SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITWATERSRAND

ELEN4020A - Data Intensive Computing

Lecturer: Prof E. J. Otoo

Laboratory Exercise 1: 3D Array Multiplication

25 February 2019

Tyson Cross 1239448 Michael Nortje 1389486 Josh Isserow 675720

Abstract

The operations of addition and multiplication with square and cubic array inputs (rank 2 and rank 3 tensors with equal dimensions and element lengths) are programmed using Python and NumPy array objects in this laboratory exercise. Error checking is implemented using simple assertion statements. The methods and motivations are discussed, and the generalised operation of tensor contraction for higher dimensions is examined with reference to the laboratory objectives.

1 INTRODUCTION

Tensors are a generalisation of dimensional arrays or vectors, with the rank of a tensor being a measure of their dimension. Tensors are a useful mathematical construct widely used in engineering and scientific applications, including machine learning applications with large data sets. For this laboratory, Python 3.6 was used for programming the functions, motivated by Python's elegant syntax and excellent overall readability. The NumPy library provides robust, performant support for manipulating *n*-dimensional data arrays. The NumPy library is a widely used scientific computing library providing a mutable array data structure used for rank 2 and rank 3 tensor representations in the laboratory exercises. NumPy array-specific functions such as `ndarray.shape` were used to return, check and compare the array dimensions and length. The laboratory results do not use any parallel code, and rely on sequential *for*-loops for computation. The implemented code is shown in Appendix I, Code 1.

2 RANK 2 TENSOR ADDITION

The function `rank2TensorAdd()` was implemented with the method shown in pseudocode in Algorithm 1. The function performs element-wise addition on two input rank 2 tensors $\mathbf{A}$ and $\mathbf{B}$, returning a new rank 2 tensor $\mathbf{C}$ as output. The function takes in three arguments: two 2d matrices (NumPy arrays) and a single integer value *N*, where *N* is the number of elements in a single row or column, i.e. the requirement is that A and B are both NxN matrices. A square 2d matrix is equivalent to a rank 2 tensor with both dimensions equal in the number of elements. The first step is initialising a new rank 2 tensor as a 2-dimensional NumPy array, with all elements set to zero. Two nested *for*-loops are then iterated through for the both dimensional indices (*i,j*) in $\mathbf{A}$ and $\mathbf{B}$, performing element-wise addition as shown in equation 1:

$$c_{ij} \mathrel{+}= a_{ij} + b_{ij} \tag{1}$$

This operation is identical to standard element-wise summing of two matrices.

2.a Error Checking

In the implemented 2d addition function, the assumption is made that both input tensors must be square and of rank 2. This requirement is checked by passing both input matrices to the function `checkRank2valid()` which uses the in-built Python `assertion` method to confirm separately

Input: tensor A (N x N), tensor B (N x N), N
Output: tensor C (N x N)
Initialize C to zero;
for *i = 1 to N* **do**
  for *j = 1 to N* **do**
    $C_{ij} \leftarrow A_{ij} + B_{ij}$
  end
end

Algorithm 1: Rank 2 Tensor addition

that both $\mathbf{A}$ and $\mathbf{B}$ are square and a third assertion checking that both input 2d NumPy arrays are the same shape. If any of the assertions fail (i.e. a required condition is not met) then the script execution is halted and an error string describing the failure is printed to the terminal output.

3 RANK 2 TENSOR MULTIPLICATION

The function `rank2TensorMult()` performs multiplication of two input 2D matrices A and B. The same validity check is performed using the `checkRank2valid()` function to ensure both input rank 2 tensors $\mathbf{A}$ and $\mathbf{B}$ are square and of the same shape. Each element of matrix C is initialised to zero. The matrix product $\mathbf{AB}$ is calculated using the summation of element-wise row and column multiplications, as shown in equation 2:

$$c_{ij} = \sum_{i=1}^{N} \sum_{j=1}^{N} a_{ik} b_{kj} \tag{2}$$

The implementation is shown as pseudocode in Algorithm 2. The function uses nested *for*-loops to perform the summations across equated indices of the row and columns of the input matrices, respectively. The innermost loop's individual row-column product is summed with the last iteration of the current element of C being calculated, with the result then stored as as the new value of the current element in C. This requires initialising the C matrix, with all elements set zeros before performing the iteration loops, to avoid accidentally adding the random contents from in the allocated memory location for the array.

3.a Error Checking

The 2d multiplication function also calls the `checkRank2Valid(A,B)` function to confirm that the input matrices are square and with with rows and columns of equal length.

Input: tensor A (N x N), tensor B (N x N), N
Output: tensor C (N x N)
Initialize C to zero;
for *i = 1 to N* **do**
  for *j = 1 to N* **do**
    for *k = 1 to N* **do**
      $C_{ij} \leftarrow C_{ij} + A_{ij} * B_{ij}$
    end
  end
end

Algorithm 2: Rank 2 Tensor Multiplication

4 Rank 3 Tensor Addition

Tensor addition requires that the inputs match in shape and size in order to be performed. The operation can be performed sequentially, but as each individual element-sum is independent of the adjacent elements within the input and output tensors, the operation could benefit from parallelisation. In the provided implementation the method is programmed using sequential, nested *for*-loops.

4.a rank3TensorAdd()

The function `rank3TensorAdd()` performs element-wise addition using the two input rank 3 tensors. The output matrix has the same shape and dimension (and total number of elements) as both of the inputs do. The method of iterating through the summation is shown in Algorithm 3.

Input: tensor A (N x N x N), tensor B (N x N x N), N
Output: tensor C (N x N x N)
Initialize C to zero;
for *i = 1 to N* **do**
  for *j = 1 to N* **do**
    for *k = 1 to N* **do**
      $C_{ijk} \leftarrow A_{ijk} + B_{ijk}$
    end
  end
end

Algorithm 3: Rank 3 Tensor Addition

4.b Error Checking

The 3d addition function also assumes that the input tensors are of the same length in all dimensions, and equal in dimensionality. It passes the two inputs to a simple error-checking function which checks these requirements with python assertion statements, which halt script execution and print a message to output in case of failure. This function is called `checkRank3Valid(A,B)`.

5 Rank 3 Tensor Multiplication

Tensor multiplication has two broad categories: the outer product, which is well defined, and the inner product, which is often an application-specific operation [1]. The required specification for the multiplication operation for the laboratory was to multiply two 3d matrices (rank 3 tensors) with the same number of elements, and produce a new 3d matrix with an equal number of dimensions to both inputs.

5.a Tensor Contraction

Tensor contraction involves equating two or more of the input tensor dimensional indices, referred to the "free variable(s)", and summing over the the remaining indices

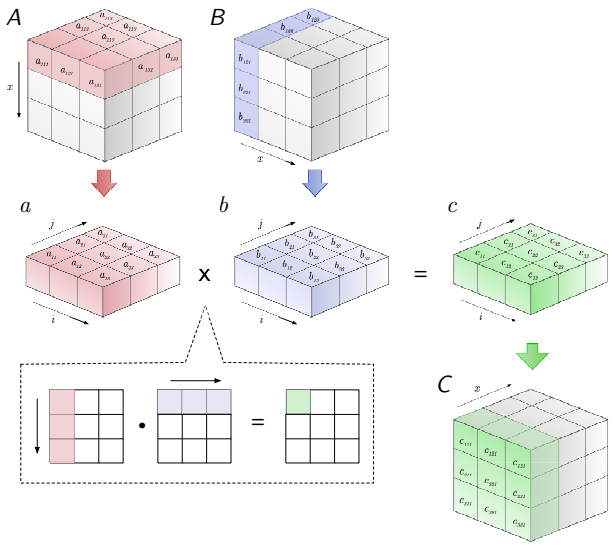

Fig. 1: Implemented rank 3 Tensor multiplication

(known as the "dummy variables") in element-wise row-column product assignment [2]. An example of single contraction between a rank 4 tensor $\mathbf{A}[\mathbf{N}][\mathbf{N}][\mathbf{N}][\mathbf{N}]$ and rank 2 tensor $\mathbf{B}[\mathbf{N}][\mathbf{N}]$, using Einstein summation notation (which drops the implied sigma symbol in the summation operation when there is a repeated index [3]), results in a rank 4 tensor $\mathbf{C}[N][N][N]$, as shown in equation 3:

$$c_{ijkm} = a_{ijkx} b_{xm} \tag{3}$$

One of the indices in the first tensor has been equated ("contracted") with an index in the second tensor. Any definition of contraction results in a reduction of dimensions, in this case with the resultant tensor having a rank of $(m + n - 2)$, where n and m are the dimensions of the original input matrices [4]. Repeated contraction would involve equating two or more of each tensors' indices making the formula for dimensional reduction $(n + m - 2r)$, where $r = 1$ for single contraction, $r = 2$ for double contraction, etc. However, this definition of an inner tensor product does not meet the requirements of the laboratory. Applying this operation to the inner product of two 3d matrices would result in a general formulation: $\mathbf{A}[N][N][N] \times \mathbf{B}[N][N][N] = \mathbf{C}[N][N][N][N]$ which possesses an additional dimension. The implementation of the laboratory function for multiplying two square 3d arrays is constructed differently.

5.b rank3TensorMult()

The implementation of the function `rank3TensorMult()` is shown in Algorithm 4. A graphical explanation of the operation is shown in Figure 1. The function takes two cubic matrices as inputs, along with *N*, where *N* is the length of all three dimensions in both input matrices. The function first initialises $\mathbf{C}$ to zero for all elements. Then a single dimension in both input tensors is equated (contracted), and a "slice" of the 2d matrix at this index in the contracted 3rd dimension is extracted from both inputs. This results in two 2d matrices, which are then multiplied by the standard 2d matrix method (performing contraction again, in the form of element-wise

dot-product for each element in the product matrix). The resultant matrix is also square and 2 dimensional, and this matrix is then assigned to a "slice" of $\mathbf{C}$, at the index in the 3rd dimension corresponding to the original contraction index. This operation "stacks" the 2d matrices (the products of the extracted input slices) resulting in a cubic tensor which has the same dimensions and total number of elements as both of the inputs rank 3 tensors.

Input: tensor A (N x N x N), tensor B (N x N x N), N
Output: tensor C (N x N x N)
Initialize C to zero;
for *x = 1 to N* **do**
  $C_{ijx} \leftarrow C_{ijx} + \left[A_{ik}B_{kj}\right]_x$
end

Algorithm 4: Rank 3 Tensor Multiplication

5.c Error Checking

The 3d multiplication function makes the same assumptions of the input matrix length and dimensionality, and calls the same error-checking function as used in the 3d addition, `checkRank3Valid(A,B)`.

5.d Multiplication with Higher Rank Tensors

The method of multiplication shown in this exercise could be generalised to higher dimensional tensors, of rank *n*. One way to accomplish this could be by using recursion, where the assignment step changes from "stacking" the matrix product to a recursive loop, assigning tensors of rank *n-1* along the contracted index, where *n* is the tensor rank (dimension of the array). However recursion is not an optimal method for computationally intensive operations, and it is not possible to easily produce parallel code with recursive functions. Use of parallel computation is vital technique the large data sets used in the relevant research areas of machine and deep learning.

6 Conclusion

Functions to perform mathematical operations with rank 2 and rank 3 tensors, of equal lengths in their respective dimensions (square or cubic) were implemented in Python, using the NumPy library for an array data structure. The steps to perform addition, and multiplication, using sequential computational processes were demonstrated in pseudocode, and the specific implementation of cubic 3d array multiplication was compared to the general definition of tensor contraction. Basic error checking was implemented for all functions. A generalised approach for implementing *n*-dimensional array multiplication was discussed.

REFERENCES

[1] A. Sringeri Vageeswara, "Tensor contraction optimizations," Ph.D. dissertation, The Ohio State University, 2015.

[2] W. Hackbusch, *Tensor spaces and numerical tensor calculus*. Springer Science & Business Media, 2012, vol. 42.

[3] R. Hunt, "University of Cambridge, Lecture Notes: Cartesian Tensors," 2015, uRL: http://www.damtp.cam.ac.uk/user/reh10/lectures/nst-mmii-chapter3.pdf. Last visited on 2019/02/23.

[4] T. Sochi, "Introduction to tensor calculus," *arXiv preprint arXiv:1603.01660*, 2016.

```
# Tyson Cross       1239448
# Michael Nortje    1389486
# Josh Isserow      675720

import numpy as np

def checkRank2Valid(A,B):
    assert np.shape(A)[0] == np.shape(A)[1], "A not square"
    assert np.shape(B)[0] == np.shape(B)[1], "B not square"
    assert np.shape(A) == np.shape(B), "Tensors not the same shape"

def checkRank3Valid(A,B):
    assert (np.shape(A)[0] == np.shape(A)[1] and np.shape(A)[1] == np.shape(A)[2]), "A not a cube"
    assert (np.shape(B)[0] == np.shape(B)[1] and np.shape(B)[1] == np.shape(B)[2]), "B not a cube"
    assert np.shape(A) == np.shape(B), "Tensors not the same rank"

def rank2TensorAdd(A, B, N):
    # Element-wise addition
    # Assumption is that both input tensors must be square, and of same rank
    checkRank2Valid(A,B)
    C = np.zeros((N,N), dtype=int)
    for i in range(N):
        for j in range(N):
            C[i][j] = A[i][j] + B[i][j]
    return C

def rank2TensorMult(A, B, N):
    # Analogous to matrix multiplication
    # Assumption is that both input tensors must be square, and of same rank
    checkRank2Valid(A,B)
    C = np.zeros((N,N), dtype=int)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i][j] += A[i][k] * B[k][j]
    return C

def rank3TensorAdd(A, B, N):
    # Assumption is that both input tensors must be cubic, and of same rank
    checkRank3Valid(A,B)
    C = np.zeros((N,N,N), dtype=int)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i][j][k] = A[i][j][k] + B[i][j][k]
    return C

def rank3TensorMult(A, B, N):
    # Iterates (with the free index 'x') through a commonly-indexed "slice" of each input rank 3 tensor.
    # This produces two 2d matrices. Matrix multiplication is performed on these two matrices.
    # The resulting 2d matrix is assigned to the C[][][x] slice.
    # Assumption is that both input tensors must be square, and of same rank
    checkRank3Valid(A,B)
    C = np.zeros((N,N,N), dtype=int)
    for x in range(N):
        C[:,:,x] = rank2TensorMult(A[x,:,:], B[:,x,:], N)

    return C

def main():
    for N in [10,20]:
        #-------------------
        #   Calculations
        #-------------------

        # Randomly generating tensors A and B (N x N) (Rank 2)
        a_2d = np.random.uniform(low=0, high=20, size=(N,N)).astype(int)
        b_2d = np.random.uniform(low=0, high=20, size=(N,N)).astype(int)

        # Randomly generating tensors A and B (N x N x N) (Rank 3)
        a_3d = np.random.uniform(low=0, high=20, size=(N,N,N)).astype(int)
        b_3d = np.random.uniform(low=0, high=20, size=(N,N,N)).astype(int)

        # 2D tensor operations
        c_2d_add = rank2TensorAdd(a_2d,b_2d,N)
        c_2d_mult = rank2TensorMult(a_2d,b_2d,N)

        # 3D tensor operations
        c_3d_add = rank3TensorAdd(a_3d,b_3d,N)
        c_3d_mult = rank3TensorMult(a_3d,b_3d,N)
```

```
        #---------------
        #    Output
        #---------------
        terminal_width = 120
        np.set_printoptions(threshold=np.inf,linewidth=terminal_width)
        title = " N = " + str(N) +": "
        print("\n",title.center(terminal_width,"*"))

        # Rank 2 Tensors output
        print("\n","Rank 2:\n","-"*7)

        print("\nA:\n\n",a_2d)
        print("\nB:\n\n",b_2d)
        print("\nC: (element-wise addition)\n\n",c_2d_add)
        print("\nC: (matrix multiplication)\n\n",c_2d_mult)

        # Rank 3 Tensors output
        print("\n","- "*int(terminal_width/2))
        print("\n","Rank 3:\n","-"*7)

        print("\nA:\n\n",a_3d)
        print("\nB:\n\n",b_3d)
        print("\nC: (element-wise addition)\n\n",c_3d_add)
        print("\nC: (tensor contraction)\n\n",c_3d_mult,"\n")


if __name__ == "__main__":
    main()
```

Code 1: lab1.py