The Ancient Game and the AI

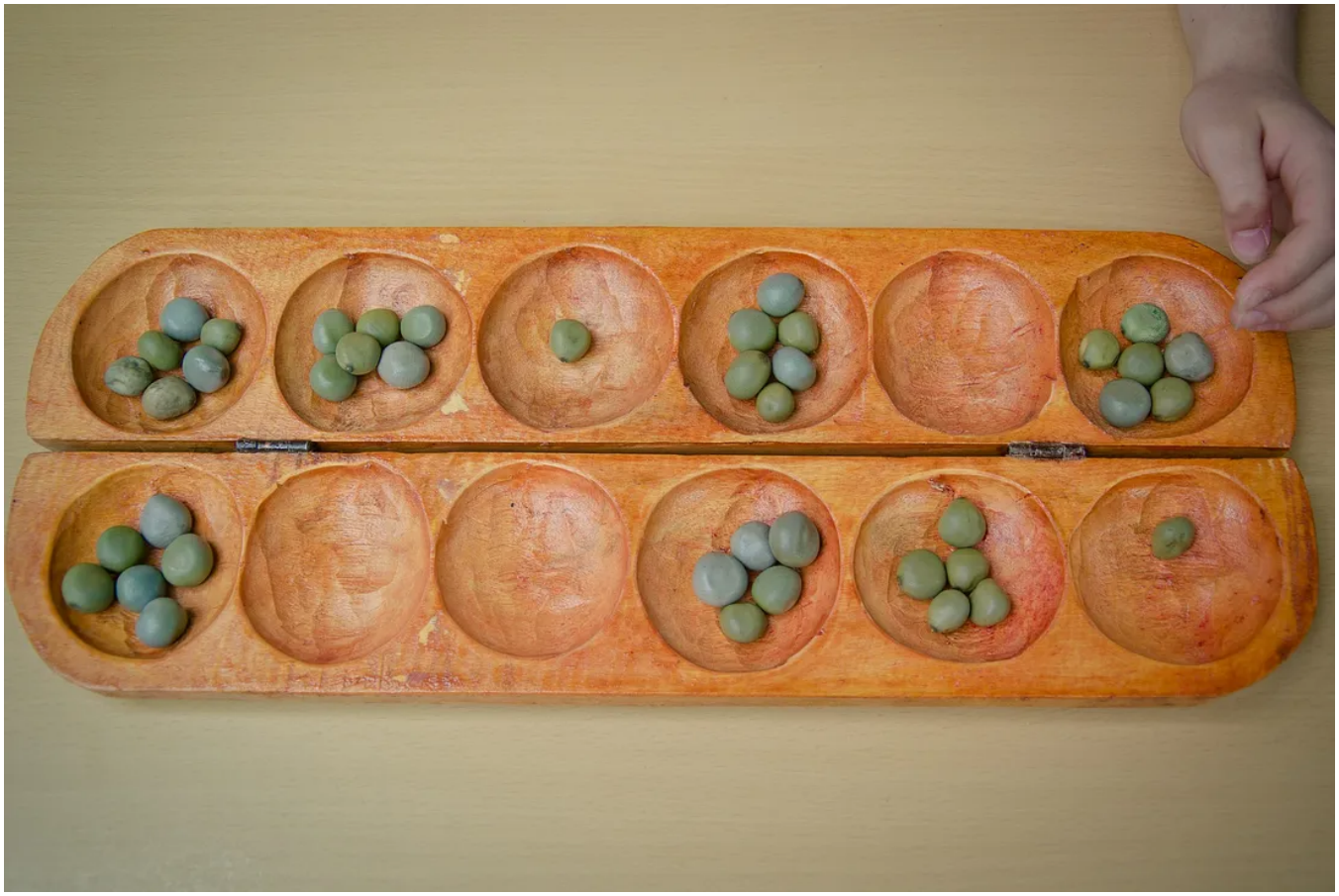
Erik Langenborg · Follow
Published in Towards Data Science
9 min read · May 14, 2017

Games are fantastic training tools, for artificial agents and humans alike. Let's step through how to codify a game, train AI bots, and determine the most effective player. Along the way, we learn about our game of choice.

By Matt DaVolio (GitHub) and Erik Langenborg (GitHub).

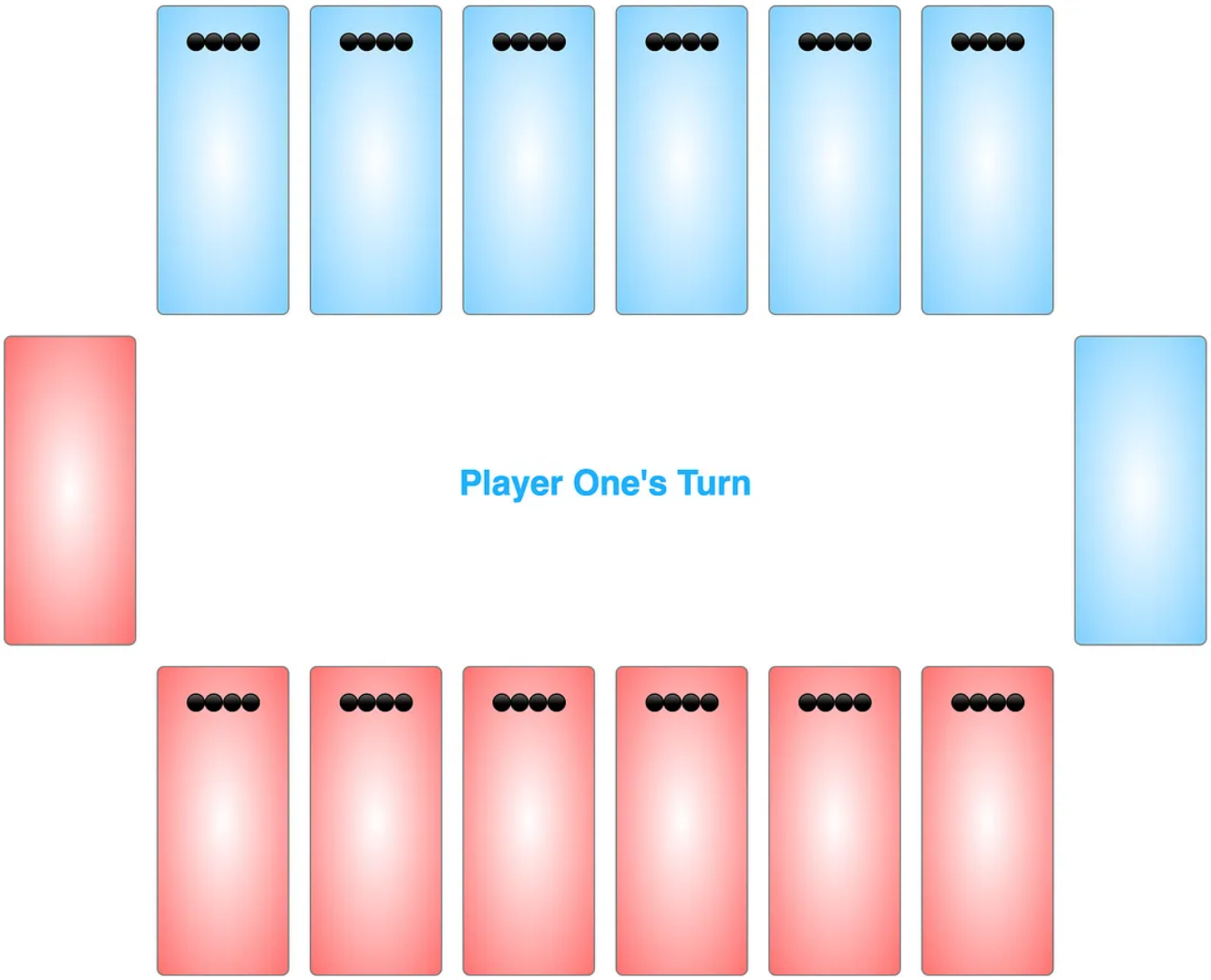
Mancala

For our purposes, we will make use of a very old game.



Mancala (Oware)

Mancala's origins date back to the 6th century. Many variations of the game have evolved through the years. This article will use the Oware version popular in the western world.



Mancala UI

The Mancala game board consists of 14 dishes, two of which are score dishes and the other 12 are split between the two players. To begin the game four stones are placed in each of the 12 non-score dishes. A move is made by a player choosing one of their six dishes which contains stones. The stones in the chosen dish are all picked up and then each stone is placed one at a time in the next dish moving in a clockwise motion. A stone can be placed in each of the twelve non-scoring dish as well as the current player's scoring dish. The opponent's score dish will be skipped over. Play alternates between players. The game is over when one player scores 25 or more points.

Two additional, common rules are applied as well. When a player's move results in

the last stone being placed in the scoring dish, they may take another turn. The capture rule occurs if two conditions are reached: the last stone a player drops on their move lands in an empty dish on the current player's side and there exists at least one stone in the adjacent dish on the opponent's side. In this case, the current player captures the last stone dropped along with all the stones in the opponent's dish.

Game State and Actions

Mancala's board consists of 14 dishes, each of which can contain any number of the stones. This makes for a simple *Array<int>* representation of the board. The only other piece of the state concerns the current player turn. A single *boolean* tracks this information. Of additional note: this state is Markov: a memoryless system where all relevant information is contained in the current state.

Viable actions are choosing one of the 12 dishes possible for collecting stones to redistribute. On every turn a player can only move on their side of the board, limiting them to six options. Further, only dishes with stones may be selected.

```
g = Game()       # Initial State
g.board()        # [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]
g.turn_player() # 1

g.move(1)        # Perform a move

g.board()        # [4, 0, 5, 5, 5, 5, 0, 4, 4, 4, 4, 4, 4, 0]
g.turn_player() # 2
```

mancala.game.usage.py hosted with ❤ by **GitHub** **view raw**

Mancala is a game featuring perfect information, like tic-tac-toe, chess, or Go. Therefore agents' view of the state is always complete and they are handed the full game to make a decision.

Agents

An agent is simply a system to make choices in the environment. This article will step through the attempted agents in rough order of complexity. The pseudo-code provided covers the main points, but visit the main repository for proper details.

Random Agent

Lets examine the baseline system: the Random Agent.

```
class AgentRandom(Agent):
    def move(self, game):
        options = Agent.valid_indices(game)
        return random.choice(options)
```

random.agent.py hosted with ❤ by **GitHub** **view raw**

The Random Agent simply looks at the available choices and picks randomly. While not much of a challenge, this AI equivalent of an identity function serves as a good baseline. In the future, the first question for any agent is: can it beat random chance?

Max Agent

Since Mancala lacks a random component, the results of any one action can be predicted with perfect accuracy. The Max Agent calculates the score from every possible move and picks the choice with the highest payoff. This is an expert agent, meaning the logic is derived from human written rules.

```python
class AgentMax(Agent):
    @staticmethod
    def _score_of_move(move_test, game):
        """Makes the move and returns the score of player one"""
        game.move(move_test)
        return game.score()[0]

    def move(self, game):
        move_options = Agent.valid_indices(game)
        available_scores = [AgentMax._score_of_move(move, game) for move in move_option

        score_max = max(available_scores)
        final_options = [move for score, move in
                         zip(available_scores, move_options)
                         if score == score_max]

        return random.choice(final_options)
```

max.agent.py hosted with ❤ by **GitHub** **view raw**

Exact Agent

An enhancement from the Max Agent takes advantage of the repeated move rule in Mancala. By favoring actions that do not surrender the turn, this agent can exploit increasing the lead when handed the turn. The first move the Exact agent searches for is any move in which the number of stones in a hole matches the distance to the player's score hole. This would allow the player to make another move. If more than one hole fits the condition, the hole closest to the score hole will be chosen. This allows the player to use this situation multiple times as a hold further away would overtake the closer hole.

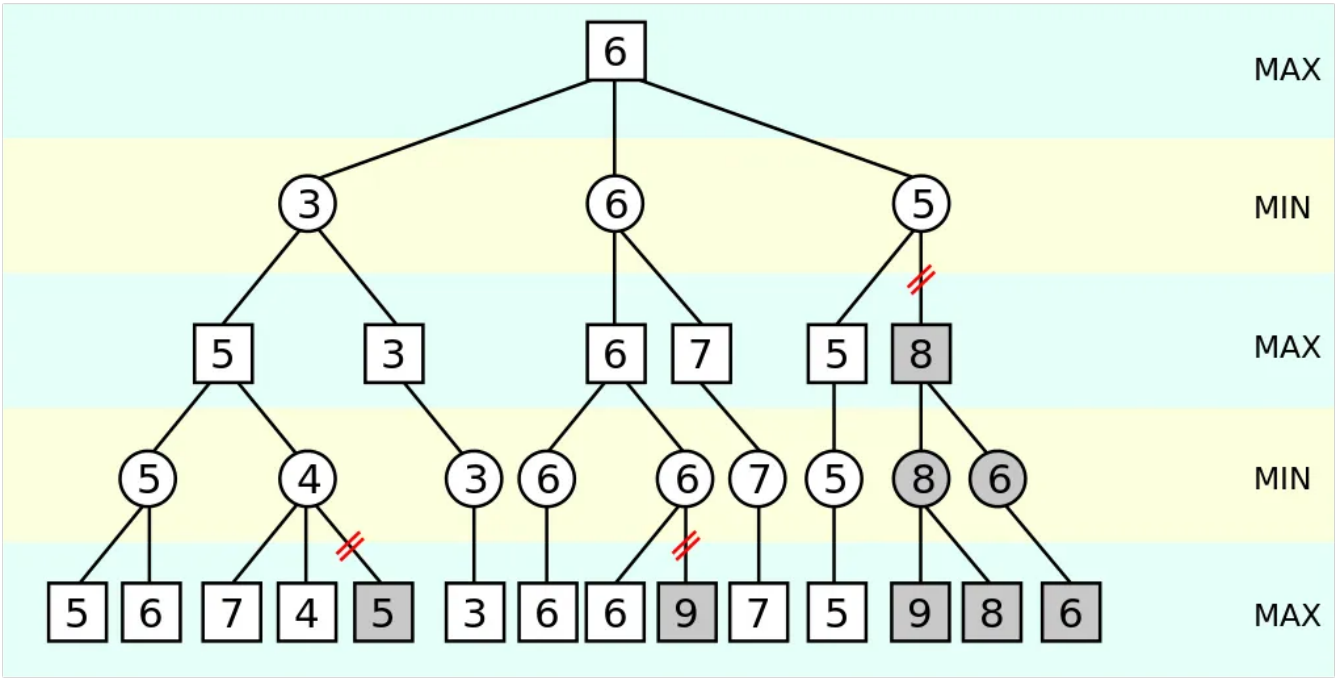
If no possible move fits the conditions, the agent falls back onto the highest possible score.

MinMax Agent

The MinMax Agent is a technique that explores possible actions to a certain depth and chooses the path that has the maximum desired result.

The agent traverses a large tree representing the choices at each tree depth. The

agent maximizes the score when the simulation gives it control and minimize the score the agent's score when the opponent would be in control. The name is derived from this pattern. When presented with a game state, the agent builds a representative tree and plays the root node which produces the maximum score for the current player. Since the number of moves increase exponentially with each depth level, this agent entails a lot of overhead.



Alpha-Beta Pruning Example [2]

Alpha-beta pruning alleviates this problem by reducing the number of branches searched during a traversal of the tree. Evaluation of branches is abandoned if the agent realizes the path cannot possibly perform better than a previously observed choices. Through the traversal, two variables on stored: alpha, the player's maximum score guaranteed for the branch, and beta, the opponent's guaranteed minimum score. During the traversal on a maximizing level (agent is playing), if the node does not increase alpha, the branch is pruned and the agent moves on. Conversely on minimizing levels, nodes are abandoned if the beta does not decrease. The observed performance increase for our chosen depths was roughly an order of magnitude.

Monte Carlo Tree Search Agent

Monte Carlo Tree Search (MCTS) has risen as a popular method in two player games

recently largely due to the success of the AlphaGo bot which includes it as a component. MCTS is an advancement on a MinMax tree search and alpha-beta

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Example Updating Q Values — [1]

randomly plays out the game to completion and records the win/loss. In effect, MCTS uses the random win/loss result in place of MinMax's fitness function. After many iterations, favorable paths are scored higher by their final results. Due to the large state space of the game, the agent requires a large number of iterations to survey enough of the available moves even at a shallow depth.

But many applications suffer with these methods because the state/action space is huge. A rough ballpark for Mancala's possible states would be upper bounded by 12^{48} (12 playable dishes, 48 available stones) $\approx 10^{50}$ — this is far too many to explore. There are a few ways to sidestep this problem, with the more successful versions detailed below.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Kocsis and Szepesvari's score for selecting nodes [3]

The basic Q-Learning attempt for Mancala involved collapsing the state space by approximating the stones in the each dish. Noting if the dish was empty or not reduced the state space to 12^2, a mere 4096 possible states and easily explored. The result fared terribly: this state representation was far too imprecise. This state representation discarded too much detail. Similar methods of breaking down the state failed and once again pushed the number of required exploration paths beyond reasonable computation.

Throughout these iterations a score for each possible node is recorded to tabulate the value of the chosen node. As each iteration is performed, the agent backpropagates the result through the tree to update the node scores.

Q Learning Agent

Deep Q Learning Agent

Q Learning is a Reinforcement Learning method to estimate the action values (Q) for every state. The actual value is derived from how the designer structures the rewards. Aside from the typical task of distilling state (often more complex than this Mancala example), the designer must select when and how much the system rewards the agent. A simple strategy might be to reward on a perceived advantageous position, but not the final desired outcome. However, most circumstances benefit from allowing the agent to learn and only reward when the end goal is reached. For Mancala, the goal is winning the game.

Deep Q Learning is a popular technique leveraging Reinforcement Q Learning with a deep neural network to represent the estimated action values (Q), eliminating the table of states/actions. The process is largely the same, but restructured to maximize efficiency for the network. An experience replay memory of *<state, action, state_next, reward>* is continually updated and sampled in order to train the network. Common techniques include using Convolutional Neural Networks (CNN) to analyze raw frames as state input. A small CNN was trained to grasp patterns in the board as a 6x2 greyscale image.

$$Q : S \times A \rightarrow \mathbb{R}$$

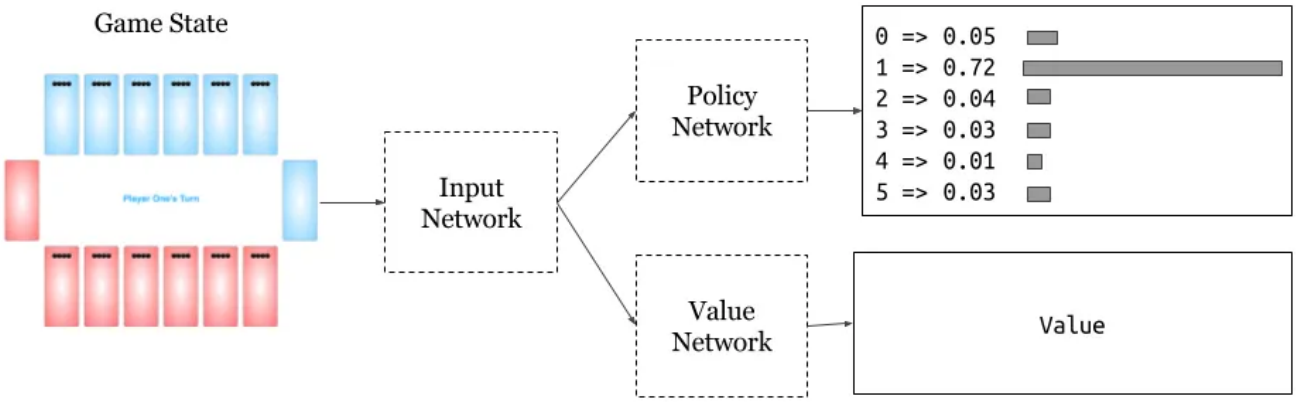
Q Values — Value of Actions given current state. [1]

Asynchronous Advantage Actor-Critic Agent

The A3C Agent is a policy estimation method that runs parallel agents instead of experience replay. Three networks work in concert: input, action, and critic. The In short, a Q function estimates how much reward the agent expects from an action

input network transforms the state to an embedded space, the policy network gives the probability distribution for available moves, and the value network describes the reward of the current state.

There are a variety of methods to discover the Q values for a system — such as SARSA (State-Action-Reward-State-Action). By walking through states, actions, and next states, the agent continually updates estimates for rewards. Some visits to a



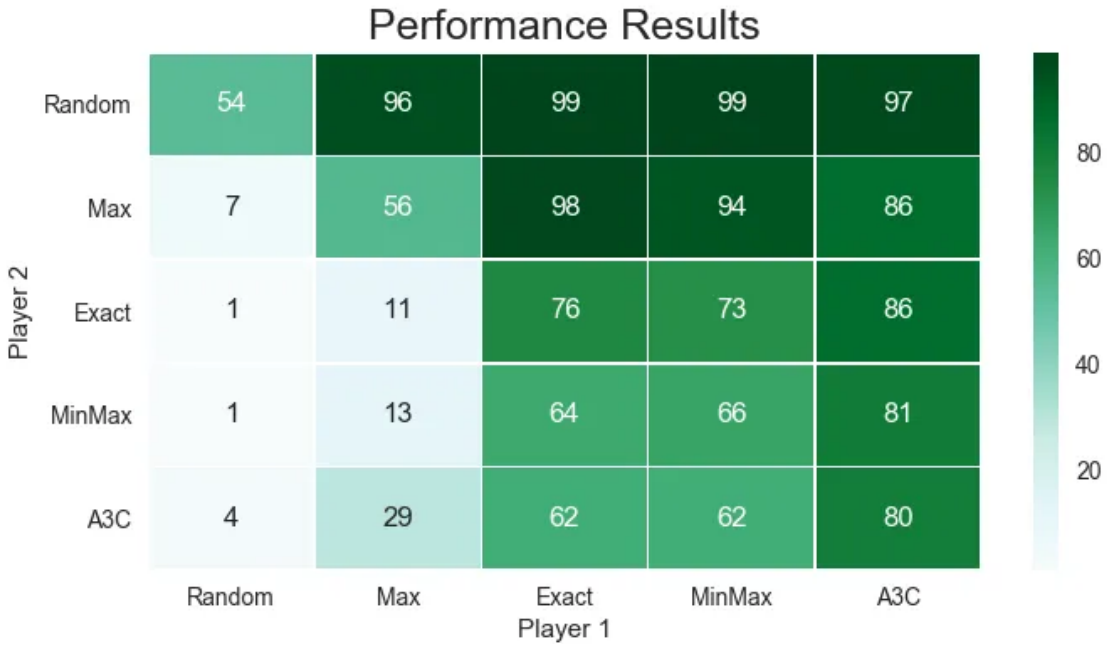
A3C Networks

Multiple agents simultaneously run the game with this shared model and update both policy and value networks based on the reward results of the actions. Aside from a linear speed up with each extra thread updating the model, the diverse games are less likely converge to similar minima (a common problem with normal actor-critic methods).

Results

The agents were compared with a simple process: play Mancala!

Every combination of agent was tested for 500 games, each starting with different seeds to allow stochastic elements to even out. Agents even played against themselves. For brevity, some ill-performing agents were omitted.



Values are % of wins for Player 1 vs Player 2

The agent performances were roughly in line with complexity: Random Agent as the worst and A3C Agent as the best. Expert agents performed better against lesser expert agents than the RL A3C Agent, while the A3C Agent hedged out the best expert agents.

Note the diagonal of agents versus themselves always favored the first player — Mancala holds a very strong first move advantage. MinMax won 62% of games against A3C as the first player, but only won 19% as the second player. Assuming they trade first move advantage, A3C would win 71.5% of games. The state-of-the-art Reinforcement Learning technique performed well.

This does not discount the expert agents: these compete and required neither expensive training nor loading of large models. The Exact Agent required only a few lines of code and was among the fastest to complete computation. By comparison, MinMax easily became infeasible when searching larger depths and A3C required loading of a model (entailing a GPU and associated libraries). Exact was a formidable opponent for most human players and does not demand heavy dependencies.

Built at the Data Science Institute at the University of Virginia.
See the code on GitHub.
By Matt DeVoe (GitHub) and Erik Langenborg (GitHub)

References

[1] https://en.wikipedia.org/wiki/Q-learning

[2] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Artificial Intelligence | Reinforcement Learning | Machine Learning | Deep Learning | Towards Data Science

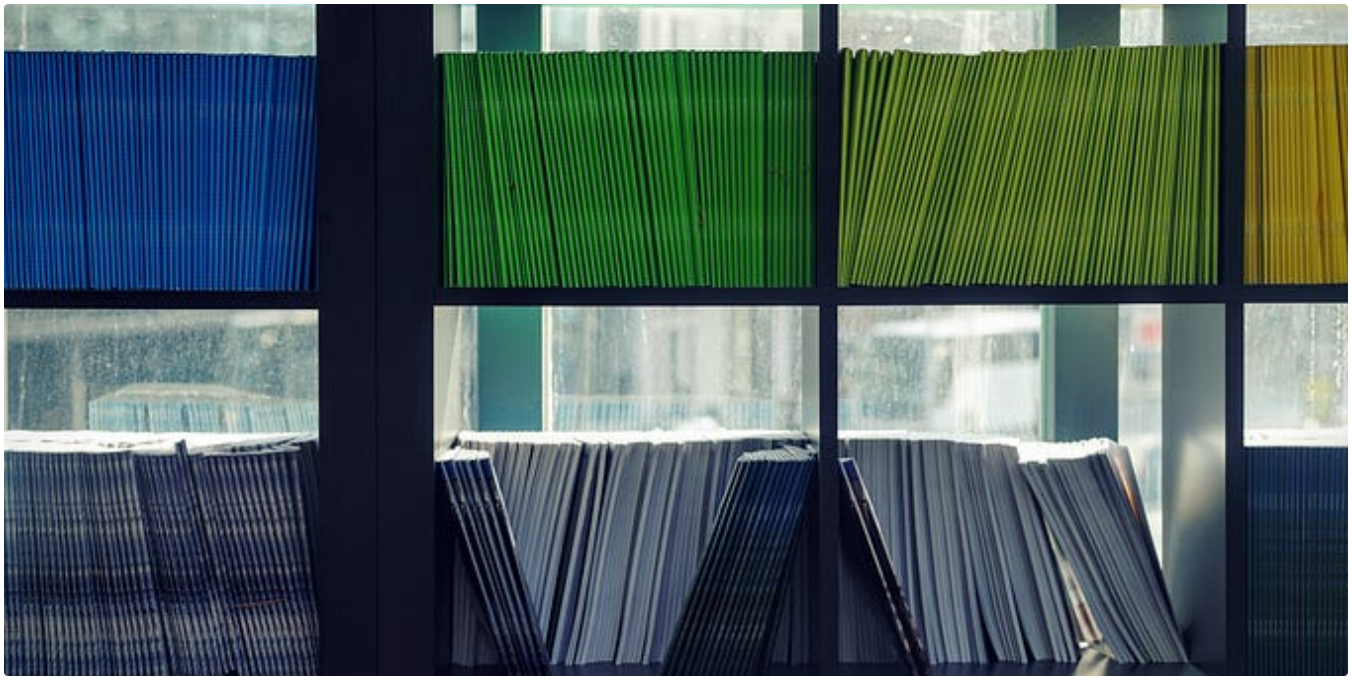
Follow

Written by Erik Langenborg

10 Followers · Writer for Towards Data Science

Data Scientist. Developer.

More from Erik Langenborg and Towards Data Science



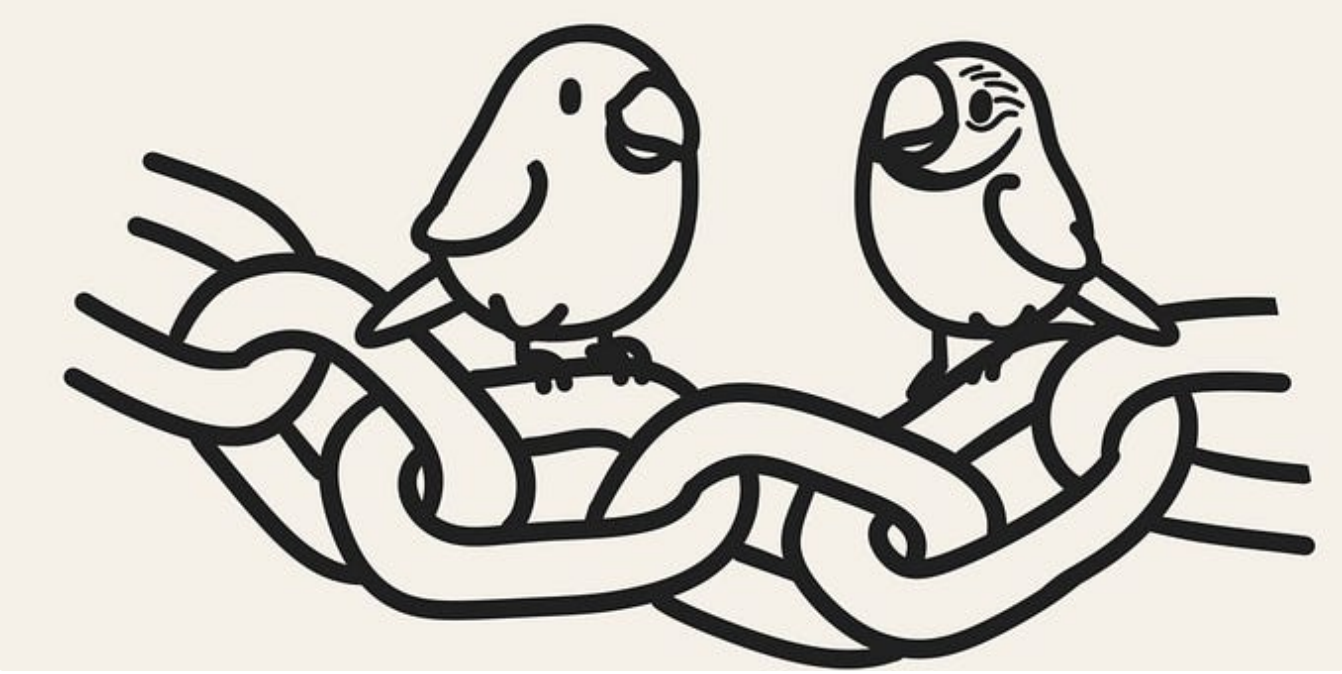
Jacob Marks, Ph.D. in Towards Data Science

How I Turned My Company's Docs into a Searchable Database with OpenAI

And how you can do the same with your docs

15 min read · Apr 25

3.5K 46

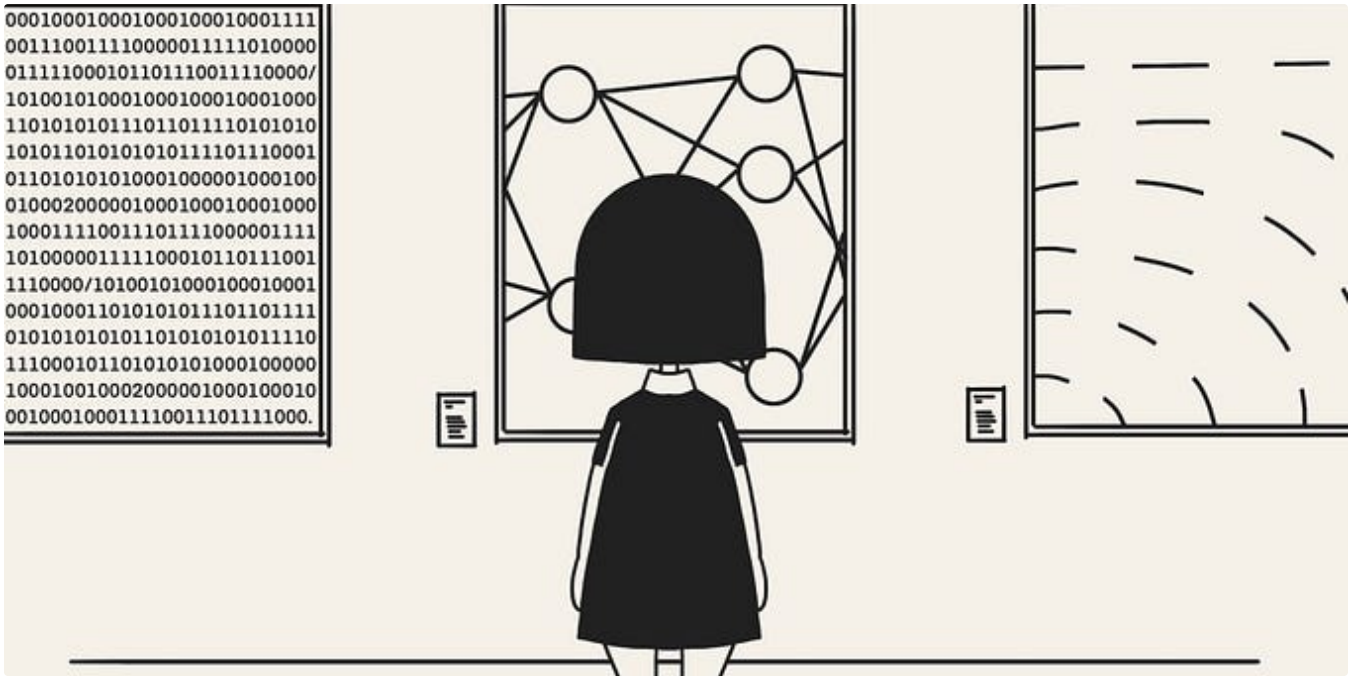


Leonie Monigatti in Towards Data Science

Getting Started with LangChain: A Beginner's Guide to Building LLM-Powered Applications

A LangChain tutorial to build anything with large language models in Python

· 12 min read · Apr 25



Leonie Monigatti in Towards Data Science

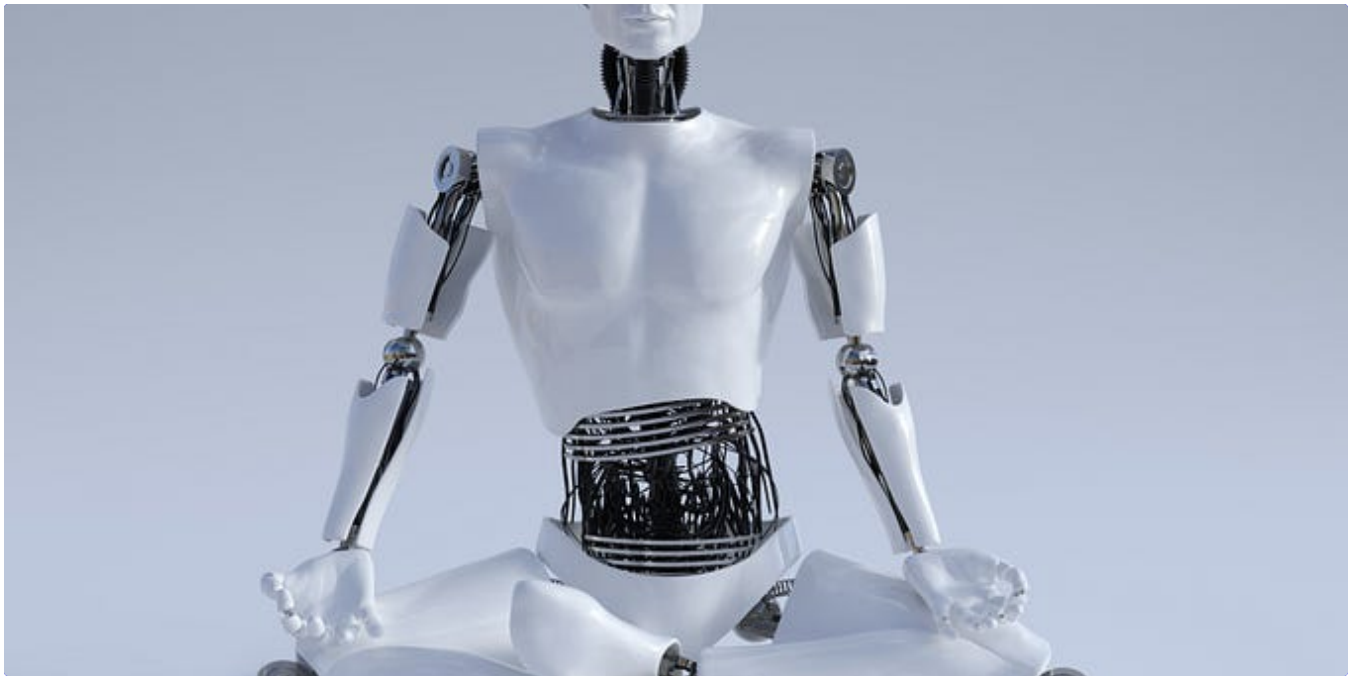
10 Exciting Project Ideas Using Large Language Models (LLMs) for Your Portfolio

Learn how to build apps and showcase your skills with large language models (LLMs). Get started today!

· 11 min read · May 15

1.1K 7

Recommended from Medium



Matt Chapman in Towards Data Science

How I Use ChatGPT With the Latest AI Trends Ahead of 99% of Data Scientist

Master ChatGPT by learning prompt engineering.

8 min read · May 17

23K 324

See all from Erik Langenborg

See all from Towards Data Science

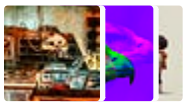
Wouter van Heeswijk, PhD in Towards Data Science

Proximal Policy Optimization (PPO) Explained

The journey from REINFORCE to the go-to algorithm in continuous control

· 13 min read · Nov 29, 2022

Lists

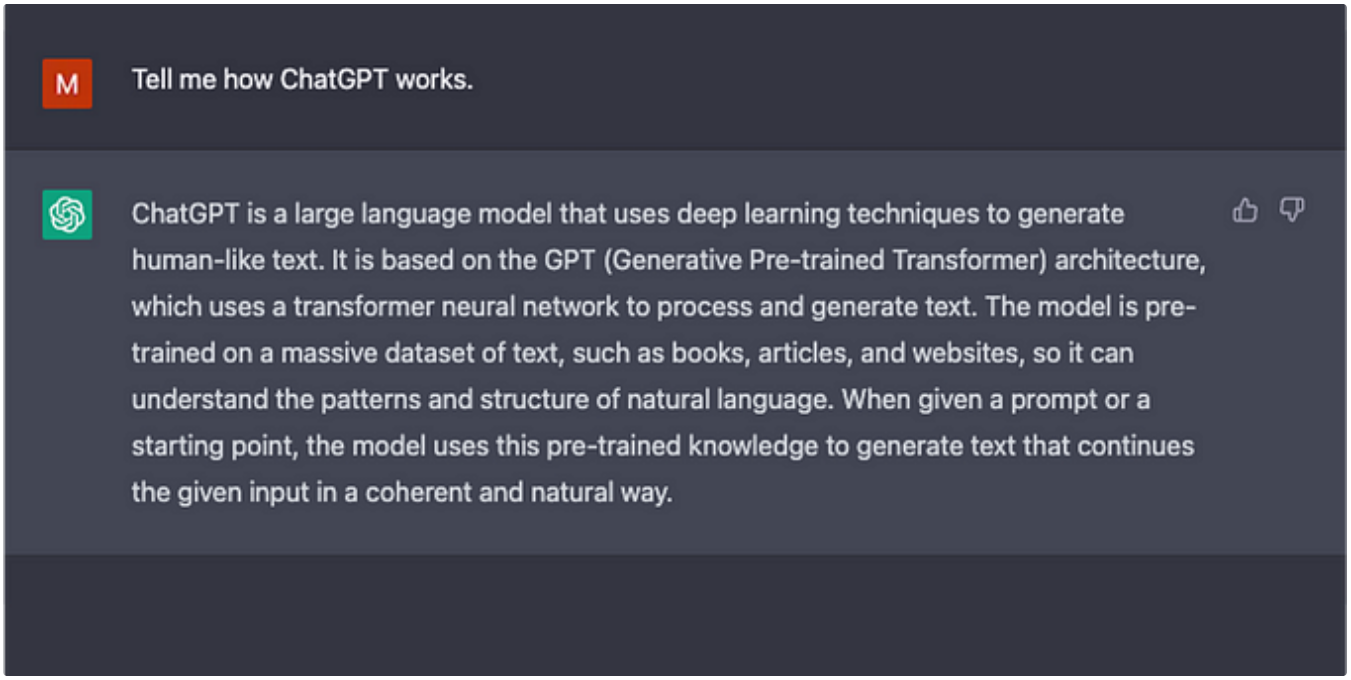


What is ChatGPT?

9 stories · 81 saves

Staff Picks

340 stories · 95 saves



Molly Ruby in Towards Data Science

How ChatGPT Works: The Models Behind The Bot

A brief introduction to the intuition and methodology behind the chat bot you can't stop hearing about.

· 8 min read · Jan 30

7.4K 125

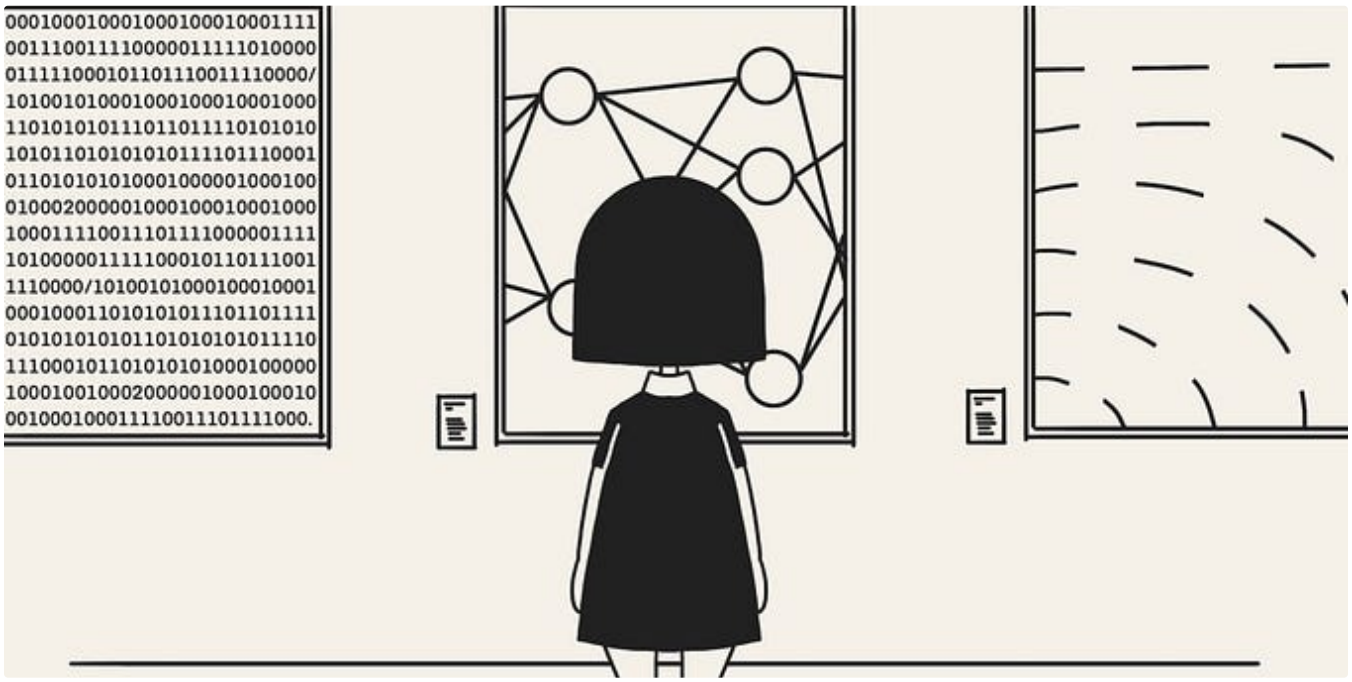
Leonie Monigatti in Towards Data Science

Getting Started with LangChain: A Beginner's Guide to Building LLM-Powered Applications

A LangChain tutorial to build anything with large language models in Python

· 12 min read · Apr 25

2.6K 20



Leonie Monigatti in Towards Data Science

10 Exciting Project Ideas Using Large Language Models (LLMs) for Your Portfolio

Learn how to build apps and showcase your skills with large language models (LLMs). Get started today!

· 11 min read · May 15

1.1K 7

Florent Poux, Ph.D. in Towards Data Science

3D Deep Learning Python Tutorial: PointNet Data Preparation

The Ultimate Python Guide to structure large LiDAR point cloud for training a 3D Deep Learning Semantic Segmentation Model with the...

· 30 min read · May 31

345 2

See more recommendations