An Insight Into Assignment 6

Krishna Murari Chivukula

May 24, 2024

Abstract

This report contains my insights and learnings from Assignment 6 of the course ID2090.

1 Treasure Hunt II

1.1 Problem Abstract

This problem dealt with using commands involved in **Git** for navigating branches and nodes, looking for Arceus. The result was an **ASCII** picture of Arceus.

1.2 Navigation Using Git

- I started off by running the given shell binary, `script.sh.x` in the home directory. This gave me a file titled `question_4.cpp`.
- Following this was more of a wild hunt , starting of with using the command `git branch` to identify which branch I was on. Then, I used `git log`, to identify the specific commit(node).
- Then , it was a series of using `git checkout(commit hash)` and `git checkout(branch name)` to navigate between different branches and nodes. Most cases it resulted in the phrase "Long way to Go" ,indicating that Arceus was not in that particular node.
- Finally , at one specific branch and node, Arceus was found, with the phrase "Congrats, You caught Arceus with the Master Ball. It is trying to come out." "Freeze it by merging this branch with this node, in MASTER".
- This was followed by the creation of a new branch named "master", done using `git branch master`. Then, the branch where Arceus was found was merged with Master, using `git merge`.

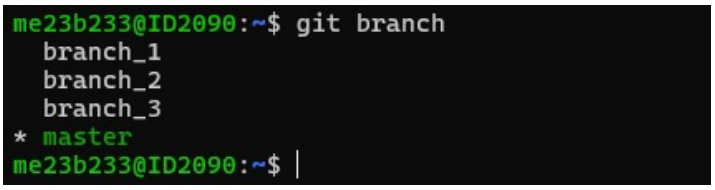

Figure 1: Currently On Created Master Branch

1.3 Commit And Branch

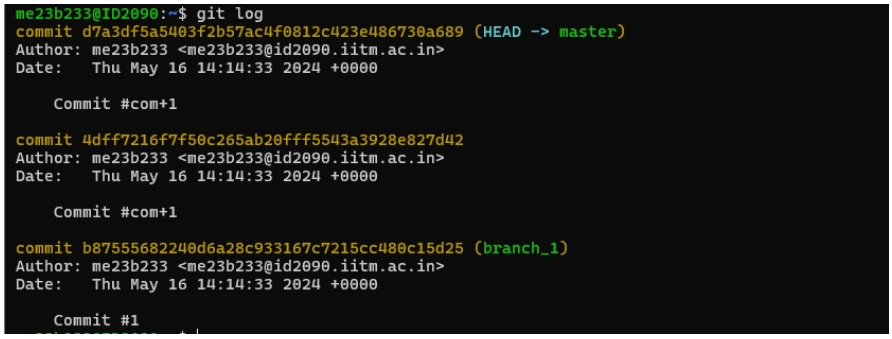

Figure 2: Showing Commit and Branch

1.4 Found Arceus

Figure 3: Found Arceus !

2 Image Processing

2.1 Problem Abstract

This problem dealt with a task of Image Processing. We were given 2 images, a normal one named `Lena.png` and a distorted version of the image named `Noisy_Lena.png`.Kernel operations were performed and an analysis was done. I used both **Python** with the main library as **OpenCV**, as well as **GNU Octave**.

2.2 Kernels and Convolution in Image Processing:

2.2.1 Kernels

In the domain of image processing, a kernel, also known as a convolution matrix, is a matrix that is used for blurring, sharpening as well as operations like edge detection in images. It achieves this by a convolution with the image itself.

- Essentially , an image is just a matrix/grid of pixels , each possessing a particular value corresponding to it's intensity. For example, in an 8-bit greyscale image each picture element has an assigned intensity that ranges from 0 to 255.
- Based on the functionality of the kernel, it's entries are also different. Below are some examples of basic kernels used in image processing.

$$\text{Edge Detection Matrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad \text{Gaussian Blur Matrix} \quad \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

- This kernel is convolved with the image matrix. The process of convolution is described below.

2.2.2 Convolution

- Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. Convolution is not traditional matrix multiplication, despite being similarly denoted by *. The general mathematical process of matrix convolution is depicted below.

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{(m-i)(n-j)} y_{(1+i)(1+j)}$$

- The convolution is performed by sliding the kernel over the image, generally starting at the top left corner, so as to move the kernel through all the positions where the kernel fits entirely within the boundaries of the image. Each kernel position corresponds to a single output pixel, the value of which is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel, and then adding all these numbers together.
- If the image has M rows and N columns, and the kernel has m rows and n columns, then the size of the output image will have $M - m + 1$ rows, and $N - n + 1$ columns.
- However, in order to do convolutionfor the bottom and right edges of the image, it is necessary to invent input pixel values for places where the kernel extends off the end of the image. Removing $n - 1$ pixels from the right hand side and $m - 1$ pixels from the bottom will fix things. We can also "pad" the image boundary with pixel values.
- An image depicting kernel convolution is depicted below.

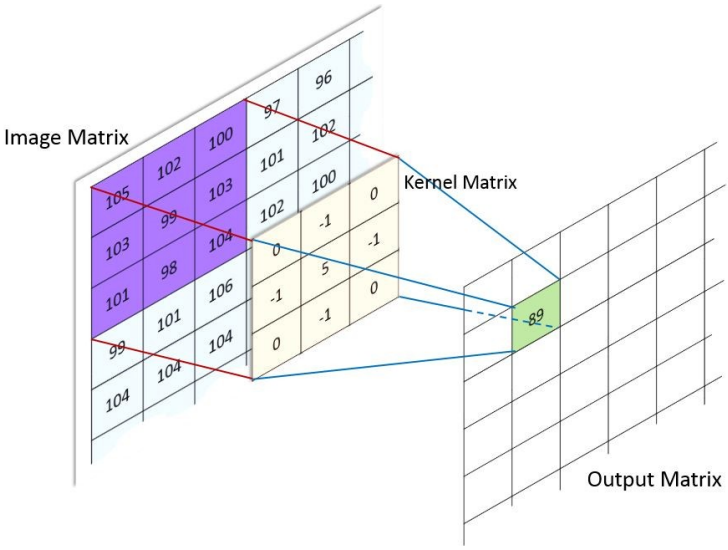

Figure 4: Convolution

2.3 Some Image Processing Operations

We will be using the normal `Lena.jpg` image for this purpose, which looks like the one below.

Figure 5: Lena.jpg

The operations we are dealing for this particular section are, namely, Edge Detection and Brightness Change.

2.3.1 Edge Detection

- For edge detection, we use the Sobel operator, which involves convolving the image with the Sobel kernels.
- Sobel filtering involves applying two 3 x 3 convolutional kernels to our image. The kernels are usually called G_x and G_y, and they are shown below These two kernels detect the edges in the image in the horizontal and vertical directions respectively.

$$G_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

```python
import cv2
import numpy as np
#Load as grayscale image,each pixel intensity 0-255.
image = cv2.imread('Lena.png', 0)

# Apply Sobel edge detection
sobelx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobely = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
sobel_combined = cv2.magnitude(sobelx, sobely)

cv2.imwrite('sobel_lena.png', sobel_combined)
```

Listing 1: Sobel Filtering for Edge Detection

- The image with only the edges detected is depicted below.The `.Sobel` function in **OpenCV** convolves both the G_x and G_y Sobel kernels with the image to produce the output.

Figure 6: Edge Detection In Action

2.3.2 Motion Blur

- Motion Blur is done by applying a convolution filter to the image with a specially designed kernel that averages pixel values along the direction of the simulated motion. The kernel size and shape determine the extent and direction of the blur.

```python
def motion_blur(image, size):
    # Create a motion blur kernel that operates horizontally
    kernel_blur = np.zeros((size, size))
    kernel_blur[int((size - 1) / 2), :] = np.ones(size)
    kernel_blur = kernel_blur / size

    # Apply the kernel to the image using cv2.filter2D
    blurred_image = cv2.filter2D(image, -1, kernel_blur)

    return blurred_image

# Load the original image
image = cv2.imread('Lena.png')
# Define the size of the motion blur kernel
kernel_size = 20
# Apply motion blur to the image
```

```
blurred_image = motion_blur(image, kernel_size)
# Display the blurred image
cv2.imwrite('motion_blurred_lena.png', blurred_image)
cv2.imshow('Motion Blurred Image', blurred_image)
```

Listing 2: Motion Blur

- For this, I used a kernel size of 20 , by modifying this value we can change the extent of motion blur. My kernel is initialised to blur in the horizontal direction. The motion blurred image is shown below.

Figure 7: Motion Blur In Action

3 Denoising Images

For this particular section , we will be dealing with a Noisy Image of Lena , titled `Noisy_Lena.png`. It is depicted below.The techniques used are called Spatial Domain Filtering.

Figure 8: Noisy Lena

3.0.1 Gaussian Blur

- Gaussian blur uses a Gaussian kernel to smooth the image, reducing noise and detail.The center of the kernel contains the highest value, and values decrease as you move away from the center.
- 1. **Kernel Size**: The size of the Gaussian kernel must be positive and odd. Larger kernels result in more smoothing.
  2. **Standard Deviation**: Controls the amount of blur. A larger value means more blur.
  3. **Function**: $cv2.GaussianBlur(image, (kernelwidth, kernelheight), sigma)$

```
'''Gaussian Blur Denoising'''

# Load the noisy image
noisy_image = cv2.imread('Noisy_Lena.png')

# Apply Gaussian blur
gaussian_blur = cv2.GaussianBlur(noisy_image, (5, 5), 0)

cv2.imwrite('denoised_lena.png', gaussian_blur)
```

Listing 3: Gaussian Blur

Figure 9: Gaussian Blur Denoised Lena

3.0.2 Median Blur

- Median blur replaces each pixel's value with the median value of the neighboring pixels, which is particularly effective at removing "salt-and-pepper" noise.
- 1. **Kernel Size**: The size of the square kernel must be positive and odd.
  2. **Effectiveness**: Particularly effective at removing salt-and-pepper noise.
  3. **Function**: $cv2.medianBlur(image, kernel_size)$

```
    '''Median Blur Denoising'''

image = cv2.imread('Noisy_Lena.png')
# Apply Median blur
median_blur = cv2.medianBlur(image, 5)

# Display the original and denoised images
cv2.imshow('Original Image', image)
cv2.imshow('Median Blurred Image', median_blur)

# Save the result
cv2.imwrite('denoised_median.png', median_blur)
```

Listing 4: Median Blur

Figure 10: Median Blur Denoised Lena

3.0.3 Non Localised Means Denoising

- I implemented this only for trial purposes, it seemed to produce some sort of a denoised image, however it was too smoothened and sort of reduced the overall quality of the image.
- This technique of denoising removes noise by comparing all the pixels in a window around a target pixel and averaging them, giving more weight to those that are more similar.

```
''' Non-Local Means Denoising'''

image = cv2.imread('Noisy_Lena.png')
# Apply Non-Local Means denoising
nlm_denoised = cv2.fastNlMeansDenoisingColored(image, None, 10, 10, 7, 21)

# Display the original and denoised images
cv2.imshow('Original Image', image)
cv2.imshow('NLM Denoised Image', nlm_denoised)

# Save the result
cv2.imwrite('denoised_nlm.png', nlm_denoised)
```

Listing 5: NLM Denoising

Figure 11: NLM Denoised Lena

4 Steps To Filter an Image Using Fourier Transform

Filtering noise from an image using Fourier Transform involves transforming the image to the frequency domain, manipulating its frequency components, and then transforming it back to the spatial domain. This technique is effective for noise with distinct frequency characteristics.

- **Transform the Image to the Frequency Domain**: Apply the Discrete Fourier Transform (DFT) to convert the image from the spatial domain to the frequency domain.
- **Manipulate the Frequency Components**: Apply a filter to the frequency components to suppress the noise.
- **Transform Back to the Spatial Domain**: Apply the Inverse Fourier Transform to convert the filtered image back to the spatial domain.

Below is the Python code that performs noise filtering using Fourier Transform with improved error handling:

```
'''Noise Handling Using Fourier Transform'''

image = cv2.imread('Noisy_Lena.png', cv2.IMREAD_GRAYSCALE)

# Apply DFT
dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shifted = np.fft.fftshift(dft)

# Create a mask for a low-pass filter
rows, cols = image.shape
crow, ccol = rows // 2 , cols // 2
mask = np.zeros((rows, cols, 2), np.uint8)
r = 30  # Radius of the low-pass filter
center = [crow, ccol]
x, y = np.ogrid[:rows, :cols]
mask_area = (x - center[0]) ** 2 + (y - center[1]) ** 2 <= r*r
mask[mask_area] = 1

# Apply the mask to the DFT shifted image
fshift = dft_shifted * mask

# Applying Inverse DFT
f_ishift = np.fft.ifftshift(fshift)
image_back = cv2.idft(f_ishift)
image_back = cv2.magnitude(image_back[:, :, 0], image_back[:, :, 1])

# Normalize the image for displaying
cv2.normalize(image_back, image_back, 0, 255, cv2.NORM_MINMAX)
image_back = np.uint8(image_back)

# Display
plt.figure(figsize=(12, 6))

plt.subplot(131), plt.imshow(image, cmap='gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])

plt.subplot(132), plt.imshow(cv2.imread('Noisy_Lena.png', cv2.IMREAD_GRAYSCALE), cmap=
    'gray')
plt.title('Noisy Image'), plt.xticks([]), plt.yticks([])

plt.subplot(133), plt.imshow(image_back, cmap='gray')
plt.title('Filtered Image (Frequency Domain)'), plt.xticks([]), plt.yticks([])

plt.show()
```

Listing 6: Noise Filtering Using Fourier Transform

4.1 Comparison with the Normal Spatial Domain Filtering

- **Performance**:
  1. **Frequency Domain Filtering**: Effective for periodic noise and can target specific frequency components. May not be as intuitive as spatial domain filters.
  2. **Spatial Domain Filtering**: Methods like Gaussian and median blur are easier to implement and work well for general noise but may not be as effective for specific noise patterns.
- **Nature of Noise**:
  1. **Frequency Domain Filtering**: Best suited for noise with distinct frequency characteristics, such as periodic noise.
  2. **Spatial Domain Filtering**: Suitable for random noise, such as Gaussian noise (Gaussian blur) and salt-and-pepper noise (median blur).
- **Boundary Conditions**:
  1. **Periodic Boundary Condition**: Implicitly handled due to the periodic nature of the DFT.

Figure 12: Fourier Transform Denoised Lena

5 References

- Git Documentation : https://git-scm.com/docs
- OpenCV Documentation:https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html
- Kernels : https://setosa.io/ev/image-kernels/
- Types of Kernels : https://en.wikipedia.org/wiki/Kernel_(image_processing)
- Convolution : https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411
- I also used StackOverFlow and ChatGPT for debugging code and digging in deeper with respect to topics pertaining to the assignment.