Target 1: Basic Game UI and Manual Input – Brief Description

I built a 9x9 Sudoku board in React by dividing it into nine 3x3 subgrids, with each cell as an editable component. Users can select a cell and input numbers from 1 to 9 via a number pad. The state management allows tracking the selected cell and updating its value, providing a clean and interactive UI ready for further game logic.

Target 2: Game Logic and Validation

Approach:

- Implemented a validateSudoku function that checks for duplicates in rows, columns, and 3x3 subgrids.
- Returned an array of conflicting cell positions to visually highlight invalid entries in the UI.
- Integrated validation logic in the SudokuGrid component to highlight conflicts dynamically.
- Added a "Check Solution" button that triggers full-board validation and displays any rule violations to the user.

Target 3: Sudoku Puzzle Generator

Approach:

- Used a backtracking algorithm to generate a fully solved Sudoku grid.
- Randomly removed numbers from the solved grid to create puzzles with varying clues.
- Controlled the difficulty by adjusting how many cells are removed (easy = fewer cells removed, hard = more).
- Ensured that each generated puzzle has a unique solution by verifying with a solution counter.
- Provided a difficulty selection feature via buttons or dropdown before generating a puzzle.

Target 4: Manual Solver

Approach:

- Allowed users to manually input numbers into a fully editable grid (isEditable: true).
- Maintained the grid state with useState and updated values via number pad input.
- On clicking "Solve," passed the current board to a backtracking solver function.
- Updated the grid with the solution if solvable, and displayed a message if unsolvable.

- (Bonus idea: planning to implement a “Hint” feature by solving the grid in memory and revealing one correct cell.)

Stretch Goal: Image-Based Solver

Approach:

- Started implementing image-based Sudoku solving by integrating image preprocessing with OpenCV to detect and warp the Sudoku grid.
- Experimented with both **Tesseract OCR** and a **TensorFlow.js digit recognition model** to extract digits from each cell.
- Began developing a backend to serve a trained TensorFlow model for more accurate digit prediction.
- Designed the image upload and extraction pipeline in the frontend, connecting it to the manual solver grid.
- Currently, the feature is incomplete; clicking the "solve" button for image-based input shows an “under development” message.
- The foundation is laid for completing OCR, digit classification, and grid population in future iterations.