Using AI to play Checkers

Introduction

AI for playing Games

AI is a field of Computer Science which deals with autonomous agents which can mimic the cognitive functions that humans associate with other human minds.

Tasks like self-driving car, playing games, answering questions etc. which require real-time decision making are some examples where AI agents are showing great promise and skill.

In this project we tried to make an AI agent which can learn to play the game of Checkers on it’s own by playing games with itself and learning the best move to be played in a given situation.

Checkers Game Play

- The number of legal positions in English draughts is estimated to be 10^{20}
- played on an 8×8 board, with checkers moving one square forward and kings moving one square in any direction.
- Captures take place by jumping over an opposing piece, and a player is allowed to jump multiple men in one move.
- Checkers promote to kings when they advance to the last rank of the board.

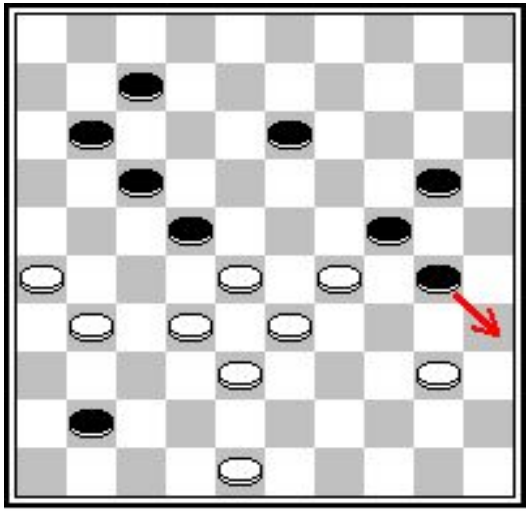
Checkers

Checkers is a strategy board game for **two** players which involves diagonal moves of uniform game pieces and mandatory captures by jumping over opponent pieces.

Rules:

- One player has dark pieces while the other has light pieces
- Players alternate turns
- A player cannot move the opponent’s piece
- A move consists of a piece moving diagonally to an adjacent unoccupied square
- If the adjacent square contains an opponent's piece, and the square immediately beyond it is vacant, the piece may be captured (and removed from the game) by jumping over it.

Source: https://en.wikipedia.org/wiki/Draughts



Suite à l'attaque fautive (30-35) des Noirs, les Blancs forcent le gain...

Source: https://upload.wikimedia.org/wikipedia/commons/3/3e/Probl%C3%A8me_Jeu_de_dames_SR.gif

Theory

Algorithms Implemented

- ★ Reinforcement Learning
- ★ TD Learning
- ★ Q Learning
- ★ Artificial Neural Network

Reinforcement Learning

Reinforcement learning is a branch of Machine learning which is concerned with training systems which can take actions in real time so as to maximize a cumulative reward.

Reinforcement learning differs from supervised and unsupervised learning in many aspects such as :-

- Correct input/output pairs are never shown. Only a reward is given to the agent corresponding to the action taken
- Focus is on on-line performance which involves finding a balance between exploration and exploitation

Temporal difference (TD)

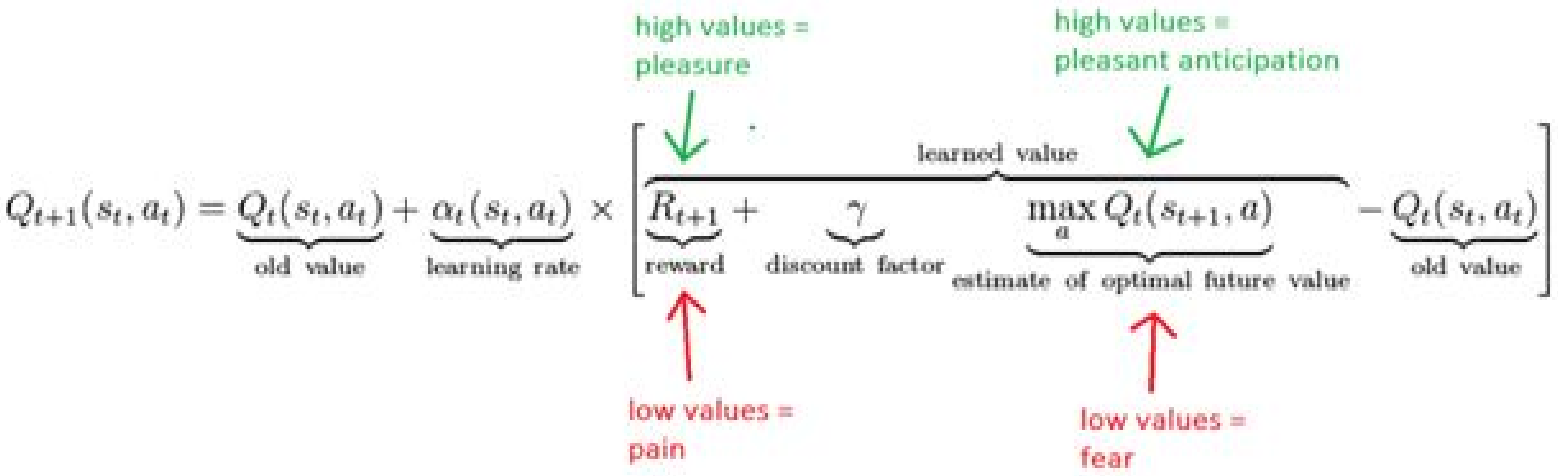
- Prediction-based Technique.
- Adjusts predictions of reward or punishment to match other.
- Final reward is the difference between the correct prediction and the our current prediction.

It is mathematically formulated using Bellman's Equation.

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s').$$

Q Learning

Q-learning is a model-free reinforcement learning technique. Q-learning can be used to find an optimal action-selection policy.



Basics of Model-free control

The basic reinforcement learning model consists of :-

S(the set of environment states)

A(the set of actions)

Rules for transitioning between states

Rules that determine the scalar immediate award for any action

Rules that describe what that agent observes

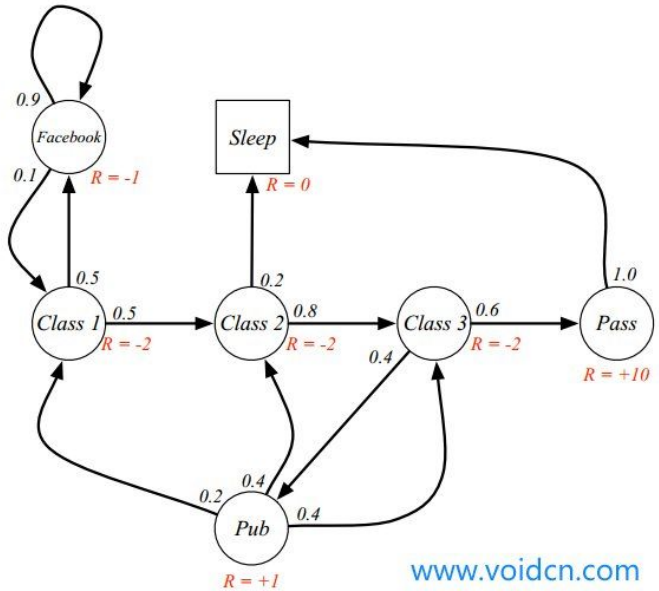
Basics of Model-free control

Usually the environment is modelled as a Markov Decision Process with a fixed set of states **S** and a fixed number of actions **A** along with rules depicting **transition** from one state to another and a **reward** attached to that.

Then using Dynamic Programming exact solutions ie. policy is evaluated which is optimal. But this can lead to **exponential time** solution so other methods like **sampling** are used.

Sampling actions from states according to different strategies like **eps-greedy** or **Boltzmann** selection rule are used to balance between **exploration** and **exploitation**.

Example: Student MRP



Source: David Silver's course on RL taught at UCL

Basics of Model-free control

But in real life cases we cannot model the environment as an MDP as the number of states is very large to be explicitly stated.

A typical episode of the activity has many states which the agent traverses through and the reward is given after the episode ends eg. the agent may play many moves but the result ie. is known after the game ends. So based on this there are **2** approaches which are proposed ie. **Monte-Carlo learning** and **TD**(Temporal Difference) based methods.

Monte Carlo based learning method

The discounted award at the time step '**t**' $G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$

The updating equation for the value at the state S_t $V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$

Points to notice:

- The value of a state is updated only after the episode ends as it needs the reward received at future time steps
- This does not work in situations where episodes do not end

Temporal Difference based learning methods

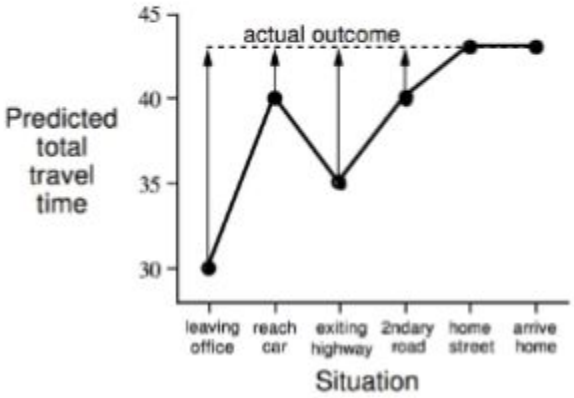
The updating equation for the value at the state S_t

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

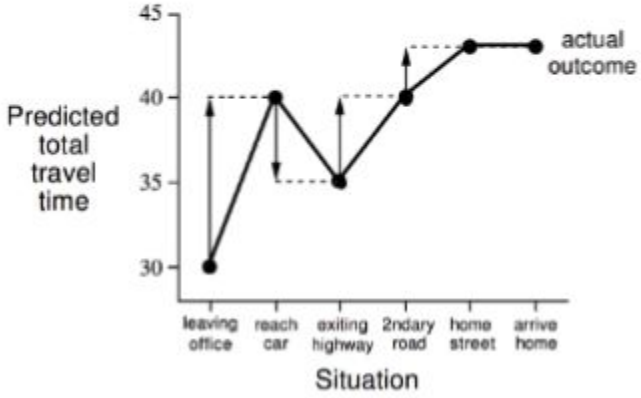
- TD learning method depends only on 1 future state for update, hence it does not require the episode to end.
- It can learn in a more online manner
- It can learn from incomplete sequences also

Difference between MC and TD methods

Changes recommended by Monte Carlo methods (α=1)



Changes recommended by TD methods (α=1)



Source: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf

Challenges in large state spaces

Although the above methods work fine in situations where we need a fast algorithm but they fail in situations where the search space is very large eg. in checkers the total number of states ~ 10^20 making it almost impossible to cover all states

But the catch is that many states are very similar to each other and they do not need to be stored separately ie. similar states will have similar value also.

Here comes the use of value function approximators which can *learn* the value of a state overtime by extrapolating the values of the states which are *nearby*.

In this project we have used a 1 hidden layer Neural Network as the function predictor and learnt it’s weights over many episodes of TD learning.

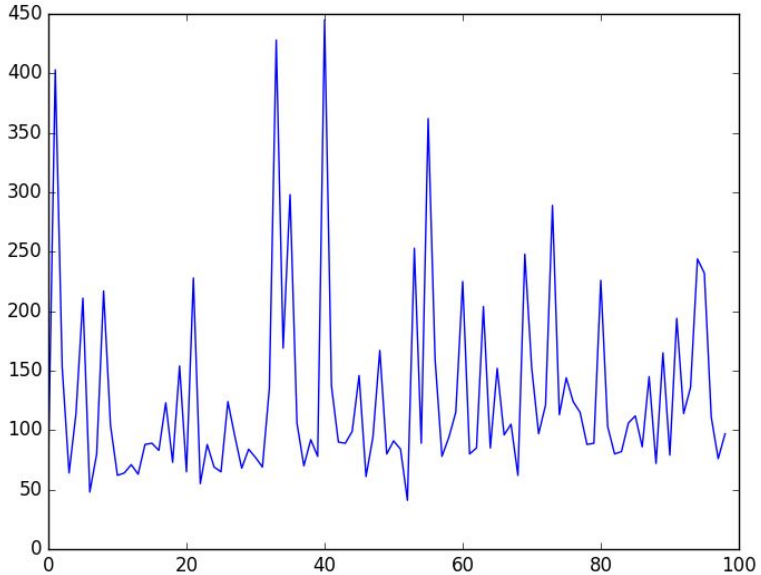
Our System

Features

- **Piece Disadvantage**
- **Piece Advantage**
- **Piece Threat**
- **Piece Take**
- **Diagonal Bridge**
- **Centre Control**
- **Advancement**
- **King Central Control**

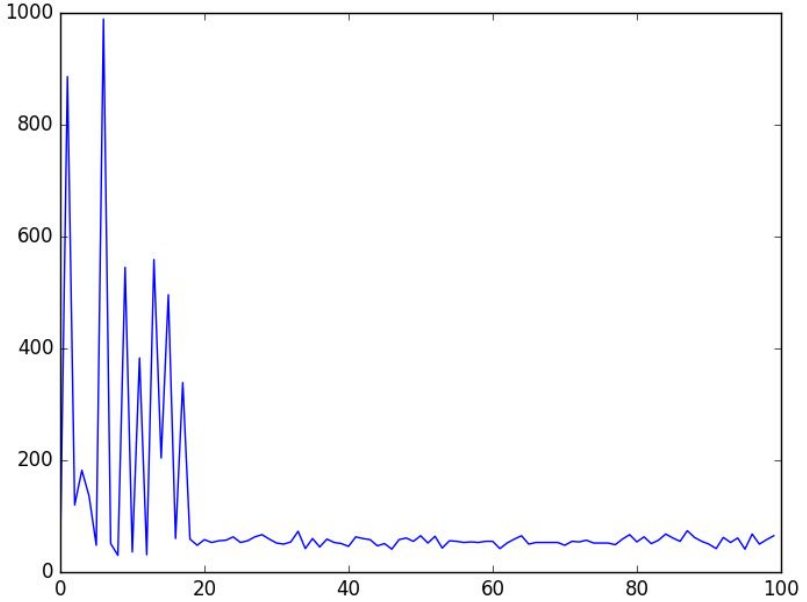
Observations

When agent and player are of equal strength, we observe that both agent and player are simultaneously learning well and both take on an average equal number of moves to defeat each other.



Case 2 :-

When one is dominant on another, we observe that number of moves required by the dominant player to defeat the weaker player decreases over time as number of epochs increase. This result is generally seen because the Neural Network becomes biased towards player.



Practical Observations

- After playing lots of games with the AI agent we observe the it has has devised its own strategy.
- The AI agent first places its pieces in positions where they cannot be cut and then rushes towards the other side of the board to make a king.
- After it has a king it rushes to bring it on the other side and uselessly wastes time playing empty moves waiting for us to make a mistake.
- As soon as we make a mistake, it pounces upon the opportunity and cuts our pieces.

Results and Conclusion

- The AI agent was successfully built using Python libraries, internet functionalities, building own Artificial Neural Network System
- The system is quite capable of playing versus single player and getting trained over period of iteration
- We have also tried to observe the impact of value of lambda on our system. An we have considered two possible scenarios on which number of moves taken to defeat the opponent is plotted.

Thanks!

Made By :
Gaurang Bansal
Shivin Srivastava