Computer Vision for investigating Chromatin

Abstract

In this work, the chromatin was examined using Computer Vision. Chromatin is a structure present within the cell nucleus. The reason behind studying chromatin structure is to analyze both efficacy and risk of cancer drugs. When a section of DNA is being expressed or repaired, chromatin assumes to be a more spread-out state, whereas during mitosis it compacts to form the chromosome. Chromatin is dynamic and responds to signals from the cell, such as DNA damage or adjustments in cellular metabolism, by changing its structural makeup. This enables the cell to control gene expression and adapt to the environment.

The goal is to predict the center position of chromatin using Computer Vision, we used Convolutional Neural Network and Object Detection Model to find the center of cell, In Object Detection Model as we created bounding boxes for each chromatin and made the assumption that the chromatin center would be located in the middle of the bounding box. After establishing the bounding boxes using the Faster R-CNN object identification model, we successfully located the center points of cells.

Keywords: -

Chromatin, Python, Keras, Convolutional Neural Network, Image Translation, Pytorch, Faster R-CNN (Detecto Model).

Table of Content

1. Introduction:

The goal is to detect center position of Chromatin present within the cell nucleus using Faster R-CNN, one of the deep convolutional networks used for object detection.

Sample Chromatin Structure

Given two types of image dictionaries for the project (NOSPP and SPP tiff format images), the tiff format is useful for representing multiple channels in a single image, and multiple pages, which can be used to store z-stacks or time-lapse sequences, where each frame is a movement of chromatin, it contains 60 to 122 frames for a single tiff file. The NOSPP middle slice image center coordinates were used as the Ground Truth.

However, we noticed that there was a little drift in NOspp photographs when compared to SPP images (the image appears to have been shifted in the upward direction for the specified folders 2021-11-02 to 2022-1-27) except the folder (2022-2-3). In order, we performed image translation based on the average difference between NOspp and Spp middle slice images.

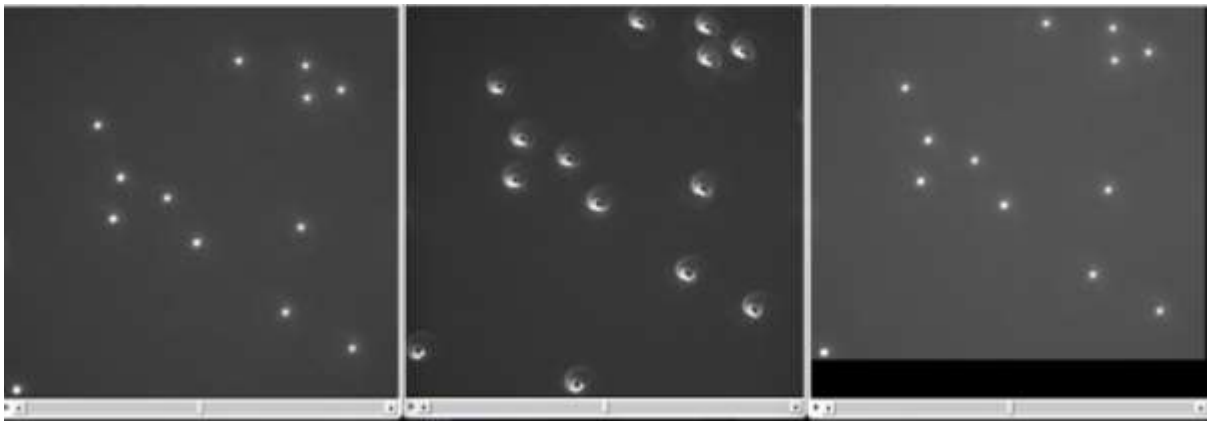
Translation on images refers to the process of moving an image by a specified distance along the x and y-axis. Translation can be used to align or register images, to adjust the position of an object within an image, or to transform an image into a different coordinate system. Each pixel is transformed to a new location using a transformation matrix. The transformation matrix specifies the amount of translation along the x and y-axis and is used to map each pixel in the original image to its new location in the translated image. After manually collecting the middle slice image centers from NOspp and SPP, the average of those difference was taken as translation parameter to adjust the NOSPP Images.

Below excel sheet shows how we selected image translation parameters. Showing translation parameters for two sample images (manually collected the x and y coordinates through Fiji.)

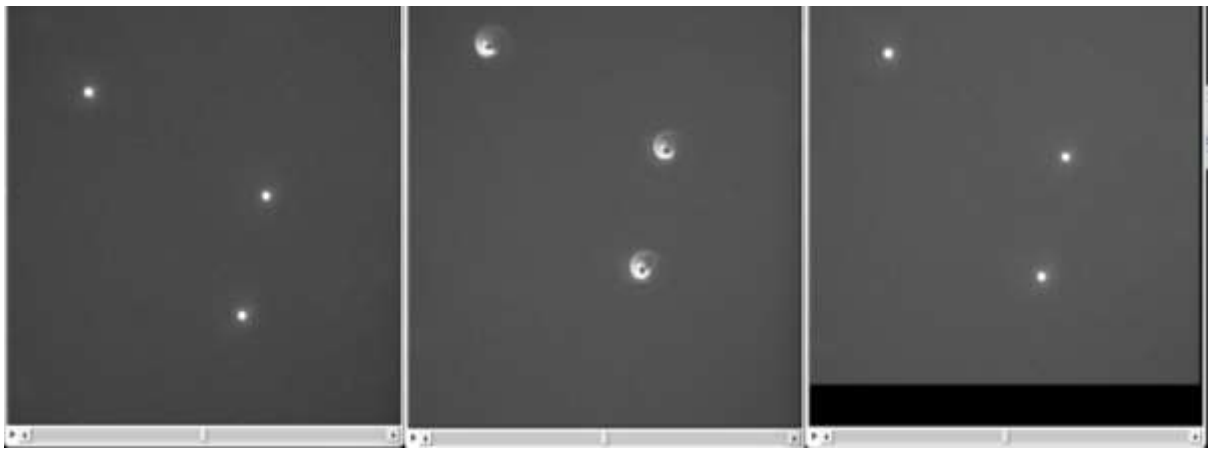
	nosppx	sppx	nosppy	sppy	diff x	diff y	avgx	avgy
1.11.22 fov3	103	102	107	52	1	55		
	298	297	371	317	1	54		
	329	328	228	174	1	54		
							1	54.333333
11.16.21 fov 3	18	14	489	441	4	48		
	120	116	151	104	4	47		
	140	137	271	224	3	47		
	149	147	218	170	2	48		
	208	205	244	197	3	47		
	246	245	301	253	1	48		
	298	296	68	20	2	48		
	358	355	390	342	3	48		
	377	374	281	233	3	48		
	383	381	75	27	2	48		
	385	382	116	68	3	48		
	427	424	105	55	3	50		
	442	439	436	390	3	46		
							2.769230769	47.769231

Code snippet for Image Translation

```
from PIL import Image
  # Open the TIFF file
  im = Image.Open(“imagepath”)
  # Get the number of frames in the TIFF file
  num_frames = im.n_frames
  # Create an empty list to store the translated frames
  translated_frames = []
  # Iterate through all the frames
  for i in range (0, num_frames):
     # Set the current frame
     im.seek(i)
     # Translate the frame
     translated_frame = im.transform(im.size, Image.AFFINE, (1, 0, x, 0, 1, y))
     # Append the translated frame to the list
     translated_frames.append(translated_frame)
# Save the translated frames to a single TIFF file
translated_frames[0].save("image”,save_all=True,append_images=translated_frames[1:])
```



Left image is the NOSPP, Middle image is the SPP image for comparison, and Right image is the translated version (translation parameters 2.7 on x axis and 47.7 on y-axis) the translation parameters for below image were x axis-1 and y axis 54.



Then we collected ground truth values from the NOSPP translated images for the use of comparing to the predicted results.

Code snippet to collect ground truth values

```
for filename in os.listdir(r'path'):
    if filename.endswith('.tif'):
        img = io.imread(f'path/{filename}')
        n_slices = img.shape[0]
        position_source = n_slices//2
        #our image is in sphere shape to retrieve center frame
        img[position_source,:,:].shape
        image = img_as_ubyte(img[position_source, :, :])
        thresh = cv2.normalize(image, None, 0, 255, cv2.NORM_MINMAX)
        thresh = cv2.threshold(thresh, 40, 255, cv2.THRESH_BINARY)[1]
        #, the values below 40 convert to 0 and remaining to 255
        labels = measure.label(thresh, connectivity=None, background=0)
        #print(np.unique(labels)), connectivity is used to find the same pixels around neighbors
        mask = np.zeros(thresh.shape, dtype="uint8")
        for label in np.unique(labels):
            if label == 0:
                continue
            labelMask = np.zeros(thresh.shape, dtype="uint8")
            labelMask[labels == label] = 255
            numPixels = cv2.countNonZero(labelMask)
            if numPixels > 6:
                mask = cv2.add(mask, labelMask)
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        cnts = contours.sort_contours(cnts)[0]
        centers = []
        for (i, c) in enumerate(cnts):
            ((cX, cY), radius) = cv2.minEnclosingCircle(c)
            centers.append((int(cX), int(cY)))
            cv2.circle(image, (int(cX), int(cY)), int(radius), (255, 0, 0), 2)
        center_file[f"{filename.split('_')[0]}_{filename.split('Fov')[1].split('_')[0]}"] = centers
```

Overall, the above code is used to detect thick pixels(spots) on a set of TIFF images and store the coordinates of the detected spots in a dictionary.

We first attempted to forecast centers using Convolutional Neural Network, but the results were unsatisfactory, making it impossible for us to rely on the prior CNN model. As a result, we picked an object identification model in which we first identify the objects before calculating their centers. Object detection models are a fundamental component of computer vision systems in general and are becoming more and more important in a range of fields.

In general, object detection models are an essential part of computer vision systems and are becoming more and more significant in a variety of sectors. This model is specifically designed for detecting instances of objects present in the images. During training phrase, the model is trained on many examples of images with annotated objects, and it uses these examples to learn how to recognize and

locate objects in new images. The result is much better than the CNN-based center point (X, Y) prediction, which had errors of 2-digit pixels.

2. Applying CNN to predict Spot Centers:

The fundamental principle behind a CNN is to automatically extract characteristics from the input data using convolutional layers. These features can subsequently be utilized to create predictions or categorize he data. CNN Architecture we used:

The first layer is a Conv2D layer with 16 filters and a kernel size of (3,3). This layer is followed by a Batch Normalization layer and a MaxPooling2D layer with a pool size of (2,2) and

The second layer we used is another Conv2D layer with 32 filters and a kernel size of (3,3). This layer is followed by another Batch Normalization layer and another MaxPooling2D layer with a pool size of (2,2).

The third layer is a Conv2D layer with 64 filters and a kernel size of (3,3). This layer is followed by another Batch Normalization layer and another MaxPooling2D layer with a pool size of (2,2).

The fourth layer we used is a Flatten layer that converts the 3D output of the previous layer into a 1D vector.

The fifth layer is a Dense layer with 16 neurons, followed by a Batch Normalization layer and a Dropout layer and the final layer is another Dense layer with 3 neurons, which corresponds to the 3 classes (X, Y coordinates, and the Z is depth of a slice) in the output.

Library

The library used for CNN model is Keras, it offers a user-friendly interface for designing a CNN's architecture, creating the model, and using data to train the model.

Training Data

The model was trained on 100 tiff images, data was labelled into a JSON format and passed to train model. Each JSON file contains information about multiple spirals in an image. We used plt.ginput() matplotlib function that allows the user to interactively select points from a plot. When called, it displays a figure and waits for the user to click on the figure with the mouse. The function then returns the coordinates of the clicked points as a list of tuples.

Code used for labelling Training data

```
import matplotlib.pyplot as plt
import numpy as np
from skimage import io
imgpath = input ("Img file path:")
# Load top image
img = io.imread(imgpath)
depthfile = input ("Depth file path:")
clnum = int (input("Column Number:"))
tmpdepth = []
with open(depthfile) as f:
for line in f:
try:
tmpdepth.append(line.split(" ")[clnum-1])
except IndexError:
pass
tmpdepth1 = tmpdepth[5:]
n_slices = img.shape[0]
plt.rcParams['figure.figsize'] = [15, 15]
depths = {}
# Get general mid slice to get a decent idea for spiral centers to label
for i in range (0, n_slices):
depths[str(i)] = tmpdepth1[i]
position_source = n_slices//2
depths = {k: v.replace('\n', '') for k, v in depths.items()}
# Pick spot XY positions
print ("Select spots. Left click to add, Right click to remove, Enter or middle click to submit full set")
plt.figure()
plt.imshow(img[position_source,:,:])
plt.axis('off')
coords = plt.ginput(-1, -1, True, 1, 3, 2)
import os
import json
# Save training data
with open (imgpath + '. json', 'w') as outfile:
json.dump({
'XY': coords,
'Z': depths
}, outfile, indent=4)
```

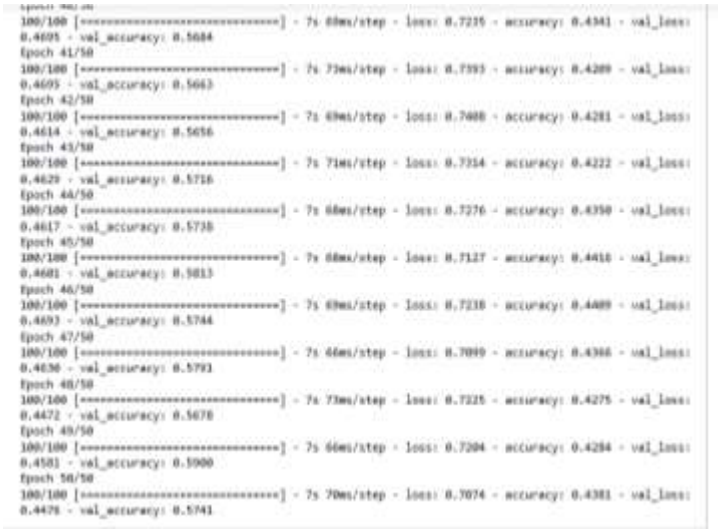
We used model.fit_generator instead of model.fit, it allows you to train the neural network using data generated by a Python generator function. We created a generator function in order to perform data augmentation, it returns a generator object that yields batches of input data and corresponding labels(coordinates) each time it is called. It takes as input a list of images, a batch size (bs), an input size (input_size), a shift value (shift), and a boolean value shuffle indicating whether the images should be shuffled before generating batches.

Code used to generate generator function

```
def tile_generator(images,bs, input_size, shift, shuffle = True):
    if shift<1:
        shift = int(input_size*shift)
    n_stacks = len(images)
    c_stack = c_spot = c_depth = 0
    while True:
        inputs = []
        targets = []
        while len(inputs) < bs:
            if shuffle:
                stack = random.choice(images)
                X, Y = random.choice(stack[0]['XY'])
                c = random.choice(list(stack[0]['Z'].items()))
                imgnum = int(c[0])
                Z = float(c[1])
            else:
                stack = images[c_stack]
                X, Y = stack[0]['XY'][c_spot]
                c = list(stack[0]['Z'].items())[c_depth]
                imgnum = int(c[0])
                Z = float(c[1])
                n_spots = len(stack[0]['XY'])
                n_depths = len(stack[0]['Z'])
                c_depth += 1
                # Move to next spot if all depths have been consumed
                if c_depth == n_depths:
                    c_depth = 0
                    c_spot += 1
                    # Move to next stack if all spots have been consumed
                    if c_spot == n_spots:
                        c_spot = 0
                        c_stack += 1
                        # Start a new epoch if all stacks have been consumed
                        if c_stack == n_stacks:
                            c_stack = 0
            xcrop, ycrop = tuple(int(p + random.uniform(-1*shift, shift)) for p in (X, Y))
            img = stack[1][imgnum][ycrop-input_size//2:ycrop+(input_size-input_size//2), \
                                xcrop-input_size//2:xcrop+(input_size-input_size//2)]
            x_train = (X - (xcrop-input_size//2))/input_size
            #print(x_train)
            y_train = (Y - (ycrop-input_size//2))/input_size
            z_vals = [float(v) for v in list(stack[0]['Z'].values())]
            z_train = (Z - min(z_vals)) / ((max(z_vals)-min(z_vals))/2) - 1
            try:
                # normalize image
                img = np.divide(img,np.max(img))
                img.reshape(input_size, input_size, 1)
            except:
                import pdb;pdb.set_trace()
            inputs.append(img.reshape(input_size, input_size, 1))
            targets.append(np.array([x_train, y_train, z_train]))
        yield(np.array(inputs),np.array(targets))
```

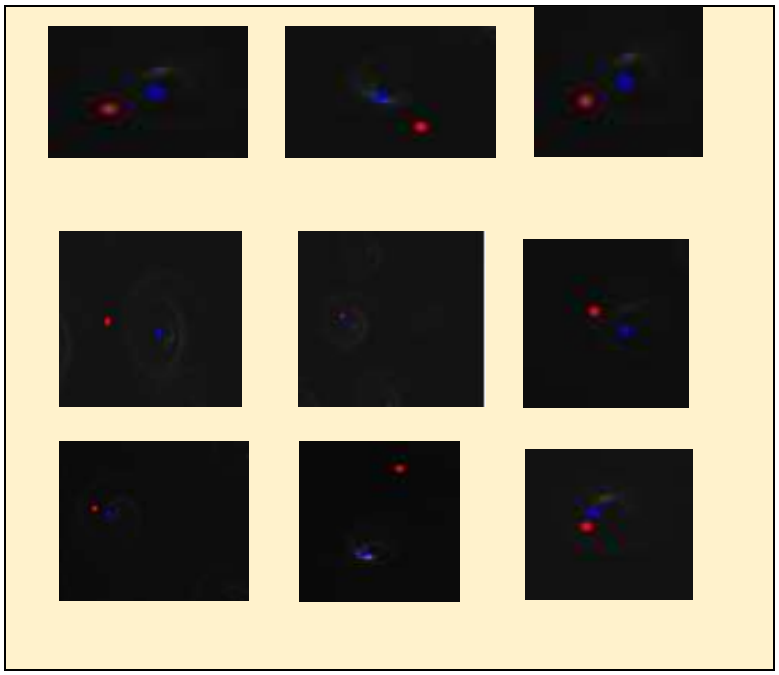
For each batch, the function selects random images from the list of images. For each image, it selects a random spot (i.e., a (x, y) coordinate) and a random depth value from the available depth values for that spot. It then crops a square patch of size input_size around the spot and depth, applies some random shift to the spot, and returns the patch as the input to the neural network. The target output for the network is a tuple of three floats: the normalized x and y coordinates of the spot, and the normalized depth z value. The result is then passed through the model as a training set.

Here is an example of the progression of the loss throughout the model's training and the model predictions.



Training Loss is around 70 % for 50 epochs

Comparing ground truth coordinates and predicted coordinates by CNN for the images 11.18.21_SPP_Fov3_1_NDTiffStack.tif and 11.23.21_SPP_Fov2_1_NDTiffStack.tif



[Blue dot is Ground Truth while Red is Predicted results]

Conclusion 1: The results were not satisfactory the difference between actual ground truth and predicted coordinates were around 2-digit pixels.

Limitations of CNN for spot detection:

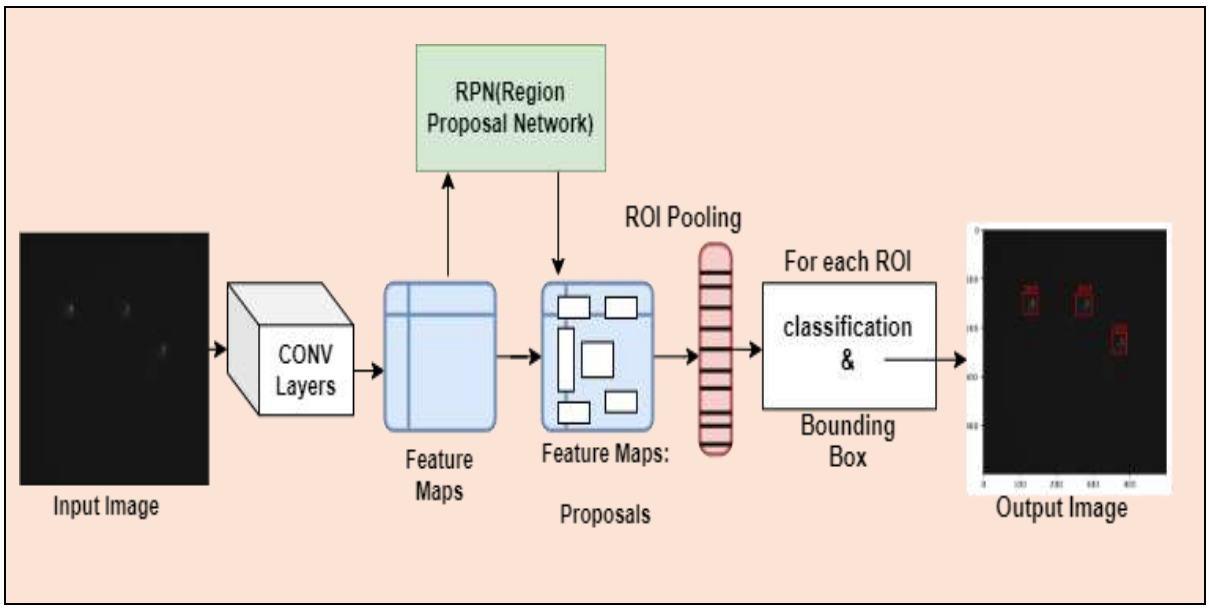
It is complicated to code and results were not accurate

3. Applying Object Detection Model to predict Spot Centers:

One of the simplest ways to detect objects is to use object detection model. It used to distinguish background and object present in the image. Some of the most used object detection models include R-CNN, Fast R-CNN, and Faster R-CNN. Those models are two-stage object detection models that consists of two stages: a region proposal stage and a classification stage. In the first stage, finds the set of regions present in the image and next classification stage the model uses convolutional neural network (CNN) to classify each region present in image as an object or background and create a bounding box (identification mark) for each object. We come with Faster R-CNN model in which the region proposals are generated by a separate network, called the region proposal network (RPN), which runs in parallel with classification network, this makes faster RCNN faster.

Faster-R-CNN Architecture:

In Faster R-CNN, first input image passing into CNN for feature maps then RPN came into play, used to generate region proposals, then generated region proposals passed into ROI Pooling after then it performs classification.



Faster R-CNN is the combination of Fast RCNN and Region Proposals. The first stage of Faster RCNN is the Region Proposal Network (RPN). In this stage, the network generates region proposals, which are used to identify the potential object regions in the image.

RoI Pooling: To handle different scales of objects, the Faster RCNN model uses RoI pooling. It takes the feature maps from RPN and crops regions of interest from the feature maps. The cropped regions are then resized to a fixed size, regardless of the size of the objects in the image.

Anchor Boxes: Anchor boxes are pre-defined bounding boxes that are used to generate the object proposals. The anchor boxes are generated based on the aspect ratios and scales of the objects in the training set.

Second Stage: The second stage of Faster RCNN is the classification and regression network. In this stage, the network takes the region proposals generated by the RPN and refines the bounding boxes around the objects.

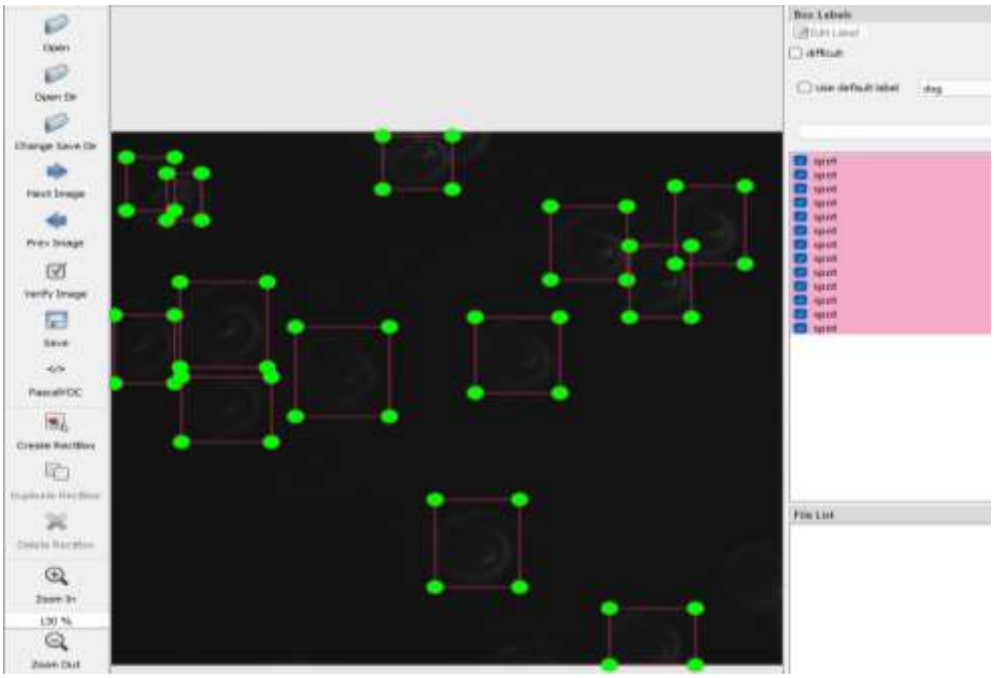
4. Implementation and Results

Library

The library used to achieve these results was detecto. The library wraps a pretrained Faster R-CNN with a Resnet50-FPN backbone for easy use with custom data and helps us apply transfer learning to our dataset.

Training Data

The model was trained on the first slices of each different FOV of the latest data. To make sure the model learns to recognize spirals that are not just in one similar state, a custom rotational transform was also applied to the training dataset. This makes it so when the spirals play out, the model should recognize different angles of their rotation and still predict the center / bounding box accurately. The data was labelled through a tiny user-friendly labelling utility LabelImg, and the data was then put into the Pascal VOC XML format. LabelImg is an open-source graphical image annotation tool that is used to annotate images for computer vision tasks, such as object detection.



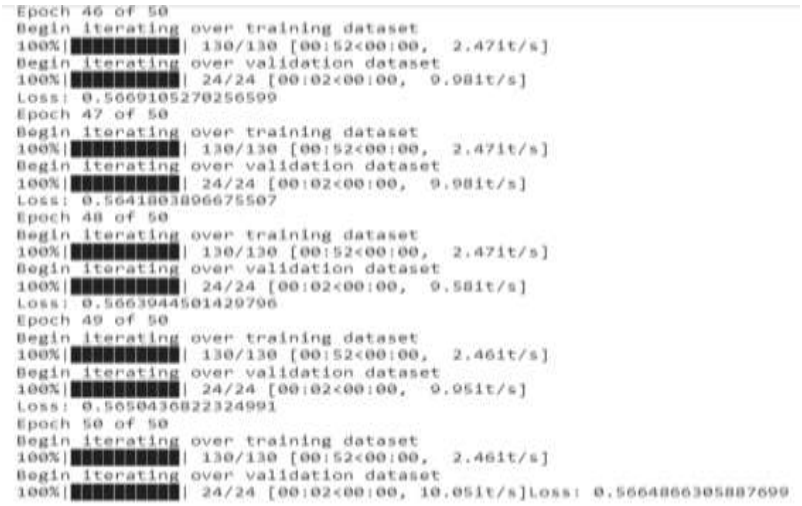
Manually creating PASCALVOC file for annotations using labelImg

we performed training on 50 epochs with the learning rate of 0.001, the overall code works using the pytorch library, PyTorch provides several high-level APIs for building and training neural networks, as well as a lower-level API for more custom and complex network design. The library also provides a range of pre-trained models that can be fine-tuned for various tasks, making it easier for practitioners to build and deploy their own AI systems. The custom transformation is used to improve the performance by modifying the image into required format and core. Data Loader allows to iterate over the training data in minibatches.

Code snippet belongs to Detecto model

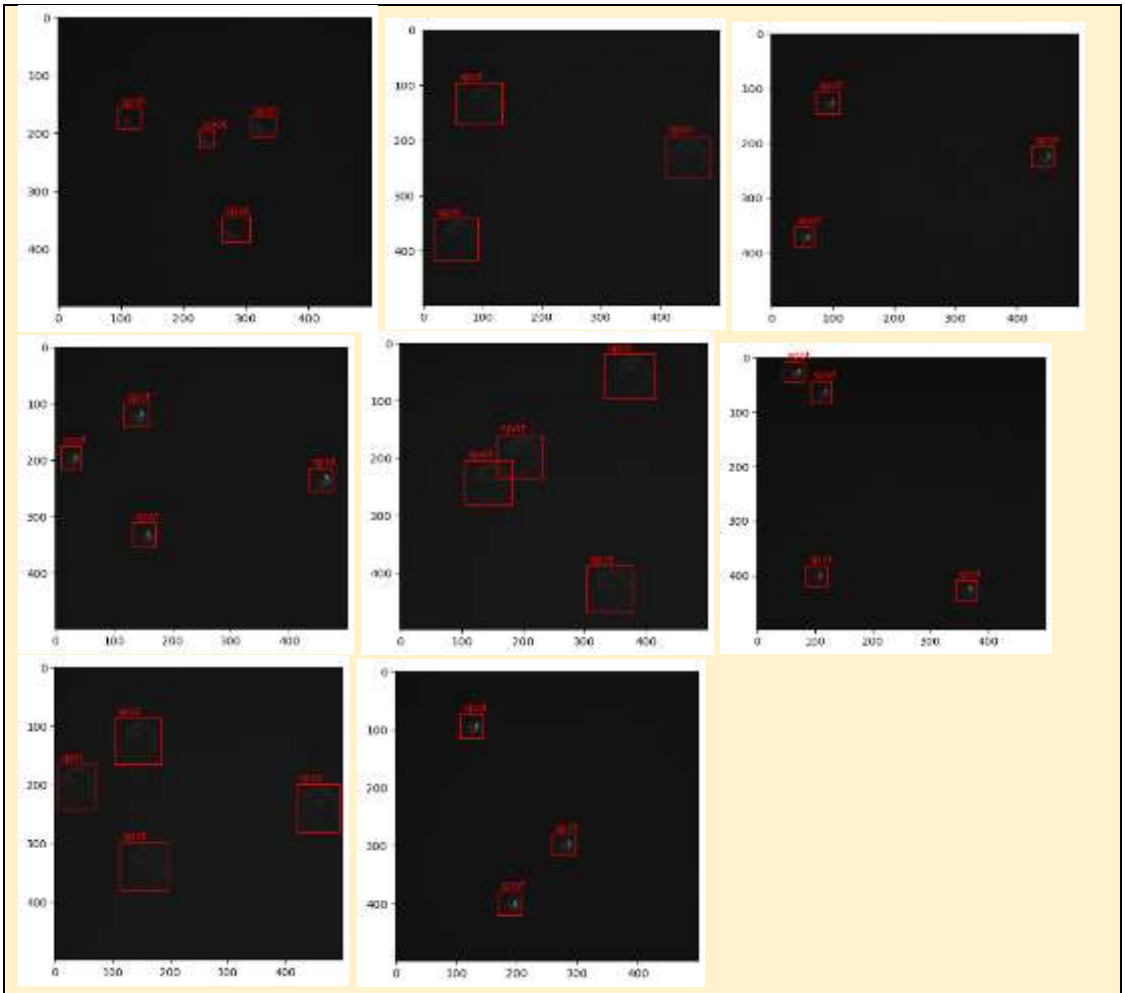
```
pip install detecto
from detecto import core, utils, visualize
from detecto. visualize import show_labeled_image, plot_prediction_grid
from torchvision import transforms
import matplotlib. pyplot as plt
import numpy as np
import torch
custom_transforms = transforms.Compose([transforms.ToPILImage(),
transforms.Resize(900), transforms.ColorJitter(saturation=0.2), transforms.ToTensor(),
utils.normalize_transform(),
])
Train_dataset = core.Dataset('path', transform=custom_transforms)
Test_dataset = core.Dataset('path')
loader = core.DataLoader(Train_dataset, batch_size=2, shuffle=False)
model = core.Model(['spot'])
losses = model.fit (loader, Test_dataset, epochs=50,lr_step_size=5,learning_rate=0.001,
verbose=True)
```

Here is an example of the progression of the loss throughout the model's training and the model predictions.



Training loss is around 50% for 50 epochs.

But compared to CNN, the training loss is relatively less for Faster-R-CNN model.



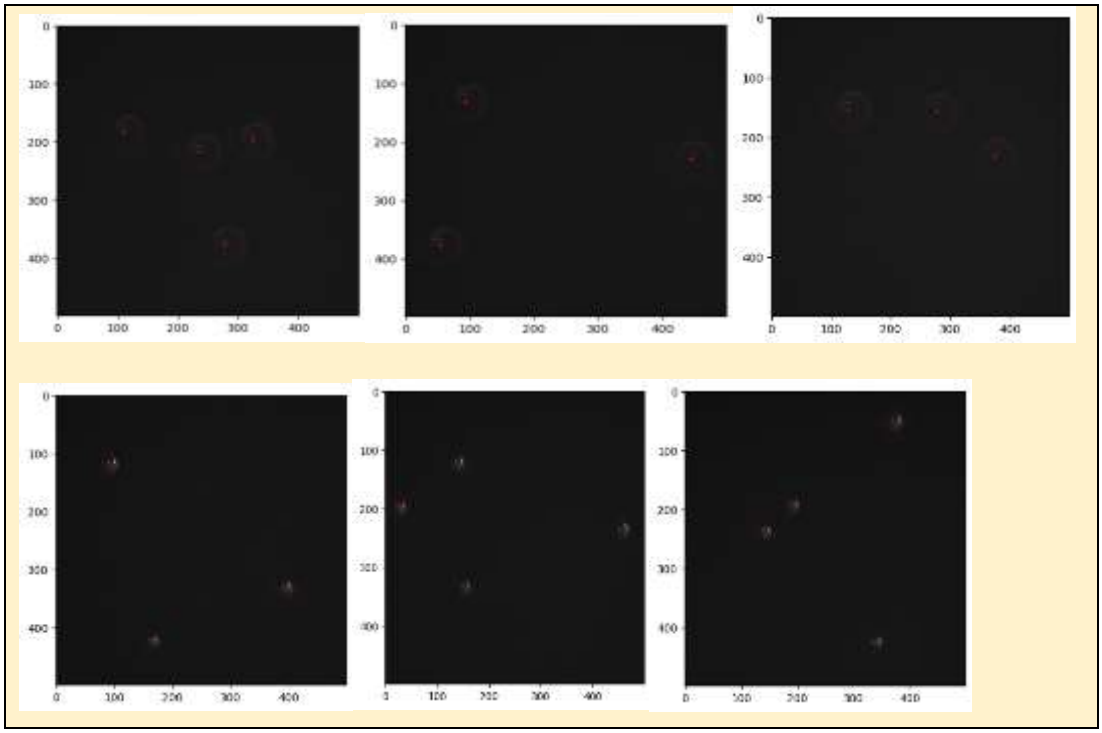
Results using Faster RCNN

We assumed that the center of bounding box will be of cell centers, Since the model generally predicts bounding boxes and uses those as its inputs, the centers were calculated from the bounding box dimensions and locations.

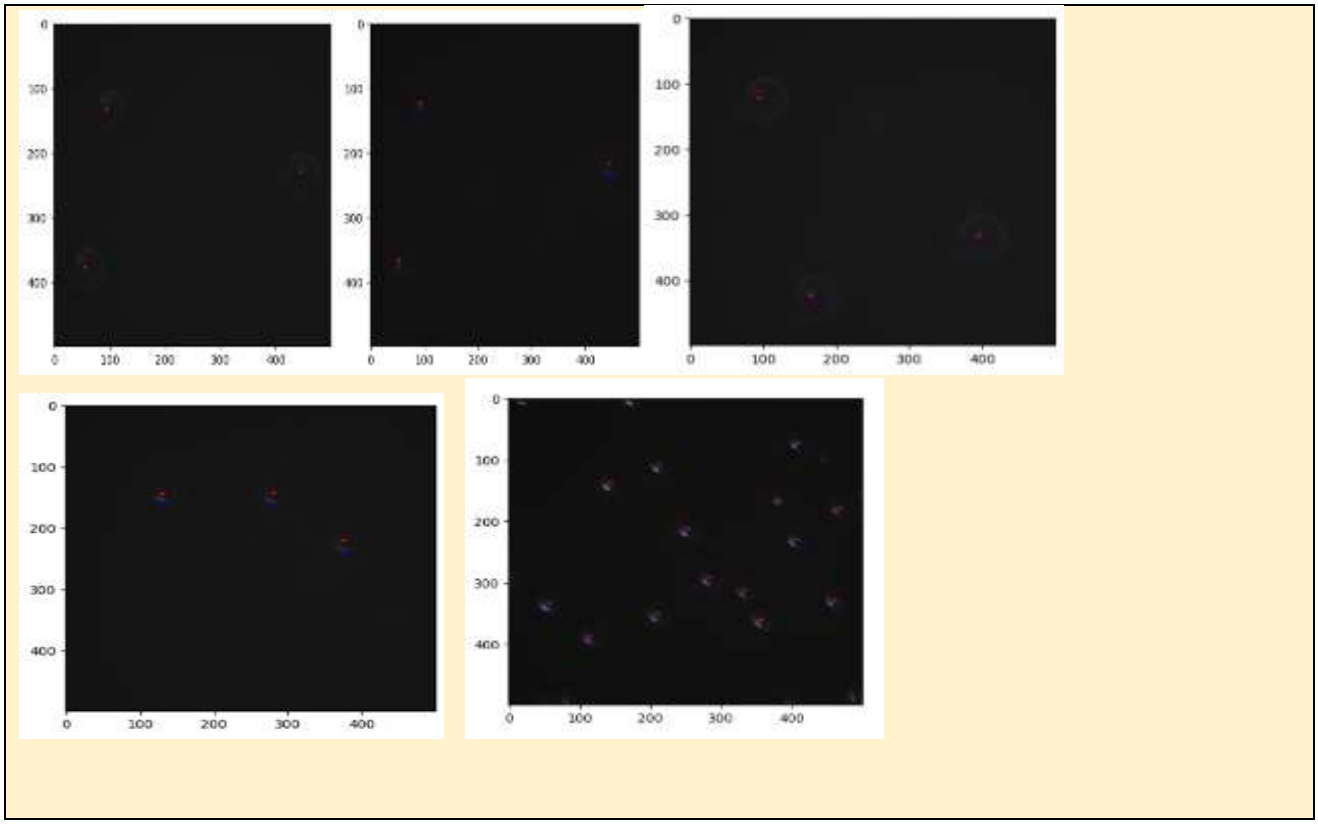
Code to find bounding box centers

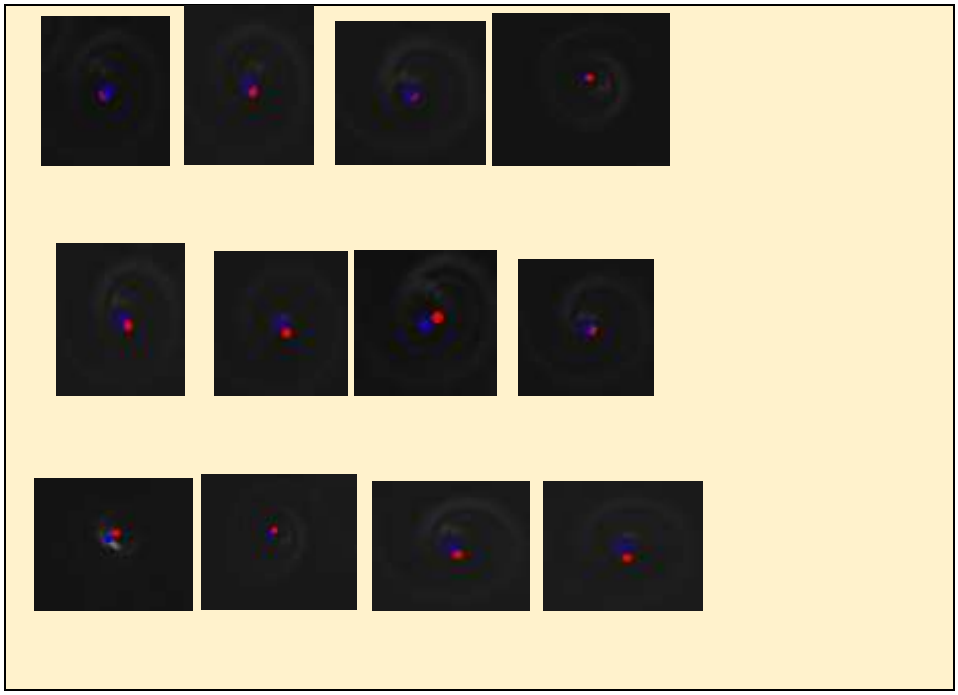
```
for box in bndboxes:
        box = tuple(box)
        xmin = int(box[0])
        xmax = int(box[2])
        ymin = int(box[1])
        ymax = int(box[3])
        predicted_center = (xmin + (xmax-xmin)//2, ymin + (ymax-ymin)//2)
```

Here are some examples of the predicted centers:



Comparing Ground Truth and Predicted Coordinates



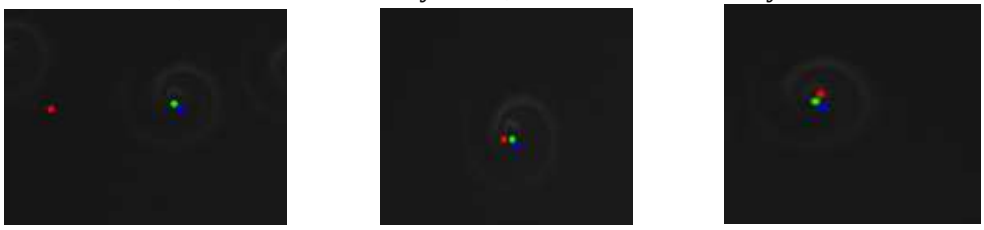


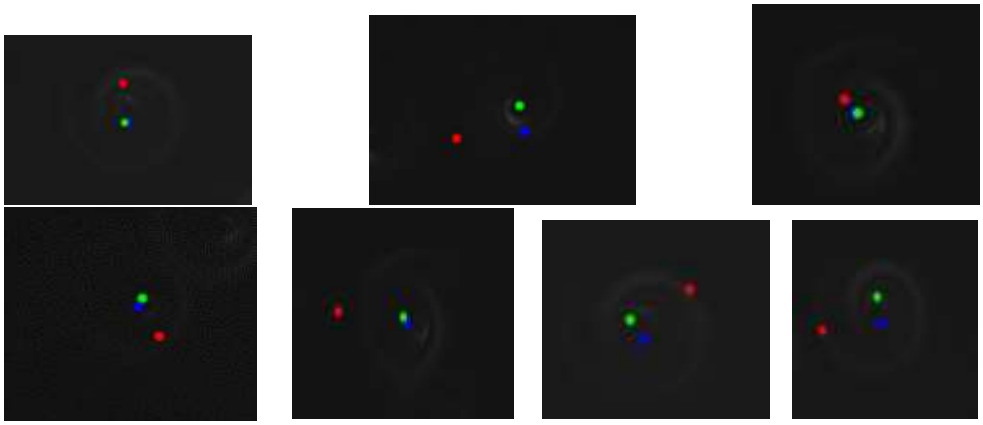
Results for the images 11.23.21_SPP_Fov2_1_NDTiffStack.tif and 2.3.22_SPP_Fov3_1_NDTiffStack.tiflast.png

5. Comparing CNN and Faster-R-CNN Results:

FileName	AverageXError CNN	AverageXError FRCNN	AverageYError CNN	AverageYError FRCNN
11.18.21_SPP_Fov3_1_NDTiffStack.tif	15.33	-1.3	7.8	4.8
11.30.21_SPP_Fov8_1_NDTiffStack.tif	-2.6	-1.5	20.3	5.3
2.3.22_SPP_Fov1, 3, 4_1_NDTiffStack.tif	12.4	4.6	13.9	9.3
11.23.21_SPP_Fov2_1_NDTiffStack.tif	8.6	-2.5	14.16	4.6
11.2.21_SPP_Fov3_1_NDTiffStack.tif	-0.3	-3.8	34.33	4.1

The Average Error for X and Y Coordinates for CNN is relatively high compared to Faster-R-CNN and CNN requires more time complexity. Here are results predicted on the random images from folders 2.3.22 and 11.18.21. **Blue : GroundTruth, Red : Predicted by CNN, Green: Predicted by Faster-R-CNN**.





6. Conclusion

The project's objective is to use Computer Vision to determine the location of a cell's center, and as the initial step in that process, we've chosen to employe one of the famous deep learning algorithm called Convolutional Neural Network. Regrettably, the outcomes were unsatisfactory, and the training process taken more time than expected. As the result, we switched to one of the Object Detection Model called Faster RCNN, it produced good results, and the model was quick and simple to use.

7. References

[1] https://www.researchgate.net/figure/Network-structure-diagram-of-Faster-R-CNN-Faster-R-CNN-is-mainly-divided-into-the_fig1_341871095
[2] Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun.
[3] https://www.researchgate.net/publication/323528969_Faster_R-CNN_based_microscopic_cell_detection.