

Medium

Search



★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Solving Sudoku by Heuristic Search



David Carmel · Follow

9 min read · Sep 17, 2021



Listen



Share

... More

Introduction

I've recently become interested in solving Sudoku puzzles, realizing that I'm really bad in it. I can handle the easy ones, however, puzzles marked as medium or difficult are too complicated for me. I guess my main weak-spots are the heavy book-keeping required, and that it is almost impossible to recover from an error done during the solution process.

Sudoku, meaning "singular number" in Japanese, consists of an $N \times N$ grid, divided into $B \times B$ blocks (e.g. a 3×3 block corresponds to a 9×9 puzzle, as presented in Figure 1). Each puzzle has some cells that have already been filled in (called clues). We have to fill in the rest of the cells with numbers in the range $[1.. N]$ such that in all rows, all columns and all blocks, every number appears exactly once. The puzzle at the right of Figure 1 is a solution to the puzzle on the left.

4	5							
		2		7		6	3	
							2	8
			9	5				
	8	6				2		
	2		6			7	5	
						4	7	6
	7			4	5			
		8			9			

4	5	3	8	2	6	1	9	7
8	9	2	5	7	1	6	3	4
1	6	7	4	9	3	5	2	8
7	1	4	9	5	2	8	6	3
5	8	6	1	3	7	2	4	9
3	2	9	6	8	4	7	5	1
9	3	5	2	1	8	4	7	6
6	7	1	3	4	5	9	8	2
2	4	8	7	6	9	3	1	5

Figure 1: Left: A Sudoku puzzle. Right: its solution

As a lazy scientist who hates hard work but also can't stand unsolved puzzles, I decided to write a Sudoku solver program based on searching over all legal assignments for a given puzzle. Unfortunately, Sudoku is an NP-complete problem [1], when generalized to $N \times N$ grids, in the sense that all known solvable algorithms do so inefficiently; in spite of the fact that if a candidate solution is given, it takes only polynomial time to check its

correctness. Therefore, a naive search approach is unlikely to be efficient so we need some domain knowledge (called heuristics in AI terminology [5]) to direct our problem solver.

There are many approaches for solving Sudoku puzzles, including CSP-based solutions [6], genetic algorithms [7], and modeling Sudoku as an exact cover problem while using the dancing links techniques [8]. Peter Norvig [2], the famous AI pioneer, published an elegant Python code for a Sudoku solver based on integrating CSP with search. In this study I follow Norvig approach by integrating common Sudoku strategies, popular among Sudoku fans, into the search process.

Heuristic search is one of my favorite AI techniques. Realizing the difficulty in searching over huge spaces, it is inspired from the ways humans solve the problem and how human strategies can be applied to improve our solver performance. This is especially useful for (the few) tasks for which humans are still superior to machines. In the following I will describe the HS-solver that is based on backtracking search, integrated with such human strategies. I'll begin with the search algorithm, and then describe the heuristics. I'll also report the results of some experiments I conducted on a popular benchmark of 95 9×9 puzzles which had already been used in Norvig study [2].

2. Backtracking

We represent a Sudoku puzzle by an NxN matrix holding all values that have already been set on the grid. At the beginning, only clue cells are set with their values, while all other cells are set to zero. In addition, we manage an auxiliary NxN matrix of BitSet objects. BitSet is a Java class that can be used to efficiently maintain a set of values, including adding, removing, and querying the membership of a specific value. Any constraint entry (i,j) represents all candidates that can be legally assigned to cell (i,j). In addition we keep for each row/column/block (a unit for short) a BitSet that maintains all values that have already been set in this unit. Finally, we also maintain a list of non-assigned cells in the puzzle.

The following Java code includes some important methods of the Sudoku class. The full Java code of the Sudoku HS-solver is available on <https://github.com/davidcarmel/SudokuSolver>.

```

public static enum UnitType {Row, Column, Block}
public int B = 3; // default block dimension
public int N = B * B; // default puzzle dimension
private int[][] puzzle = null; // puzzle table - at the end should be fully assigned
private BitSet[][] constraints = null; // validating the legality of a new assignment
private List<Pair<Integer, Integer>> nonAssignedCells = null; // non-assigned cells

private int getBlockIndex(int i, int j) {
    // cr & cc are the coordinates of the block's top left cell
    int cr = ((i - 1) / B) * B + 1;
    int cc = ((j - 1) / B) * B + 1;
    return (cr - 1) / B * B + cc / B + 1; // (cr,cc) --> [1..N]
}

public List<Integer> getCellCandidates(int i, int j) {
    List<Integer> vals = new ArrayList<>();
    this.constraints[i][j].stream().forEach(vals::add);
    return vals;
}

public boolean isLegalAssignment(int i, int j, int val) {
    // return true iff val is not assigned already to one of (i,j) units
    return (!constraints[i][0].get(val) && !constraints[0][j].get(val)
        && !constraints[N + 1][getBlockIndex(i, j)].get(val));
}

public boolean isPuzzleSolved() {
    for (int i = 1; i <= N; i++) {
        if ((this.constraints[i][0].nextClearBit(1) < N + 1)
            || (this.constraints[0][i].nextClearBit(1) < N + 1)
            || (this.constraints[N + 1][i].nextClearBit(1) < N + 1))
            return false;
    }
    return true;
}

```

Figure 2: Data structure and some important methods of the Sudoku class

Our first trial for solving the puzzle is by a naive backtracking algorithm. Backtracking is a classical *depth-first search*. It first examines if the puzzle already been solved. If not, it collects all non assigned cells and sort them by their cardinality (preferring cells with fewer candidates). It then traverses over the sorted list, inspecting the cell's candidates. If a value is legal in this cell, it sets it and the method applies itself recursively on the modified puzzle. If all values fail we back up to a higher level seeking for an alternative track. As already been mentioned, while this algorithm is guaranteed to find a solution, if exists, this process is inefficient and may take forever to run. Here is the code for the backtracking algorithm.

```

static Sudoku solve(Sudoku sudoku) {
    if (sudoku.isPuzzleSolved())
        return sudoku;

    // collect all non-assigned cells in the puzzle
    List<Triple<Integer, Integer, Integer>> nonAssignedCells =
        sudoku.getNonAssignedCellsWithCardinality();
    if (nonAssignedCells == null) // no solution
        return null;
    if (nonAssignedCells.size() == 0) // no cells to assign - puzzle solved
        return sudoku;

    // sort the non-assigned cell according to their cardinality in increasing order
    nonAssignedCells.sort(new CellComparator());

    for (Triple<Integer, Integer, Integer> triple : nonAssignedCells) {
        List<Integer> vals = sudoku.getCellCandidates(triple.getLeft(), triple.getMiddle());
        for (Integer val : vals) {
            if (sudoku.isLegalAssignment(triple.getLeft(), triple.getMiddle(), val)) {
                Sudoku newSudoku = sudoku.clone();
                newSudoku.setCellValue(triple.getLeft(), triple.getMiddle(), val);
                Sudoku solved = solve(newSudoku);
                if (solved != null && solved.isPuzzleSolved())
                    return solved;
            }
        }
        return null; //this pass reaches dead-end - return to the upper level
    }
    return null;
}

```

Figure 3: Solving Sudoku by the backtracking algorithm

We ran this function over the 95 puzzles, when limiting the search to 100M calls. 93 out of the 95 puzzles can be solved, with an average of 1.7M calls per puzzle, in 18.2 seconds per puzzle on my laptop. Well, this takes too much time and certainly is not satisfying.

Strategy	#Solved puzzles	#calls/puzzle	max #calls	avg time/puzzle (sec)
backtracking (100M)	93/95	1,772,609.25	22,849,956	18.17

Table 1: Search results of Backtracking algorithm when limited to 100M calls.

3. Heuristics

Can we do better? hopefully — this is what we would like to explore. Our hypothesis is that having some domain knowledge on the problem space can help our solver to navigate the vast domain and save us a lot of precious time. We will apply some common Sudoku strategies [3] in order to improve our backtracking algorithm. It turns out that all strategies are closely related to CSP, focusing on reducing the number of legal candidates per cell (the constraints). The goal is that any cell will have only one legal candidate and hence the puzzle is solved. The difference is that while in CSP, constraints are given and we look for a solution that obey them, in our case our strategy is to reduce the number of constraints by eliminating inconsistent ones. The less constraints we have on legal assignments, the smaller the search domain to explore.

Strategy I: Candidate Reduction

If a value is being set to a cell, then it can't be set to any other cells of its units. Therefore, we can safely remove

any cell candidate if it is already being set in one of its units. Furthermore, when only one candidate is left for a cell, it can be safely set thus entailing additional candidate reduction.

Calling this strategy by the HS-solver, prior to running the search process, enables solving all puzzles, including P5, and P41, the two puzzles for which backtracking failed. The results reveals significant reduction in the search efforts. On average, search is reduced to 23.8K calls per puzzle (98.6% improvement with respect to naive search) while the maximum calls over all puzzles is reduced from 22.M to 377K. In particular, the problematic puzzles P5 and P41 are now being solved in a reasonable number of calls (and run-time) by invoking this strategy.

Strategy II: Uniqueness in Unit:

The 2nd strategy also applies a very simple rule: If a candidate is legal in **only one** of the unit cells, then it can be safely nailed to this cell. Of course, any new cell setting entails further reduction in candidate number for the other cells of its units.

We apply this strategy, together with candidate reduction, as part of the backtracking search. The average number of calls per puzzle is reduced to 100 (98% improvement) and the maximum calls over all puzzles to 1403, in 25ms per puzzle. Solving P5 and P41 is a piece of cake. It seems that we have cracked this benchmark.

Strategy III: Hidden pairs

A pair of candidates is called hidden if it occurs in exactly two unit cells, and none of its members is a candidate in the other unit cells. Then, it follows that the two unit's cells holding this pair must be assigned with one of the pair members, therefore all other candidates for these two cells can be dropped. Note that we do not know which pair member should be set in which cell, however any candidate reduction is blessed.

Well, the average number of calls is reduced to 40.8, and the maximum number of calls over all puzzles is only 331. The number of calls continues to drop for the P5 and P4 tricky puzzles.

Strategy IV: Naked pairs:

A pair is called naked if it is lonely in a cell. If a pair is naked in two unit cells, then it can be dropped from all the other unit cells. following the same logic of hidden pairs.

Strategy	#Solved puzzles	avg # calls	max #calls	avg time/puzzle (sec)	P5 (#calls)	P41 (#calls)	Inkala-2006 (#calls)	Inkala-2010 (#calls)
backtracking (<100M)	93	1,772,609.25	22,849,956	18.17	---	---	335,638	10,008
+Candidate Reduction	95	23,827.84	377,252	0.72	192,900	150,150	6294	184
+ uniqueness	95	100.41	1403	0.025	451	388	15	13
+Hidden pairs	95	40.768	331	0.029	194	48	5	13
+ naked pairs	95	26.19	252	0.029	132	31	3	13

Table 2: Search performance with the heuristic strategies.

Table 2 presents the results of backtracking, and after adding each of the strategies. The average number of calls is reduced to 26.2 and the maximum calls over all puzzles to 252. The problematic tricky puzzles are also solved smoothly. The average run-time per puzzle is 29 ms. The two right columns relate to difficult puzzles described in the following. We can conclude that our code is robust to handle quite complicated 9×9 puzzles.

4. Man vs Machine

What about really hard puzzles? Norvig [2], mentioned a Finnish mathematician, Arto Inkala, who proposed the ([Inkala-2006 puzzle](#)) as “the most difficult Sudoku-puzzle known so far” and the [Inkala-2010](#) puzzle as “the most difficult puzzle I’ve ever created.” Looking at the right columns in Table 2, we can see that these two puzzles can be solved with a few calls, in a few milliseconds . Norvig also provided a benchmark of 11

“difficult” puzzles (eleven puzzles). None of them really challenges the HS-solver (10.4 calls on average, 25ms). This is not surprising, problems seem difficult to humans are not necessarily difficult to machines and vice versa. Norvig also generated artificial puzzles by shuffling existing ones, looking for a “killer puzzle” that will challenge his own solver. The puzzle shown in Figure 2 is such a killer. This is not an “official” Sudoku puzzle (you will not find it in a Newspaper) because it has multiple solutions (Norvig reported 13 different solutions). Interestingly, it only takes 19 calls, in 195ms, for the HS-solver to converge to the solution shown on the right.

.	6	.	.	.
.	5	9	8
2	8	.	.	.
.	4	5
.	.	3
.	.	6	.	.	3	.	5	4
.	.	.	3	2	5	.	.	6
.
.

1	3	8	2	4	6	5	7	9
6	5	9	1	3	7	2	4	8
2	7	4	5	9	8	1	6	3
7	4	5	6	8	2	3	9	1
8	1	3	4	5	9	6	2	7
9	2	6	7	1	3	8	5	4
4	8	7	3	2	5	9	1	6
3	6	2	9	7	1	4	8	5
5	9	1	8	6	4	7	3	2

Figure 4: A killer puzzle generated by Norvig [2], on the left, and a solution found by the HS-solver on the right.

5. Conclusions

It turns out that Sudoku, considered by many as a great challenge for human intellect, (and in particular, a great challenge for myself), does not provide a real challenge to naive search program plus some basic strategies. Are there puzzles out there that our HS-solver will not be able to solve in a reasonable time? It is very likely that the answer is yes; it would be great if we can find such examples. One direction is to move to larger puzzles of 16×16 or even 25×25 , unfortunately, I could not find any reliable benchmark for such puzzles. If you are familiar with, please let me know. There are many more complicated Sudoku strategies in the arsenal such as X-wing, Y-wing, Sword-fish and more [3]. At present, I was unable to find any evidence that they are really needed, but hopefully they will be helpful for really difficult puzzles to be found in the future.

References

- [1] Ercsey-Ravasz, Mária, and Zoltán Toroczka. “The chaos within Sudoku.” *Scientific reports* 2.1 (2012): 1–8.
- [2] Peter Norvig. Solving Every Sudoku Puzzle.
- [3] Tip on Solving Sudoku puzzles. <https://www.kristanix.com/sudoku/sudoku-solving-techniques.php>
- [4] Mathematics_of_Sudoku. https://en.wikipedia.org/wiki/Mathematics_of_Sudoku
- [5] Heuristics: Intelligent Search Strategies for Computer Problem Solving. Judea Pearl
- [6] Simonis, Helmut. Sudoku as a Constraint Problem. Eleventh International Conference on Principles and Practice of Constraint Programming. (2005)
- [7] Mantere, Timo, and Janne Koljonen. “Solving, rating and generating Sudoku puzzles with GA.” *IEEE congress on evolutionary computation*. 2007.
- [8] Donald Knuth. Dancing links, arXiv (2000).

Sudoku Solver

Heuristic Search

[Follow](#)

Written by David Carmel

21 Followers · 29 Following

Principal Applied Scientist, Amazon, Israel

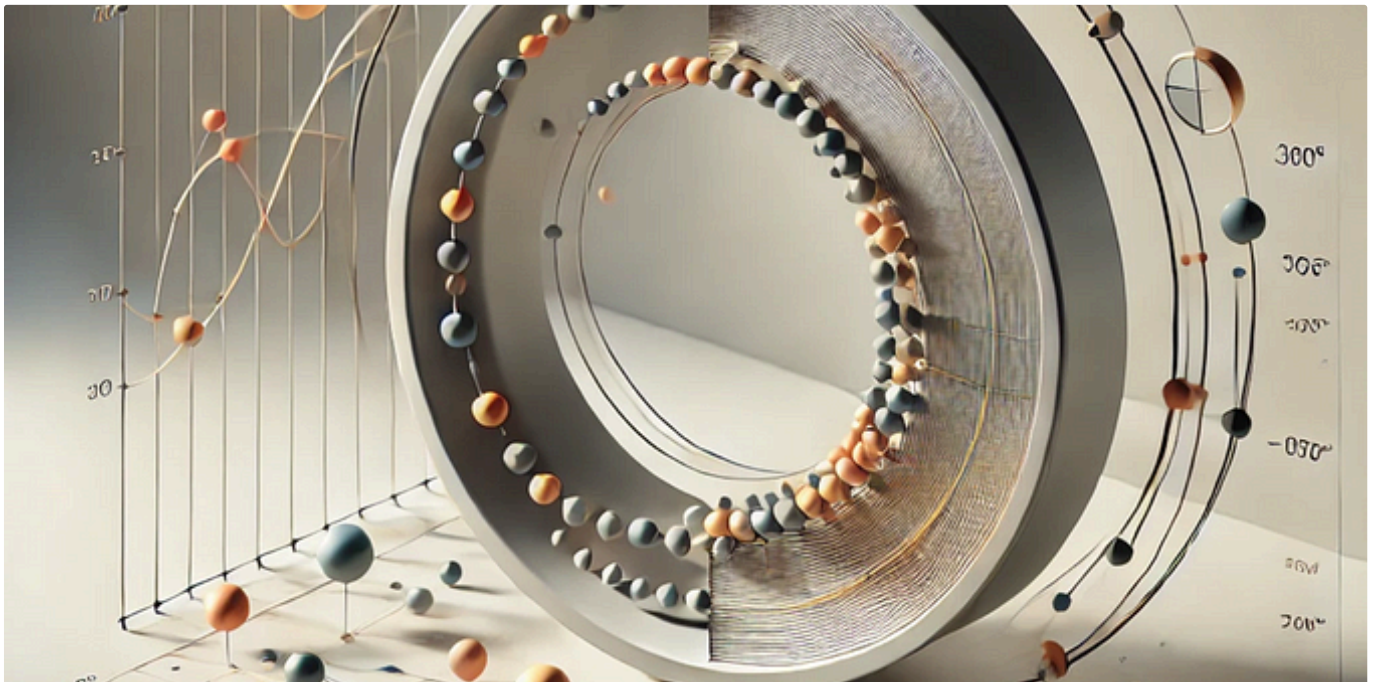
No responses yet



What are your thoughts?

[Respond](#)

Recommended from Medium





Martin Timms

Extending Scikit-Learn to Use Modulo Distance in K-Nearest Neighbors

A Guide for Engineers

★ Oct 29 🖱 1

 Edward Scrivner 

Dijkstra's Algorithm

Understanding the Shortest Path Algorithm

★ Oct 16 🖱 52 💬 1

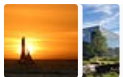


Lists



Staff picks

786 stories · 1498 saves



Stories to Help You Level-Up at Work

19 stories · 892 saves



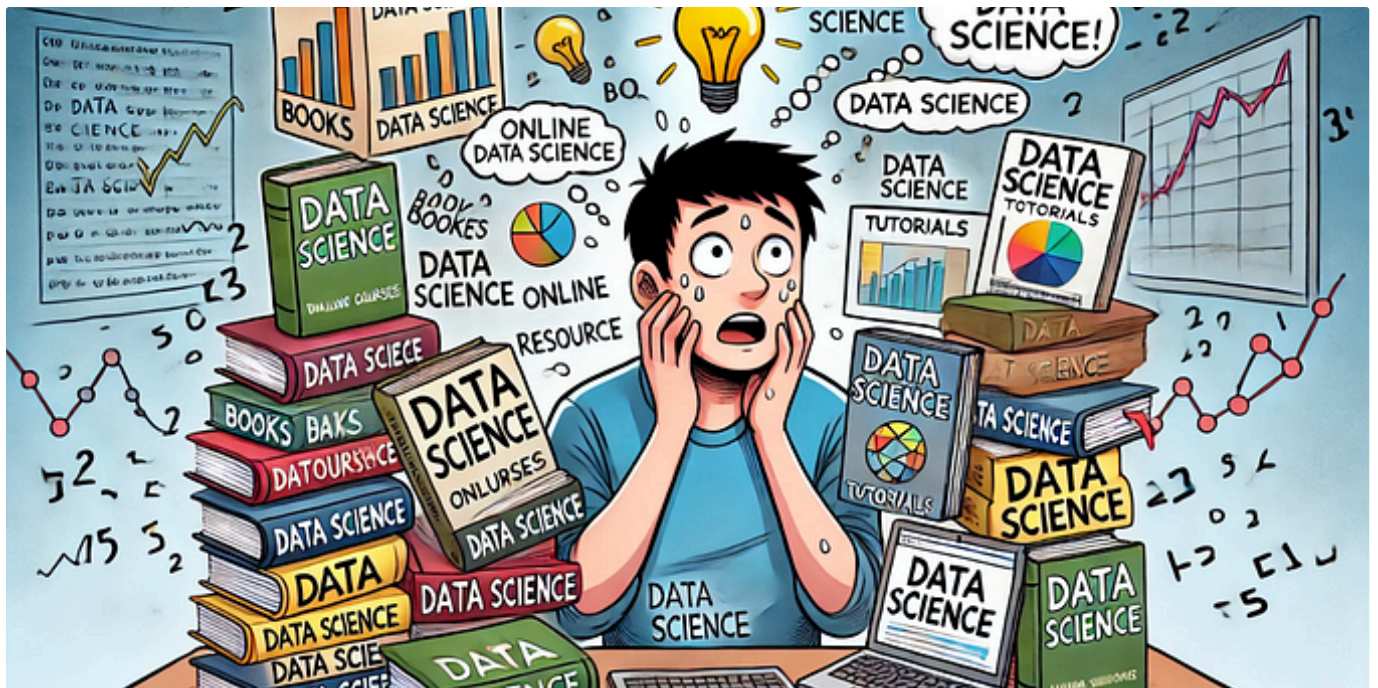
Self-Improvement 101

20 stories · 3129 saves



Productivity 101

20 stories · 2646 saves

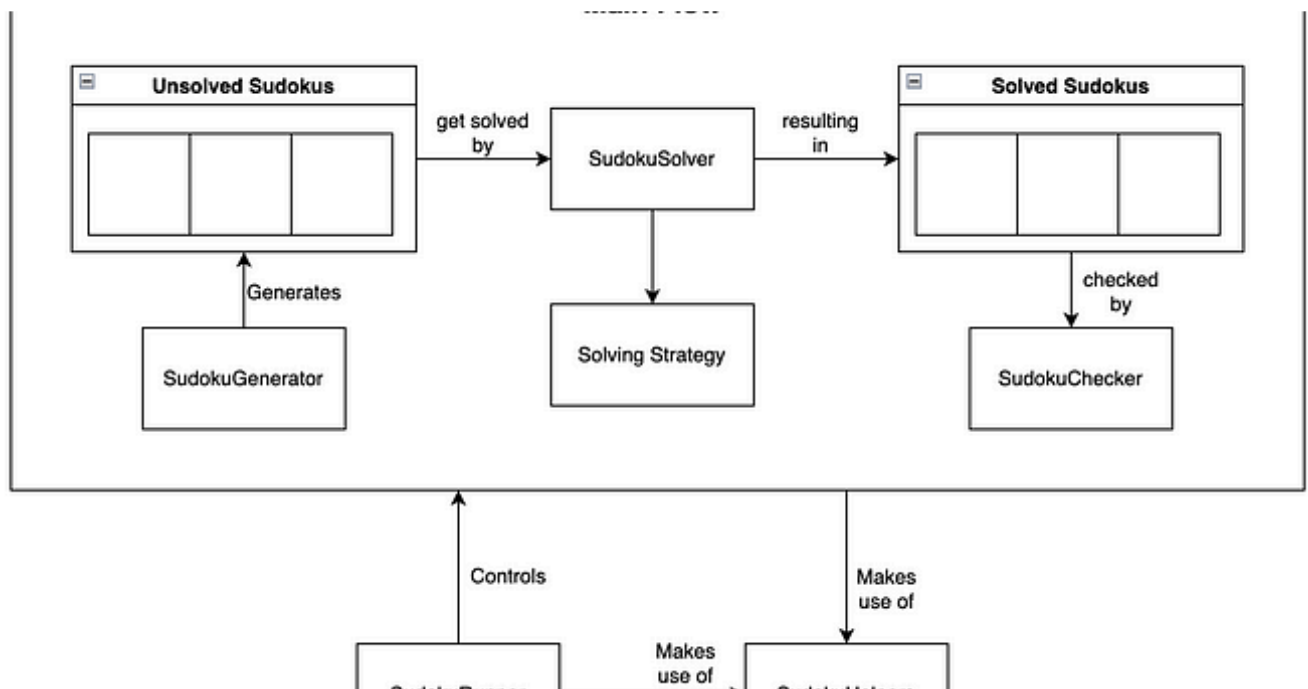


Amit Yadav

K-Means Clustering Pseudocode and Implementation

Hey, is this you?

Jul 18



spooky man

Quick Sudoku Solver | Pt.1

Laying the groundwork

Jun 14 6





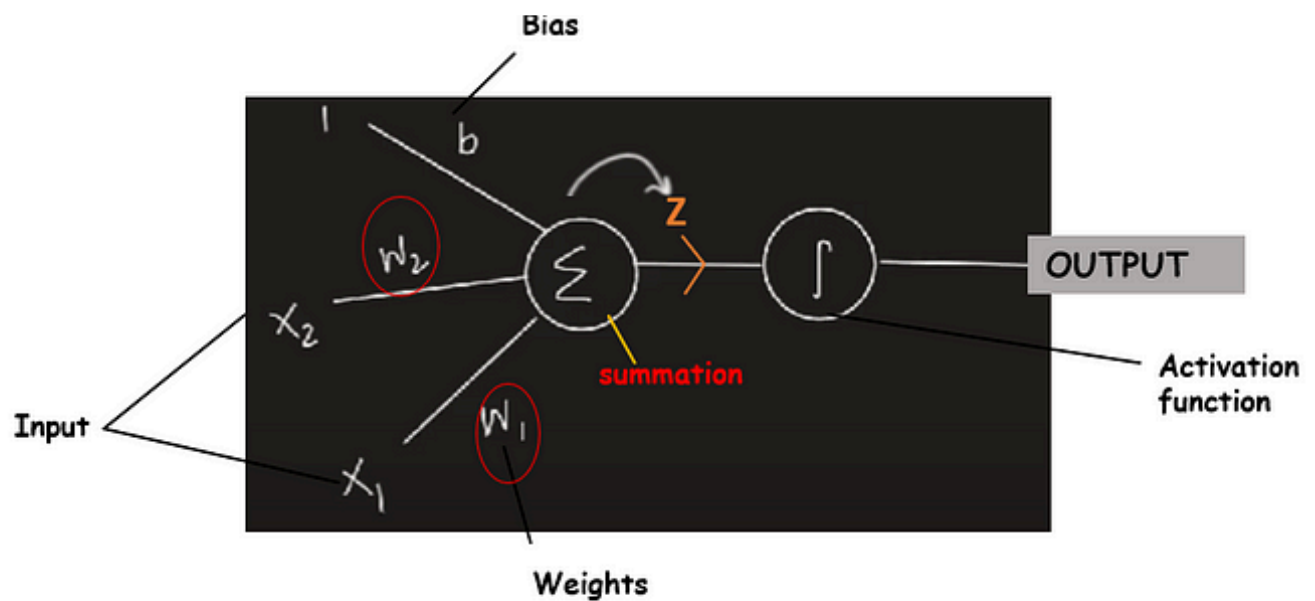
Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

★ Oct 26 🖱 7.4K 💬 111

🔖 ⋮



$$Z = W_1X_1 + W_2X_2 + b$$

Akanksha Verma, MSc Data Science

Perceptron

A perceptron is a simple type of Artificial Neural Network (ANN) algorithm developed by Frank Rosenblatt in 1957. It is the basic unit of a...

Oct 4 🖱 52

🔖 ⋮

[See more recommendations](#)