2373. Largest Local Values in a Matrix

Easy Array Matrix

Leetcode Link

Problem Description

The given problem involves an `n x n` integer matrix named `grid`. The objective is to generate a new integer matrix called `maxLocal` with the dimensions `(n - 2) x (n - 2)`. For each cell in `maxLocal`, its value should be the largest number found in the corresponding `3 x 3` submatrix of `grid`. To clarify, the submatrix corresponds to a sliding window centered around the cell `(i + 1, j + 1)` in the original grid, with `i` and `j` denoting row and column indices, respectively, in `maxLocal`. Essentially, it can be visualized as overlaying a `3 x 3` grid on top of the main grid, moving it one cell at a time, and at each position, recording the maximum value found in this overlay into the new matrix.

Intuition

To solve this challenge, the intuitive approach is to perform a nested loop traversal across the main grid that captures every possible `3 x 3` submatrix. We start this process one row and one column in from the top-left of the grid (since the edge rows and columns don't have enough neighbors for a full `3 x 3` submatrix) and end one row and one column before the bottom-right. At each iteration of the loop, we find the largest value from the current `3 x 3` submatrix and store it in the corresponding cell in the `maxLocal` matrix.

More specifically, the iteration will have two loops: one for traversing rows `i` and another for columns `j`. For each position `(i, j)`, we look at rows `i` to `i + 2` and columns `j` to `j + 2`, creating a `3 x 3` region. Making use of a list comprehension and the `max()` function, we find the largest value in that region and save it as the value of `maxLocal[i][j]`. This two-step process - finding the `3 x 3` submatrix and then the maximum value within it - is repeated until the entire `maxLocal` matrix is filled.

Solution Approach

The implementation utilizes a straightforward brute-force algorithm to address the problem. This strategy leverages the capability of nested loops and list comprehension for easy iteration through the matrix.

Here's a step-by-step walkthrough of the algorithm, using Python as the reference language:

1. Determine the size `n` of the input `grid`.
2. Initialize the `maxLocal` matrix filled with zeros, with the size `(n - 2) x (n - 2)`, which will eventually store the largest values.
3. Start with two nested loops, where `i` iterates through rows from `0` to `n - 2`, and `j` iterates through columns from `0` to `n - 2`. These ranges are chosen to ensure we can always extract a `3 x 3` submatrix centered around `grid[i + 1][j + 1]`.
4. For each position `(i, j)`, extract the contiguous `3 x 3` submatrix. This is done by another nested loop, or in this case, a list comprehension, that iterates through all possible `x` and `y` coordinates in this submatrix, with `x` ranging from `i` to `i + 2` and `y` ranging from `j` to `j + 2`.
5. Calculate the maximum value within this `3 x 3` submatrix using the `max()` function applied to the list comprehension, which iterates over the range of `x` and `y` and accesses the values in `grid[x][y]`.
6. Assign this maximum value to the corresponding cell in `maxLocal[i][j]`.

By using a list comprehension within the nested loops to calculate the maximum, the implementation avoids the need for an explicit inner loop for exploring the `3 x 3` submatrix. This makes the code more concise and readable.

Additionally, the approach does not require any extra data structures aside from the `maxLocal` matrix which is being filled in, indicating an in-place algorithm with no additional space complexity.

The final result will be the `maxLocal` matrix, which now contains the largest number from every contiguous `3 x 3` submatrix of the original `grid`.

```python
class Solution:
    def largestLocal(self, grid: List[List[int]]) -> List[List[int]]:
        n = len(grid)
        ans = [[0] * (n - 2) for _ in range(n - 2)]
        for i in range(n - 2):
            for j in range(n - 2):
                ans[i][j] = max(
                    grid[x][y] for x in range(i, i + 3) for y in range(j, j + 3)
                )
        return ans
```

The above code snippet corresponds to the entire algorithm discussed, simplified into a Python method definition within a Solution class.

Example Walkthrough

Let's assume we have the following `4 x 4` grid:

```
9 9 8 1
5 7 5 1
7 8 6 2
4 3 2 0
```

We want to apply the algorithm to create a `maxLocal` matrix with dimensions `(4 - 2) x (4 - 2)`, which is `2 x 2`.

Here are the steps of the algorithm in action:

1. Start with `i=0` and `j=0`. The `3 x 3` submatrix is:

   ```
   9 9 8
   5 7 5
   7 8 6
   ```

   The maximum value in this submatrix is `9`. So, `maxLocal[0][0]` will be `9`.

2. Move to `i=0` and `j=1`. The `3 x 3` submatrix is:

   ```
   9 8 1
   7 5 1
   8 6 2
   ```

   The maximum value in this submatrix is `9`. So, `maxLocal[0][1]` will be `9`.

3. Next, `i=1` and `j=0`. The `3 x 3` submatrix is:

   ```
   5 7 5
   7 8 6
   4 3 2
   ```

   The maximum value in this submatrix is `8`. So, `maxLocal[1][0]` will be `8`.

4. Finally, `i=1` and `j=1`. The `3 x 3` submatrix is:

   ```
   7 5 1
   8 6 2
   3 2 0
   ```

   The maximum value in this submatrix is `8`. So, `maxLocal[1][1]` will be `8`.

After filling all the cells in `maxLocal`, we get the final matrix:

```
9 9
8 8
```

This `2x2` matrix represents the maximum values from each `3 x 3` sliding window within the original `4 x 4` grid.

Python Solution

```python
from typing import List

class Solution:
    def largestLocal(self, grid: List[List[int]]) -> List[List[int]]:
        # Determining the size of the given grid.
        grid_size = len(grid)

        # Preparing the answer grid with the reduced size (n-2) since we are looking for
        # 3x3 local grids and each local grid reduces the dimension by 2 on each axis.
        largest_values_grid = [[0] * (grid_size - 2) for _ in range(grid_size - 2)]

        # Iterating over each cell that will be the top-left corner of a 3x3 grid.
        for row in range(grid_size - 2):
            for col in range(grid_size - 2):

                # Finding the largest value in the current 3x3 local grid.
                largest_value = max(
                    grid[i][j] for i in range(row, row + 3) for j in range(col, col + 3)
                )

                # Storing the largest value found in the corresponding cell of the answer grid.
                largest_values_grid[row][col] = largest_value

        # Returning the answer grid containing all the largest values found in each 3x3 local grid.
        return largest_values_grid

```

Java Solution

```java
class Solution {

    // Method to find the largest element in every 3x3 subgrid
    public int[][] largestLocal(int[][] grid) {
        // Determine the size of the grid
        int gridSize = grid.length;

        // Initialize the answer grid with a reduced size
        // because the border elements can't form a complete 3x3 subgrid
        int[][] maxLocalValues = new int[gridSize - 2][gridSize - 2];

        // Iterate through the grid, considering each 3x3 subgrid
        for (int i = 0; i <= gridSize - 3; ++i) {
            for (int j = 0; j <= gridSize - 3; ++j) {
                // Initialize max value for the current 3x3 subgrid
                int maxVal = Integer.MIN_VALUE;

                // Nested loops to go through each element in the 3x3 subgrid
                for (int x = i; x <= i + 2; ++x) {
                    for (int y = j; y <= j + 2; ++y) {
                        // Update maxVal if a larger value is found
                        maxVal = Math.max(maxVal, grid[x][y]);
                    }
                }

                // Assign the largest value in the 3x3 subgrid to the answer grid
                maxLocalValues[i][j] = maxVal;
            }
        }
        // Return the answer grid containing the largest values of each subgrid
        return maxLocalValues;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<vector<int>> largestLocal(vector<vector<int>>& grid) {
        int size = grid.size(); // Get the size of the input grid.
        // Initialize the result grid with dimensions (size - 2) x (size - 2) to accommodate the 3x3 window reduction.
        vector<vector<int>> result(size - 2, vector<int>(size - 2));
        // Loop through each cell where a 3x3 grid can start.
        for (int row = 0; row < size - 2; ++row) {
            for (int col = 0; col < size - 2; ++col) {
                // Iterate through each cell within the current 3x3 window.
                for (int localRow = row; localRow <= row + 2; ++localRow) {
                    for (int localCol = col; localCol <= col + 2; ++localCol) {
                        // Update the corresponding cell in the result grid with
                        // the maximum value seen so far in the 3x3 window.
                        result[row][col] = max(result[row][col], grid[localRow][localCol]);
                    }
                }
            }
        }
        // Return the populated result grid after processing the entire input grid.
        return result;
    }
};

```

Typescript Solution

```typescript
function largestLocal(grid: number[][]): number[][] {
    const gridSize = grid.length; // The size of the input grid
    // Initialize the result grid with dimensions (gridSize - 2) x (gridSize - 2)
    const resultGrid = Array.from({ length: gridSize - 2 }, () => new Array(gridSize - 2).fill(0));

    // Iterate through the grid to fill the resultGrid with the maximum values from each local 3x3 grid
    for (let row = 0; row < gridSize - 2; row++) {
        for (let col = 0; col < gridSize - 2; col++) {
            let localMax = 0; // The maximum value in the current 3x3 grid

            // Iterate over the 3x3 grid starting at (row, col)
            for (let k = row; k < row + 3; k++) {
                for (let z = col; z < col + 3; z++) {
                    // Find the maximum value in the local 3x3 grid
                    localMax = Math.max(localMax, grid[k][z]);
                }
            }

            // Assign the maximum value found to the corresponding position in resultGrid
            resultGrid[row][col] = localMax;
        }
    }

    // Return the result grid which contains the largest values found in each local 3x3 grid
    return resultGrid;
}

```

Time and Space Complexity

The provided code snippet is used to find the maximum local element in all 3×3 subgrids of a given 2D grid. Here is an analysis of its time and space complexity:

Time Complexity:

The time complexity of the code is determined by the nested loops and the operations within them. The outer two `for` loops iterate over `(n - 2) * (n - 2)` elements since we're examining 3×3 subgrids and thus can't include the last two columns and rows for starting points of our subgrids. For each element in the answer grid, we find the maximum value within the 3×3 subgrid, which involves iterating over 9 elements (3 rows by 3 columns).

Therefore, the time complexity is: `O((n-2)*(n-2)*9)`, which simplifies to `O(n^2)`, assuming that the max operation within a constant-sized subgrid takes constant time.

Space Complexity:

The space complexity is determined by the additional space used by the algorithm, not including the input. In this case, we're creating an output grid `ans` of size `(n - 2) * (n - 2)` to store the maxima of each 3×3 subgrid, which represents the additional space used.

Thus, the space complexity is: `O((n-2)*(n-2))` which simplifies to `O(n^2)` as the size of the `ans` grid grows quadratically with the input size `n`.