**HEXAPAWN**



APPLICATIONS OF AI AND MACHINE LEARNING

Tanay Patil

Sriram S

Raunak Ramesh Battu

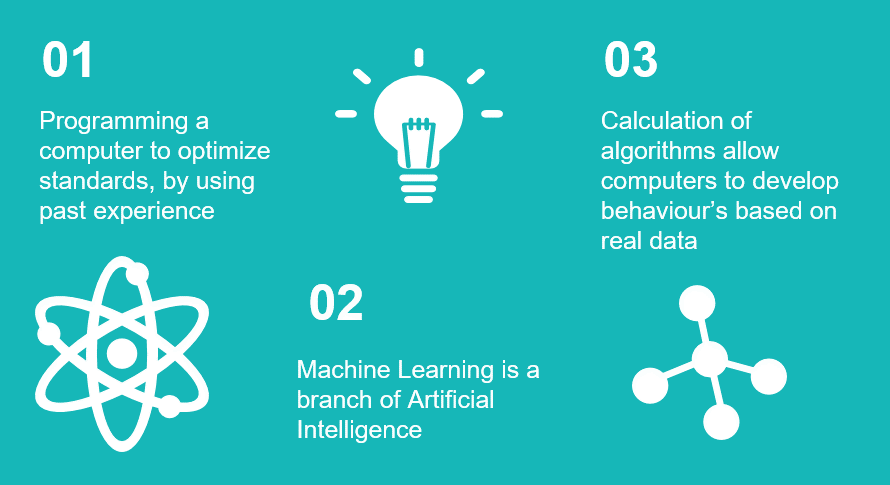
**Acknowledgment**

We are thankful for the help and cooperation of the school authorities, our computer science teacher, Mrs. Manjula, and Mrs. Uzma Fathima, for the successful completion of our investigatory project, “Hexapawn AI”

**Table of Contents**

* What is Machine Learning
* What is Hexapawn?
* Flow Chart, Design, Output
* Conclusion, Bibliography

**What is Machine Learning?**



Machine learning is a study of computer algorithms that improve automatically through experience. It is seen as a subset of artificial intelligence but she is running out of them to build a mathematical model based on sample data understanding data to make predictions are decisions without being explicitly programmed to do so.

ABSTRACT

Hexapawn is a deterministic two-player game invented by Martin Gardner. It is played on a rectangular board of variable size, for example on a 3x3 board or a chessboard. On a board of size 3x3, each player begins with three pawns, one for each square in the row closest to them. The goal of each player is to advance their pawns to the opposite end of the board or to prevent the other player from moving. Hexapawn also happens to be one of the simplest demonstrations and applications of Machine Learning.

Purpose

* Hexapawn was designed to illustrate mechanical game-players by Martin Gardner, the well-known Scientific American mathematics columnist.
* Till now, we punished our computer when it makes a mistake, right? But what if we reward it for every good move? By punishing out “computer” whenever it makes a mistake we increase the probability that the computers move each round leads to its success.
* But what if we reward it for every good move?  
  If we reward the computer for every good move it makes that would reduce the probability of a losing marble appearing by increasing the probability of a matchbox generating a winning marble. The computer would eventually still reach perfect play because I’ll still remove the losing marbles.
* This may not be the first machine learning algorithm, there is Turing’s machine and there may be a lot of others. But we can for sure say that this is learning, our “computer” gets better every time.

**How Does the Game Work?**

* As in chess, each pawn may be moved in two different ways: it may be moved one square forward, or it may capture a pawn one square diagonally ahead of it.
* A pawn may not be moved forward if there is a pawn in the next square.
* A player loses if he/she has no legal moves or the other player reaches the end of the board with a pawn.
* Unlike chess, the first move of a pawn may not advance it by two spaces.

**SYSTEM REQUIREMENTS**

Required:

Operating System with a built-in/external Python Interpreter

Python Modules

* Tkinter
* Random
* Copy
* Time

Python IDEs and Code Editors

* Python IDLE
* Visual Studio Code
* Anaconda
* PyCharm

**Flow Chart and Design**

A picture containing diagram

Description automatically generated

**Python Libraries Used**

Copy

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations.

Tkinter

The Tkinter package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk & Tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Random

Functions in the random module depend on a pseudo-random number generator function random(), which generates a random float number between 0.0 and 1.0. random. random(): Generates a random float number between 0.0 to 1.0. The function doesn't need any arguments.

**What is a ‘sprite’?**

* It’s a small graphics pixmap image
* It’s normally stored in off-screen VRAM
* It’s ready to be copied to on-screen VRAM
* The copying operation is called a ‘BitBlt’
* Several sprites can support an animation
* Let’s see how to create a sprite array, then synchronize BitBlts with Vertical Retrace

**Future Possible Improvements**

* Arrange your sprites in an array
* Write an animation loop
* Incorporate movement in sprite’s location
* Study the source-code for higher efficiency.
* Let user control direction with arrow-keys
* Store your arrays in offscreen VRAM
* Use modules such as *TURTLE* to improve the user interface and better animations.
* Use Pyautogui to run computer vs computer simulations.

**Source Code**

import random

import sys

from copy import deepcopy

import pickle

import os

WRAP=55

WHITE=1

BLACK=-1

EMPTY=0

class HexpawnException(Exception):

pass

class IllegalCoordinate(HexpawnException):

pass

class IllegalMove(HexpawnException):

pass

fh=open("MovesetPermutations.txt","r")

MoveSet=fh.readlines()

InitialMoveSet=MoveSet[0:19]

InitialMoveSetVar=''

for permutation in InitialMoveSet:

InitialMoveSetVar+=permutation.rstrip(',\n')+','

InitialMoveSetVar=InitialMoveSetVar.rstrip(',')

InitialMoves=tuple(eval(InitialMoveSetVar))

MoveSetConfigs=MoveSet[19:38]

ConfigMoveSetVar=''

for permutation in MoveSetConfigs:

ConfigMoveSetVar+=permutation.rstrip(',\n')+','

ConfigMoveSetVar=ConfigMoveSetVar.rstrip(',')

ConfigMoveSet=tuple(eval(ConfigMoveSetVar))

fh.close()

initialWinsLosses={"Wins":0,"Losses":0}

try:

fh=open("Scores.dat","rb+")

except FileNotFoundError:

f=open("Scores.dat","wb")

pickle.dump(initialWinsLosses,f)

f.close()

fh=open("Scores.dat","rb+")

WinLoose=pickle.load(fh)

fh.close()

def update(WinOrLoose):

f=open("Scores.dat","rb+")

f1=open("temp.dat","wb")

data=pickle.load(f)

if WinOrLoose=='Win':

newvar=data["Wins"]

data['Wins']=newvar+1

pickle.dump(data,f1)

f.close()

f1.close()

elif WinOrLoose=='Loose':

newvar=data["Losses"]

data['Losses']=newvar+1

pickle.dump(data,f1)

f.close()

f1.close()

os.remove("Scores.dat")

os.rename("temp.dat","Scores.dat")

class Game:

\_initial\_moves = InitialMoves

\_initial\_board = [

BLACK, BLACK, BLACK,

EMPTY, EMPTY, EMPTY,

WHITE, WHITE, WHITE

]

def \_\_init\_\_(self):

self.wins = WinLoose['Wins']

self.losses = WinLoose['Losses']

self.winner = None

self.message = None

self.board = deepcopy(self.\_initial\_board)

self.moves = deepcopy(self.\_initial\_moves)

self.x = self.y = 0

self.configs = ConfigMoveSet

#Show state of board after every move

def reset(self):

self.board = deepcopy(self.\_initial\_board)

self.winner = None

self.message = None

#Uses Machine Learning to remove losing strategies

def game\_over(self, winner, message=None):

# If the player won

if winner is WHITE:

self.wins += 1

update("Win")

if message:

self.message = message

else:

self.message = 'You win.'

# Remove losing strategy from moves list

self.moves[self.x][self.y] = 0

# If machine won

elif winner is BLACK:

self.losses += 1

update("Loose")

if message:

self.message += '\n' + message

else:

self.message += '\nI win.'

# Winner should only ever be player or machine

else:

raise HexpawnException

self.winner = winner

def overview(self):

return 'Games lost = {}, Games won = {}, Total Games = {}.'.format(

self.losses, self.wins, self.losses + self.wins

)

#Printing Gameboard

def fnr(x):

rval = {

1: 3, 3: 1,

4: 6, 6: 4,

7: 9, 9: 7

}

return rval.get(x, x)

# Player Move

def white\_move(m1, m2, game):

assert game.winner is None

game.message = None

# Ensure move is on the board

if m1 not in range(1, 10) or m2 not in range(1, 10):

raise IllegalCoordinate

# Ensure player is moving their own piece

if game.board[m1 - 1] is not WHITE:

raise IllegalCoordinate

# Ensure player isn't moving onto their own piece

if game.board[m2 - 1] is WHITE:

raise IllegalMove

# Ensure if moving diagonally its onto an opponents piece

if m2 - m1 != -3 and game.board[m2 - 1] is not BLACK:

raise IllegalMove

# Ensure the user is not trying to move left, right, or down

if m2 > m1:

raise IllegalMove

# Make sure if moving forward, destination is unoccupied

if m2 - m1 == -3 and game.board[m2 - 1] is not EMPTY:

raise IllegalMove

# Enusre if the player is moving forward it is within allowable range

if m2 - m1 < -4:

raise IllegalMove

# Ensure user isn't moving from bottom left of board to top right

if m1 == 7 and m2 == 3:

raise IllegalMove

# Perform move

game.board[m1 - 1] = EMPTY

game.board[m2 - 1] = WHITE

# Check if any player pieces have reached the far row

if WHITE in (game.board[0], game.board[1], game.board[2]):

game.game\_over(WHITE)

return

# Check if all of machine's pieces have been captured

if BLACK not in game.board:

game.game\_over(WHITE)

return

# Check if machine can move forward

for i in range(6):

if game.board[i] is BLACK:

if game.board[i + 3] is EMPTY:

return

# Check if machine can capture a piece

for i in (0, 1, 3, 4):

if game.board[i] is BLACK and game.board[i + 4] is WHITE:

return

for i in (1, 2, 4, 5):

if game.board[i] is BLACK and game.board[i + 2] is WHITE:

return

# If machine has no valid moves player wins

game.game\_over(WHITE)

#Machine Move

def black\_move(game):

assert game.winner is None

game.message = None

strategies = list()

r = None

for game.x in range(19):

current\_config = list(game.configs[game.x])

mirrored\_config = list(current\_config)

mirrored\_config[0] = current\_config[2]

mirrored\_config[3] = current\_config[5]

mirrored\_config[6] = current\_config[8]

mirrored\_config[2] = current\_config[0]

mirrored\_config[5] = current\_config[3]

mirrored\_config[8] = current\_config[6]

if game.board == current\_config:

r = False

break

elif game.board == mirrored\_config:

r = True

break

assert r is not None

# Uses previously played data to implement move

for i in range(4):

if game.moves[game.x][i] != 0:

strategies.append(i)

# If a move cannot be found, machine loses

if not strategies:

game.game\_over(WHITE, message='I lose.')

return

# Get machine's move

game.y = random.choice(strategies)

move = divmod(game.moves[game.x][game.y], 10)

if r:

move = (fnr(move[0]), fnr(move[1]))

# Perform move

game.board[move[0] - 1] = EMPTY

game.board[move[1] - 1] = BLACK

game.message = 'I move from {} to {}'.format(move[0], move[1])

# Check if any machine pieces have reached end of board

if BLACK in (game.board[6], game.board[7], game.board[8]):

game.game\_over(BLACK)

return

# Check if all of player's pieces have been captured

if WHITE not in game.board:

game.game\_over(BLACK)

return

# Check if player can move forward

for i in range(3, 9):

if game.board[i] is WHITE:

if game.board[i - 3] is EMPTY:

return

# Check if player can capture a piece

for i in (4, 5, 7, 8):

if game.board[i] is WHITE and game.board[i - 4] is BLACK:

return

for i in (3, 4, 6, 7):

if game.board[i] is WHITE and game.board[i - 2] is BLACK:

return

game.game\_over(BLACK, message='Human Loses, Machine Wins.')

from tkinter import \*

COLOUR1 = 'black',

COLOUR2 = 'white',

TILE\_SIZE = 6

#Start Game

def main():

game = Game()

gui = GUI(game)

gui.mainloop()

class GUI(Tk):

def \_\_init\_\_(self, game):

self.game = game

self.m1 = None

# Set up Tkinter

Tk.\_\_init\_\_(self)

self.title("Hexapawn")

self.resizable(0, 0)

self.wpawn = PhotoImage(file='white.gif')

self.bpawn = PhotoImage(file='black.gif')

self.empty = PhotoImage(file='empty.gif')

self.tk.call('wm', 'iconphoto', self.\_w, self.wpawn)

self.notice = StringVar()

Label(

self, textvariable=self.notice, width=45, height=2,

font="Sans 12 bold"

).pack(pady=10)

# Create game board

self.tiles = list()

self.tile\_frame = Frame()

self.tile\_frame.pack(padx=60, pady=(0, 60))

self.new\_game\_button = Button(

self, text='NEW GAME', command=self.enable

)

self.new\_game\_button.pack\_forget()

# Create game board tiles

color = COLOUR1

i = 0

for row in range(3):

for col in range(3):

tile = Button(self.tile\_frame)

tile.config(

relief=FLAT,

bg=color,

activebackground=color,

command=lambda i=i: self.player\_selected(i)

)

tile.grid(column=col, row=row)

self.tiles.append(tile)

color = COLOUR1 if color == COLOUR2 else COLOUR2

i += 1

self.set\_pieces()

#Movement of Pawns

def set\_pieces(self):

for i in range(9):

piece = self.game.board[i]

if piece is BLACK:

self.tiles[i].config(image=self.bpawn)

self.tiles[i].image = self.wpawn

elif piece is WHITE:

self.tiles[i].config(image=self.wpawn)

self.tiles[i].image = self.bpawn

else:

self.tiles[i].config(image=self.empty)

self.tiles[i].image = self.empty

#Determines tile clicked on

def player\_selected(self, position):

if self.m1 is not None:

self.move(self.m1, position)

self.m1 = None

else:

if self.game.board[position] is WHITE:

self.m1 = position

#Calculates Moves (Whether Illegeal or Not)

def move(self, m1, m2):

try:

white\_move(m1 + 1, m2 + 1, self.game)

except IllegalMove:

self.notice.set('Illegal move.')

return

except IllegalCoordinate:

self.notice.set('Illegal coordinates.')

return

self.set\_pieces()

if self.game.winner:

self.notice.set(self.game.message.replace('\n', '-- '))

self.notice.set(

'{}\n{}'.format(self.notice.get(), self.game.overview()))

self.disable()

return

black\_move(self.game)

self.set\_pieces()

self.notice.set(self.game.message.replace('\n', '-- '))

if self.game.winner:

self.notice.set(

'{}\n{}'.format(self.notice.get(), self.game.overview()))

self.disable()

return

#Disables gameboard interaction

def disable(self):

for tile in self.tiles:

tile.config(state='disabled')

self.new\_game\_button.pack(pady=16)

self.tile\_frame.pack(padx=60, pady=0)

#Disables gameboard interaction

def enable(self):

for tile in self.tiles:

tile.config(state='normal')

self.game.reset()

self.set\_pieces()

self.new\_game\_button.pack\_forget()

self.tile\_frame.pack(padx=60, pady=(0, 60))

#Start game

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT**

(After playing 25 games)

|  |  |
| --- | --- |
|  | |
| Icon  Description automatically generated  Black Pawn | Icon  Description automatically generated  White Pawn |

**Conclusion**

Our Hexapawn Machine uses reinforcement learning to figure out what is the best action to take in any given state. In principle, during the learning process we “punish” the machine each time it makes a mistake by taking a set of moves away from him, making him less likely to make a mistake. Another possible solution would be to reward the machine whenever it makes a good play by pushing forward the used set of moves, making him more likely to choose the action that won before.

**Bibliography**

* [**https://www.youtube.com/watch?v=sz7UAZNgGg8**](https://www.youtube.com/watch?v=sz7UAZNgGg8)
* [**http://cs.williams.edu/~freund/cs136-073/GardnerHexapawn.pdf**](http://cs.williams.edu/~freund/cs136-073/GardnerHexapawn.pdf)
* [**http://www.mscroggs.co.uk/menace/**](http://www.mscroggs.co.uk/menace/)
* [**https://prezi.com/paoljqexnos3/hexapawn/?frame=daa6ec7c35d48ab6ef1fd5e61aa0ba380e9d1400**](https://prezi.com/paoljqexnos3/hexapawn/?frame=daa6ec7c35d48ab6ef1fd5e61aa0ba380e9d1400)

* **<http://www.mscroggs.co.uk/blog/19>**