1 Introduction to Approximate Multipliers

Did you understand the concept of a multiplication table? Now it is important, as the topic for today is to implement the very same program as in the homework, but with an **Approximate Multiplication**. Last time, we dealt with an Accurate Multiplication via an integer-based look-up table. This time, on top of using an integer-based look-up table (which in itself can be regarded as an estimation since it cuts floating point values) we will use a simpler look-up table that gives us the result of an integer multiplication not accurately but approximately. So to speak, 3*4 is approximately 10.

What is an Approximate Multiplication table?
The approximate multiplication table is a coarser version of the accurate look-up table. Coarse means that there will be patches within the table that yield the same result, whereas the accurate table always has a different result for each integer multiplication (as multiplication tables should be). The advantage of having an approximate table comes when implementing this on the hardware. We then can save time calculating a multiplication.

1.1 Subtask 1: Replacing Accurate Multiplication By Approximate Multiplication

Start by downloading `Approximate Mult1.npy` from the resource folder on moodle and load it as a numpy matrix:

```
Multiplier_Approx=np.load("Approximate Mult1.npy")
```

This time, the approximate look-up table is given. Plotting it as an image you will notice that it looks similar to the original version. However, to see the difference, you can for example plot the difference between both images (the difference between `Multiplier` and `Multiplier_Approx`.) Plotting the difference as an image will show you that indeed they differ.

1.2 Subtask 2: Implementing Look-up Based Vector and Matrix Multiplication (Approximate Version)

In this subtask we are going to define a function called `My_Mult_approx(a,b)` that takes two parameters `a` and `b` that are converted into a numpy array. The purpose of this function will be to use the look-up table to multiplicate two numpy arrays of the same shape element-wise. This is a subprocess of the convolution, where adding up all values of the result of this element-wise multiplication leads to the value of the next pixel in the next layer.

1. **Function Definition**

   At the heart of this subtask, we have a custom multiplication function named `My_Mult_Approx`. It takes two inputs, `a` and `b`, and aims to perform element-wise multiplication. Let's start by defining the function:

   ```
   def My_Mult_Approx(a, b):
   ```

   This function takes in two arrays, `a` and `b`, representing the operands for multiplication.

2. **Data Preparation**

   Next, the function prepares the data for computation. It converts `a` and `b` into NumPy arrays to ensure compatibility:

   ```
       a = np.array(a)  # Convert data 'a' to a NumPy array
       b = np.array(b)  # Convert data 'b' to a NumPy array
   ```

   This step is essential for ensuring consistent data types.

3. **Determining Array Shapes**

   The code then proceeds to determine the shapes of the input arrays `a` and `b`. This is crucial for understanding the dimensions of the operands:

```
a_shape = np.shape(a)  # Get the shape of array 'a'
```

   This information helps tailor the multiplication operation accordingly.

4. **Reshaping 'b' (if needed)**

   In the case where `b` is not a one-dimensional array, it is reshaped to match the shape of `a`. This ensures that element-wise multiplication is possible between the two arrays:

```
b = np.reshape(b, a_shape)  # Reshape 'b' to match the shape of 'a'
```

   Reshaping 'b' aligns the dimensions for multiplication.

5. **Result Matrix Initialization**

   The result matrix `res` is initialized with zeros to store the multiplication results. Its shape matches the shape of `a`:

```
res = np.zeros(a_shape)  # Initialize the result matrix with zeros
```

   `res` is where the multiplication results will be stored.

6. **Element-wise Multiplication**

   The core of the function involves element-wise multiplication. The specific implementation depends on the dimensions of the input arrays, as indicated by the use of conditionals.

   If `a` and `b` are one-dimensional arrays, the code performs element-wise multiplication for each element of the arrays and stores the result in `res`. It accesses precomputed values from the `Multiplier_Approx` array based on the elements of `a` and `b` after applying an offset of 128:

```
if len(a_shape) == 1:
    for i in range(np.shape(a)[0]):
        res[i] = Multiplier_Approx[int(a[i]) + 128, int(b[i]) + 128]
```

   This part handles one-dimensional input arrays and leverages precomputed values for multiplication.

   If `a` and `b` are two-dimensional arrays, the code employs nested loops to traverse through the elements. It calculates the multiplication result for each element in the arrays and assigns it to the corresponding position in `res`. Similar to the one-dimensional case, the code uses precomputed values from the `Multiplier_Approx` array based on the elements of `a` and `b` after applying the offset:

```
if len(a_shape) == 2:
    for i in range(a_shape[0]):
        for j in range(a_shape[1]):
            res[i, j] = Multiplier_Approx[int(a[i, j]) + 128, int(b[i, j]) + 128]
```

   This part handles two-dimensional input arrays, performing element-wise multiplication and utilizing precomputed values for efficient computation.

7. **Returning the Result**

   After performing the element-wise multiplication and populating the `res` matrix with the results, the function concludes by returning the result:

```
return res  # Return the matrix containing the multiplication results
```

   The function returns the `res` matrix, which now holds the element-wise multiplication results of the input arrays `a` and `b`.

The function is now prepared to perform element-wise multiplication of the input arrays `a` and `b`, and the result will be stored in the matrix `res`. The code is structured to handle both one-dimensional and two-dimensional arrays, making it versatile for various applications.

1.3 Subtask 3: Implementing Look-up Based Convolution (Approximate Version)

In this subtask, you will implement a convolution function named `My_Conv2d_LT_Approx`. The goal is to perform convolution using a look-up based approach with an approximate version of multiplication.

1. **Function Definition**

   First we define the function that will use the approximate look-up table:

   ```
   def My_Conv2d_LT_Approx(a, b):
   ```

   The function takes two input arrays, `a` and `b`, which represent the data and the filter, respectively.

2. **Data Preparation**

   The arrays that are handed to this function should be prepared in the usual way:

   ```
       a = np.array(a)  # Convert data to a NumPy array
       b = np.array(b)  # Convert filter to a NumPy array
   ```

   This step ensures that `a` and `b` are treated as NumPy arrays, making them suitable for array operations.

   ```
       a_shape = np.shape(a)  # Get the shape of the data
       b_shape = np.shape(b)  # Get the shape of the filter
   ```

   Understanding the dimensions of `a` and `b` is crucial for setting up the convolution process.

3. **Result Matrix Initialization**

   ```
       res_shape1 = np.abs(a_shape[0] - b_shape[0]) + 1  #
           Calculate the first dimension of the result
       res_shape2 = np.abs(a_shape[1] - b_shape[1]) + 1  #
           Calculate the second dimension of the result
   ```

   The shape of the result matrix is based on the difference in dimensions between the data and filter.

   ```
       res = np.zeros([res_shape1, res_shape2])  # Create a
           matrix to store convolution results
   ```

   This matrix is where the convolution results will be stored.

4. **Convolution in a Loop**

   Having prepared all the ingredients, we can start the convolution process. Similar to how it was done in the previous exercise we will go through a nested loop for each pixel of the resulting image.

   ```
       for i in range(res_shape1):
           for j in range(res_shape2):
               res[i, j] = np.sum(
                   My_Mult_Approx(np.flip(b), a[i:i +
                       b_shape[0], j:j + b_shape[1])
               )
   ```

   Here's how it works: For each element in the result matrix, we take a region from the input data `a` of the same size as the filter `b`, then we perform element-wise multiplication between this region and the filter (after flipping it horizontally and vertically). After the multiplication, we sum up all the resulting values to obtain the final convolution result for that position in the result matrix. This process repeats for all elements in the result matrix, effectively convolving the data and filter to produce the output.

5. **Returning the Result**

   Finally, the function returns the result matrix `res`, which now holds the convolution results of `a` and `b` using the look-up based convolution approach with approximate multiplication.

   ```
       return res  # Return the matrix containing the
           convolution results
   ```

This revised explanation follows the same format as Subtask 2, provides a breakdown of each part, and uses an enumerated list to make the explanation more structured.

1.4 Subtask 4: Checking the Results

1. **Calculate the Look-up Based Convolution (Approximate Version) Result**

   You've implemented your custom convolution function `My_Conv2d_LT_Approx` in the previous subtask. It's time to put it to the test:

```
Result_Imp_LT_Approx = My_Conv2d_LT_Approx(img, fltr)
    # Calculate the convolution result using your
    implementation with approximate multiplication
```

2. **Check the Results**

   It's time for the grand comparison! Let's check if the results from your look-up based convolution (approximate version) and the Python library match. We'll calculate the error value:

```
checkup3 = np.sum(np.abs(Result_Imp_LT_Approx -
    Result_Python))  # Calculate the error value by
    comparing the two results
```

3. **Display the Error Value**

```
print("The Error Value is:", checkup3)  # Display the
    error value to see how closely your
    implementation of look-up based convolution (
    approximate version) matches the Python library's
    results
```

   The Verdict

   You've checked the results! If the error value is close to zero, it means your implementation of look-up based convolution (approximate version) is doing a great job. The smaller the error, the closer your results are to the Python library's results.