Assignment 4: Converting math equations into code

In our lab, most of our research work comes from reading papers and converting the algorithms they discuss into code. While you
might think of math as just ways to analyze single value numbers, in
image processing, we expand a lot of math formulas from their original
single-value (scalar, as it's known) form to arrays (mathematically, referred
to as matrices) of numbers.

In fact, a whole branch of math is dedicated to manipulating matrices.
This branch of mathematics is known as linear algebra. This
assignment is designed to teach you the very basics of linear algebra
and single variable calculus through mathematical algorithms, then will
have you convert those algorithms into code.

This, in turn, simulates a bit like what being a research assistant
(not just a programmer) in this lab is like- it is very much about
being able to understand advanced math and computer science concepts,
and converting them into practically-implemented code.

Step 0: What are matrices?

Matrices are defined, mathematically, as rectangular arrays of numbers
we can perform mathematical operations on. Some examples of matrices
are:

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 3 & 2 \\ 2 & 0.3 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -0.1 \\ 2 & 5 & -9 \\ 3 & 3 & 3 \end{bmatrix}$$

You might recognize them, as basically being the math version of 2D built-in arrays of
`int` or `double` variables. Thus, whenever you get confused trying to visualize
a matrix, think of it simply as a built-in array.

We define the **height** of a matrix as the number of rows, and the
width of a matrix as the number of columns.

Step 1: The dot product

(Note: Some of you may have encountered dot products in your math class.
Please still read the definition of the dot product below because it
is a more CS-focused definition we will be using to implement this in code)

Definition of the Dot Product

Consider two matrices: X and w. X has a height of N and a width of M. w has a height of M and a width of P. Thus, we propose the following definitions with respect to the dot product:

1) We consider two matrices, X and w to have a valid dot product between them if $width(X) = height(w)$. Note that

with this definition, the dot product is not commutative.

2) We consider that the output matrix of the dot product will have height N and width P given the dimensions of X and w above.

3) We can derive the value at row i and column j in the output matrix as follows:

$$O_{i,j} = \sum_{k=0}^{M} X_{i,k} * w_{k,j}$$

Breaking Down the Definition

The entire dot product definition above may have seemed confusing. This is ok- even when we read papers in this lab, it takes a few times to understand what is going on. We can try to break down what is happening in the definition further.

Definition **1** defines the conditions for a valid dot product. We can only have a dot product if the width (number of columns) of the first matrix is the same as the height (number of rows) of the second matrix. Using this definition, we can come up with a few examples:

1)

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = valid$$

2)

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 0.1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -0.1 \\ 2 & 5 & -9 \\ 3 & 3 & 3 \end{bmatrix} = valid$$

3)

$$\begin{bmatrix} 4 & 3 & 2 \\ 2 & 0.3 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = invalid$$

4)

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} = invalid$$

So our code must be able to produce some sort of error if it encounters bad dimensions. We should keep this in mind when we move on to writing an implementation of the dot product.

Definition **2** Defines the output dimensions of a matrix. This will be useful when we treat the output matrix as a built-in matrix of `double[][]` - then we will know which values to fill in in the brackets.

Try to derive the dimensions of the output matrix for the two valid examples above. Then check your answer below:

1)
$\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ = 2 x 1 matrix

2)
$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 0.1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -0.1 \\ 2 & 5 & -9 \\ 3 & 3 & 3 \end{bmatrix}$ = 2 x 3 matrix

Definition **3** finally tells us how to compute the dot product. It is usually best, when understanding tough mathematical formulas, to start with a concrete example. Let's start with example 1 above.

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Let us treat this as a math problem first. We start by knowing that our output matrix has 2 positions. Let us refer to these output positions as $O_{0,0}$ and $O_{1,0}$ respectively.

Let us start by calculating $O_{0,0}$. We thus know, that $i = 0$ and $j = 0$ in our formula. So we can rewrite the formula from definition 3 as

$$O_{0,0} = \sum_{k=0}^{M} X_{0,k} * w_{k,0}$$

So we know that we have to multiply all the elements in the 0th row of the first matrix by all the values in the 0th column of the second. Doing that, we get, for

$$O_{0,0} = 1 * 1 + 2 * 2 = 5$$

Repeating again for $O_{1,0}$:

$$O_{1,0} = 2 * 1 + 3 * 2 = 8$$

With that, we can thus write the output as:

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$$

Now, how does this help us write our code? We know that we must fill in each element in the output matrix, as our formula only gives an answer for a specific position in the output matrix. So our Java code will start as follows:

```
for(int i = 0 ; i < outputMatrix.length ; i++){
    for(int j = 0 ; j < outputMatrix[0].length ; j++){
        //some code here
    }
}
```

Then, we know the summation from definition 3 goes inside the for loop. Summations, in CS, are also just for loops. Use that as a start to your code.

You will note I am being a bit sparse on hints at this point- I've given you basically everything you will need to implement the dot product, but you must look carefully at definition 3 to implement the rest.

Your implementation will go in `MathOperations` in the function `dotProduct`.

Step 2: Implementing functions on your own

In this part, we will implement two common functions in image processing and their derivatives. These two functions are known as the rectify (RELU) function, and the exponential linear unit (ELU) function.

Here, most of the code-writing will be on your own as well.

1) The rectify (RELU) function

We define the rectify function as follows:

$$rectify(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

Write your implementation in the function `RELU` in `MathOperations`.

2) The Exponential linear unit (ELU) function

We define the exponential linear unit function as follows:

$$elu(x) = \begin{cases} e^x - 1 & x \leq 0 \\ x & x > 0 \end{cases}$$

Write your implementation in the function `ELU` in `MathOperations`.

Step 3: Derivatives of functions

In case you haven't reached single variable calculus yet, you should first review the math notes here.

We also start by defining 5 common derivatives. let b be any constant value. Then

1)

$$\frac{d}{dx}b = 0$$

2)

$$\frac{d}{dx}x = 1$$

3)

$$\frac{d}{dx}e^x = e^x$$

4)

$$\frac{d}{dx}b * x = b$$

5)

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

Using these functions, you can thus implement the derivatives of both `RELU` and `ELU`. Think about which derivative rules you will want to use in each step. Remember also that piecewise functions can be broken up into various smaller derivatives as well.

Step 4: Implementing the convolutional operator

Your final assignment step (*sniffles*) is to implement the convolutional operator. This is a variation (*kind of) of the dot product, but instead of multiplying a full input row i by a full weight column j to produce each output element, we instead multiply values across a small square of the first matrix. This produces an output matrix, that, in image processing, tends to reflect characteristics of the image that are local to one area. Think, for example, edges of an object in an image, as an example of a local feature of an image.

Our definition of convolution will follow from our definition of the dot product.

1) We define the output dimensions of the convolution of input matrix I by weight matrix w as $height(O) = height(I) - height(w) + 1$, and $width(O) = width(I) - width(w) + 1$. This means the output matrix is typically smaller than the input matrix. However, in our version of convolution, we will place our input matrix at

the center of a matrix of zeros, “padding” the matrix with zeros, where the amount of padding $= height(w) - 1$ for $height(O)$ and $= width(w) - 1$ for $width(O)$.

2) We define the output of convolution as the following. For any output position $O_{i,j}$, and any padded input matrix I:

$$O_{i,j} = \sum_{k=0}^{height(w)} \sum_{l=0}^{width(w)} I_{i+k,j+l} * w_{k,l}$$

This is a very confusing equation to look at at first. So let’s try to break it down with an example

let $I = \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 4 & 5 \\ 10 & 11 & 12 & 13 \\ 0 & 1 & 3 & 8 \end{bmatrix}$ and let $w = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$.

First, according to our definition, we must “pad” the input matrix with $floor(width(w)/2)$ zeros on each dimension. This means we must pad I with 1 row of zeros on the left, right, top and bottom of the matrix. The resulting padded matrix looks like:

$$I_{padded} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 4 & 5 & 0 \\ 0 & 2 & 3 & 4 & 5 & 0 \\ 0 & 10 & 11 & 12 & 13 & 0 \\ 0 & 0 & 1 & 3 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Next, using definition **2**, we know our output matrix will be 4 x 4, because these were the dimensions of the original, unpadded input.

Next, let us start by deriving output $O_{0,0}$. Using definition **3**, our formula is:

$$O_{0,0} = \sum_{k=0}^{3} \sum_{l=0}^{3} I_{k,l} * w_{k,l}$$

,

where I here is actually equal to the I_{padded} we defined above. Plugging in the values:

$$O_{0,0} = 0 * 0 + 0 * 1 + 0 * 0 + 0 * 1 + 1 * 1 + 2 * 0 + 0 * 0 + 1 * 2 + 0 * 3 = 3$$

Next, we can derive our output for $O_{0,1}$. Our formula now becomes:

$$O_{0,1} = \sum_{k=0}^{3} \sum_{l=0}^{3} I_{k,l+1} * w_{k,l}$$

Plugging in the values again:

$$O_{0,1} = 0 * 0 + 0 * 1 + 0 * 0 + 1 * 1 + 2 * 1 + 4 * 0 + 2 * 0 + 1 * 3 + 0 * 4 = 6$$

Our complete output matrix will look like:

$$O = \begin{bmatrix} 3 & 6 & 10 & 14 \\ 13 & 18 & 23 & 27 \\ 12 & 25 & 30 & 38 \\ 10 & 12 & 16 & 24 \end{bmatrix}$$

Now, to implement the convolutional operator, you will again want to break down the formula into for loops. Your code should look something like

```
for(int i = 0 ; i < outputMatrix.length ; i++){
    for(int j = 0 ; j < outputMatrix[0].length ; j++){
        // Formula goes here
    }
}
```

But where the formula goes, you will have two inner for loops- one for each dimension of weight matrix w.

Implement your function in `Convolution`. You will test it using `TestConv1`, `TestConv2` and `TestConv3` in the `TestMathOperations` file in the testing folder.

Step 5: Your next steps

Now that you have learned the basics of programming and research work at this lab, you now have two choices as to what assignment you would like to do next.

1) QA Testing Fundus Annotation SofTware (FAST) with me. In this assignment, you will be tasked with testing and debugging a software I am currently producing to help a clinic label eye images for us. The tool works a bit like Microsoft Paint, but contains a few extra features using image processing techniques to make the annotation of the fundus images a bit easier. Specifically, you will write unit tests (the tests you run to check your code works) for many major components of FAST. If FAST fails a test it is not supposed to, you will be tasks with attempting to debug FAST, and seeing if you can localize the error. This project will involve a lot of programming at a reasonably high level in the Java language, so if you liked learning Java and want to continue on programming, this is the project for you.

2) Annotation of optic disks. In this assignment, you will be tasked with using FAST to label a part of the eye known as the optic disk. The optic disk is very obvious in most eye images- it's a bright circle either to the left or the right of the image. Your job will be to draw clean, accurate bounds around the optic disk in about 600 eye images. While this job can get a bit repetitive, if you found programming in Java to be boring or frustrating, annotating the optic disks in the fundus images can provide us clean, accurate data to improve FAST with.