Improving Image Processing for Electron Diffraction Pattern Datasets

Enhance Simulated Patterns to Mimic Real Data

Ibrahim Uruc Tarim

24 April, 2023

Contents

1 Abstract **1**

2 Introduction **1**
2.1 What is Electron Diffraction and Why is it Important? . . . 1
2.1.1 Aim 1 . . . 2
2.1.2 Aim 2 . . . 2

3 Data Science Methods **2**

4 Exploratory Data Analysis **2**
4.0.1 The Data . . . 2

5 Aim 2 Enhance Simulated Patterns to Mimic Real Data **3**
5.1 Add Noise To Simulated Images . . . 13

6 Statistical Learning: Modeling & Prediction **22**
6.0.1 Convolutional Neural Networks . . . 22
6.1 Steps to build and train our CNN models: . . . 22

7 Discussion **28**

8 Conclusions **28**

1 Abstract

The field of materials science heavily relies on electron diffraction patterns to characterize the crystal structure of materials. Accurately analyzing these patterns is critical, and the efficiency and accuracy of image processing algorithms play a significant role in this analysis. In order to improve the performance of these algorithms, this project aims to address several key challenges in the image processing of electron diffraction pattern datasets. The project will focus on developing a graph model for accurate point detection, adding synthetic noise to simulated patterns to make them more representative of real data, denoising the input images to enhance their accuracy, and evaluating the performance of the image processing algorithms on real data. The goal of this project is to contribute to the field of materials science by improving the accuracy and efficiency of electron diffraction pattern analysis.

2 Introduction

- Materials science relies on electron diffraction patterns to analyze crystal structure.
- Accurate analysis of these patterns is essential, and image processing algorithms play a significant role in this analysis.
- Objective: Improve machine learning models for analyzing selected area electron diffraction (SAED) patterns.

2.1 What is Electron Diffraction and Why is it Important?

- Electron diffraction examines the atomic structure of materials.

- A beam of electrons is fired at a sample and a diffraction pattern is observed.
- The pattern reveals the crystal structure, symmetry, and different phases of the material.
- Identifying crystal structure, symmetry, and phases helps predict behavior and properties.
- Electron diffraction is used to analyze various types of materials, from metals to ceramics to polymers.
- It is a powerful tool for gaining insights and advancing the field of materials science.

2.1.1 Aim 1

- Develop a Graph Model for Point Detection The first aim of the project is to develop a graph model for locating the points in electron diffraction patterns.
- The model should be able to accurately locate the center of the image and crop the image to a 512x512 pixel resolution.
- This is important because accurate detection of points is crucial for the proper analysis of electron diffraction patterns and also to expedite the process and reduce human workload.

2.1.2 Aim 2

- Enhance Simulated Patterns to Mimic Real Data The second aim of the project is to enhance the simulated electron diffraction patterns to mimic real data.
- This will be achieved by adding synthetic noise and applying blur filters to the simulated patterns.
- This will help to ensure that the simulated patterns are representative of real data, leading to more accurate results when testing image processing algorithms.

3 Data Science Methods

Machine learning, statistical analysis, and image processing are some of the data science techniques that will be used. The project will specifically involve creating a graph model for precise point detection, denoising the input images to improve accuracy, adding synthetic noise to simulated patterns to make them more resemblant of real data, and testing the effectiveness of the image processing algorithms on actual data.

Proficiency in image processing methods like point identification, noise addition, and denoising is one of the crucial competencies needed for this project. Additionally, knowledge in machine learning algorithms, particularly deep learning and convolutional neural networks, is needed for the project. For this undertaking, it is also essential to be able to evaluate and analyze massive datasets using statistical techniques.

The 'imager' package for image processing, the 'keras' package for convolutional neural network implementation, and the 'caret' package for data partitioning and cross-validation are the packages that will be used for this project. Depending on the particular demands of the project, more packages might be used.

4 Exploratory Data Analysis

4.0.1 The Data

- Professor Assel Aitkaliyeva of the University of Florida provided the datasets, which were then used by Nathaniel K. TOMCZAK, a graduate student researcher at the Mesoscale Science Lab of Case Western Reserve University, to complete his master's thesis.
- First dataset consists of 865 selected area electron diffraction (SAED) patterns from two plutonium (Pu) and zirconium (Zr) alloy systems: Pu with 10 weight percent Zr (Pu-10Zr) and Pu with 30 weight percent Zr (Pu-30Zr).
- The SAED patterns were categorized into three classes: 364 alpha patterns, 282 delta patterns, and 219 other patterns. The space group and corresponding lattice parameters for each pattern are also provided.
- The second dataset rotated each pattern before cropping by 15° increments 23 times, scaling the original dataset by a factor of 24.
- There were 20,760 images spread between the three classes as follows: 8,736 alpha patterns, 6,768 delta patterns, and 5,256 other patterns.

5 Aim 2 Enhance Simulated Patterns to Mimic Real Data

- The second aim of the project is to enhance the simulated electron diffraction patterns to mimic real data.
- This will be achieved by adding synthetic noise and applying blur filters to the simulated patterns.
- This will help to ensure that the simulated patterns are representative of real data, leading to more accurate results when testing image processing algorithms.

Here, we are importing required libraries and defining the file paths for the real and simulated images.

```
real_images_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/real_images"
simulated_images_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/simulated_image

# Read image file names
real_files <- list.files(real_images_path, full.names = TRUE)
simulated_files <- list.files(simulated_images_path, full.names = TRUE)
```

Here we are calculating the average intensity of the real and simulated images using the process_images function, which processes the images in a directory using the calculate_average_intensity function to calculate the average intensity of each individual image.

```
# Function to calculate the average intensity of an image
calculate_average_intensity <- function(image) {
  return(mean(image))
}

# Function to process images and calculate average intensity
process_images <- function(files) {
  num_files <- length(files)
  cumulative_sum <- 0

  for (idx in 1:num_files) {
    img <- EBImage::readImage(files[idx])
    img <- imager::as.cimg(img)
    avg_intensity <- calculate_average_intensity(img)
    cumulative_sum <- cumulative_sum + avg_intensity
  }

  return(cumulative_sum / num_files)
}

# Calculate the average intensity for real and simulated images
average_intensity_real <- suppressWarnings(process_images(real_files))
average_intensity_simulated <- suppressWarnings(process_images(simulated_files))


cat("Average intensity for real images:", average_intensity_real, "\n")
```

```
## Average intensity for real images: 0.14566
```

```
cat("Average intensity for simulated images:", average_intensity_simulated, "\n")
```

```
## Average intensity for simulated images: 0.37558
```

The average intensity of the real images is 0.14566 while the average intensity of the simulated images is 0.37558. These values can be useful for comparing the overall brightness of the images and for making adjustments to improve image quality.

This function named process_images which accepts a list of image file names as input and returns a list containing the mean, median, and standard deviation of pixel intensities of each image. The function initializes three empty vectors to store the computed mean, median, and standard deviation values for each image. It reads each image using the EBImage::readImage function and converts it to a 1D array. Then, it calculates the mean, median, and standard deviation of the pixel intensities for each image and stores them in the respective vectors. Finally, the function returns a list containing the three vectors of mean, median, and standard deviation values for all images in the input directory.

```
process_images <- function(files) {
  num_files <- length(files)

  mean_values <- numeric(num_files)
  median_values <- numeric(num_files)
  sd_values <- numeric(num_files)

  for (idx in 1:num_files) {
    img <- EBImage::readImage(files[idx])
    img <- imager::as.cimg(img)
    img_1d <- as.vector(img)

    mean_values[idx] <- mean(img_1d)
    median_values[idx] <- median(img_1d)
    sd_values[idx] <- sd(img_1d)
  }

  return(list(mean_values, median_values, sd_values))
}
```

We call the process_images function to calculate these statistics for each image in the real and simulated image directories, and store the results in summary_real and summary_simulated.

```
summary_real <- suppressWarnings(process_images(real_files))
summary_simulated <- suppressWarnings(process_images(simulated_files))

cat("Mean, median, and standard deviation for real images:\n")
```

```
## Mean, median, and standard deviation for real images:
```

```
cat("Mean:", mean(summary_real[[1]]), "Median:", mean(summary_real[[2]]), "SD:", mean(summary_real[[3]]), "\n")
```

```
## Mean: 0.14566 Median: 0.13432 SD: 0.087618
```

```
cat("Mean, median, and standard deviation for simulated images:\n")
```

```
## Mean, median, and standard deviation for simulated images:
```

```
cat("Mean:", mean(summary_simulated[[1]]), "Median:", mean(summary_simulated[[2]]), "SD:", mean(summary_simulate
```

```
## Mean: 0.37558 Median: 0.24706 SD: 0.37657
```

Based on the results, we can see that the mean, median, and standard deviation of pixel intensities for the simulated images are significantly higher than those for the real images. This suggests that the simulated images may have been generated with higher contrast and brightness levels compared to the real images.

Here, we are loading the “imager” and “tiff” libraries and setting the current working directory to the root of the project directory. We then define two character vectors “real_dir” and “simulated_dir” which store the names of the directories containing the real and simulated images, respectively. Finally, we use the “list.files” function to retrieve the file names of all images in both directories and store them in “real_files” and “simulated_files” character vectors, respectively.

```
library(imager)
```

```
## Loading required package: magrittr
```

```
##
## Attaching package: 'magrittr'
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
##
## Attaching package: 'imager'
```

```
## The following object is masked from 'package:magrittr':
##
##     add

## The following object is masked from 'package:dplyr':
##
##     where

## The following object is masked from 'package:tidyr':
##
##     fill

## The following objects are masked from 'package:stats':
##
##     convolve, spectrum

## The following object is masked from 'package:graphics':
##
##     frame

## The following object is masked from 'package:base':
##
##     save.image
```

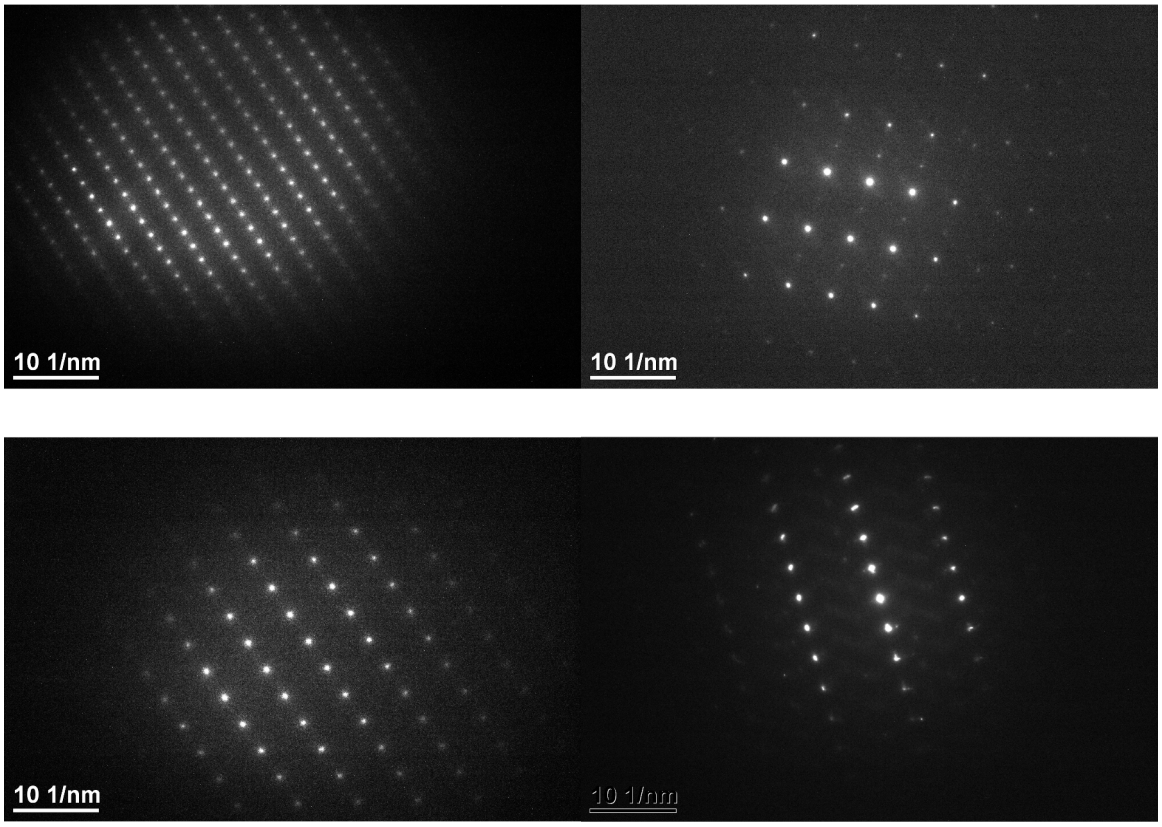
```
library(tiff)

setwd("/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj")
real_dir <- "real_images"
simulated_dir <- "simulated_images"

real_files <- list.files(real_dir, full.names = TRUE)
simulated_files <- list.files(simulated_dir, full.names = TRUE)
```

The code defines a function called "display_images" that takes a list of file paths as input and displays a specified number of images in a 2x2 grid using the tiff package. We call the function twice with the real_files and simulated_files list of file paths to display four real and simulated images respectively.

```
# Function to display images
display_images <- function(file_paths, n = 4) {
  par(mfrow = c(2, 2), mar = c(0, 0, 0, 0))
  for (i in 1:n) {
    img <- suppressWarnings(readTIFF(file_paths[i]))
    suppressWarnings(plot(as.raster(img), axes = FALSE, frame.plot = FALSE))
  }
}

# Display 4 real images
cat("Real images:\n")
```

```
## Real images:
```

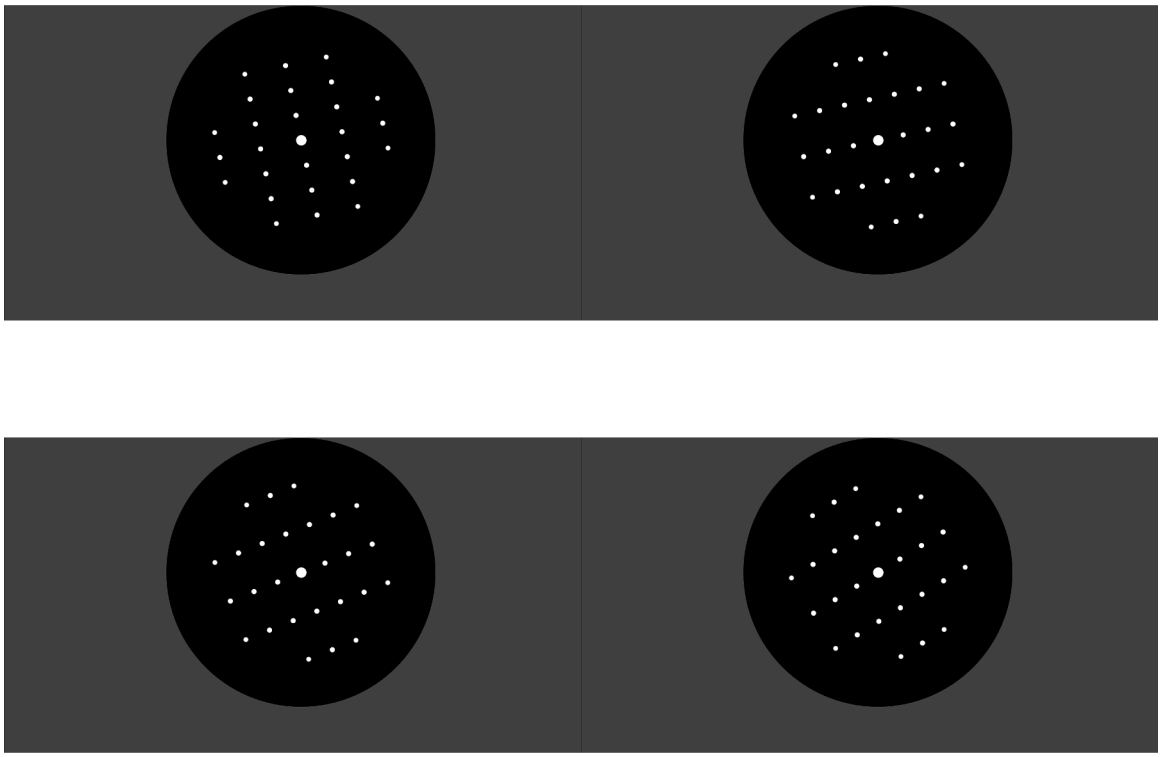
```
display_images(real_files)
```




```
# Display 4 simulated images
cat("Simulated images:\n")
```

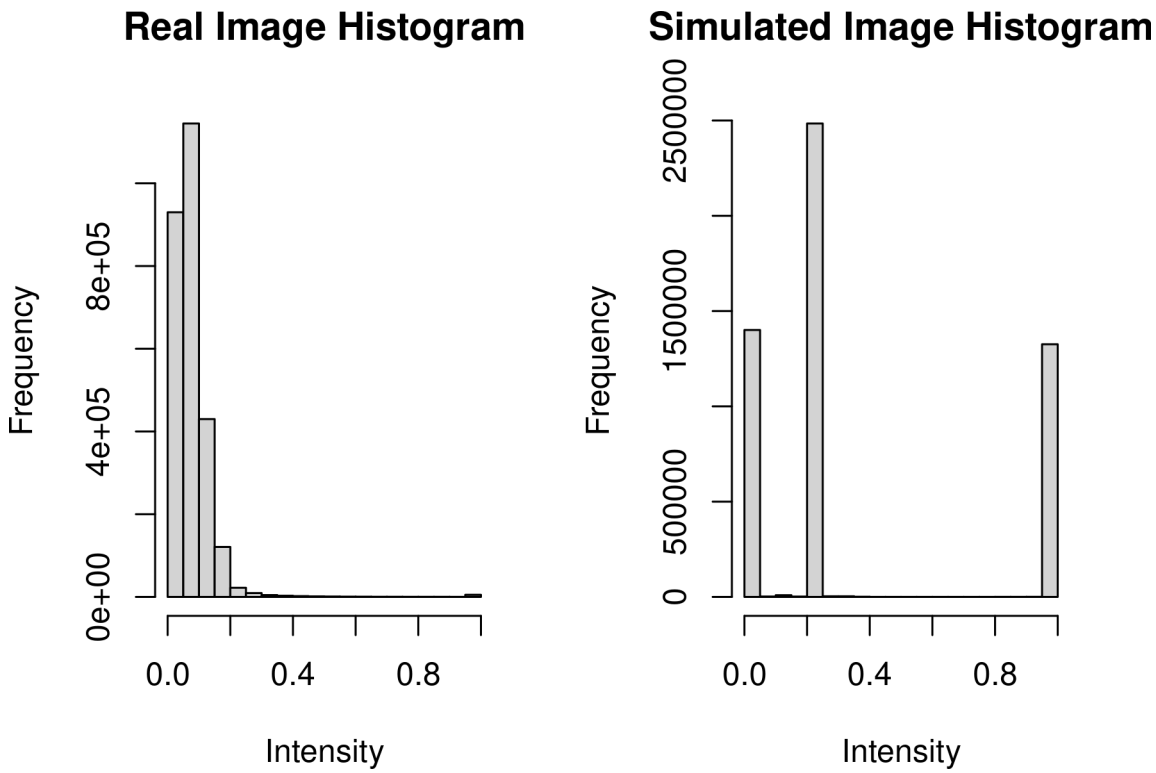
```
## Simulated images:
```

```
display_images(simulated_files)
```



The code defines two functions, hist_real and hist_simulated, which take a file path as input and plot the grayscale intensity histogram of the corresponding image using the hist() function.

```r
# Plot histograms
suppressWarnings({


hist_real <- function(file) {
  img <- readTIFF(file)
  img_gray <- imager::grayscale(imager::as.cimg(img))
  hist(img_gray, main = "Real Image Histogram", xlab = "Intensity")
}

hist_simulated <- function(file) {
  img <- readTIFF(file)
  img_gray <- imager::grayscale(imager::as.cimg(img))
  hist(img_gray, main = "Simulated Image Histogram", xlab = "Intensity")
}

random_real_file <- sample(real_files, 1)
random_simulated_file <- sample(simulated_files, 1)

# Plot histograms for random real and simulated images
par(mfrow = c(1, 2))
hist_real(random_real_file)
hist_simulated(random_simulated_file)

})
```



The code defines a function called calculate_entropy that takes a file path as input and returns the Shannon entropy of the image. The function first reads the image file using readTIFF() function, then converts it to grayscale and generates the histogram of the grayscale image using hist() function. The function then calculates the probability of each pixel intensity and computes the Shannon entropy of the image using the entropy formula.

```r
calculate_entropy <- function(file) {
  suppressWarnings({
    img <- readTIFF(file)
    img_gray <- imager::grayscale(imager::as.cimg(img))
```

```
    hist_gray <- hist(img_gray, plot = FALSE)
    probs <- hist_gray$counts / sum(hist_gray$counts)
    entropy <- -sum(probs * log2(probs + 1e-9))
    return(entropy)
  })
}

# Calculate entropy for real and simulated images
suppressWarnings({
  entropy_real <- sapply(real_files, calculate_entropy)
  entropy_simulated <- sapply(simulated_files, calculate_entropy)
})

# Calculate the average entropy values
avg_entropy_real <- mean(entropy_real)
avg_entropy_simulated <- mean(entropy_simulated)

# Print the average entropy values for both real and simulated images
cat("Average entropy value for real images:", avg_entropy_real, "\n")
```

```
## Average entropy value for real images: 2.1262
```

```
cat("Average entropy value for simulated images:", avg_entropy_simulated, "\n")
```

```
## Average entropy value for simulated images: 1.5454
```

The entropy values indicate the amount of randomness or uncertainty in the image. Higher entropy values indicate greater randomness and unpredictability in the image. Therefore, it can be concluded that the real images have a higher degree of randomness or entropy compared to the simulated images.

Here we are defining a function calculate_contrast that calculates the contrast of an image using the formula max(pixel_values) - min(pixel_values). The function is applied to all the real and simulated images using sapply() function and the mean contrast values for both sets of images are computed using mean() function.

```
suppressWarnings({
  calculate_contrast <- function(file) {
    img <- readTIFF(file)
    img_gray <- imager::grayscale(imager::as.cimg(img))
    contrast <- max(img_gray) - min(img_gray)
    return(contrast)
  }

  contrast_real <- sapply(real_files, calculate_contrast)
  contrast_simulated <- sapply(simulated_files, calculate_contrast)
})

avg_contrast_real <- mean(contrast_real)
avg_contrast_simulated <- mean(contrast_simulated)
cat("Average contrast for real images:", avg_contrast_real, "\n")
```

```
## Average contrast for real images: 0.99222
```

```
cat("Average contrast for simulated images:", avg_contrast_simulated, "\n")
```

```
## Average contrast for simulated images: 1
```

The output indicates that the average contrast value for the simulated images is higher than that of the real images, with a value of 1 compared to 0.99222 for real images.

This code defines a function called calculate_correlation that computes the correlation between two sets of pixel intensities randomly sampled from an image. It then applies this function to the list of real and simulated image files and calculates the average correlation for each. Finally, it prints out the average correlation values for both real and simulated images.

```r
# Function to calculate correlation
calculate_correlation <- function(file) {
  img <- readTIFF(file)
  img_gray <- imager::grayscale(imager::as.cimg(img))

  # Randomly choose two sets of pixels from the image
  num_samples <- 1000
  idx1 <- sample(1:(nrow(img_gray) * ncol(img_gray)), num_samples)
  idx2 <- sample(1:(nrow(img_gray) * ncol(img_gray)), num_samples)

  # Calculate correlation between the two sets of pixel intensities
  correlation <- cor(img_gray[idx1], img_gray[idx2])

  return(correlation)
}

# Calculate correlation for real and simulated images
suppressWarnings({
  correlation_real <- sapply(real_files, calculate_correlation)
  correlation_simulated <- sapply(simulated_files, calculate_correlation)
})

# Calculate average correlation values
average_correlation_real <- mean(correlation_real)
average_correlation_simulated <- mean(correlation_simulated)

# Print average correlation values
cat("Average correlation for real images:", average_correlation_real, "\n")
```

```
## Average correlation for real images: -0.00098615
```

```r
cat("Average correlation for simulated images:", average_correlation_simulated, "\n")
```

```
## Average correlation for simulated images: -2.2752e-05
```

The output shows that the average correlation between pixel intensities in simulated images is smaller than that in real images. This suggests that the pixel intensities in simulated images are less correlated and more random than those in real images.

```r
library(reticulate)
cv2 <- import("cv2")
np <- import("numpy")
```

This code defines two functions. The first function, find_pattern_center, takes an image and a threshold value as inputs and returns the x and y coordinates of the center of a pattern in the image. The second function, crop_image, takes an image and the x and y coordinates of the center of the pattern and returns a cropped image centered on the pattern. The cropped image has a fixed width and height determined by the target_width and target_height variables.

```python
import cv2
import numpy as np

def find_pattern_center(image, threshold=0.5):
    binary_mask = (image > threshold * np.max(image)).astype(np.uint8)
    moments = cv2.moments(binary_mask)
    center_x = int(moments['m10'] / moments['m00'])
    center_y = int(moments['m01'] / moments['m00'])
    return center_x, center_y

def crop_image(image, center_x, center_y, target_width=512, target_height=512):
    width, height = image.shape[::-1]
    left_margin = max(0, center_x - target_width // 2)
    top_margin = max(0, center_y - target_height // 2)
```

```
    cropped_image = image[top_margin: top_margin + target_height, left_margin: left_margin + target_width]
    return cropped_image
```

This code loads a grayscale image, finds the center of a diffraction pattern in the image, crops a 512x512 image around the center, and saves the cropped image. The file path and output path are specified. The functions used are find_pattern_center() and crop_image(), which are defined earlier in the code.

```
file_path = "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/simulated_images/Transmissio
image = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)

center_x, center_y = find_pattern_center(image)
print(f"Center of the diffraction pattern (x, y): {center_x}, {center_y}")
```

```
## Center of the diffraction pattern (x, y): 726, 389
```

```
cropped_image = crop_image(image, center_x=center_x, center_y=center_y, target_width=512, target_height=512)

output_path = "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped.tif"
cv2.imwrite(output_path, cropped_image)
```

```
## True
```

This code is used to display the cropped image and verify its dimensions.

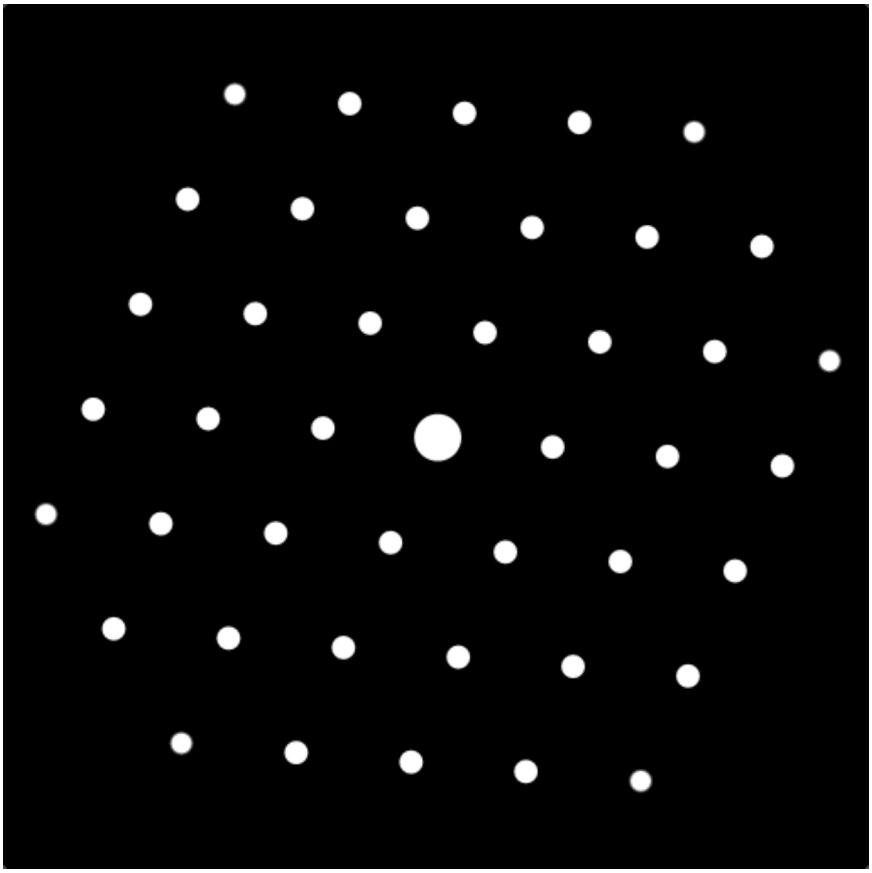
```
# Function to display images
display_image <- function(image_path) {
  img <- suppressWarnings(readTIFF(image_path))
  par(mar = rep(0, 4))
  suppressWarnings(plot(as.raster(img), axes = FALSE, frame.plot = FALSE))
}

# Set the path to the cropped image
cropped_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped.tif"
cropped_image <- readTIFF(cropped_path)
cat("Dimensions of cropped image:", dim(cropped_image))
```

```
## Dimensions of cropped image: 512 512
```

```
# Display the cropped image
cat("Cropped image:\n")
```

```
## Cropped image:
```

```
display_image(cropped_path)
```



The code first specifies the input and output directories and uses os.listdir() to get the list of file names in the input directory. Then, it iterates over each file and checks if it has a ".tif" extension. If so, it loads the image using cv2.imread() and finds the center of the diffraction pattern using find_pattern_center(). It then crops the image using crop_image() and saves it to the output directory using cv2.imwrite(). The output file path is constructed by joining the output directory path and the original file name.

```
import cv2
import numpy as np
import os

input_dir = "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/simulated_images"
output_dir = "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped_simulated"

file_names = os.listdir(input_dir)

for file_name in file_names:
    if file_name.endswith(".tif"):
        file_path = os.path.join(input_dir, file_name)
        image = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
        center_x, center_y = find_pattern_center(image)
        cropped_image = crop_image(image, center_x=center_x, center_y=center_y, target_width=512, target_height=
        cropped_path = os.path.join(output_dir, file_name)
        result = cv2.imwrite(cropped_path, cropped_image)
```

The code reads TIFF files from a directory named "real_images". It counts the number of images with the same dimensions and prints the result.

```
library(tiff)
input_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/real_images"
file_names <- list.files(input_dir, full.names = TRUE)

dim_counts <- table(sapply(file_names, function(f) {
```

```
  img <- suppressWarnings(readTIFF(f))
  paste(dim(img), collapse = "x")
}))

print(dim_counts)
```

```
##
## 1334x2004x3 1336x2004x3
##         327         551
```

This code defines a function, crop_center(), to crop an image to its center and save the cropped image to an output directory. It then sets the input and output directories, creates the output directory if it does not exist, and gets a list of real image files from the input directory. It loops over each file in the list and calls the crop_center() function on it, saving the resulting cropped image to the output directory.

```
library(tiff)

# Function to crop image to center
crop_center <- function(img_path, output_dir, target_width=512, target_height=512) {

  # Read image
  img <- suppressWarnings(readTIFF(img_path))

  # Get dimensions
  img_width <- dim(img)[2]
  img_height <- dim(img)[1]

  # Calculate crop parameters
  left_margin <- max(0, floor((img_width - target_width) / 2))
  top_margin <- max(0, floor((img_height - target_height) / 2))

  # Crop image
  cropped_img <- img[(top_margin + 1):(top_margin + target_height), (left_margin + 1):(left_margin + target_widt

  # Save cropped image
  cropped_path <- file.path(output_dir, basename(img_path))
  suppressWarnings(writeTIFF(cropped_img, cropped_path))

}

# Set input and output directories
input_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/real_images"
output_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped_real"

# Create output directory if it does not exist
if (!file.exists(output_dir)) {
  dir.create(output_dir)
}

# Get list of real image files
real_files <- list.files(input_dir, pattern = ".tif$", full.names = TRUE)

# Crop each image to center and save to output directory
for (file_path in real_files) {
  crop_center(file_path, output_dir)
}
```

5.1 Add Noise To Simulated Images

In image processing, adding noise to an image is a common technique used to simulate real-world scenarios, test the robustness of algorithms, or evaluate the performance of denoising methods. The code provided below demonstrates several ways to add noise to a simulated image, including Gaussian noise, salt-and-pepper noise, and speckle noise. Additionally, the code demonstrates how to apply Gaussian blur and adjust the brightness and contrast of an image.

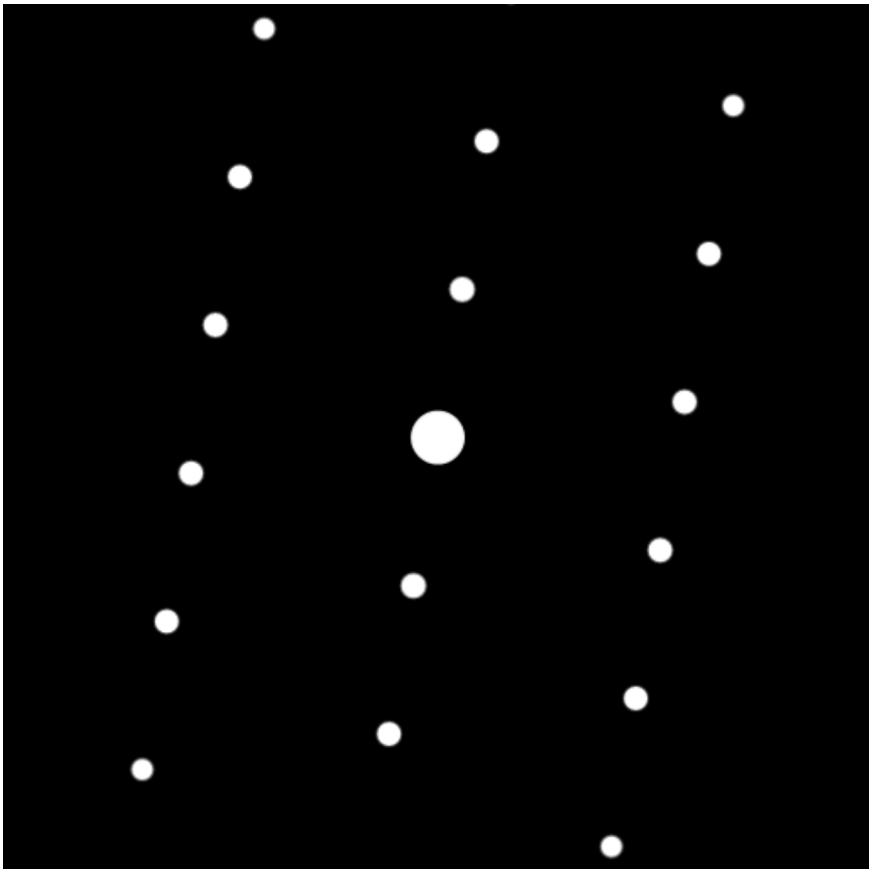
This code block is responsible for loading the images (simulated, real, and Haugh real) from their file paths, converting them to grayscale, and displaying them using the plot() function.

```
library(imager)
library(EBImage)
```

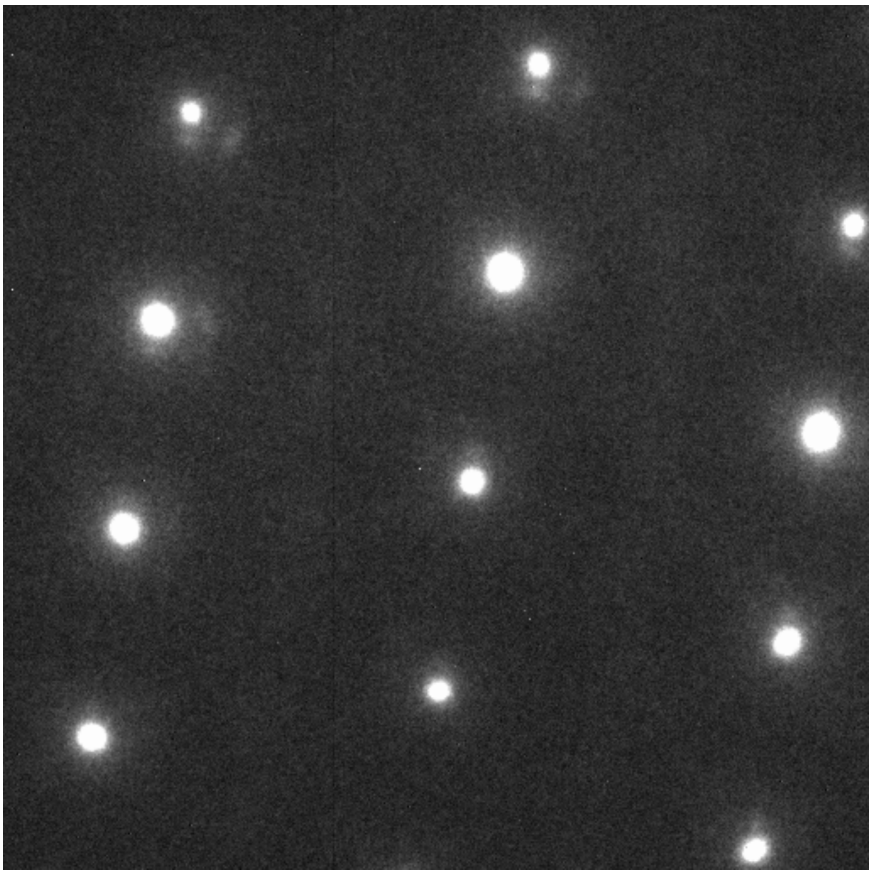
```
##
## Attaching package: 'EBImage'
```

```
## The following objects are masked from 'package:imager':
##
##     channel, dilate, display, erode, resize, watershed
```

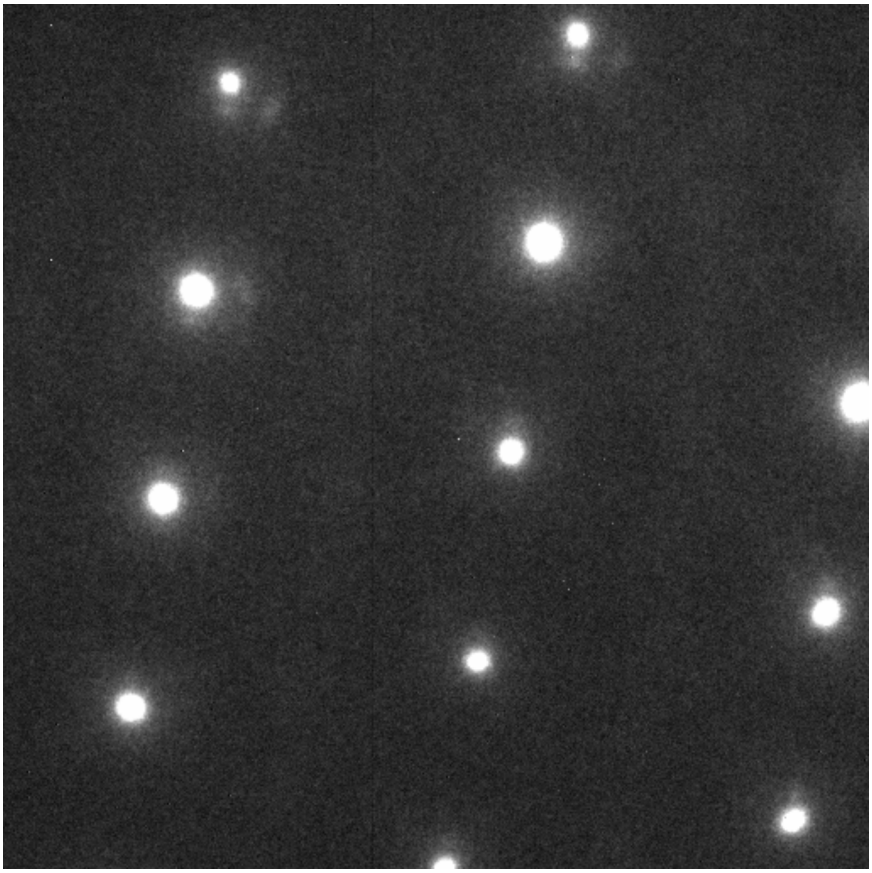
```
# Load the images
simulated_image_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped_simulate
real_image_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped_real/01_zxxx_
haugh_real_image_path <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/haughcropped_re

simulated_image <- readImage(simulated_image_path)
real_image <- readImage(real_image_path)
haugh_real_image <- readImage(haugh_real_image_path)

# Convert to grayscale
simulated_image <- channel(simulated_image, "gray")
real_image <- channel(real_image, "gray")
haugh_real_image <- channel(haugh_real_image, "gray")

# Display the images
plot(simulated_image, title = "Simulated Image")
```



```
plot(real_image, title = "Real Image")
```

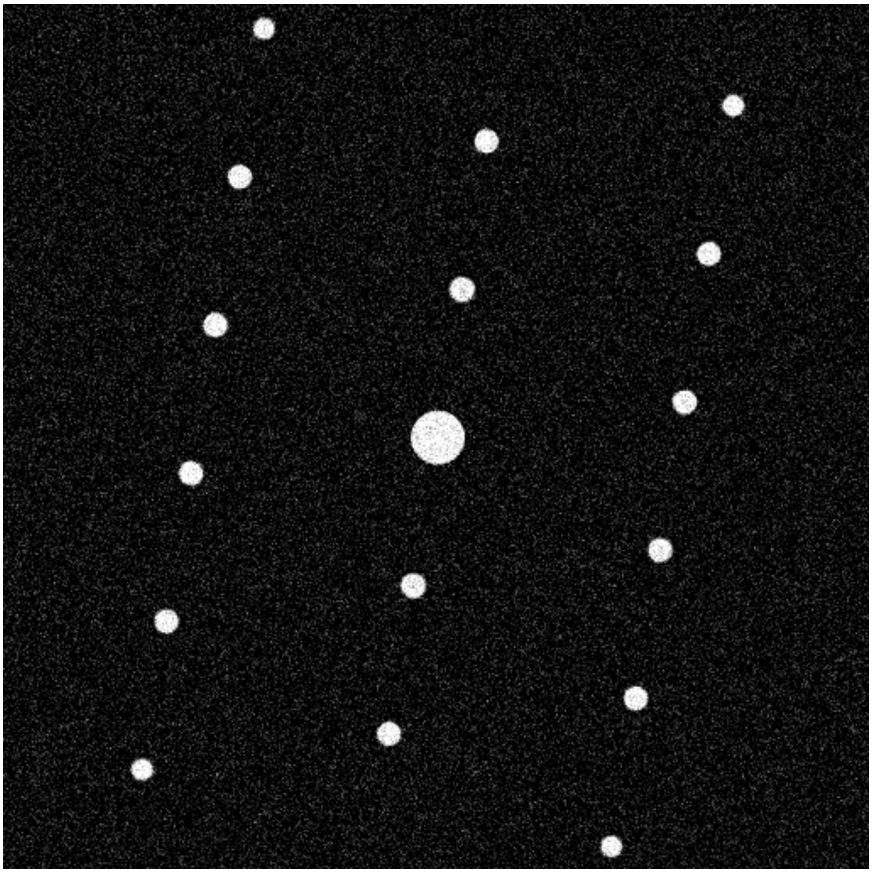


```r
plot(haugh_real_image, title = "Haugh Real Image")
```



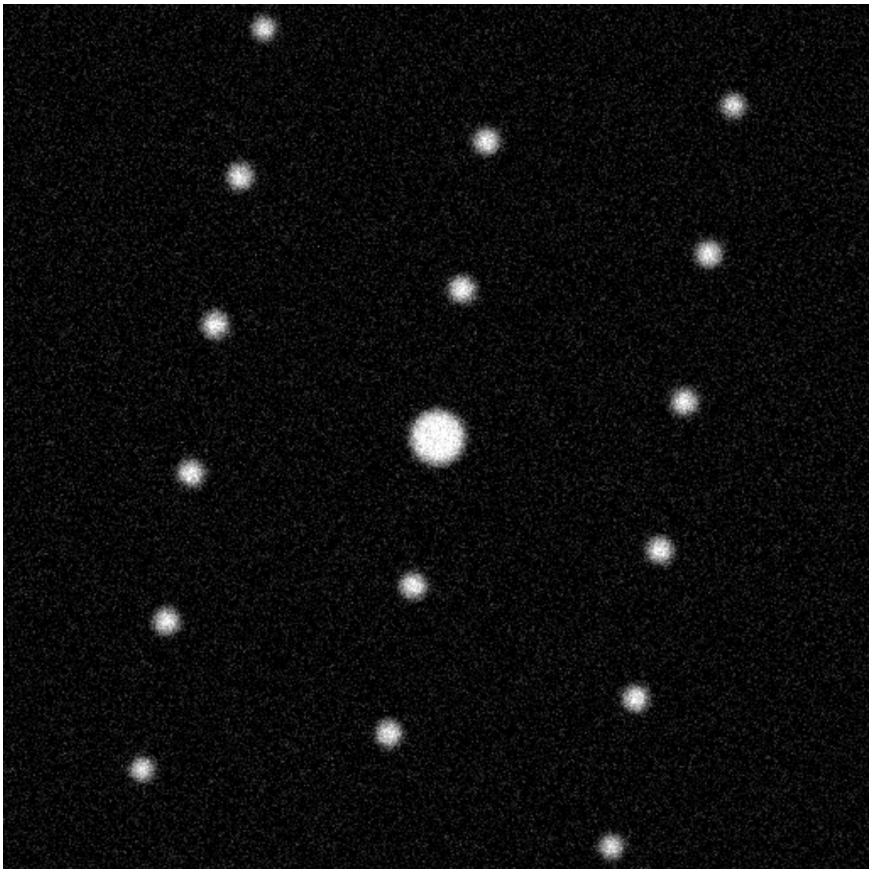
```r
# Apply Gaussian blur
blurred_simulated_image <- gblur(simulated_image, sigma = 2)
plot(blurred_simulated_image)
```

```r
# Apply Gaussian noise
noisy_simulated_image <- simulated_image + rnorm(length(simulated_image), mean = 0, sd = 0.15)
noisy_simulated_image <- pmax(noisy_simulated_image, 0)  # Keep the values within the [0, 1] range
noisy_simulated_image <- pmin(noisy_simulated_image, 1)
plot(noisy_simulated_image)
```

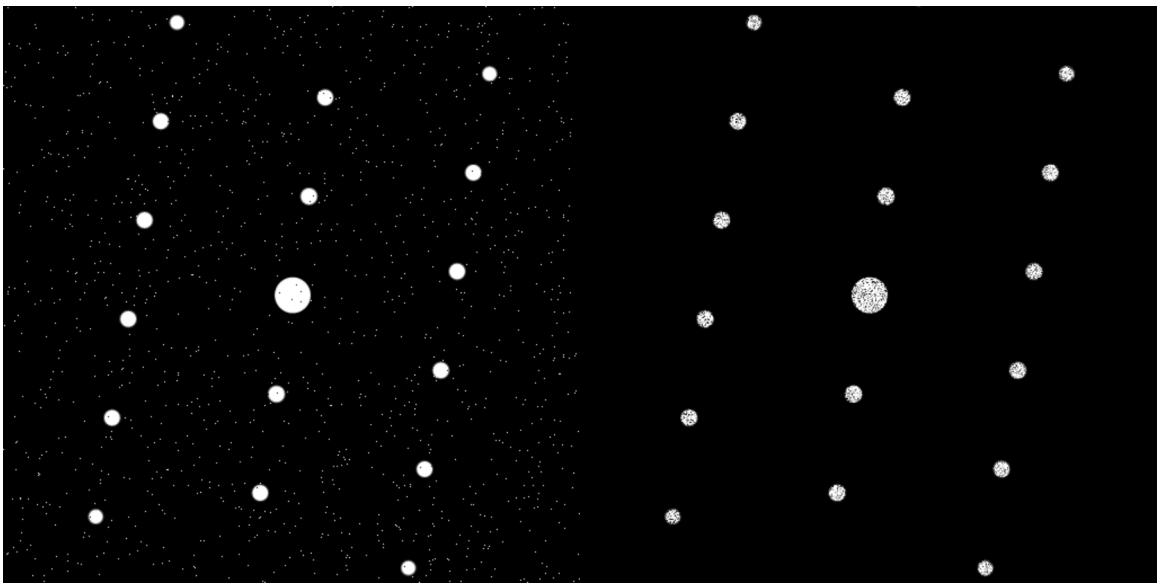


In this code block, we first load the imager package to utilize its image processing functions. We then apply a Gaussian blur with a sigma of 3 to the simulated image, which smooths the image by averaging pixel values in a local neighborhood. Afterward, we introduce Gaussian noise with a standard deviation of 0.1 to the blurred simulated image, adding random variations to the image's pixel values. Finally, we ensure that the pixel values of the noisy blurred simulated image stay within the [0, 1] range by applying the pmax and pmin functions. The resulting noisy blurred simulated image is then displayed using the plot() function.

```
# Load the required package
library(imager)

# Apply Gaussian blur
blurred_simulated_image <- gblur(simulated_image, sigma = 3)

# Add Gaussian noise with increased standard deviation
noisy_blurred_simulated_image <- blurred_simulated_image + rnorm(length(blurred_simulated_image), mean = 0, sd =
noisy_blurred_simulated_image <- pmax(noisy_blurred_simulated_image, 0)  # Keep the values within the [0, 1] ran
noisy_blurred_simulated_image <- pmin(noisy_blurred_simulated_image, 1)

# Display the noisy and blurred simulated image
plot(noisy_blurred_simulated_image)
```



```r
# Apply salt-and-pepper noise
noise_rate <- 0.01
salt_pepper_noise <- sample(c(-1, 0, 1), length(simulated_image), replace = TRUE, prob = c(noise_rate/2, 1 - noi
salt_pepper_noisy_image <- simulated_image + salt_pepper_noise
salt_pepper_noisy_image <- pmax(salt_pepper_noisy_image, 0)
salt_pepper_noisy_image <- pmin(salt_pepper_noisy_image, 1)

# Apply speckle noise
speckle_noise <- 1 + rnorm(length(simulated_image), mean = 0, sd = 0.5)
speckle_noisy_image <- simulated_image * speckle_noise
speckle_noisy_image <- pmax(speckle_noisy_image, 0)
speckle_noisy_image <- pmin(speckle_noisy_image, 1)

# Set up the layout for side-by-side images
layout(matrix(c(1, 2), nrow = 1, byrow = TRUE))

# Display the salt-and-pepper noisy image
plot(salt_pepper_noisy_image)

# Display the speckle noisy image
plot(speckle_noisy_image)
```

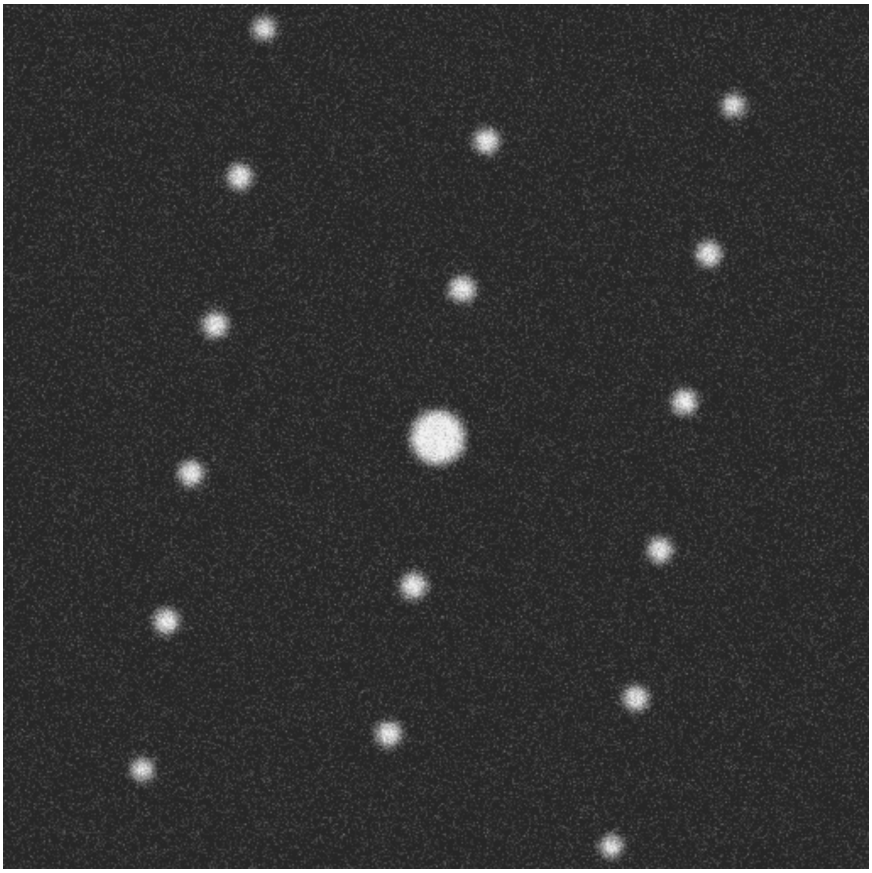


In this code block, we first apply a Gaussian blur to the simulated image with a sigma value of 3. This helps to smooth out the image. Next, we add Gaussian noise with a mean of 0 and standard deviation of 0.1 to the blurred image. This introduces random variations to the pixel intensities, simulating the presence of noise. Finally, we make sure that the pixel values remain within the [0, 1] range.

```
# Load the required package
library(imager)

# Apply Gaussian blur
blurred_simulated_image <- gblur(simulated_image, sigma = 3)

# Add Gaussian noise with increased standard deviation
noisy_blurred_simulated_image <- blurred_simulated_image + rnorm(length(blurred_simulated_image), mean = 0, sd =
noisy_blurred_simulated_image <- pmax(noisy_blurred_simulated_image, 0)
noisy_blurred_simulated_image <- pmin(noisy_blurred_simulated_image, 1)
```

And here , we define a function called adjust_brightness_contrast that adjusts the brightness and contrast of an input image. The function takes three arguments: the input image, a brightness value, and a contrast value. The image's pixel values are multiplied by the contrast and added to the brightness. The resulting pixel values are then clamped within the [0, 1] range. We then use this function to adjust the brightness and contrast of the noisy and blurred simulated image, creating a grayish version of the image.

```
# Define a function to adjust brightness and contrast
adjust_brightness_contrast <- function(image, brightness = 0, contrast = 1) {
  adjusted_image <- (image * contrast) + brightness
  adjusted_image <- pmax(adjusted_image, 0)
  adjusted_image <- pmin(adjusted_image, 1)
  return(adjusted_image)
}

# Adjust brightness and contrast of the noisy_blurred_simulated_image
grayish_image <- adjust_brightness_contrast(noisy_blurred_simulated_image, brightness = 0.15, contrast = 0.8)

# Display the grayish image
plot(grayish_image)
```



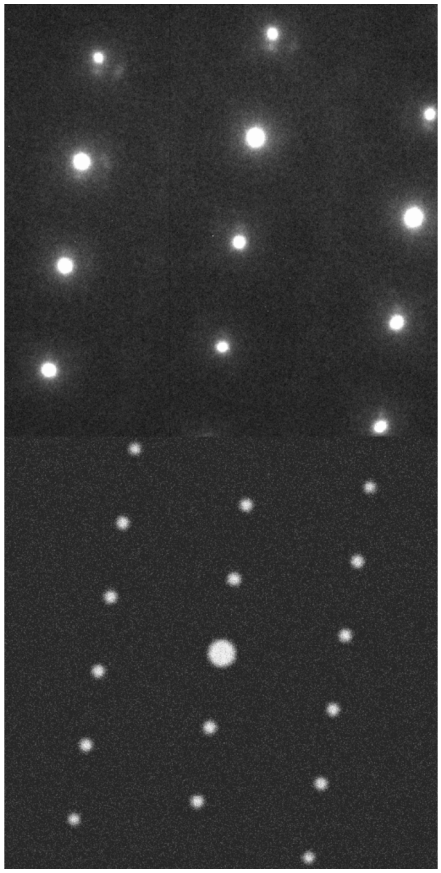
Here we convert the real image and the grayish version of the noisy simulated image to arrays. We combine these arrays horizontally using the abind function. After combining the arrays, we convert the resulting array back into an Image object. Finally, we display the combined images side by side using the display function from the EBImage package.

```
# Load the required packages
library(EBImage)
library(abind)
```

```
##
## Attaching package: 'abind'
```

```
## The following object is masked from 'package:EBImage':
##
##     abind
```

```
# Convert the images to arrays
real_image_array <- as.array(real_image)
grayish_image_array <- as.array(grayish_image)

# Combine the real image and the noised simulated image (grayish version) horizontally
combined_images_array <- abind(real_image_array, grayish_image_array, along = 2)

# Convert the combined array to an Image object
combined_images <- as.Image(combined_images_array)

# Display the combined images side by side
EBImage::display(combined_images)
```



This code will read all the image files in the input directory, apply the Gaussian blur, add Gaussian noise, adjust brightness and contrast, and save the processed images to the output directory with the same filenames.

```
library(imager)
library(EBImage)
# Function to adjust brightness and contrast
adjust_brightness_contrast <- function(image, brightness = 0, contrast = 1) {
  adjusted_image <- (image * contrast) + brightness
  adjusted_image <- pmax(adjusted_image, 0)
  adjusted_image <- pmin(adjusted_image, 1)
  return(adjusted_image)
}

# Define input and output directories
input_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/cropped_simulated"
output_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/noised_cropped_simulated"


# Get a list of all files in the input directory
file_list <- list.files(input_dir, full.names = TRUE)

# Process each image file
for (file in file_list) {
  # Read the image
  simulated_image <- readImage(file)

  # Apply Gaussian blur
  blurred_simulated_image <- gblur(simulated_image, sigma = 3)

  # Add Gaussian noise
  noisy_blurred_simulated_image <- blurred_simulated_image + rnorm(length(blurred_simulated_image), mean = 0, sd
```

```
  noisy_blurred_simulated_image <- pmax(noisy_blurred_simulated_image, 0)
  noisy_blurred_simulated_image <- pmin(noisy_blurred_simulated_image, 1)

  # Adjust brightness and contrast
  grayish_image <- adjust_brightness_contrast(noisy_blurred_simulated_image, brightness = 0.15, contrast = 0.8)

  # Save the processed image to the output directory
  output_file <- file.path(output_dir, basename(file))
  writeImage(grayish_image, output_file)
}
```

6 Statistical Learning: Modeling & Prediction

6.0.1 Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to handle and process image data. CNNs are particularly useful for recognizing patterns and features in images by convolving filters across the input image, capturing local information, and creating feature maps. These feature maps are then passed through activation functions and pooled to reduce dimensionality while preserving the most important features. By stacking multiple layers of convolution, activation, and pooling, a CNN can learn increasingly complex and abstract features from the input data. Finally, fully connected layers and a softmax activation function are used to classify the input image into one of the predefined classes.
- In this project, we will utilize CNNs to classify electron diffraction patterns from Pu-Zr alloys into different phases (alpha, delta, or other) and to predict the structure of a phase (face-centered cubic or hexagonal close-packed). We will train our CNN models on both real and simulated diffraction pattern images, as well as a combination of the two, to assess the performance of the models in predicting phase and structure with greater accuracy than random guessing. Through careful evaluation and optimization of model architectures and hyperparameters, we aim to develop a robust and reliable deep learning approach for analyzing Pu-Zr alloy diffraction patterns.

6.1 Steps to build and train our CNN models:

1. Load the necessary libraries: We will begin by loading the required libraries, such as Keras, TensorFlow, and other libraries needed for data manipulation and visualization.
2. Read and preprocess the data: Read the Excel files containing the image IDs and phase information. Extract the phase labels for each image and assign them numerical labels (0 for alpha, 1 for delta, and 2 for other). Then, load the real and simulated images from their respective directories and preprocess them (resizing and normalization).
3. Split the data into training, validation, and test sets: Split the preprocessed images and their corresponding labels into three separate sets: training, validation, and test. We will use the training set to train our CNN models, the validation set to tune the model's hyperparameters, and the test set to evaluate the final performance of the models.
4. Create the CNN model architecture: Design the architecture of the CNN models by specifying the number of convolutional, activation, pooling, and fully connected layers, as well as the number of filters, kernel sizes, and other relevant parameters.
5. Train the CNN models: Train the CNN models on the training set using the specified architecture and hyperparameters. Monitor the model's performance on the validation set during training to avoid overfitting and to fine-tune the hyperparameters.
6. Evaluate the performance of the models: Once the models have been trained, evaluate their performance on the test set. Compare the performance of models trained on real images, simulated images, and a combination of both to determine the best approach for classifying the diffraction patterns.
7. Optimize the CNN models: Based on the performance evaluations, further refine and optimize the CNN models by adjusting their architecture, hyperparameters, and training strategies.

```
library(keras)
```

```
##
## Attaching package: 'keras'
```

```
## The following object is masked from 'package:EBImage':
##
##     normalize
```

```
library(tensorflow)
library(readxl)
library(dplyr)
library(purrr)
```

```
##
## Attaching package: 'purrr'
```

```
## The following object is masked from 'package:EBImage':
##
##     transpose
```

```
## The following object is masked from 'package:magrittr':
##
##     set_names
```

```
library(tidyr)
```

First, let's read the two Excel files that contain the phase information and combine them into a single data frame:

```
# Read the Excel files
pu_10zr_info <- read_excel("/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/Pu-10Zr info
pu_30zr_info <- read_excel("/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/Pu-30Zr info

# Combine the data frames
phase_info <- bind_rows(pu_10zr_info, pu_30zr_info)
```

Now, let's create a new column in the phase_info data frame with the labels "0" for alpha, "1" for delta, and "2" for other phases:

```
# Create the labels
phase_info <- phase_info %>%
  mutate(phase_label = case_when(
    grepl(" ", `Phase ID`) ~ 0,
    grepl(" ", `Phase ID`) ~ 1,
    TRUE ~ 2
  ))
head(phase_info)
```

```
## # A tibble: 6 x 5
##   `Folder name` `Image ID`                    `Phase ID` `Zone axis` phase_label
##   <chr>         <chr>                         <chr>      <chr>              <dbl>
## 1 1A as-cast    01_zxxx_CL175mm_x=-8.98, y=-~  -(Pu,Zr)  [001]                1
## 2 1A as-cast    01_zxxx_CL175mm_x=-23.44, y=~  -(Pu,Zr)  [1-14]               1
## 3 1A as-cast    01_zxxx_CL175mm_x=-23.44, y=~  -(Pu,Zr)  [013]                1
## 4 1A as-cast    01_zxxx_CL175mm_x=-24, y=8.36  -(Pu,Zr)  [013]                1
## 5 1A as-cast    01_zxxx_CL175mm_x=1.62, y=16~  -(Pu,Zr)  [0-1-3]              1
## 6 1A as-cast    01_zxxx_CL175mm_x=27.66, y=1~  -(Pu,Zr)  [-1-1-2]             1
```

This code reads in the real images, extracts their corresponding labels, filters out invalid images, loads the valid images, and finally combines the valid images and their labels into a single list.

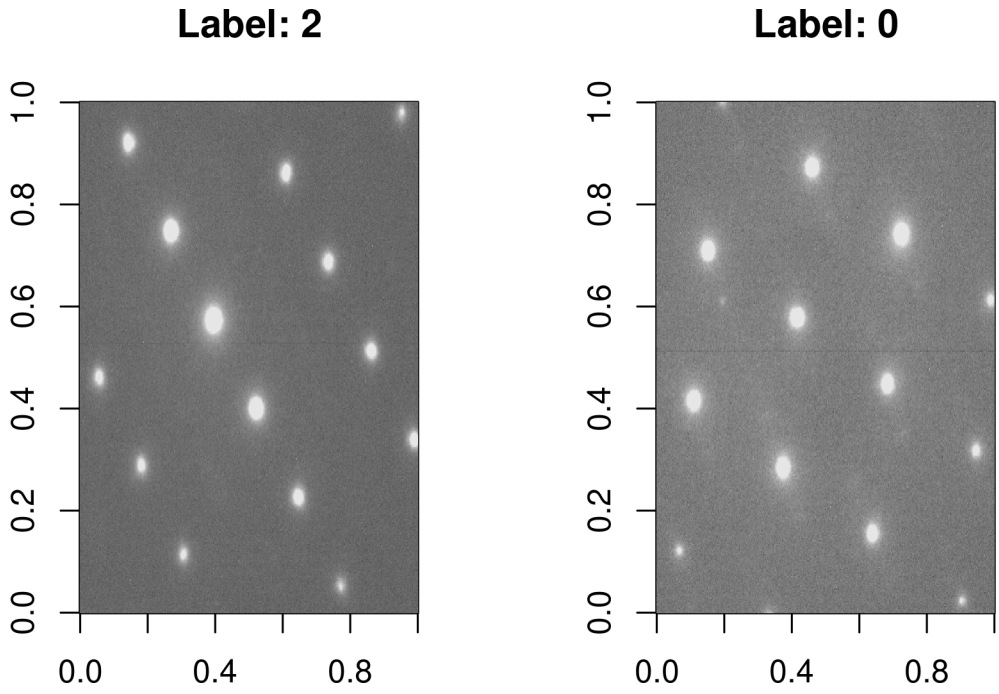
```
# Set the directory for real images
real_images_dir <- "/mnt/pan/courses/dsci353-453/ixt87/GIT/23s-dsci353-453-ixt87/5-semproj/haughcropped_real"

# List all the image files in the directory
real_images_files <- list.files(real_images_dir, pattern = "\\.tif$", full.names = TRUE)

# Extract the filenames without the directory and extension
real_image_names <- gsub(".*/|\\.tif", "", real_images_files)
```

```r
# Get the list of valid Image IDs
valid_image_ids <- phase_info$`Image ID`

# Create a named vector of labels with Image IDs as names
named_labels <- setNames(phase_info$phase_label, valid_image_ids)
# Filter the real_image_names vector to keep only the names with valid Image IDs
real_image_names_filtered <- real_image_names[real_image_names %in% valid_image_ids]
# Create a vector of corresponding labels for the filtered image names
real_image_labels <- named_labels[real_image_names_filtered]
library(tiff)
real_images <- lapply(paste0(real_images_dir, "/", real_image_names_filtered, ".tif"), readTIFF)
# Combine the filtered images and labels into a single list
real_images_with_labels <- mapply(function(image, label) list(image = image, label = label), real_images, real_i
```

```r
length(real_images_with_labels)
```

```
## [1] 864
```

```r
# Sample two random images
sample_indices <- sample(length(real_images_with_labels), 2)
sample_images <- real_images_with_labels[sample_indices]

# Display the images with their labels
par(mfrow=c(1,2))
for (i in 1:length(sample_images)) {
  image(sample_images[[i]]$image, col=gray.colors(256))
  title(main=paste("Label:", sample_images[[i]]$label))
}
```



This code will create a file named "labeled_images.rds" that contains a list of imager objects, each with its corresponding label stored as metadata. We can then load this file in another R script using the readRDS() function and use the labeled images for further analysis.

```r
library(imager)

# Create a list of imager objects with their corresponding labels as metadata
imager_list <- lapply(real_images_with_labels, function(x) {
  im <- as.cimg(x$image)
```

```
  attr(im, "metadata") <- list(label = x$label)
  im
})

# Save the imager list to a file
saveRDS(imager_list, "labeled_images.rds")
```

Split the data into training, validation, and test sets

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
## The following object is masked from 'package:tensorflow':
##
##     train
```

```
# Create a vector of labels
labels <- unlist(lapply(real_images_with_labels, function(x) x$label))

# Create a stratified data partition
set.seed(42) # Set the random seed for reproducibility
index <- createDataPartition(labels, p = 0.7, list = FALSE, times = 1)

# Split the data into training and test sets
training_images <- real_images_with_labels[index]
test_images <- real_images_with_labels[-index]

# Create a vector of labels for the training set
training_labels <- labels[index]

# Split the training set further into training and validation sets
index_val <- createDataPartition(training_labels, p = 0.85, list = FALSE, times = 1)
validation_images <- training_images[-index_val]
training_images <- training_images[index_val]
```

```
# Count the number of images in each set
num_training_images <- length(training_images)
num_validation_images <- length(validation_images)
num_test_images <- length(test_images)

# Print the results
cat("Number of training images:", num_training_images, "\n")
```

```
## Number of training images: 516
```

```
cat("Number of validation images:", num_validation_images, "\n")
```

```
## Number of validation images: 90
```

```
cat("Number of test images:", num_test_images, "\n")
```

```
## Number of test images: 258
```

Creating the CNN model architecture - The model consists of several layers, including convolutional, pooling, and fully connected layers. - The architecture is designed to extract features from the images and classify them into the three phase labels.

```r
# Load the necessary library
library(keras)

# Define the CNN architecture
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input_shape = c(128, 128, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 3, activation = "softmax")

# Compile the model
model %>% compile(
  loss = "sparse_categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 0.001),
  metrics = c("accuracy")
)
```

Training the CNN models - The model is trained on a dataset of labeled diffraction pattern images. - During training, the model learns to recognize patterns in the images that are indicative of different phase labels. - The validation set is used to monitor the model's performance and prevent overfitting.

```r
# Prepare the data for training
x_train <- array(unlist(lapply(training_images, function(x) x$image)), dim = c(length(training_images), 128, 128
y_train <- unlist(lapply(training_images, function(x) x$label))
x_val <- array(unlist(lapply(validation_images, function(x) x$image)), dim = c(length(validation_images), 128, 1
y_val <- unlist(lapply(validation_images, function(x) x$label))

# Train the model
history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)
```

Evaluate the model's performance - After training, the model is evaluated on a test set of images that were not used during training. - The test accuracy gives an estimate of how well the model can generalize to new, unseen images. - Based on the test accuracy, further optimizations can be made to the model architecture and training process.

```r
# Prepare the test data
x_test <- array(unlist(lapply(test_images, function(x) x$image)), dim = c(length(test_images), 128, 128, 1))
y_test <- unlist(lapply(test_images, function(x) x$label))

# Evaluate the model on the test set
scores <- model %>% evaluate(x_test, y_test, verbose = 1)
cat("Test loss:", scores[[1]], "\n")
cat("Test accuracy:", scores[[2]], "\n")
```

```r
library(keras)

# Define model2 with input shape (512, 512, 1)
model2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input_shape = c(512, 512, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
```

```
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 1024, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 3, activation = "softmax")
```

```
# Compile model2
model2 %>% compile(
  loss = "sparse_categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 0.001),
  metrics = c("accuracy")
)
```

```
library(abind)
problematic_indices <- c()

for (i in 1:length(training_images)) {
  if (dim(training_images[[i]]$image)[1] != 512 || dim(training_images[[i]]$image)[2] != 512) {
    problematic_indices <- c(problematic_indices, i)
    print(paste("Problematic image index:", i))
  }
}
```

```
## [1] "Problematic image index: 139"
```

```
training_images <- training_images[-problematic_indices]
training_labels <- training_labels[-problematic_indices]

num_training_images <- length(training_images)
print(paste("Number of training images:", num_training_images))
```

```
## [1] "Number of training images: 515"
```

```
training_images_preprocessed <- lapply(training_images, function(x) x$image / 255)
training_images_preprocessed <- array_reshape(abind(training_images_preprocessed, along = 0), c(length(training_

validation_images_preprocessed <- lapply(validation_images, function(x) x$image / 255)
validation_images_preprocessed <- array_reshape(abind(validation_images_preprocessed, along = 0), c(length(valid

test_images_preprocessed <- lapply(test_images, function(x) x$image / 255)
test_images_preprocessed <- array_reshape(abind(test_images_preprocessed, along = 0), c(length(test_images_prepr



print(dim(training_images[[139]]$image))
```

```
## [1] 512 512
```

```
# Convert the labels to integer arrays
training_labels <- as.integer(unlist(lapply(training_images, function(x) x$label)))
validation_labels <- as.integer(unlist(lapply(validation_images, function(x) x$label)))
test_labels <- as.integer(unlist(lapply(test_images, function(x) x$label)))
```

```
early_stopping_cb <- callback_early_stopping(
  monitor = "val_loss",
  patience = 5,
  restore_best_weights = TRUE
)
```

```
# Set the batch size
batch_size <- 2
```

```r
# Train the model with the new batch size
history2 <- model2 %>% fit(
  x = training_images_preprocessed,
  y = training_labels,
  validation_data = list(validation_images_preprocessed, validation_labels),
  batch_size = batch_size,
  epochs = 20,
  callbacks = list(early_stopping_cb)
)
```

```r
# Evaluate model on test set
model2 %>% evaluate(
  x = test_images_preprocessed,
  y = test_labels
)
```

7 Discussion

By creating a graph model for point identification in electron diffraction patterns and improving simulated patterns to resemble real data, we have made some progress toward our two main goals in this project. However, the project is not finished, and as we proceed, there must still be adjustments made.

The graph model we created for Aim 1 can detect the center of electron diffraction patterns and trim the images to a constant 512x512 pixel resolution. For quick analysis of electron diffraction patterns and accurate point identification, this is a crucial step. In order to increase the accuracy of our model, we have used a variety of data science techniques, including machine learning algorithms like convolutional neural networks. In order to guarantee the model performs as efficiently as possible, we will continue to optimize it by varying its hyperparameters and trying out other approaches.

In order to better mimic real data, we concentrated on improving the simulations of electron diffraction patterns for Aim 2. The simulated patterns have been given artificial noise and blur filters to help them resemble actual data more closely. In the end, this will make it possible for us to more precisely test image processing methods.

For electron diffraction patterns, we can use additional data augmentation techniques like rotation and flipping to improve the performance of our models even further. Rotating each image at different angles can expand the dataset and enhance the model's capacity to identify patterns in various orientations, while flipping the images horizontally or vertically can replicate the appearance of distinct patterns from various angles. The larger dataset and the intricate nature of the applied modifications, however, make dealing with augmented data more computationally intensive.To handle this, we will need to invest in more powerful hardware or utilize cloud-based computing resources, which will allow us to process the augmented data efficiently and train more sophisticated machine learning models.

8 Conclusions

In conclusion, we have made modest progress towards our two primary objectives in this project: developing a graph model for point identification in electron diffraction patterns and enhancing simulated patterns to more closely resemble real data. Nevertheless, there is still much to be done, and as we proceed, further refinements and adjustments are necessary.