Improving Image Processing for Electron Diffraction Pattern Datasets

Enhance Simulated Patterns to Mimic Real Data

Ibrahim Uruc Tarim

30 April, 2023

Contents

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.

2.2 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.

2.3 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.

2.4 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, 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.

3 Exploratory Data Analysis

3.1 The Data

- 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 $15\circ$ 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. Here, we are importing required libraries and defining the file paths for the real and simulated images.

4 Aim 2: Enhance Simulated Patterns to Mimic Real Data

4.1 Data Cleaning

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.

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.

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

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

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

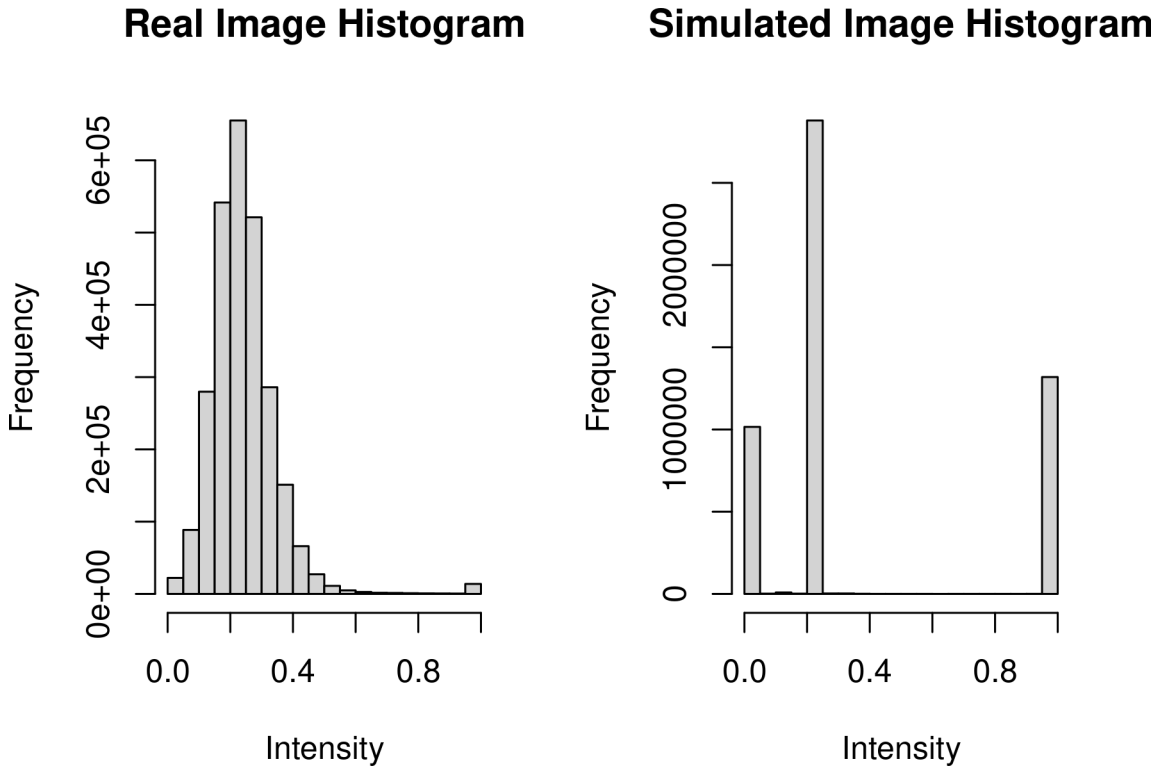
```
## Mean: 0.37558 Median: 0.24706 SD: 0.37657
```

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.

4.1.1 Visualizations for Exploratory Analysis

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.

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.



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.

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.

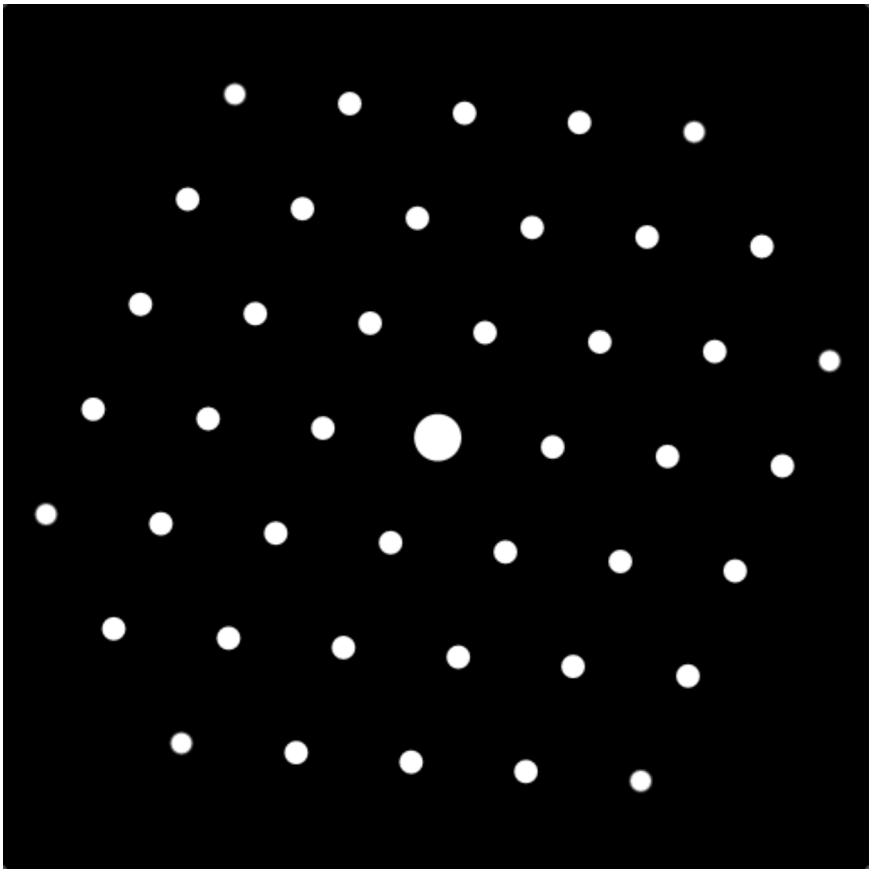
```
## Center of the diffraction pattern (x, y): 726, 389
```

```
## True
```

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

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

```
## Cropped image:
```



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.

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.

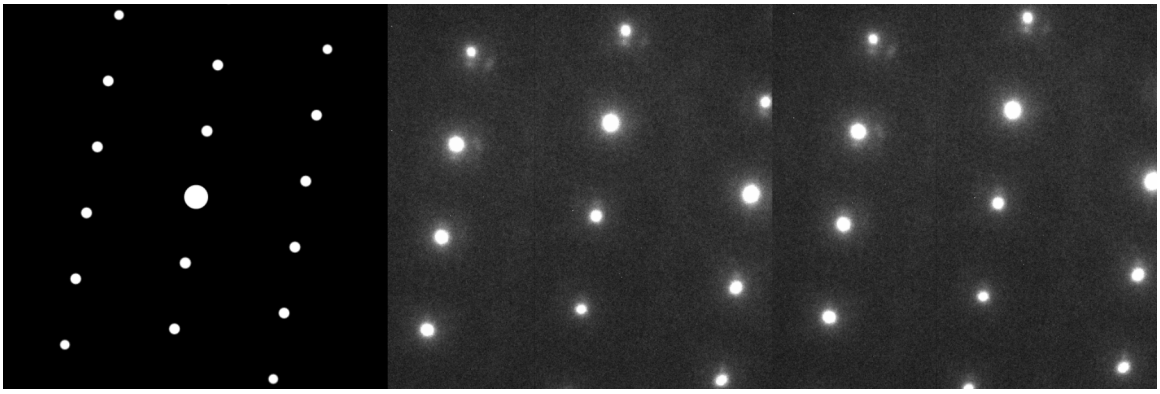
```
##
## 1334x2004x3 1336x2004x3
##         327         551
```

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.

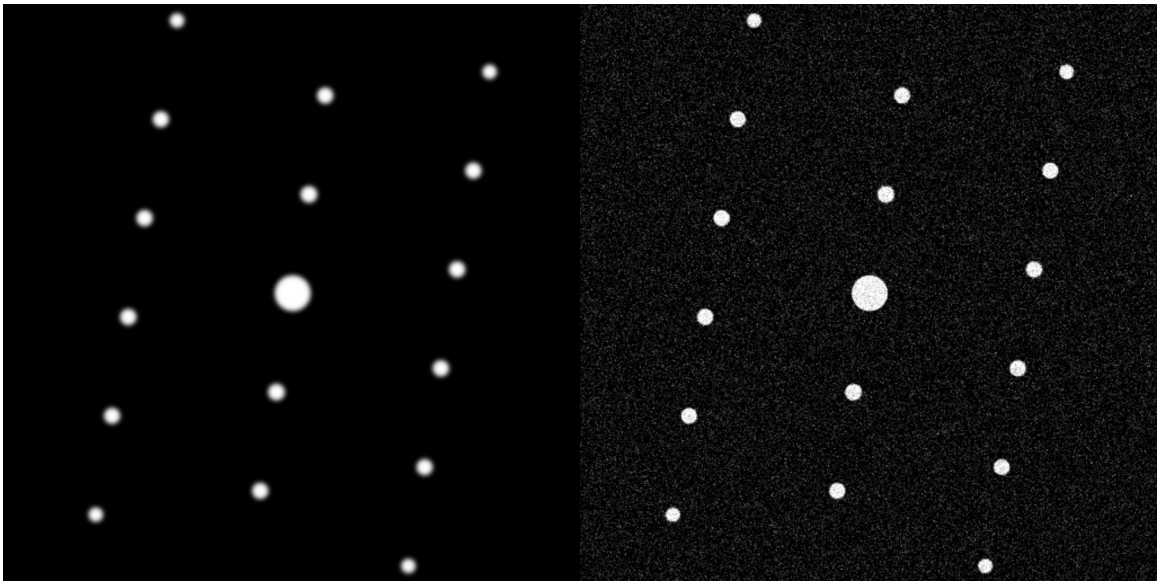
4.2 Adding Noise To Simulated Images

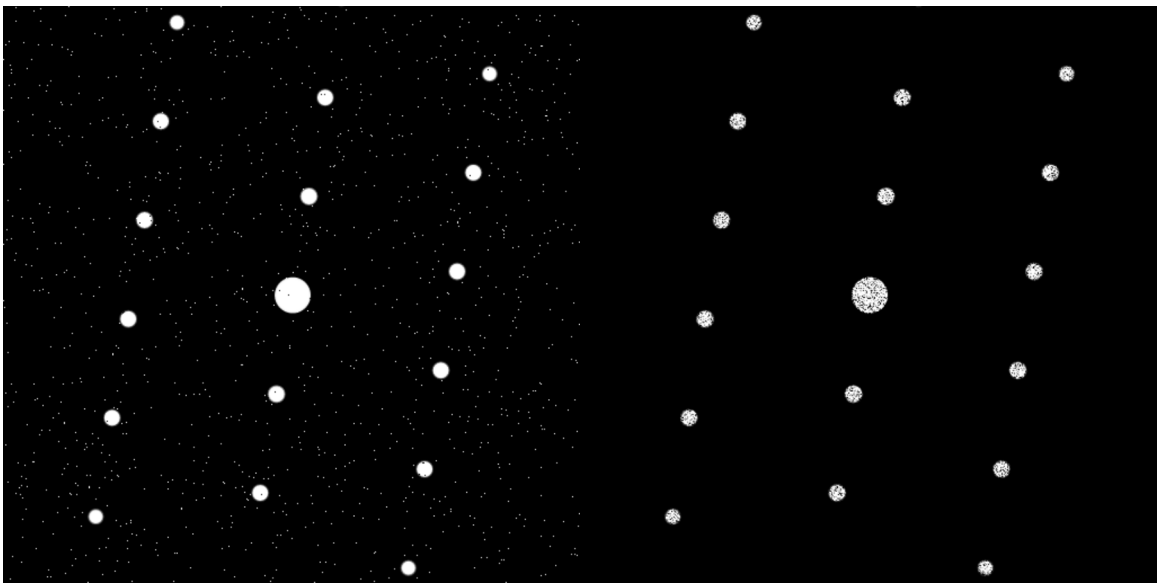
In image processing, adding noise to an image is a common technique used to simulate real-world scenarios, test the robustness of algorithms, or evaluate the performance of denoising methods. The code provided below demonstrates several ways to add noise to a simulated image, including Gaussian noise, 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.



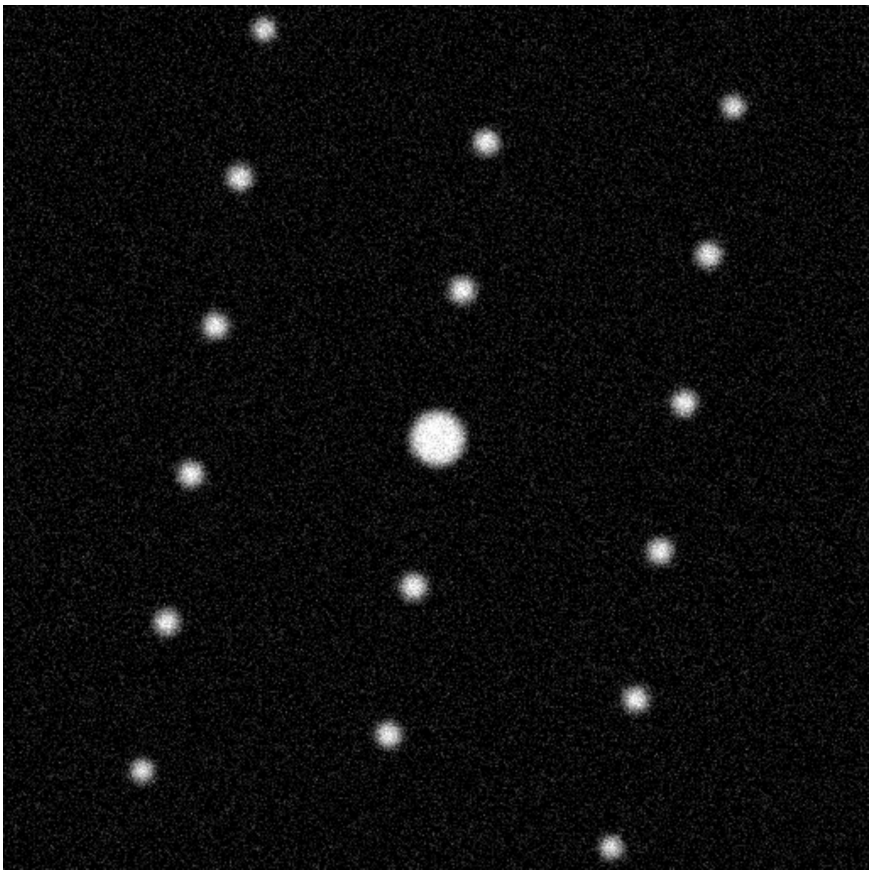
In this section, we will explore various image noising techniques to make simulated diffraction patterns resemble experimental patterns more closely. By applying different types of noise and blur to the simulated images, we can better understand the impact of these factors on the pattern recognition process and improve the performance of our model when dealing with real-world data. We will apply Gaussian blur, Gaussian noise, salt-and-pepper noise, and speckle noise to the simulated images, and examine their effects on the image quality. Furthermore, we will display the resulting noisy images side by side for easier comparison. Through this investigation, we aim to identify the most suitable combination of noise and blur that can help us create more realistic simulated diffraction patterns, thus enhancing the robustness of our predictive models in handling experimental data.





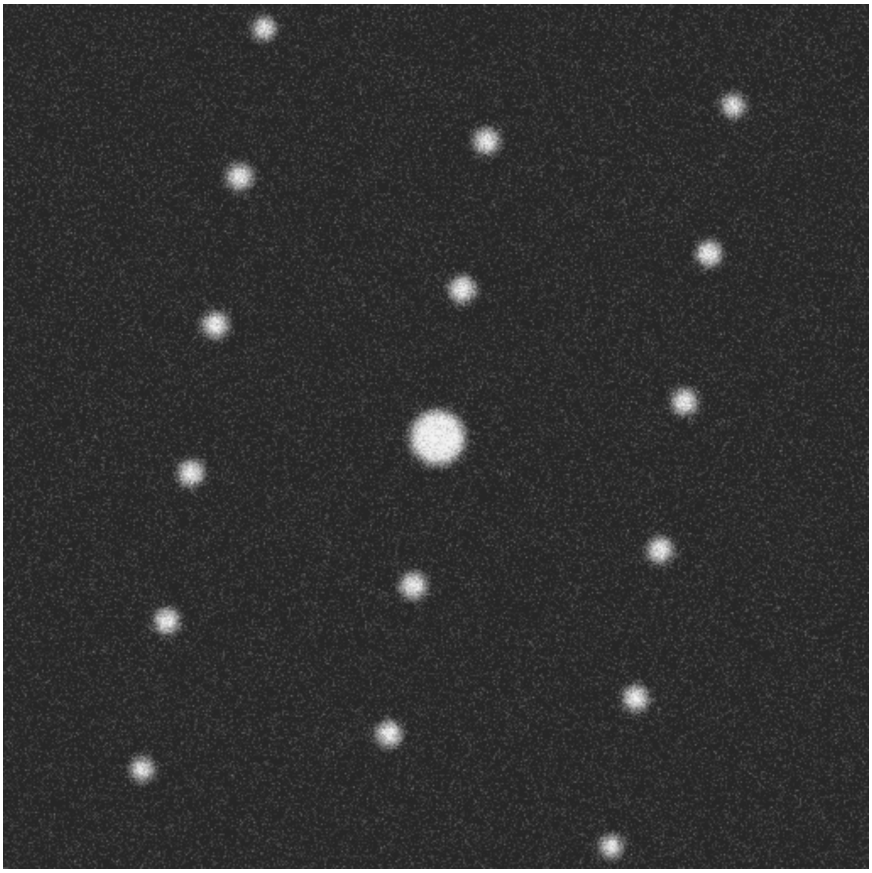
In this code block, we then apply a Gaussian blur with a sigma of 3 to the simulated image, which smooths the image by

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.

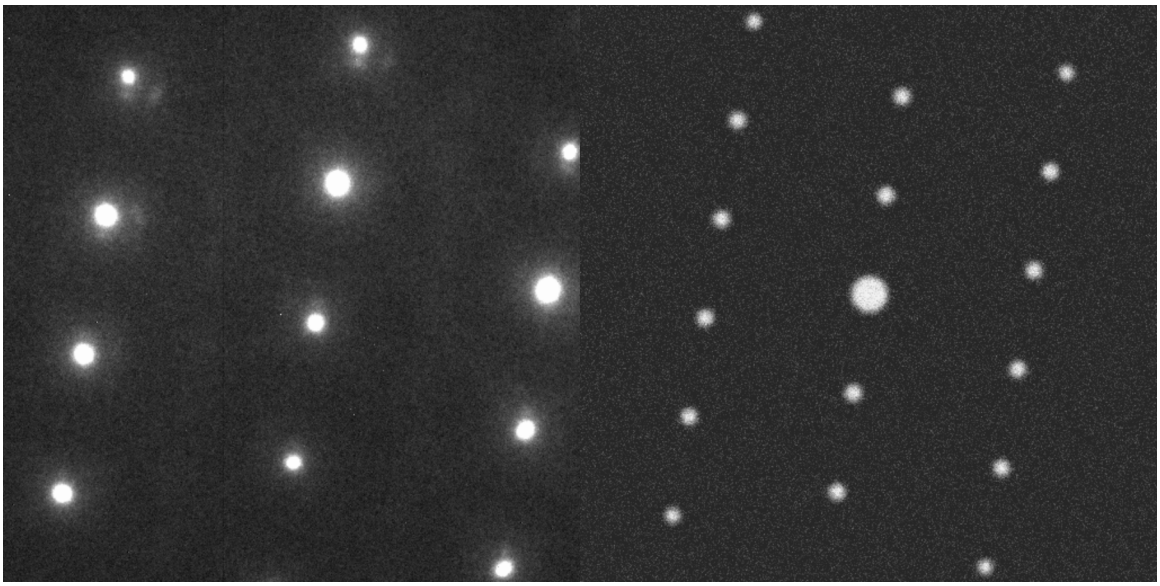


And here we first apply a Gaussian blur to the simulated image with a sigma value of 3. This helps to smooth out the image. Next, we add Gaussian noise with a mean of 0 and standard deviation of 0.1 to the blurred image. This introduces random variations to the pixel intensities, simulating the presence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

And next 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.



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.



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.

5 Statistical Learning: Modeling & Prediction

5.0.1 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.

5.1 Steps to build and train our CNN models:

1. Load the appropriate libraries: We will start by loading the libraries needed for data processing and visualization, including Keras, TensorFlow, and other libraries.
2. Read the Excel files containing the image IDs and phase information, then preprocess the data. Affix number labels to the phase labels for each image (0 for alpha, 1 for delta, and 2 for other). Then, preprocess (resize and normalize) the real and simulated images by loading them from their respective directories.
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. Our CNN models will be trained using the training set, validated using the validation set, and tested against the test set to determine how well the models performed overall.
4. Specify the filters, kernel sizes, and other relevant parameters, as well as the number of convolutional, activation, pooling, and fully connected layers, to construct the CNN models' architecture.
5. Train the CNN models: Use the provided architecture and hyperparameters to train the CNN models on the training set. To prevent overfitting and to fine-tune the hyperparameters, keep track of the model's performance on the validation set throughout training.
6. Assess the models' performance: After the models have been trained, assess how well they performed on the test set. To choose the most effective method for identifying the diffraction patterns, compare the performance of models trained on actual photos, simulated images, and a mix of both.
7. Improve and optimize the CNN models based on performance evaluations by modifying the models' architecture, hyperparameters, and training methods.

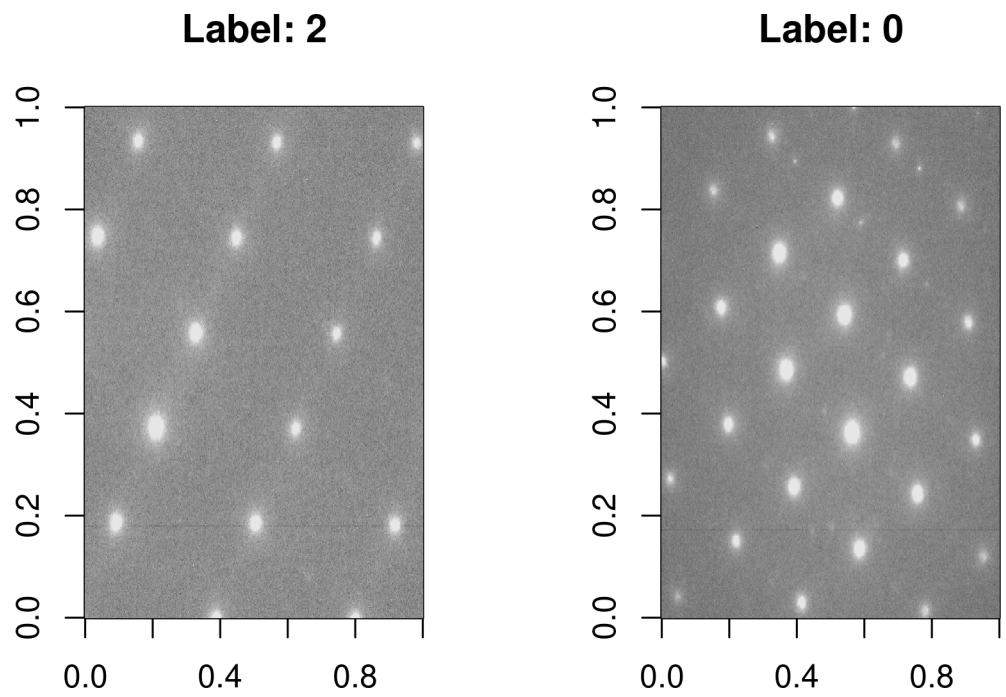
First, let's read the two Excel files that contain the phase information and combine them into a single data frame:

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:

```
## # 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.

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



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.

Split the data into training, validation, and test sets

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

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

```
## 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.

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.

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.

```
## Test loss: 1.1385
```

```
## Test accuracy: 0.34884
```

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

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

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

6 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.

7 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.