until convergence and then update the policy as follows:

Policy evaluation : Fixed policy, until convergence of the values

$$V_{i+1}^{\pi_k}(s) = \sum_{s'} T(s, \pi, s')[R(s, \pi) + \gamma V_i^{\pi_k}(s)]$$

Policy iteration : find the best action according to the one step-look ahead:

$$\pi_{k+1}(s) = argmax_a \sum_{s'} T(s, a, s')[R(s, a) + \gamma V_k^{\pi}(s)]$$

4.1.3 Comparasion between Value Iteration and Policy Iteration

Value iteration
Updates Utilities/Values of each state as well as the policy at each iteration of the algorithm

Policy iteration
Updates the Utilities/Values of each state for a fixed policy π at each iteration of the algorithm and only occasionally updates the the policy
Convergence Both Algorithms converge to the true values (not proven in this report)

4.1.4 Reinforcement learning

Now that we have cover MDP's it is finally time to tackle Reinforcement learning. In reinforcement learning we still assume an MDP with the following properties:

1. A set of state s $\in$ S
2. A set of actions a $\in$ A
3. A transition function $T(s, a, s')$, which represents the probability to move to state s' from s under the action a
4. A reward function $R(s, a, s')$, which represents the reward of moving to state s' from s under the action a

And we are looking for the optimal policy π^*. However, there is a twist in this scenario. We do not know $R(s, a, s')$ or $T(s, a, s')$ in reinforcement learning, i.e. we do not know which states are good or bad and we must actually try all the states to learn from experience !!!. An example of reinforcement learning would be the walking robot dog, an experiment by Kohl and Stone in 2004, where they were trying to teach a robot dog to get from point A to B as fast as possible, given movement restrictions:

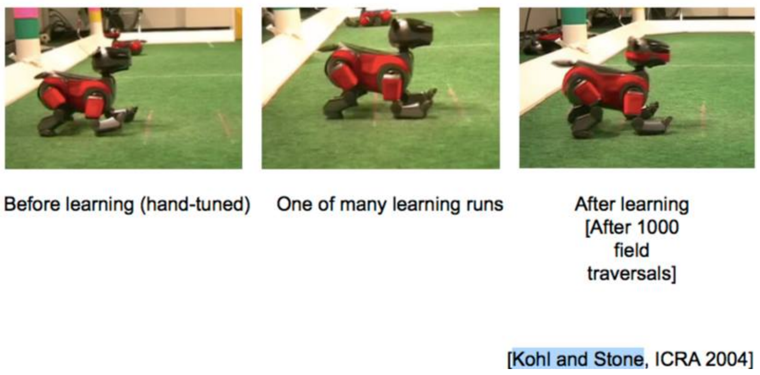

Before learning (hand-tuned)
One of many learning runs
After learning [After 1000 field traversals]
[Kohl and Stone, ICRA 2004]

Figure 10. Learning to walk by a robot after reinforcement learning.

There are 2 main branches of reinforcement learning, **Model-Based RL**, where one creates a model for $R(s, a, s')$ and $T(s, a, s')$ and treats it as a normal MDP or Model-Free-Based RL, which by-passes the need of $R(s, a, s')$ and $T(s, a, s')$, using methods like : **Temporal difference learning** and **Q-learning**. In the following report we are going to focus on model-free-based RL and we are going to take a closer look at temporal difference and Q learning.

4.1.5 Temporal Difference Learning/Q-learning

Let's first write down the update algorithm as it will be easier to explain:

$$sample/experience = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

$$V^{\pi}(s) \longleftarrow V^{\pi}(s) + (1 - \alpha)(sample - V^{\pi}(s))$$

Observation: We update V(s) each time we experience (s,a,s',r) and note that s' that are more likely to occur given a $\pi(s)$ will contribute more to the state s, which makes sense. In addition, the policy π is fixed and V(s) becomes a running average, being constantly updates by the experiences (s,a,s',r).

4.1.6 Example of temporal difference learning on Tic-Tac-Toe

(Credit goes to gabrieledcjr : github : https://github.com/gabrieledcjr/TicTacToe)
In order to apply TD-learning on the game of Tic-Tac-Toe, we need to first of all, define the setting for the learning process. This process will involve no prior knowledge of the game and will purely rely on playing against itself for about 200000 times, in order to learn the optimal policy. We start off by creating a data file called tictactoe.dat, where we have all the board configuration saved and value each board either 0, 1, -1, 1/2 for draw, win, loss, no winner yet respectively. The format use is the following: for example '1:2:1:2:1:2:1:0:0 1.0', where 1 represents the first player and 2 the second. Each value in the sequence represents a move from the respective player in the location of the board and the final number represents the value of the state if this board configuration. In this example we see that player 1 won in the first

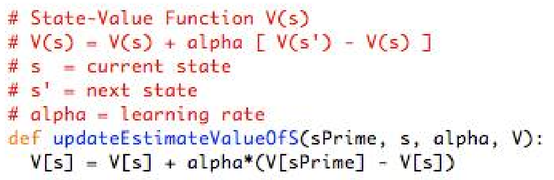
column. Note: This whole setting can be seen as the grid world where we basically have values for each state and an action/move will direct us to another state with specific value and so on. Now that we have this out of the way we can start focusing on how we can apply TD-learning to this problem. (For full code please refer to appendix or https://github.com/gabrieledcjr/TicTacToe) First of all we need a function that will allow us to take the greedy move ϵ times.

```
part1 of the code
'''

def greedyMove(states, V, player):
  maxVal = 0
  maxIndex = 0

  # looking at potential next moves
  nextMoves = getListOfBlankTiles()
  # choosing one of the moves as base case in order to enter the loop
  boardIndex = nextMoves.pop()
  board[boardIndex] = player
  maxIndex = states.index(board)
  maxVal = V[maxIndex]
  board[boardIndex] = 0

  # looping through all the potential moves and extracting the
  # boardIndex (the optimal/ greedy move) as well as the maxIndex(row number in
  for i in nextMoves:
    board[i] = player
    idx = states.index(board)
    if V[idx] > maxVal:
      boardIndex = i
      maxIndex = idx
      maxVal = V[idx]
    board[i] = 0

  return boardIndex, maxIndex
```

Next we need a function that can read in the **'.dat file'** and update the file after each game with the new values of the states:

```
'''
 def saveStatesToFile(filename, states, V, totalStates):
     # opening the data file in the 'write' option
     fp = open(filename, "w")
     for index in range(0, totalStates):
         # overwriting the old file with the new values
         state_string = ':'.join(map(str,states[index]))
         value_string = str(V[index])
         fp.write("%s %s\n" %(state_string, value_string))
     fp.close()

 def loadStatesFromFile(fp, states, V):
     total = 0
     while True:
         line = fp.readline()
         if line == "":
             break
         first_split = line.split(' ')
         state = map(int,first_split[0].split(':'))
         value = float(first_split[1])
         states.append(state)
         V.append(value)
         total = total + 1
     return total
```

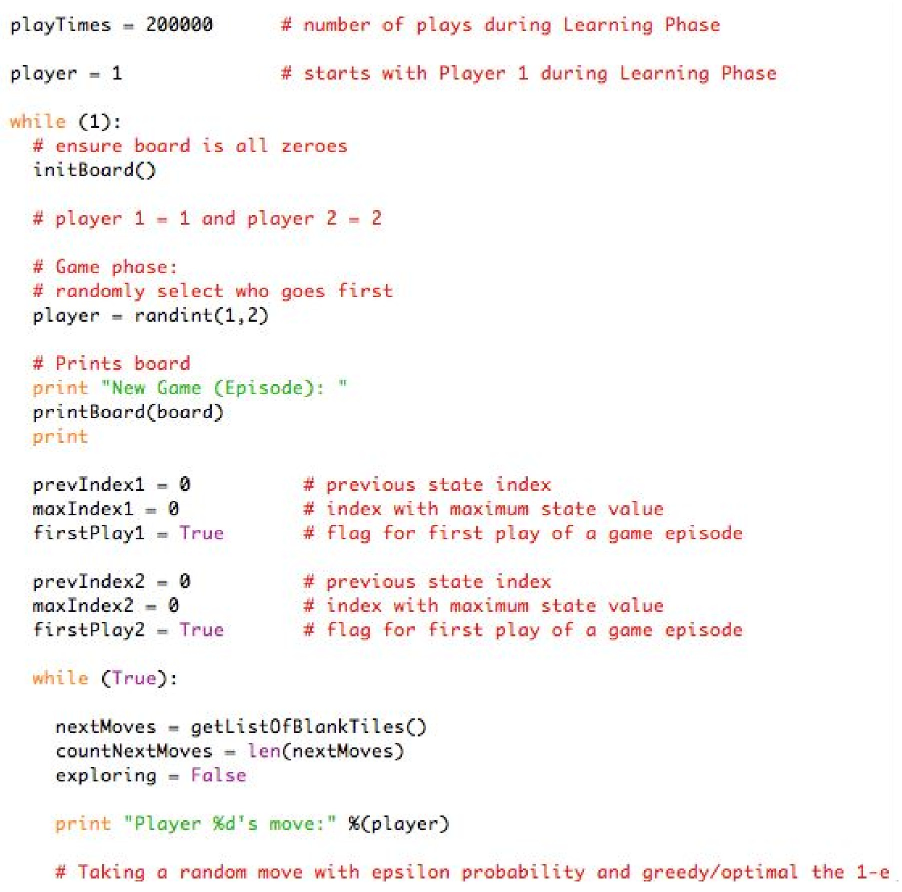
After this, the probably most important bit is the TD-learning update, which is this simple code allowing us to update the values after each action



```
# State-Value Function V(s)
# V(s) = V(s) + alpha [ V(s') - V(s) ]
# s  = current state
# s' = next state
# alpha = learning rate
def updateEstimateValueOfS(sPrime, s, alpha, V):
  V[s] = V[s] + alpha*(V[sPrime] - V[s])
```

Having set up the algorithm it is now time to teach the program the optimal policy, by letting it play against itself. The way we achieve this is by actually have to tictactoe.dat files one for player and and one for player 2. in every game these 2 play against each other, they will follow their own policy which is given by their respective .dat file. The winner of each game (out of 200000) updates their policy according to the TD-learning algorithm, explained above. Hence after 200000 iterations the .dat files for both player will have drastically changed from the initial file as it should have now converged to the optimal policy.

Note: In order to obtain the optimal policy as fast as possible, We have first of all increased the exploration and learning rate in the beginning, in order for it to explore the as many states as possible. Afterwards, we steadily decreased these 2 parameters 0, as we do not want random moves in our optimal policy.



```
playTimes = 200000      # number of plays during Learning Phase

player = 1              # starts with Player 1 during Learning Phase

while (1):
  # ensure board is all zeroes
  initBoard()

  # player 1 = 1 and player 2 = 2

  # Game phase:
  # randomly select who goes first
  player = randint(1,2)

  # Prints board
  print "New Game (Episode): "
  printBoard(board)
  print

  prevIndex1 = 0          # previous state index
  maxIndex1 = 0           # index with maximum state value
  firstPlay1 = True       # flag for first play of a game episode

  prevIndex2 = 0          # previous state index
  maxIndex2 = 0           # index with maximum state value
  firstPlay2 = True       # flag for first play of a game episode

  while (True):

    nextMoves = getListOfBlankTiles()
    countNextMoves = len(nextMoves)
    exploring = False

    print "Player %d's move:" %(player)

    # Taking a random move with epsilon probability and greedy/optimal the 1-e
```

So finally, after all this training process we have now discovered the optimal policy for tic-tac-toe, without actually telling it how the game works. All we told the algorithm was whether a state is good or bad. This is very similar to the MiniMax algorithm but as mentioned in the MDP section we cannot use MiniMax in non deterministic searches. TD-learning however will still work. We are going to illustrated this later on na example with Pacman, where we are going to use a small variation of TD-learing called Q-learning.

However, the problem with TD value learning is that it is basically a model-free policy evaluation method and just gives you the values at each state according to the policy. However, if we want to turn these values into a new policy, we can revert back to the Q-Values we discussed a while back, which in short represent the expected utility of a state s after taking the action a and thereafter acting optimally. Hence the idea is, instead of updating the V(s), we are going to update Q(s,a), which is called Q-learning. N.B. : Q-learning is basically TD learning, but instead of updating the utility/value of a state s, it also takes into account the action at each state, hence allowing us to determine which action will lead to the highest reward/utility.

Q-learning is very similar to to TD learning with only minor tweaks:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*(s',a')]$$

$$sample/experience = R(s,a,s') + \gamma max_{a'} Q(s',a')$$

Hence the update is the following:

$$Q(s,a) \longleftarrow Q(s,a) + (1-\alpha)(sample - Q(s,a))$$

and the **optimal policy** is obtained after multiple iterations of the updates by:

$$\pi(s) = argmax_a Q^*(s,a)$$

i.e. taking the action with the hightest utility/reward

4.2 Q-learning Properties

1. Q-learning will converge to the optimal policy
2. Convergence will heavily depend on your alpha/learning rate. Ideally you want it to decrease over time as the policy is converging
3. It does not matter how you choose your actions when training. The algorithm will learn the optimal policy without even following it.

The following picture should well illustrate the what Q-values are and how they are used to determine the optimal policy after a few iterations of the algorithm.

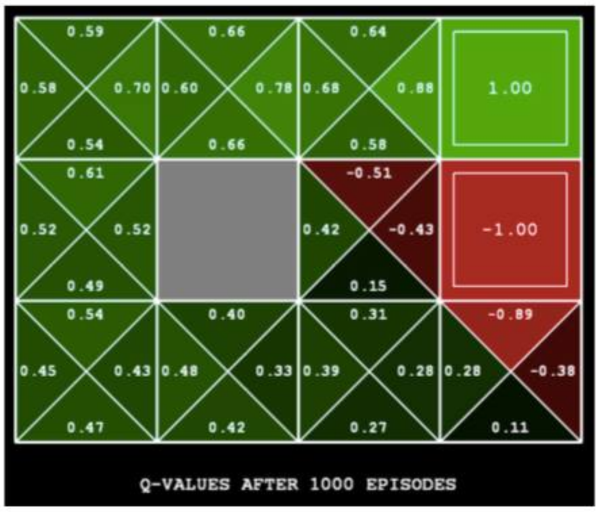


Figure 11. Q values grid associated for GridWorld game.

One last thing we need to discuss before we jump into some examples is **"Exploration vs Exploitation"** during the training of your agent. Exploration refers to taking an action whose utility is not yet fully explored, whereas Exploitation is taking the current optimal action, i.e. argmax of the Q(s,a). It is very important to have a balance between Exploration and Exploitation as too much exploration will slow down convergence by a lot, as it will have to go through way too many scenarios before converging to the optimal policy and too much exploitation will blind sight you in a way that you never explore better options than your current one. You will never discover if there are better methods, as you only stick to the action you initially consider to be the best.

One way of controlling Exploration vs Exploitation is by using the very simple $\epsilon - greedy$ method, which basically just says : take a random action ϵ of the time and the currently optimal action 1-ϵ of the time.This means that ideally we want ϵ to be quite big in the beginning of the training sessions and to decrease as we get closer to convergence. When then using the trained data we obviously put ϵ to zero to it plays the optimal policy.

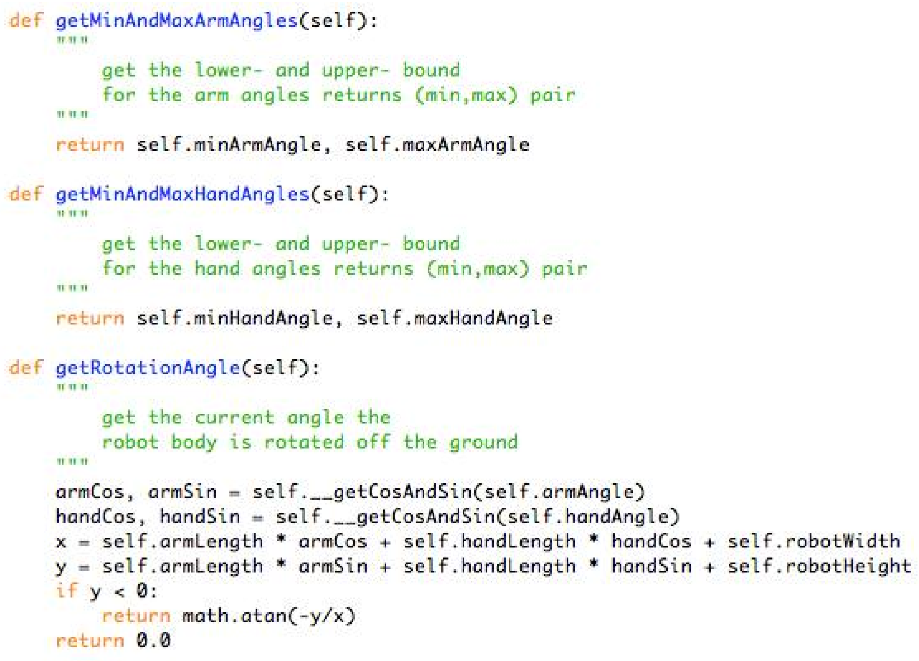
In the project the techniques from previous section were used following CS188 course offered by UC Berkeley [11]. The tasks were further enhanced in accordance to needs of the project.

4.3 Crawler

Crawler is 'robot machine' which consists of the body and one arm, which drags the body through the floor. For the optimal speed of how the body is dragged two angles need to be defined at any given time and space. This is rather a complicated task for which reinforcement learning can be used to make the crawler find optimal values of the angles. For the project the GUI was used to show how Crawler is moving.

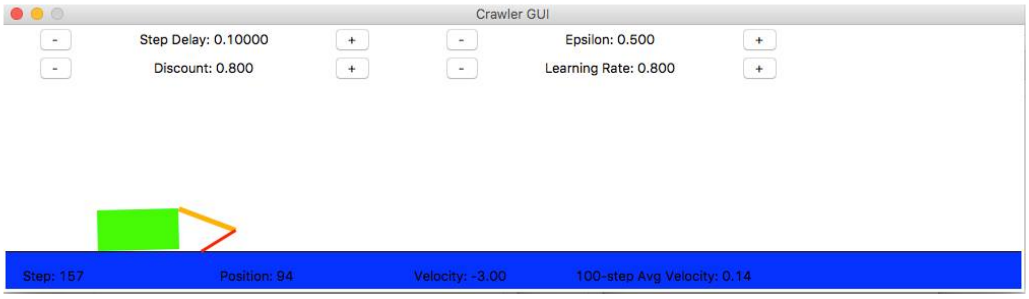
As the game is continuous the algorithm could be developed so it's dynamically changing the pace of how fast the Crawler learns, and wether it chooses to explore or exploit (as explained in previous section). There is also an option of choosing the discount rate, which is the importance assigned to the most recent steps. Additionally, it displays the average speed of the crawler and how many 'steps' did it take to reach the particular state.

Here is a screenshot of the code responsible for the angles control.



```
def getMinAndMaxArmAngles(self):
    """
        get the lower- and upper- bound
        for the arm angles returns (min,max) pair
    """
    return self.minArmAngle, self.maxArmAngle

def getMinAndMaxHandAngles(self):
    """
        get the lower- and upper- bound
        for the hand angles returns (min,max) pair
    """
    return self.minHandAngle, self.maxHandAngle

def getRotationAngle(self):
    """
        get the current angle the
        robot body is rotated off the ground
    """
    armCos, armSin = self.__getCosAndSin(self.armAngle)
    handCos, handSin = self.__getCosAndSin(self.handAngle)
    x = self.armLength * armCos + self.handLength * handCos + self.robotWidth
    y = self.armLength * armSin + self.handLength * handSin + self.robotHeight
    if y < 0:
        return math.atan(-y/x)
    return 0.0
```

Initial the crawler looks like this:



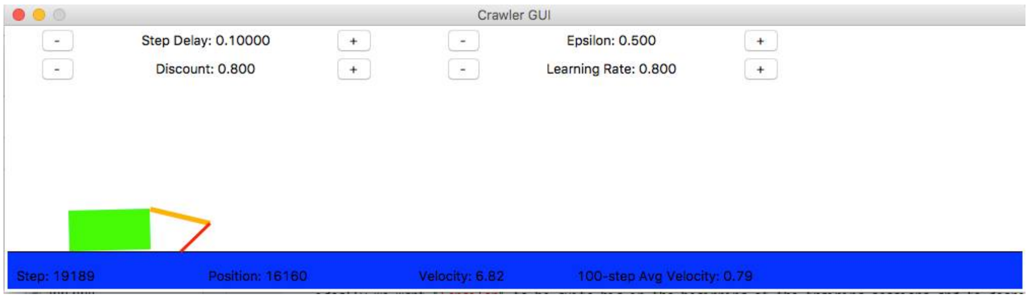
Step Delay: 0.10000
Discount: 0.800
Epsilon: 0.500
Learning Rate: 0.800

Step: 157
Position: 94
Velocity: -3.00
100-step Avg Velocity: 0.14

Figure 12. The Crawler movement initially.

It's initial speed is around 0.2 (average of 100 steps). After leaving the algorithm training for 2 hours, major improvememnts can be noticed. The speed increased 5-fold to around 1 (average of 100 steps).

After leaving the alogrithm for one full night - **10 hours** in total, the Crawler reaches near perfection with very high speed of movements.



Step Delay: 0.10000
Discount: 0.800
Epsilon: 0.500
Learning Rate: 0.800

Step: 19189
Position: 16160
Velocity: 6.82
100-step Avg Velocity: 0.79

Figure 13. The Crawler movemements after 10 hours of training.

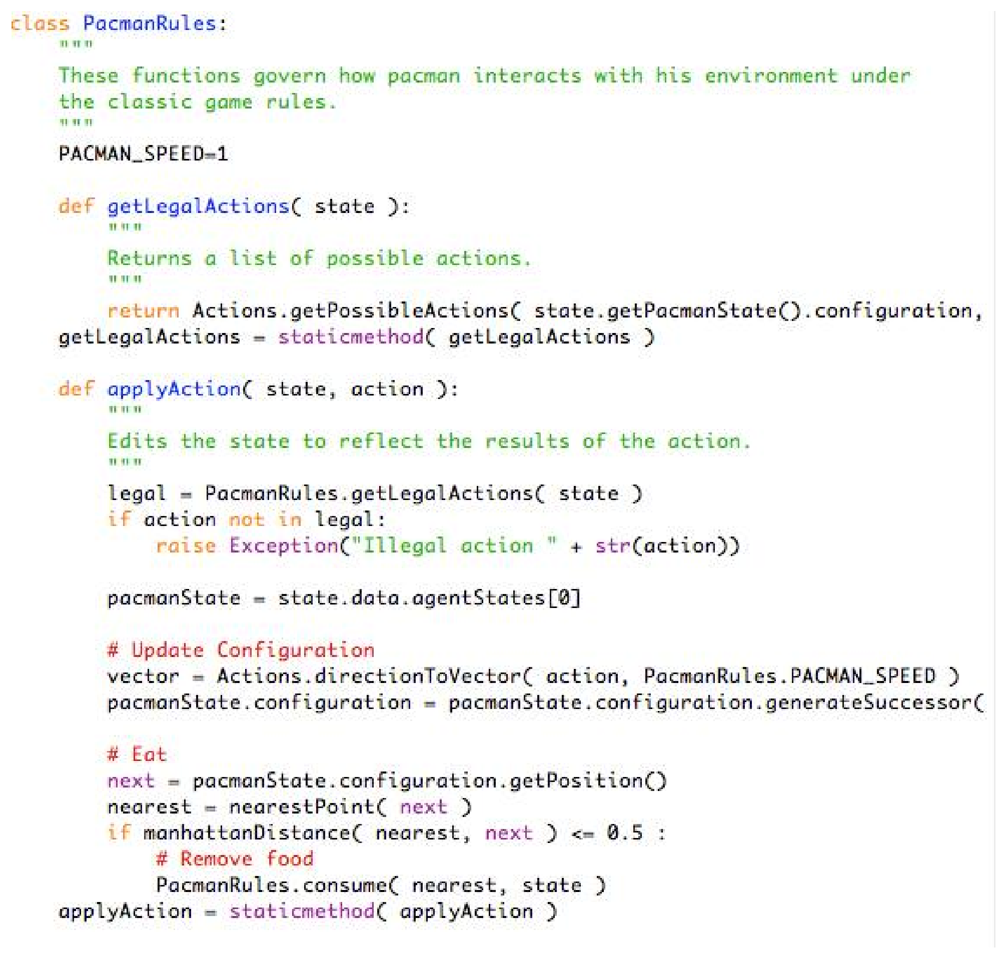
The same algorithm can be adjusted to also work on another game - Pacman.

4.4 Pacman

Pac-Man is an arcade game developed by Namco and first released in Japan in May 1980. It was created by Japanese video game designer Toru Iwatani. It was licensed for distribution in the United States by Midway and released in October 1980. Immensely popular from its original release to the present day, Pac-Man is considered one of the classics of the medium, and an icon of 1980s popular culture. Upon its release, the game—and, subsequently, Pac-Man derivatives—became a social phenomenon that yielded high sales of merchandise and inspired a legacy in other media, such as the Pac-Man animated television series and the top-ten hit single "Pac-Man Fever". Pac-Man was popular in the 1980s and 1990s and is still played in the 2010s.[10]

Pacman AI will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter testing mode. When testing, Pacman's self.epsilon and self.alpha will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default.

Here is the example of bit of the code used for Pacman rules.



```
class PacmanRules:
    """
    These functions govern how pacman interacts with his environment under
    the classic game rules.
    """
    PACMAN_SPEED=1

    def getLegalActions( state ):
        """
        Returns a list of possible actions.
        """
        return Actions.getPossibleActions( state.getPacmanState().configuration,
    getLegalActions = staticmethod( getLegalActions )

    def applyAction( state, action ):
        """
        Edits the state to reflect the results of the action.
        """
        legal = PacmanRules.getLegalActions( state )
        if action not in legal:
            raise Exception("Illegal action " + str(action))

        pacmanState = state.data.agentStates[0]

        # Update Configuration
        vector = Actions.directionToVector( action, PacmanRules.PACMAN_SPEED )
        pacmanState.configuration = pacmanState.configuration.generateSuccessor(

        # Eat
        next = pacmanState.configuration.getPosition()
        nearest = nearestPoint( next )
        if manhattanDistance( nearest, next ) <= 0.5 :
            # Remove food
            PacmanRules.consume( nearest, state )
    applyAction = staticmethod( applyAction )
```

After running 2000 training games Pacman algorithm was challenged with 100 games. It won 84 of those games. If the algorithm was left for longer then it would make the AI even better. 2The graphics attached include GUI for Pacman.

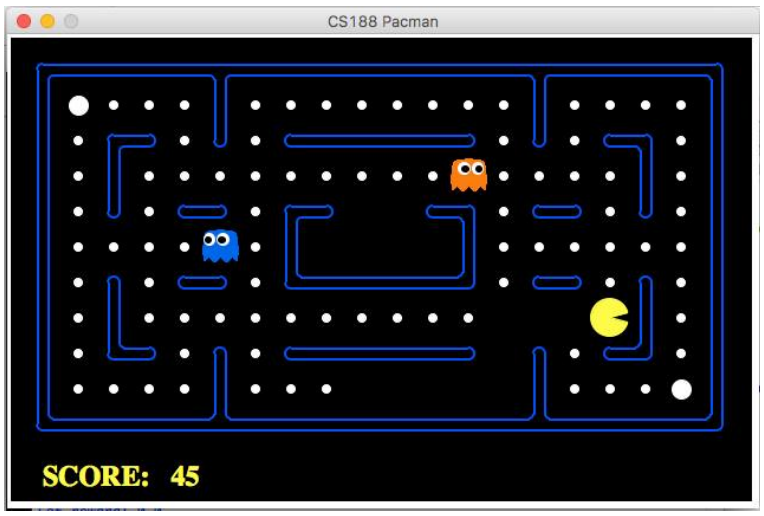


Figure 14. The Pacman graphical user interface.

To further improve the project neural networks could be used to save the time needed for doing the training.

5 Conclusion

Our 8 week journey summarized. The project effectively consisted of multiple sections focusing on how AI could be applied to Discontinuous and Continuous games and environments. Starting with some less complicated examples in discontinuous games, we worked with Tic-Tac-Toe and Connect 4. Gaining a fair understanding of how the reward system works we successfully applied MiniMax and Alpha Beta Pruning algorithms. With increasing complexity of the games we need to adjust our algorithms - for Connect4 we tried to implement Heuristic Search which evaulated the siutation on the board at any given moment in a better way. Since we knew this approach would not be effective in Continuous game due to excessive computation time we had to reconsider our strategy and tackle the problem with Reinforcement Learning.

Due to self-taught nature of our coding abilities we faced some difficulties in even reading the code at first, let alone applying it to a game. Picking up a course on edX and with several books suggested by our supervisor and several computing department professors specializing in machine learning, after 2 weeks of intense studying we were ready to continue and make relevant progress in the project. Becoming familiar with the abilities that Reinforcement Learning had, the way Q-Learning, Temporal Difference Learning operates and the whole idea of an algorithm training system, we starting with applying both methods to tic-tac-toe for better understanding and then expanded to Pacman and a 2 leg Crawler. The most fascinating discovery to us was that the algorithm for both was actually the same.

By the end of 8 weeks we managed to gain a very good insight in how AI operates, the numerous applications and possibilities in this field. Drastically improved our coding abilities. Completed a bug proof code for 2 Continious and 2 Discontinious games - over 60 pages of Code (GitHub-https://github.com/mjs114/urop).

References

[1] István S. N. *What is Artificial Inteligence?.*
`http://www.ucs.louisiana.edu/ isb9112/dept/phil341/wisai/WhatisAI.html`

[2] *So, what exactly is a Neural Network?.*
`http://www.ai-junkie.com/ann/evolved/nnt2.html`

[3] Jason Brownlee. *Supervised and Unsupervised Machine Learning Algorithms.*
`http://machinelearningmastery.com/`

[4] Richard Jones *Machine Learning, Part I: Supervised and Unsupervised Learning.*
`http://www.aihorizon.com/essays/generalai/`

[5] Rich Sutton. *A brief introduction to reinforcement learning.*
`http://www.cs.ubc.ca/ murphyk/Bayes/pomdp.html`

[6] CS312 Recitation 21. *Minimax search and Alpha-Beta Pruning.*
`https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm`

[7] Dave Marshall *Heuristic Search.*
`http://www.cs.cf.ac.uk/Dave/AI2/node23.html`

[8] Maureen Caudill *Heuristic Search.*
`Neural Network Primer:  Part I`

[9] Josef Burger *A Basic Introduction to Neural Networks.*
`http://pages.cs.wisc.edu/ bolo/shipyard/neural/local.html`

[10] Namco Bandai Games Inc. *Bandai Namco press release for 25th Anniversary Edition.*
`http://bandainamcogames.co.jp`

[11] UC Berkeley *UC Berkeley CS188 Intro to AI.*
`http://ai.berkeley.edu/home.html`

[12] Figure 1. - Michael Nielson *Neural Networks and Deep Learning.*
`http://neuralnetworksanddeeplearning.com/chap1.html`

[13] Figure 2. - Tim Kane *National Affairs the Political Prisoners' Dilemma.*
`https://balanceofeconomics.com/2013/01/03/`

[14] Figure 3. *Computer Systems Colloquium.*
`http://web.stanford.edu/class/ee380/Abstracts/4g4g.gif`

[15] Figure 4. - Ian Barland *Connect 5 Strategies.*
`http://www.owlnet.rice.edu/ comp210/02fall/Labs/Lab15/`

[16] Figure 5. - Mark Lee *The Agent Environment Interface.*
`https://webdocs.cs.ualberta.ca/ sutton/book/ebook/node28.html`

[17] Figure 6-11. UC Berkley CS188
`http://ai.berkeley.edu/home.html`