MATRX

A Matrix Manipulation Language

September 15th, 2018

Co-Manager: Katie Pfleger (kjp2157)
Co-Manager: Julia Sheth (jns2157)
Language Guru: Alana Anderson (afa2132)
System Architect: Pearce Kieser (pck2119)
Tester: Nicholas Sparks (ns3284)

Introduction

Machine learning (ML) is the area of computational science that focuses on analyzing and interpreting patterns and structures in data in order to enable learning, reasoning, and decision making. While the use of machine learning has been around since the turn of the century, it has only recently become mainstream in the industry. Today, 51% of enterprises across a variety of industries are deploying machine learning in production.[1] In fact, job titles such as "machine learning engineer," "deep learning engineer," and "data scientist" are already widely used terms. As these engineers will tell you, though, machine learning really boils down to one thing: *matrices*.

A matrix is a two-dimensional array of scalars with one or more columns and one or more rows. Matrix manipulations are often essential to machine learning algorithms, where they are used as the input data when training algorithms. However, implementing these operations in common programming languages (such as C, C++, or python) can be extremely complicated and time-consuming. While libraries and tools with more robust matrix manipulation tools exist, they are often expensive and syntactically complex. With this motivation, we have decided to build a simple language that supports matrix operations by design.

[1] Lorica, B., & Nathan, P. (2018). The State of Machine Learning Adoption in the Enterprise. O'Reilly Media, Inc.

Language Description

Inspired by our work both in the classroom and in the industry, MATRX is designed for the modern software engineer. The syntax of our language is simple and intuitive, allowing users to focus more on building and less on syntactical details. The high-level components of our language are designed to support the mathematical manipulations and computations of matrices.

1. **Matrix Manipulations**: A matrix in our language is simply defined as an array of arrays (*see Example 2 below*). Additionally, our language supports a variety of basic matrix manipulations such as determinant, dot product, transpose, scalar multiplication, addition, multiplication, and inverse. We also provide a built in function for printing matrices, which we call `printm`.
2. **Data Types**: In addition to our own `Matrix` type, our language supports basic primitives including `int`, `double`, `float`, `char`, and `string`.
3. **User-Defined Functions**: We allow users to define their own functions for working with primitives (*see Example 1 below*) and manipulating matrices.
4. **Control Flow**: Our language supports `if/else`, `for`, and `while` control flows, similar to the C language (*see Example 1 below*).
5. **Operators:** In the spirit of simplicity, our language uses the same standard arithmetic and logical operators as the C language (*see Example 1 below*).

Types of Programs

MATRX is an imperative programming language designed to allow software engineers to work with matrices in a simple and intuitive way. There are several important use cases that our language will aim to support. As computer science majors, we have all taken a course in linear algebra and longed for a simple way to check over our computations. Furthermore, we have seen such computations arise again and again in our work with machine learning algorithms. MATRX will allow users to perform such calculations quickly and easily.

Example 1: We demonstrate the basic syntax of our language, including a user-defined function to compute the greatest common divisor of two integers.

```
int gcd(int a, int b) {
      /* example of a simple user-defined function */
      while (a != b) {
            if (a > b)
                  a -= b;
            else
                  b -= a;
      }
      return a;
}
```

```
int main () {
      int a = 5;
      int b = 10;

      int d = gcd(a, b);
      printf("%d", d); // prints 5

      return 0;
}
```

Example 2: We show a simple program to exhibit the built-in declaration and manipulation of matrices in our language.

```
int main () {
      Matrix m1 = [[0, 1]
                   [2, 3]]; // matrix declaration
      print(m1);   // prints [0, 1]
                   //        [2, 3]

      Matrix m2 = [[4, 5]
                   [6, 7]]; // matrix declaration
      printm(m2); // prints [4, 5]
                   //        [6, 7]

      Matrix m3 = m1 * m2; // matrix multiplication
      printm(m3);  // prints [6, 7]
                   //        [26, 31]

      return 0;
}
```

The following is the C-equivalent of the above code:

```
typedef struct Matrix {
  int*     vals;
  size_t   rows;
  size_t   cols;
} Matrix;

Matrix new_matrix(size_t rows, size_t cols) {
  Matrix m;
  m.vals = malloc(sizeof(int) * rows * cols);
  m.rows = rows;
  m.cols = cols;
  return m;
}
```

```
void free_matrix(Matrix* m) {
  free(m->vals);
}

int* access_cell(Matrix *m, size_t row, size_t col) {
  return &m->vals[row*m->rows + col];
}

void print_matrix(Matrix *m) {
  for (int i = 0; i < m->rows; i++) {
    printf("[");
    for (int j = 0; j < m->cols; j++) {
      printf(" %d", *access_cell(m, i, j));
      if (j < m->cols - 1) {
        printf(",");
      }
    }
    printf(" ]\n");
  }
}

int dot_product(Matrix* m1, size_t m1_row, Matrix* m2, size_t m2_col) {
  int dp = 0;
  for (size_t m1_col = 0; m1_col < m1->cols; m1_col++) {
    dp += *access_cell(m1, m1_row, m1_col) * *access_cell(m2, m1_col,
m2_col);
  }
  return dp;
}

Matrix multiply_matrices(Matrix* m1, Matrix* m2) {
  if (m1->cols != m2->rows) {
    Matrix empty;
    empty.vals = NULL;
    empty.rows = 0;
    empty.cols = 0;
    return empty;
  }
  Matrix ret = new_matrix(m1->rows, m2->cols);

  for (size_t m1_row = 0; m1_row < m1->rows; m1_row++) {
    for (size_t m2_col = 0; m2_col < m2->cols; m2_col++) {
      *access_cell(&ret, m1_row, m2_col) = dot_product(m1, m1_row, m2,
m2_col);
    }
  }

  return ret;
}

int main() {
	Matrix m1 = new_matrix(2, 2);
	for (int i = 0; i < 2; i++) {
		for (int j = 0; j < 2; j++) {
			// in the language: m.vals[i][j] = i*j;
			int* cell = access_cell(&m1, i, j);
			*cell = (m1.rows*i)+j;
		}
	}

	Matrix m2 = new_matrix(2,2);
	for (int i = 0; i < 2; i++) {
		for (int j = 0; j < 2; j++) {
			// in the language: m.vals[i][j] = i*j;
			int* cell = access_cell(&m2, i, j);
			*cell = 4+(m2.rows*i)+j;
		}
	}

	print_matrix(&m1);
	print_matrix(&m2);
	Matrix m3 = multiply_matrices(&m1, &m2);
	print_matrix(&m3);

	return 0;
}
```