Renninger_self_assessment

January 23, 2020

1 Task

Matrix multiplication is a fundamental linear algebraic operation, working knowledge of which is a prerequisite to understanding modern neural networks. In this assignment, you will implement matrix multiplication in pure Python (`python_matmul`) and in Python using NumPy (`numpy_matmul`). Skeletal versions of the two functions are provided in the included Python source file.

This assignment is an opportunity for you to assess your preparedness for the course. It does not count towards your final grade.

[0]:
```
import numpy as np
import copy
from timeit import default_timer as timer
import statistics
```

[0]:
```
def python_matmul(X, Y):
    """
    Multiply 2-dimensional numpy arrays using pure Python.

    Parameters
    ----------
    X : list
        A list of M elements, each of which is an N-element list.
    Y : list
        A list of N elements, each of which is a K-element list.

    Returns
    ----------
    A list of M elements, each of which is a K-element list.
    """
    if not isinstance(X, list):
        raise ValueError('X must be list')
    if not isinstance(Y, list):
        raise ValueError('Y must be list')
    if not isinstance(X[0], list):
        raise ValueError('X must be 2-dimensional')
    if not isinstance(Y[0], list):
        raise ValueError('Y must be 2-dimensional')
    if not len(X[0]) == len(Y):
        raise ValueError('Column length of X must equal row length of Y')

    # need the dimensions of the input matrices
    M = len(X)
    N = len(Y)
    K = len(Y[0])

    # need this initialization so that you get unique row initializations
    outMatrix = [[0] * K for _ in range(0, M)]

    for i in range(0, M):
        for j in range(0, K):
            for k in range(0, N):
                outMatrix[i][j] += X[i][k] * Y[k][j]

    return outMatrix
```

[0]:
```
def numpy_matmul(X, Y):
    """
    Multiply 2-dimensional numpy arrays using PEP-0465 infix operator.

    Parameters
    ----------
    X : np.ndarray
        An MxN array.
    Y : np.ndarray
        An NxK array.

    Returns
    ----------
    An MxK array.
    """

    if not isinstance(X, np.ndarray):
        raise ValueError('X must be ndarray')
    if not isinstance(Y, np.ndarray):
        raise ValueError('Y must be ndarray')
    if not X.ndim == 2:
        raise ValueError('X must be 2-dimensional')
    if not Y.ndim == 2:
        raise ValueError('Y must be 2-dimensional')
    if X.shape[1] != Y.shape[0]:
        raise ValueError(
            'Columns of X ({:d}) must equal rows of Y ({:d}'.format(
                X.shape[1], Y.shape[0]))
    return X @ Y
```

Now, to make sure we have properly implented both functions (excluding input sanitization), we run `numTests` tests. Each test samples uniform distributions to randomly draw compatible matrix sizes (i.e. M, N, K) and the matrix elements for X and Y.

[16]:
```
numTests = 10000

alg_1_times = []
alg_2_times = []

for i in range(0, numTests):

    # uniformly sample compatible dimensions of the matrices to test
    M = int(np.random.uniform(low=2, high=50))
    N = int(np.random.uniform(low=2, high=50))
    K = int(np.random.uniform(low=2, high=50))

    # uniformly sample compatible matrices to test
    X = np.random.uniform(low=-10, high=10, size=(M, N))
    Y = np.random.uniform(low=-10, high=10, size=(N, K))

    # comparing the two implentations
    start = timer()
    mat_mult_alg_1 = np.array(python_matmul(np.ndarray.tolist(X),
                                            np.ndarray.tolist(Y)))
    finish = timer()
    alg_1_time = finish - start
    alg_1_times.append(alg_1_time)

    start = timer()
    mat_mult_alg_2 = numpy_matmul(X, Y)
    finish = timer()
    alg_2_time = finish - start
    alg_2_times.append(alg_2_time)

    np.testing.assert_array_almost_equal(mat_mult_alg_1,
                                         mat_mult_alg_2, decimal=12)

# if you reach here without any failed assertions, no tests have failed
print('The Two Matrix Multiplication Implementations are the Same in',
      numTests, 'tests')
print('Avg. time for base python matrix multiplication: ',
      statistics.mean(alg_1_times))
print('Avg. time for numpy python matrix multiplication: ',
      statistics.mean(alg_2_times))
```

```
The Two Matrix Multiplication Implementations are the Same in 10000 tests
Avg. time for base python matrix multiplication:  0.0021540813361000345
Avg. time for numpy python matrix multiplication:  1.1463073500863175e-05
```

The two implementations are quite different in terms of average multiplication speed. As expected, the `numpy` implementation is, on average, many orders of magnitude faster than my completely unoptimized matrix multiplication implementation for the chosen test set of matrices.

2 Survey

Briefly describe your (research) interest in deep learning/machine learning in the cell below.

I am a researcher in the aerospace department, where I focus on safe autonmy. My current research lies in the intersection of machine learning, formal methods, and control theory. In more detail, I work on ways to learn task specifications as probabilistic automata from human demonstrations of correct autonomous system beavior in order to formally synthesize correct controllers for these same autonomous systems. This is all done with the goal of improving the interpretability and correctness of the control of safety-critical autonomous systems.

As such, my interest in Deep Learning is primarily motivated by its use in Deep Inverse Reinforcement Learning, which is attacking the same problem I work on with the modern deep RL framework. A paper recently published in Science shows the use of deep RL in a framework that allows for guarantees on the learned behavior. This sort of work is of great interest to me and would make an excellent comparison to my own work.