Can Genetic Algorithms Affect AI Models?

By Jayden Yin and Marcus Leung

Table of Contents

Table of Contents 2
Acknowledgements and Citations 3
Time Line 4
Background Research 10
- Connect Four 10
- Neural Networks 10
- How To Apply Neural Networks to Connect Four 10
- Training a Neural Network 11
- Genetic Algorithms 11
- Genetic Algorithms For Neural Networks 11
- Setup Of Genetic Algorithms 11

Scientific Question And Problem 13
Motivation 14
Hypothesis 15
Experimental Design And Procedure Materials 16
- Procedure 16
  - Comparison method 16
  - Reinforcement Learning method 16
  - Genetic Algorithm Process 17

Observations 20
- Raw Data 20
- Subjective Observations 20
- **Managed Data** **21**

Results and Conclusion 22
Applications, Improvements, and Future 23
- Efficiency of training 23
- Experimental resources 23
- Other questions 23

Sources of Error 24
Glossary 25
- Core AI & Machine Learning Terms 25
  - Artificial Intelligence (AI) 25
  - Machine Learning (ML) 25
- Neural Network–Specific Terms 26
- Training & Learning Methods 28
- Game AI & Decision-Making Terms 29
- Programming & Development Terms 29
- Hardware & Compute Terms 31
- General Terms 31

Sources 31
Appendix - Source code 34

Acknowledgements and Citations

We, Marcus Leung and Jayden Yin, would personally like to thank all the people who helped us throughout this project, including our supportive parents and our teacher, Ms. Lai. Thank you to Westmount Charter for creating and managing our school fair. Also, to the people who helped set up the Calgary Youth Science Fair event, and all the judges who helped judge our project and citations. Once again, special thanks to everyone who helped us and to all those involved in the Calgary Youth Science Fair project.

Time Line

2025

Oct. 3: Decided on topic, connect 4 AI. Set up a server for coding

Oct. 4: Started logbook, started background research, Connect 4

Oct. 5: Background research, Connect 4

Oct. 6: Background research, Connect 4

Oct. 7: planning logistics

Oct. 8: Background research, AI, Connect 4

Oct. 9: Background research, AI. Planning logistics

Oct. 10: Background research, AI. Planning logistics

Oct. 11: Coding, finished background research, AI, Connect 4. Planning logistics.

Oct. 12: Planning logistics

Oct. 13: Planning logistics

Oct. 14: Started coding

Oct. 15: Continue coding, building AI model

Oct. 17: Met up. Setup version control and has a backup on GitHub. Deleted empty file. Remote access established. Decided to make our own engine. Created working interactive game with invalid moves

Oct. 18 Created to-do list

Oct. 19 Finished game engine, Added AI building to the To-Do list

Oct. 20 Added Draws to game engine

Oct. 22 Made a skeleton for AI

Oct. 23 Researched about simple AIs and reinforcement learning, https://www.youtube.com/watch?v=8392NJjj8s0

https://roboticsproject.readthedocs.io/en/latest/ConnectFourAlgorithm.html?utm_source=chatgpt.com

Oct. 24 Deciding AI+ML method, implemented “play one game”, found resources, implemented neural nets

Oct. 25 Started planning trifold on slides

Oct. 28 Converted lists to numpy array to be compatible with the neural net

Oct. 29 Added the ability for loading and saving a model to a file

Oct 29-Nov 12 No progress, on fall break

Nov. 13 Implemented checking which of 2 models is stronger via play one game

Nov. 14 Tested play_series and play_one_game

Nov. 15 Researched the use of genetic algorithms.

Nov. 16 Fixed the double definitions of play_move, swapped sides so AI is always player 1, implemented a simple AI training system, cleaned small errors and typos in code.

Nov. 17 Fixed swapping sides so AI is player one so we don’t waste brainpower teaching, as player 1 and 2

Nov. 18 Finished first copy of genetic algorithm code including creating a population of models, playing them against each other and ranking them as well as the part of mutating that doesn’t change the model.

Nov. 19 fixed bugs in the genetic algorithm part. Added get_mutant. 10% of the model changes its weights by -2 to 2

Nov. 20. Found bug in win counting, slowly genetic algorithm code is coming together

Nov. 21 Tested some training. No progress observed

Nov. 22 sped up a genetic generation step by running games in parallel. Tested and changed some parameters, 40 models in a generation, 100 games per series. We will possibly change the model’s shape/ size

Nov. 23-24 Researching about what size and shape our neural net should be. Continued training, no progress observed

Nov. 24 Added convolution to the neural nets

Nov. 25-26 Researched and decided on recording sequence of moves and reinforcing the nets knowledge

Nov. 27 Implemented recording and reinforcing a sequence of moves, genetic

Nov. 28 Created slides for the slideshow part of the science fair and a doc for the report

Nov. 29 Found bug where the AI plays invalid moves

Nov. 30 Working to fix the bug where the AI plays invalid moves. Loss integration function causing the weights magnitude to get too large. Reward function after win/ loss has been adjusted to a more minimal change. Fixed loss function. Invalid moves were fixed due to the change in the loss function

Dec. 1 Created title for report

Dec. 4 Outlined the requirements for the report

Dec. 5 Filled in acknowledgements section in report

Dec. 6 Found a bug in the code where the AI wasn’t learning. Fixed it by removing relu

Dec. 7 Restarted training with random weights, added another convolutional: 4 layers, 2 convolution 2 regular

Dec. 10 lengthened background information

Dec. 11 lengthened background research

Dec. 12 Met up and created a question, lengthened background research on neural nets, started slideshow, added some randomness to moves (one in a hundred is random)

Dec. 13 Added background research for neural nets

Dec 14 recorded AI performance

Dec 15 Did most of the slides that we can do before data

Dec 16 Checked AI performance

Dec 17 finished the rest of the slides we can do before data is received

Dec 18 Fixed a small bug in the code

Dec 19 Worked on diagrams in the slideshow, implemented compete to test the models against each other

Dec 20 Recorded model performance

Dec 22 Recorded model performance

Dec. 23 Stopped training due to lack in progress

Dec 24 Decided to increase the size of the neural net to 6 layers, 5 convolutional and 1 linear

Dec. 25 Christmas

Dec. 26 Boxing day

Dec. 27. Increased the size of the neural net to 6 layers. Restarted training with modified AI

Dec 28. N/A

Dec. 29 Recorded AI performance

Dec. 30 Begun Trifold planning

Dec. 31 Recorded AI performance

2026

Jan. 1 Discussed issues with AI performance

Jan. 2 Started trifold planning and organization

Jan. 3 Added more randomness to the AI's, occasionally plays a random move. Recorded AI performance

Jan. 4 Worked on procedure slide and in the logbook

Jan. 5 Recorded AI performance, added slightly more randomness to moves

Jan. 6 Worked on hypothesis in logbook, removed playing random moves when against a human

Jan. 7 Decided to stop training, recorded data

Jan. 8 Worked more on the procedure slides

Jan. 9 Met up and starting making graphs and chart to represent our data

Jan. 10 Redid our background research

Jan. 11 Developed the logbook, added experimental design and procedure.

Jan. 12 Framed outline for conclusion and observations

Jan. 13 Formatted Logbook

Jan. 14 Changed grammatical errors in logbook and improved conventions and flow.

Jan. 15 Created and completed conclusions in the logbook, also completed the forms

Jan. 16 Met up and completed the Applications, Improvements, and Future section in the logbook

Jan. 17 Made and put in some flow charts to explain the training process

Jan. 18 Started the 5 page summary

Jan. 20 Worked on the hypothesis of the 5 page summary

Jan. 21 Expanded the 5 page summary with more text

Jan. 22 Added some flow charts to the 5 page summary

Jan. 23 Met up and finished the 5 page summary as well as the glossary

Jan. 24 Started writing down what to put on the trifold

Jan. 25 Wrote down the contents of the trifold up to application, improvement and future.

Jan. 26 Edited and improved the final report.

Jan. 27 Completed the 5 page summary. Started to plan the specifics of the trifold layout

Jan. 28 Modified the 5 page summary

Jan. 29 Started to put in content for trifold

Jan. 30 Input the majority of the content for the trifold

Jan. 31 Finished the trifold content and printed it out

Feb. 1 Modified and reprinted some areas of the trifold, stated to glue pages down

Feb. 2 Practiced for the presentation and kept gluing pages on the trifold

Feb. 3 Practiced the presentation with the trifold

Feb. 4 Edited our presentation and continued practising

Feb. 5 Last minute practice and printed out our logbook and 5 page summary

Feb. 6 Westmount Charter School science fair

Background Research

Connect Four

Connect Four has a 6 vertical by 7 horizontal board where players take turns dropping different colour pieces. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of your own colour pieces. In the case that the board fills up without either player connecting four in a row, then it is considered a tie. Invented by Howard Wexler, and was first sold by Milton Bradley in 1974. The game is actually very complicated because there are 4,531,985,219,092 (about 4.5 trillion) possible situations the board can be in. Making the game very difficult and not so simple.

Neural Networks

Neural networks are a simplified computer model of neurons in the brain. A neural network processes an input through several layers of neurons and emits an output. The processing is controlled by numeric weights on each neuron, which control how the neuron activates other neurons in the network.

The neural network can recognize patterns in the input through the connections in the neurons. AIs for games usually use neural networks because humans also look for patterns in the shape of a game board to decide what moves to make.

How To Apply Neural Networks to Connect Four

- The input to the network is the state of the board (where pieces are placed on the board, and whose turn it is).
- The output of the network is a number for each possible move. The highest-numbered move is where the neural net recommends playing.

Training a Neural Network

For a game like Connect Four, a neural network can learn how to play well if it is taught which moves are good and which are bad. Teaching the network uses a process called Reinforcement Learning, where we modify the weights in the network.

When we know a move is good, we update the weights so that this move is more likely to be played in the future if the inputs are similar. When we know a move is bad, we update the weights so that this move is less likely to be played in the future. This means that good moves get rewarded over time and bad moves get penalized.

We also need to play random moves on occasion, to experiment with new approaches. This is similar to how a human might experiment with new ideas when learning to play a new game.

Training a Connect Four neural net traditionally involves playing the model against itself over many games, and teaching the model good moves and bad moves as it wins and loses games.

Genetic Algorithms

Genetic Algorithms are a method to optimize a solution by randomly modifying parameters. Genetic algorithms were inspired and modeled around natural selection, where genes in living things randomly mutate over time, and gradually the living things adapt to fit their environment as the individuals with the best fit tend to reproduce more.

We can think of a set of parameters as “genes”. Random modifications to the parameters can be “mutations”. Parameters with the best fit are mutated to produce new parameters, while parameters that do not fit are discarded. Like with natural selection, we expect the population of parameters to gradually adapt to fit better.

Genetic Algorithms For Neural Networks

For neural networks, the weights on the neurons are the “genes”. We can produce mutations by randomly modifying the weights, and a model is more “fit” if it wins more games versus other models.

To implement a genetic algorithm, this means we need to maintain a population of models and have them compete against one another.

Setup Of Genetic Algorithms

1. Create a population of neural network models
2. Each model plays games against every other model
3. Rank the models by the number of games won

4. Remove some of the worst-performing models
5. Refill the population by cloning and mutating the remaining models to get back to the original size
6. Repeat from step 2

Scientific Question And Problem

Can Connect 4 AI training benefit from genetic algorithms?

Genetic algorithms mimic real-life populations, and we want to know if AI training can be improved by having AI models compete like real-world individuals do.

Motivation

AI is a growing field and improving training can make it more efficient and effective.

We are interested in AI and enjoy playing games, and it is fun to see how strong an AI player can get.

Hypothesis

We hypothesize that genetic algorithms implemented into the Connect 4 AI model will perform slightly better than the non-genetic algorithms model. We think that the genetic algorithms model will win more games against a normally-trained model.

Experimental Design And Procedure Materials

The materials used in our project were mainly composed of computers, as our project was based on AI and ML. Those materials include:

- The server running the AI has an AMD Ryzen 5 5600X, 32GB of RAM, more than a terabyte of storage, all paired with AMD's most powerful and advanced graphics card, powered with AI capabilities, released on March 6th, 2025, and that is the AMD Radeon RX 9070XT.
- Other materials we used were apps, such as Microsoft Visual Studio Code, Google Chat, Google Docs, Microsoft DevHub, GitHub, Git, and other websites and minor apps.
- We used Python as our coding language as well as using pytorch for building the AI
- Used AMD ROCm software through the Pytorch library to build neural network models

Procedure

First, we implemented the rules of Connect Four in a Python program, and allowed neural network models to choose moves to play in a game.

Our procedure then compares genetic algorithm training against normal training without genetic algorithms. Normal training uses neural networks with reinforcement learning. Genetic algorithms however use genetic algorithms on top of normal training

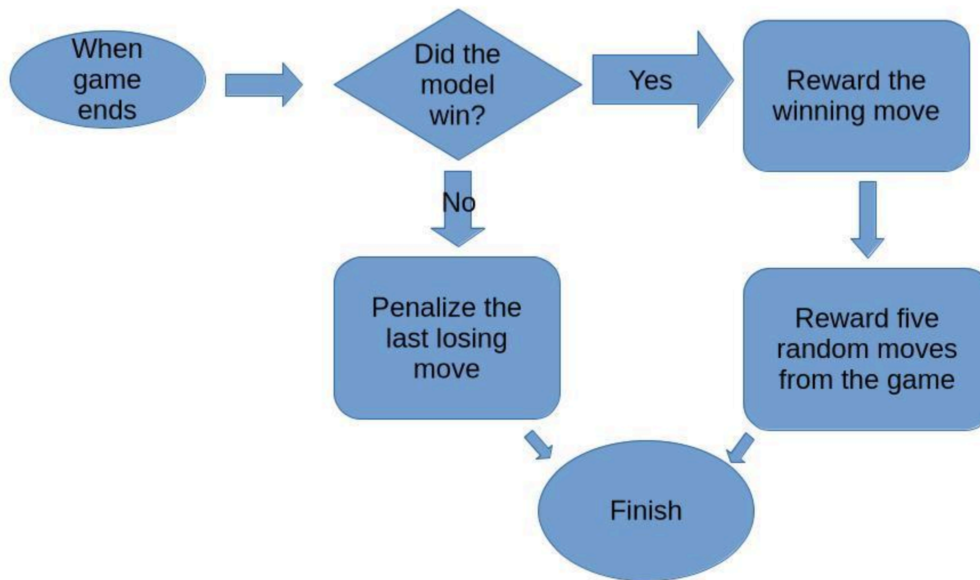
Comparison method

To judge whether one model is stronger than another, we have them play 2000 games against one another. We consider the model that wins more games to be stronger. We check the strongest normal and genetic models against one another periodically.

Reinforcement Learning method

We train models by playing a series of 100 Connect Four games between one model and another. Humans are not involved in training or supervising the process.

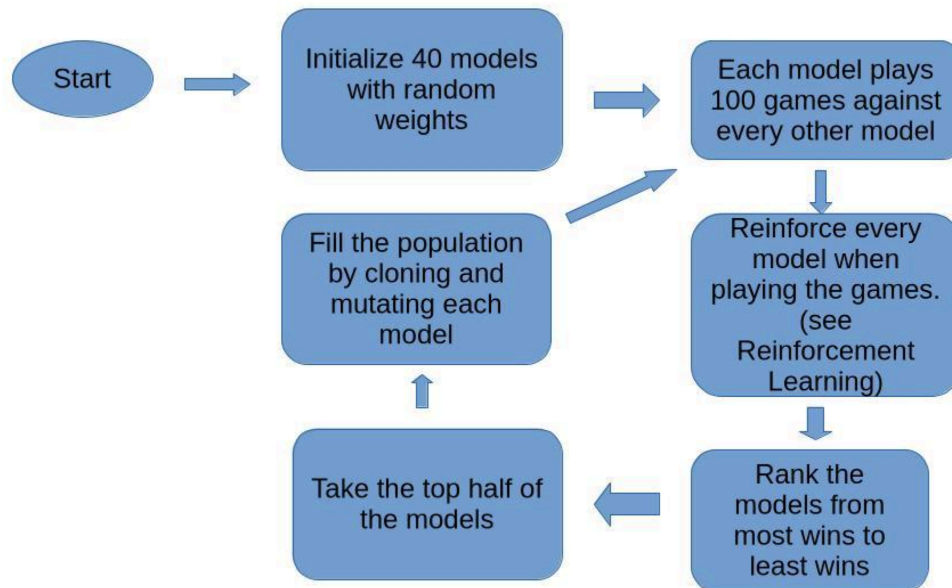
After each game, we consider the last winning move to be “good” because it won the game. The loser's last move is considered “bad” because it did not prevent the loss. On the winning side, we also choose 5 random moves from the list of moves the model played and consider those “good” as well, because they should have led to the winning situation.



Genetic Algorithm Process

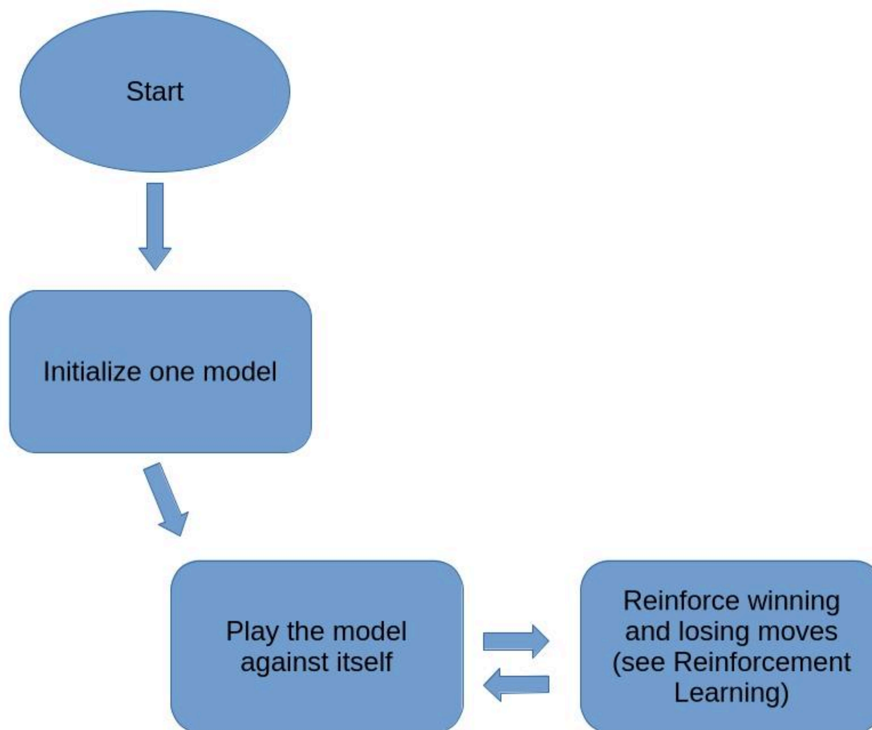
We have chosen these parameters for our genetic algorithm:

- The population size is 40 neural network models
- After all of the models play each other, the bottom half are removed
- Fill the population back to 40 with copies of the remaining models, mutating 1% of weights
- Mutation is done by multiplying weights by a random number between -2.0 and 2.0



Normal Training process

Normal training only maintains one neural network model, and continuously plays it against itself.



Training Time

Both normal and genetic training systems were run in parallel together, and we periodically compared the top genetic model with the normal to see which was stronger.

Training ran continuously for 20 days.

Observations

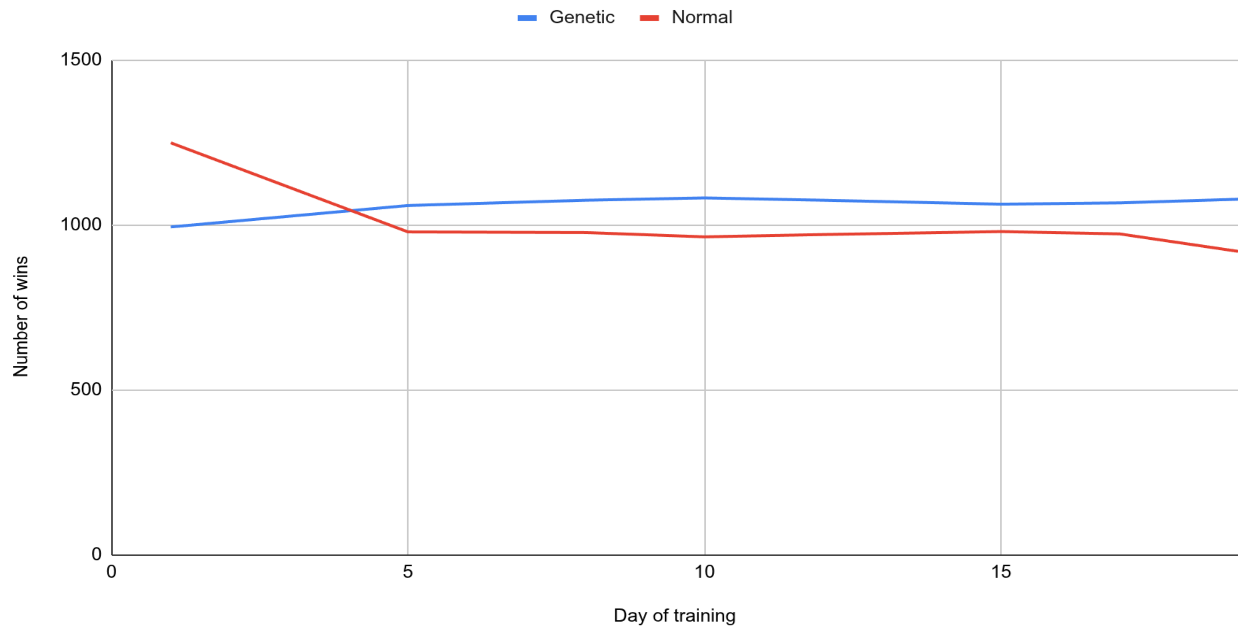
Raw Data

Date Recorded	Wins By Normal Model	Wins By Genetic Model	Genetic Wins Over Normal
Dec. 20 2025	1250	995	-255
Dec. 24 2025	980	1060	80
Dec. 27 2025	978	1076	98
Dec. 29 2025	965	1083	118
Dec. 31 2025	972	1076	104
Jan. 3 2026	981	1064	83
Jan. 5 2026	974	1068	94
Jan. 7 2026	921	1079	158

Subjective Observations

- Both the genetic and normal AI models usually played in the same column
- They would move to another column if the first one was full
- Neither seemed to react to block a winning move

Managed Data



Results and Conclusion

By the end of training, the genetic model won slightly more games over the normal model. This result validates our hypothesis to some extent.

However, by observing the moves that the models played, neither one seemed to understand how to win the game. They mostly played in the same column, not really reacting to an opponent's moves.

Despite that, the genetic model still performed slightly better than the normal model. We do not fully understand why this is the case, since both models always seemed to play in the same places.

As a result, we are not very convinced that our hypothesis is actually true. It is hard to conclude that genetic algorithms are definitely better than normal training, but more investigation is justified.

Applications, Improvements, and Future

Efficiency of training

If genetic algorithms are able to achieve better results in the same amount of training time, then they could help reduce the amount of resources needed to get a result. This would mean we can save electricity and time developing AI.

Genetic algorithms work when fitness of models can be easily measured. When the fitness can be measured easily, this technique could be applied.

Experimental resources

We did not observe much improvement in the play of either normal or genetic models. It's possible that if we used more time and compute power that we would have achieved better results.

Other questions

- Could we apply this technique to other games?
- What other applications are there for this technique besides games?

Sources of Error

There were potential sources of error in our project:

- We wrote the code for the Connect Four game engine and training process. There might be bugs in the code due to our inexperience, that were preventing training from working effectively. There might also be other programming techniques that we did not use that would perform better.

- We tried our best to choose good genetic algorithm parameters, such as the size of the population, the number of games in a series, the mutation rate, and so on. It's possible that better choices for these parameters would give better results.

- Our time and compute power was limited during this experiment. We trained for 20 days on one computer.

- The neural network parameters include the size and number of layers, and the types of computation that each layer performs. There are many possible choices for the shape of the network, and we might have chosen a shape that was not very effective for learning Connect Four.

Glossary

Core AI & Machine Learning Terms

Artificial Intelligence (AI)

The field of computer science focused on creating machines or software that can perform tasks that normally require human intelligence, such as learning, reasoning, and decision-making.

Machine Learning (ML)

A branch of artificial intelligence in which computers learn patterns from data and improve their performance without being explicitly programmed for every task.

AI Model

A computer program or mathematical system that has been trained on data to make predictions or decisions.

Training (AI Training)

The process of teaching an AI model by showing it data and adjusting its internal values so it produces better outputs.

Model Performance

A measure of how well an AI model completes its intended task, often evaluated using accuracy or success rate.

Autonomous / Autonomously

Able to operate independently without direct human control.

Parameters

Internal numeric values in a model that influence how input data is processed.

Optimization

The process of adjusting a model's parameters to improve its performance.

Algorithm

A defined set of step-by-step instructions used by a computer to solve a problem.

Genetic Algorithm (GA)

An optimization method inspired by natural selection that improves solutions through mutation, selection, and reproduction.

Non-Genetic Algorithms

Algorithms that improve performance without using evolutionary or biological principles.
Natural Selection (AI Context)
A process where better-performing models are selected to continue while weaker ones are discarded.

Population (of Models)
A group of AI models trained and evaluated together during genetic training.

Fitness / Fit
A score that represents how well a model performs compared to others.

Mutation
A small random change made to a model's parameters to create variation.

Genes (AI Meaning)
Values in a model, such as weights, that are modified and passed on during genetic training.

Cloning (Models)
Creating a copy of an AI model, often before applying mutations.

Ranking (Models)
Ordering models based on their performance or fitness scores.

Neural Network–Specific Terms

Neural Network
A computing system inspired by the human brain, made up of connected artificial neurons that process information.

Neural Net / Neural Nets
Shortened terms for neural networks.

Artificial Neuron
A basic unit in a neural network that receives input, applies weights, and produces output.

Layers
Groups of neurons in a neural network that process data in stages.

Weights
Numeric values that control how strongly inputs affect a neuron's output.

Activation
The output produced by a neuron after applying a mathematical function.

Input Layer
The first layer of a neural network that receives raw data.

Output Layer
The final layer of a neural network that produces the model's prediction.

Linear Neural Network
A neural network that uses only linear mathematical operations.

Convolutional Neural Network (CNN)
A neural network designed to recognize patterns in structured data such as images or game boards.

Convolution
A mathematical operation that scans input data to detect patterns.

Two-Dimensional Network
A neural network that processes data arranged in rows and columns.

Model Shape / Size
The number of layers and neurons in a neural network.

Loss Function
A calculation that measures how incorrect a model's output is.

Reward Function
A rule that assigns positive feedback when a model performs well.

Penalty
Negative feedback given to discourage poor or invalid actions.

Magnitude (of Weights)
The size or absolute value of a model's weight numbers.

Random Weights
Initial weight values chosen randomly before training begins.

ReLU (Rectified Linear Unit)
An activation function that outputs zero for negative values and the input value for positive values.

Feature / Pattern Detection
The process of identifying important structures or regularities in data.

Training & Learning Methods

Reinforcement Learning
A learning method where an AI improves through rewards and penalties based on its actions.

Move Reinforcing
Strengthening certain actions by rewarding moves that lead to success.

Self-Play
Training where an AI plays against itself to learn strategies.

Trial and Error (AI Meaning)
A learning process where actions are tested and adjusted based on results.

Rewarding
Providing positive feedback to reinforce good actions.

Penalizing
Applying negative feedback to discourage poor actions.

Exploration (Random Moves)
Occasionally choosing random actions to discover new strategies.

Training Generation
One full cycle of evaluation and improvement in genetic training.

Parallel Training
Running multiple training processes at the same time to increase speed.

Training Parameters
Settings that control how training is performed.

Training Logs
Recorded data showing how a model performs during training.

Restarted Training
Beginning the training process again, usually with new settings or weights.

Game AI & Decision-Making Terms

Game Engine
Software that controls the rules, logic, and mechanics of a game.

AI Engine
The component of a game engine responsible for decision-making.

Invalid Moves
Actions that break the rules of the game.

Board State / State of the Board
A snapshot of all current positions and conditions in a game.

Game Policy
A strategy that determines which action an AI should take in a given state.

Evaluation
Assessing how good a move or model is.

Prediction
A decision or output produced by a model based on current data.

Monte Carlo Tree Search (MCTS)
A decision-making algorithm that simulates many possible future moves to choose the best one.

Programming & Development Terms

Python
A high-level programming language commonly used for AI and data science.

NumPy
A Python library used for working with numerical data and arrays.

Array
An ordered collection of values stored in memory.

Version Control
A system that tracks changes to code over time.

Git
A version control system used to manage code changes.

GitHub
An online platform for storing and sharing Git repositories.

Repository
A storage location for code and related files.

IDE (Integrated Development Environment)
Software that provides tools for writing and debugging code.

VS Code
A popular IDE developed by Microsoft.

Compiler
Software that converts code into a form a computer can execute.

Binary
A system of numbers using only 0s and 1s, used by computers.

Remote Access
Connecting to another computer over a network.

Server
A computer that provides services or resources to other computers.

Parallel Processing
Executing multiple computations at the same time.

CPU Training
Training AI using a computer's main processor.

GPU Training
Training AI using a graphics processor for faster computation.

Hardware & Compute Terms

CPU (Central Processing Unit)
The main processor responsible for executing instructions.

GPU (Graphics Processing Unit)
A processor designed for parallel computations, commonly used in AI.

Integrated Graphics
Graphics processing built into the CPU.

Dedicated GPU
A separate graphics processor designed for high-performance tasks.

Processing Power
The ability of hardware to perform computations efficiently.

General Terms

Model
A system used to represent and solve a problem.

Simulation
A computer-based imitation of a real or theoretical process.

Iteration
One repeated step in a process.

Baseline
A reference point used for comparison.

Performance Metrics
Measurements used to evaluate how well a model performs.

Sources

References

GeeksforGeeks. (2016, June 14). *Minimax Algorithm in Game Theory | Set 1 (Introduction)*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/minimax-algorithm-in-game-theory-set-1-introduction/

GeeksforGeeks. (2017a, June 29). *Genetic Algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/genetic-algorithms/

GeeksforGeeks. (2017b, August 21). *Introduction to Convolution Neural Network*. GeeksforGeeks. https://www.geeksforgeeks.org/machine-learning/introduction-convolution-neural-network/

GeeksforGeeks. (2017c, September 8). *What is Artificial Intelligence(AI)?* GeeksforGeeks. https://www.geeksforgeeks.org/artificial-intelligence/what-is-artificial-intelligence-ai/

GeeksforGeeks. (2019, January 14). *ML | Monte Carlo Tree Search (MCTS)*. GeeksforGeeks. https://www.geeksforgeeks.org/machine-learning/ml-monte-carlo-tree-search-mcts/

Leung, S. (n.d.). *Genetic Algorithms* [Personal communication].

PyGAD - Python Genetic Algorithm! — PyGAD 3.2.0 documentation. (n.d.). Pygad.readthedocs.io. https://pygad.readthedocs.io/

Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2021). Monte Carlo Tree Search: A Review of Recent Modifications and Applications. *ArXiv:2103.04931 [Cs]*. https://arxiv.org/abs/2103.04931

Wikipedia. (2019, February 18). *Artificial Intelligence*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Artificial_intelligence

Wikipedia Contributors. (2019, November 16). *Connect Four*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Connect_Four

Zvornicanin, E. (2022, April 11). *Convolutional Neural Network vs. Regular Neural Network | Baeldung on Computer Science*. Www.baeldung.com. https://www.baeldung.com/cs/convolutional-vs-regular-n

Appendix - Source code

```
import sys
import torch
import torch.nn as nn
import torch.nn.functional as F
import random
import numpy as np
import math
import copy
import pickle
import argparse
#from torch.multiprocessing import Pool, set_sharing_strategy,
set_start_method

ROWS, COLS = 6, 7
player = 1
board = np.zeros((ROWS,COLS), dtype=int)

def print_board():
    for r in range(ROWS):
        for c in range(COLS):
            sys.stdout.write(f"{board[r][c]}")
        print("")
    print("")

DRAW = 0
WIN = 1
KEEP_PLAYING = 2
INVALID_MOVE = 3

random_move_threshold = 0.25

def drop_piece(col):
    global player
    for row in reversed(range(ROWS)):
        if board[row,col] == 0:
            board[row,col] = player
            if check_win(player):
                #print(f"Player {player} wins")
                #print_board()
                reset_board()
                return WIN
            check_draw()
            if check_draw():
                #print('Draw!')
                reset_board()
                return DRAW
            player = 2 if player == 1 else 1
            return KEEP_PLAYING
    print("Invalid move! You went above the board, try again")
    return INVALID_MOVE
```

```python
def check_win(p):
    for r in range(ROWS):
        for c in range(COLS - 3):
            if all(board[r,c+i] == p for i in range(4)): return True
    for r in range(ROWS - 3):
        for c in range(COLS):
            if all(board[r+i,c] == p for i in range(4)): return True
    for r in range(ROWS - 3):
        for c in range(COLS - 3):
            if all(board[r+i,c+i] == p for i in range(4)): return True
            if all(board[r+3-i,c+i] == p for i in range(4)): return True
    return False

def check_draw():
    for c in range(COLS):
        if board[0,c] == 0:
            return False
    return True

def reset_board():
    global board, player
    board = np.zeros((ROWS, COLS), dtype=int)
    player = 1

# represents the data of an AI model
class Connect4Model(nn.Module):

    def __init__(self, input_dim, output_dim):
        super(Connect4Model, self).__init__()
        self.conv1 = nn.Conv2d(2, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 32, 3, padding=1)
        self.conv4 = nn.Conv2d(32, 32, 3, padding=1)

        self.policy_conv = nn.Conv2d(32, 2, 1)
        self.policy_fc = nn.Linear(2 * ROWS * COLS, COLS)
        self.win_count = 0
        # list of (board_state tensor, move) tuples
        self.moves = []

    def __lt__(self, other):
        return self.win_count < other.win_count

    # this function tells pytorch how to apply the layers of the neural net
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))

        x = F.relu(self.policy_conv(x))
        return self.policy_fc(x.flatten())

    # save the model to a file
    def save(self, filename):
```

```
        torch.save(self.state_dict(), filename)

    # load the model from a file
    def load(self, filename):
        self.load_state_dict(torch.load(filename, weights_only=True))
        self.eval()

    def board_to_tensor(self, board, player):
        # convert to (2, 6, 7) tensor.  first (6,7) array is current
        # player's pieces, second (6,7) array is opponent's
        tensor = torch.zeros((2,6,7), dtype=torch.float32)
        for r in range(ROWS):
            for c in range(COLS):
                value = board[r][c]
                if value == player:
                    tensor[0,r,c] = 1.0
                elif value != 0:
                    tensor[1,r,c] = 1.0
        return tensor.cuda()

    # ask the model which move it should play, given the board
    # position as a tensor from board_to_tensor.  Returns a column to
    # play in.
    def play_move_tensor(self, board_tensor, mask):
        # with low probability, play a random legal move - this is
        # how we explore new solutions and try to learn from them if
        # they're good/bad
        if random.random() < random_move_threshold:
            # get indexes of legal moves from the mask
            legal = list([x[0] for x in enumerate(mask) if x[1] == 0.0])
            move = random.choice(legal)
        else:
            with torch.no_grad():
                outputs = self.forward(board_tensor)
            # apply the mask to the outputs and return the highest-rated
move
            output_tensor = outputs - mask
            move = torch.argmax(output_tensor).item()
        self.moves.append((board_tensor.detach(), move))
        return move

    # ask the model which move it should play, given the board
    # position and this model's player.  Returns a column to play in.
    def play_move(self, board, player):
        # construct a mask of valid values - an entry in the list is
        # True if it's legal to move there
        mask = torch.tensor([0.0 if board[0, i] == 0 else math.inf for i in
range(COLS)]).cuda()
        board_tensor = self.board_to_tensor(board, player)
        return self.play_move_tensor(board_tensor, mask)

    # return a copy of this model, with some weights randomly mutated
    def get_mutant(self):
        mutant = copy.deepcopy(self)
        # modify the mutant
```

```
        with torch.no_grad():
            for param in mutant.parameters():
                for val in param.flatten():
                    # 1% mutation rate
                    if random.random() < 0.01:
                        # mutate the parameter by multiplying by the range
-2.0 to 2.0
                        val *= random.uniform(-2.0, 2.0)
        return mutant


def reward_move(model, board_tensor, move, reward, optimizer, criterion):
    outputs = model(board_tensor)
    # we can make the target have the same outputs, but with a
    # slightly larger value for this move
    target = outputs.clone()
    if reward > 0:
        target[move] = torch.max(outputs) + reward
    else:
        target[move] = torch.min(outputs) + reward
    loss = criterion(outputs, target)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

def reward_model(model, reward):
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

    # reward the last move
    reward_move(model, model.moves[-1][0], model.moves[-1][1], reward,
optimizer, criterion)

    # remove the winning move and reward a random set of 5 of the
    # remaining moves
    if reward == 1.0:
        model.moves.pop()
        random.shuffle(model.moves)
        for move in model.moves[:5]:
            reward_move(model, move[0], move[1], reward, optimizer,
criterion)

# have two models play a game until one wins.
# if a model is None then a human needs to play.
# needs to return which model wins the game.
# Also need a way to say it was a draw
def play_one_game(model1, model2):
    players = [model1, model2]
    reset_board()
    if model1 is not None:
        model1.moves.clear()
    if model2 is not None:
        model2.moves.clear()
    player_index = random.randint(0,1)
    while True:
```

```
        m = players[player_index]
        if m is None:
            print_board()
            print(f"Type a column number. Player {player}'s move:")

            try:
                move = int(input())
                if move == 0:
                    print('Quiting...')
                    sys.exit(0)
                move = move - 1

            except (ValueError, IndexError):
                print("Invalid move! Type a valid column, try again")
                continue
        else:
            move = m.play_move(board, player_index + 1)
            #print(f'AI player {player} moved in column {move + 1}')
        result = drop_piece(move)
        if result == DRAW:
            return
            break
        elif result == WIN:
            if m is None:
                print("You won")
            # reinforce winning move
            if m is not None:
                reward_model(m, 1.0)
            # punish losing move
            other_m = players[(player_index + 1) % 2]
            if other_m is not None:
                reward_model(other_m, -1.0)
                return player_index
                break
        elif result == INVALID_MOVE:
            if m is None:
                # human made a mistake, let them try again
                continue
            else:
                print("AI was not supposed to make an invalid move")
                sys.exit(0)

        player_index = (player_index + 1) % 2

# Play a series of games between two models, and return the win counts of
each
def play_series(model1, model2, number_of_games):
    model1_wins = 0
    model2_wins = 0
    for i in range(number_of_games):
        result = play_one_game(model1, model2)
        if result == 0:
            model1_wins += 1
        elif result == 1:
            model2_wins += 1
```

```python
    return (model1_wins, model2_wins)

# play the ith model against models numbered higher in the given list of
models
def play_i(i, models, number_of_games):
    all_wins = [0] * len(models)
    for j in range(i+1, len(models)):
        wins = play_series(models[i], models[j], number_of_games)
        print(f"model {i} won {wins[0]}, model {j} won {wins[1]}")
        all_wins[i] = wins[0]
        all_wins[j] = wins[1]
    return all_wins

class GeneticAlgorithm:
    def __init__(self, num_models):
        self.num_models = num_models
        self.models = []
        self.generation = 0
        for i in range(num_models):
            self.models.append(Connect4Model(ROWS * COLS, COLS))

    # updates win count on each model
    def everyone_play_everyone(self):
        # initialize everyone's count to 0
        for model in self.models:
            model.win_count = 0

        def store_result(all_wins):
            for m, w in zip(self.models, all_wins):
                m.win_count += w

#         # allocate a list to store async results from running models in
parallel
#         async_results = [None] * self.num_models
#
#         # assume a 12-thread CPU for now
#         with Pool(12) as p:
#             # async play each model against the rest
        for i in range(self.num_models):
            store_result(play_i(i, self.models, 100))

    # mutate survivors until population is full
    def fill_population(self):
        survivor_count = len(self.models)
        while len(self.models) < self.num_models:
            to_mutate = self.models[random.randint(0, survivor_count - 1)]
            self.models.append(to_mutate.get_mutant())

    def generation_step(self):
        # have everyone play everyone else
        self.everyone_play_everyone()

        # rank the models by decreasing number of wins
        self.models.sort(reverse=True)
```

```
        print(f"generation {self.generation} best model won
{self.models[0].win_count} games, worst won {self.models[-1].win_count}")

        # keep only the top half of the models
        num_keep = self.num_models // 2
        self.models = self.models[:num_keep]

        # mutate the surviving models into a new population
        self.fill_population()
        self.generation += 1


if __name__ == "__main__":
#     set_start_method('spawn')

#     model = Connect4Model(ROWS * COLS, COLS)
#     play_one_game(None, model)

    parser = argparse.ArgumentParser(prog='connect4_engine')
    parser.add_argument('type', choices=['genetic', 'normal', 'compete'])
    parser.add_argument('action', choices=['play', 'train'])
    args = parser.parse_args()

    if args.type == 'compete':
        # play at full strength, no random moves
        random_move_threshold = 0.0

        if args.action != 'play':
            raise RuntimeError('compete can only play')
        with open('genetic_data', 'rb') as f:
            genetic = pickle.load(f)
        with open('normal_data', 'rb') as f:
            normal = pickle.load(f)
        wins = play_series(genetic.models[0], normal.models[0], 2000)
        print(f'genetic won {wins[0]}, normal won {wins[1]}')
        sys.exit(0)

    if args.type == 'genetic':
        filename = 'genetic_data'
        num_models = 40
    elif args.type == 'normal':
        filename = 'normal_data'
        num_models = 1

    try:
        with open(filename, 'rb') as f:
            gen = pickle.load(f)
            gen.num_models = num_models
    except:
        gen = GeneticAlgorithm(num_models)
        for model in gen.models:
            torch.nn.init.uniform_(model.conv1.weight)
            torch.nn.init.uniform_(model.conv2.weight)
            torch.nn.init.uniform_(model.conv3.weight)
            tornich.nn.init.uniform_(model.conv4.weight)
```

```
        torch.nn.init.uniform_(model.policy_conv.weight)
        torch.nn.init.uniform_(model.policy_fc.weight)

for model in gen.models:
    model.cuda()

if args.action == 'train':
    if args.type == 'genetic':
        for _ in range(2):
            gen.generation_step()
            with open(filename, 'wb') as f:
                pickle.dump(gen, f)
    elif args.type == 'normal':
        # Just have the model play a lot of games with itself, since
        # we reinforce winning/losing moves as we go
        play_series(gen.models[0], copy.deepcopy(gen.models[0]), 2000)
        with open(filename, 'wb') as f:
            pickle.dump(gen, f)
elif args.action == 'play':
    # play at full strength, no random moves
    random_move_threshold = 0.0

    play_one_game(None, gen.models[0])
```