



Stony Brook University

**Milestone 2: Image classification using CNN and object segmentation.**

ESE 577: Deep Learning Algorithms and Software.

Spring 2024.

03/23/2024

Suvab Baral, 115646344

Sakshi Nagapure, 116000503

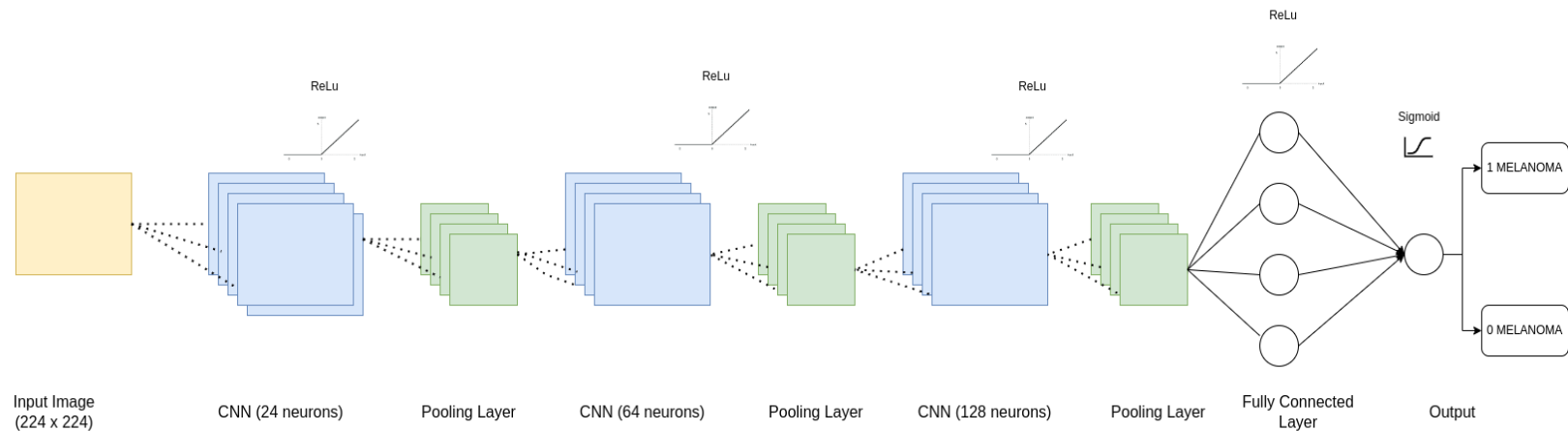
# Table of Contents

1. Introduction.....	3
2. Image Classification.....	4
A. Data structure.....	4
B. Data cleaning.....	5
C. Train test split.....	6
D. Lazy loading of images.....	7
E. CNN model architecture.....	8
F. Training.....	9
3. Image Classification: Results.....	10
A. Learning in each epoch.....	10
B. Accuracy and loss plot.....	10
C. Prediction of unseen images.....	11
D. Comparing results with advanced model.....	12
3. Image segmentation.....	13
A. Methodology.....	13
B. Output.....	15
C. Results.....	17
5. Conclusion.....	18

# 1. Introduction

In the area of medical image analysis, the task of accurate classification and segmentation plays a pivotal role in diagnosing diseases and assisting medical professionals in decision-making. In pursuit of advancing the field, this report embarks on a comprehensive exploration into leveraging deep learning techniques for image classification and segmentation, focusing specifically on dermatological images from the ISIC 2019 challenge dataset.

The primary objective of this study is twofold. Firstly, utilizing TensorFlow and Keras, we build Convolutional Neural Network (CNN) from scratch to classify images from the International Skin Imaging Collaboration (ISIC) 2019 Challenge dataset. This dataset is known for its diverse range of dermatological conditions, presenting a challenging yet promising opportunity to delve into the intricacies of image classification. Although it has images belonging to 8 different classes, we just randomly choose 2 for the classes: Melanoma and Basal cell carcinoma (BCC)



Following the evaluation of our self-built CNN, we pivot to compare its accuracy with an optimized model, wherein advanced architectures and hyperparameter tuning are employed to potentially enhance classification performance.

Transitioning from classification to segmentation, the second part of our report delves into object segmentation using a pre-trained PyTorch Mask R-CNN ResNet-50-FPN model. Object segmentation involves delineating and identifying specific objects or regions within images. Leveraging the power of deep learning, we apply the Mask R-CNN architecture to perform image segmentation on three previously unseen dermatological images. By employing a pre-trained model, we aim to assess the segmentation accuracy and the model's ability to generalize to novel data, offering valuable insights into its practical applicability in real-world scenarios.

## 2. Image Classification: CNN Architecture

We've developed a CNN Architecture to classify two out of the nine skin cancers using the demoscopic images available in the ISIC 2019 challenge.

[<https://challenge.isic-archive.com/data/#2019>]. The two skin cancers we have chosen are Melanoma and Basal cell carcinoma.

The dataset is scattered among the images in a folder and a csv file that contains the labels for each image that can be used for training. First, we studied the data organization of the images and the csv file.

### I. Data structure:

The initial data structure consists of a folder with images and a csv file containing the image names and its labels.

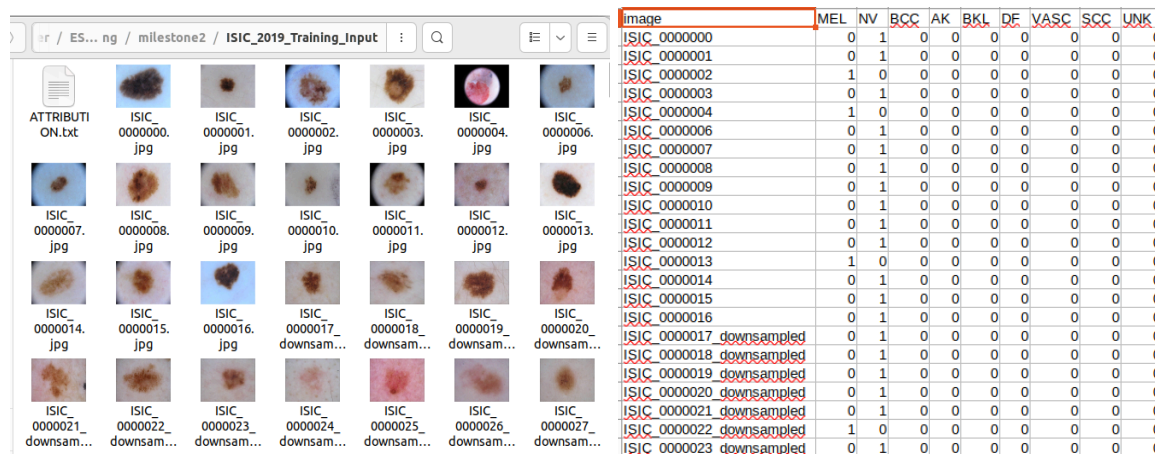


Image	MEL	NV	BCC	AK	BKL	DF	VASC	SCC	UNK
ISIC_0000000	0	1	0	0	0	0	0	0	0
ISIC_0000001	0	1	0	0	0	0	0	0	0
ISIC_0000002	1	0	0	0	0	0	0	0	0
ISIC_0000003	0	1	0	0	0	0	0	0	0
ISIC_0000004	1	0	0	0	0	0	0	0	0
ISIC_0000006	0	1	0	0	0	0	0	0	0
ISIC_0000007	0	1	0	0	0	0	0	0	0
ISIC_0000008	0	1	0	0	0	0	0	0	0
ISIC_0000009	0	1	0	0	0	0	0	0	0
ISIC_0000010	0	1	0	0	0	0	0	0	0
ISIC_0000011	0	1	0	0	0	0	0	0	0
ISIC_0000012	0	1	0	0	0	0	0	0	0
ISIC_0000013	1	0	0	0	0	0	0	0	0
ISIC_0000014	0	1	0	0	0	0	0	0	0
ISIC_0000015	0	1	0	0	0	0	0	0	0
ISIC_0000016	0	1	0	0	0	0	0	0	0
ISIC_0000017_downsampled	0	1	0	0	0	0	0	0	0
ISIC_0000018_downsampled	0	1	0	0	0	0	0	0	0
ISIC_0000019_downsampled	0	1	0	0	0	0	0	0	0
ISIC_0000020_downsampled	0	1	0	0	0	0	0	0	0
ISIC_0000021_downsampled	0	1	0	0	0	0	0	0	0
ISIC_0000022_downsampled	1	0	0	0	0	0	0	0	0
ISIC_0000023_downsampled	0	1	0	0	0	0	0	0	0

The dataset is scattered among the images in a folder and a csv file that contains the labels for each image that can be used for training. First, we studied the data organization of the images and the csv file.

We then clean this large dataset into a much smaller dataset containing only the required images that belong to Melanoma and Basal cell carcinoma. All other images are not considered right now, so we avoid loading them into the program.

The required dataset now totals around 8000 images with close to 4500 images for melanoma and 3300 images for BCC.

```
----- Selecting MEL and BCC images-----  
----- There are a total of 7845 images.-----  
----- MEL image count: 4522-----  
----- BCC image count: 3323-----
```

## II. Data cleaning:

Since we are only using the data of Melanoma and Basal cell carcinoma, we separate these data from the rest of the data that are irrelevant to us for this study.

We go through all the items in the dataset and separate those image names that have the label MEL or BCC as 1.

```
def filterImageNamesForLabels(csvFile, outputCsvFile, label1Index,  
label2Index):  
    with open(csvFile, 'r', newline='') as infile, open(outputCsvFile, 'w',  
newline='') as outfile:  
        reader = csv.reader(infile)  
        writer = csv.writer(outfile)  
        melCount = 0  
        bccCount = 0  
        writer.writerow(['image_name', 'MEL', 'BCC'])  
        for row in reader:  
            if row[label1Index] == '1.0' or row[label2Index] == '1.0':  
                if (row[label1Index] == '1.0'):  
                    melCount += 1  
                    labels.append(1)  
                else:  
                    bccCount += 1  
                    labels.append(0)  
            melCount += 1 if row[label1Index] == '1.0' else 0  
            bccCount += 1 if row[label2Index] == '1.0' else 0  
            writer.writerow([row[0], row[label1Index], row[label2Index]])  
        printInFormat('There are a total of ' + str(melCount + bccCount) + '  
images.')  
        printInFormat('MEL image count: ' + str(melCount))  
        printInFormat('BCC image count: ' + str(bccCount))
```

Since the problem now becomes a binary classification problem, we assign the labels for those image names having MEL=1 as '1' and those images having BCC=1 as '0'. The final output csv file now looks like the following:

2090	ISIC_0034012	1	0
2091	ISIC_0034015	0	1
2092	ISIC_0034022	1	0
2093	ISIC_0034026	0	1
2094	ISIC_0034028	1	0
2095	ISIC_0034034	1	0
2096	ISIC_0034036	1	0
2097	ISIC_0034046	1	0
2098	ISIC_0034047	0	1
2099	ISIC_0034048	1	0
2100	ISIC_0034049	1	0
2101	ISIC_0034050	1	0
2102	ISIC_0034051	1	0
2103	ISIC_0034052	1	0

We only have 3 columns, the image name with label MEL=1 or BCC=1.

Basically after all the cleaning, pruning and data isolation is complete, we have two variables

**image\_paths** : an array of paths of images belonging to MEL or BCC.

**labels** : an array of labels containing 1 or 0 representing each of the. 1 representing that the image is of Melanoma.

Notice here, that we are only loading the image paths as of now and not the actual images in order to save the program memory. We load the images during training the model.

### III. Train-test split:

We use the widely popular *train\_test\_split* module from *sklearn* in order to split our dataset in a 80 - 20 train test split.

```
def trainTestSplitDataset(images, labels):
    imageSet = np.array(images)
    labelSet = np.array(labels)
    return train_test_split(imageSet, labelSet, test_size=0.2,
random_state=42)
```

```
X_train_path, X_test_path, y_train, y_test =
trainTestSplitDataset(imagesPaths, labels)
```

## IV. Lazy loading of images:

Once we have divided the image paths and the labels into training and testing data, we prepare a lazy loader to prepare loading the images for training.

When we tried to load the entire dataset containing 8000 images into the program, the system was unable to handle the load of the data causing it to crash. So, it was not possible for us to load all the images into the program at once.

We pivoted to use the lazy loading mechanism for the images. *Tensorflow* provides a mechanism called ImageDataGenerators

([https://www.tensorflow.org/api\\_docs/python/tf/keras/preprocessing/image/ImageDataGenerator](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator)) to support lazy loading of large datasets.

```
IMAGE_HEIGHT = 224
IMAGE_WIDTH = 224
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
```

We normalize the pixels of the images between 0 and 1 in order to reduce the complexity.

```
train_df = pd.DataFrame({'image_paths': X_train_path, 'labels':
y_train.astype(str)})
test_df = pd.DataFrame({'image_paths': X_test_path, 'labels':
y_test.astype(str)})
```

```
train_generator = train_datagen.flow_from_dataframe(
    dataframe=train_df,
    x_col='image_paths',
    y_col='labels',
    target_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
    batch_size=32,
    class_mode='binary'
)
```

```
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    x_col='image_paths',
    y_col='labels',
    target_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
```

```

        batch_size=32,
        class_mode='binary'
    )

```

This helps create a data flow to pass into the CNN model where the images will be loaded in batches. Dividing the images into batches, helped to improve the runtime and training time of the program.

The given size of the image is 1024 x 768 which is very large to process for a non-GPU system. So we also resize the images into a simple 224 x 224 pixel size.

## V. CNN model architecture:

Finally, we define our CNN model after several iterations of testing with different types of architectures.

```

model = Sequential()
model.add(Input(shape=(IMAGE_HEIGHT, IMAGE_WIDTH, 3)))
model.add(Conv2D(28, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

```

The architecture of the CNN we have defined is as follows:

1. 1 Input Layer
2. 3 Convolutional and Max Pooling hidden layers  
Consisting of 24, 64 and 128 hidden neurons.
3. 1 Fully connected layer  
Consisting of 64 hidden neurons.
4. 1 Output layer for classification.



The max pooling layer is added after each convolutional layer and the extra fully connected layer is added at the end in order to capture every minute details of the image that might be missed due to resizing the image into a smaller size.

We tried different activation functions including softmax and tanh and found that 'relu' was most suited and produced a better accuracy in terms of binary classification of images.

We also use the standard adam optimizer with the default learning rate of 0.01 and binary cross-entropy as the loss function since we divided the problem into a binary classification problem.

## **VI. Training:**

We train the model using the train\_generators that we created above across 10 and 20 epochs and plot the accuracy and losses.

The validation is conducted using the test\_generators from the test dataset.

```
history = model.fit(train_generator, epochs=10,  
validation_data=test_generator)
```

On most iterations the model shows the accuracy between 70 - 82 %.

# 3. Image Classification: Results

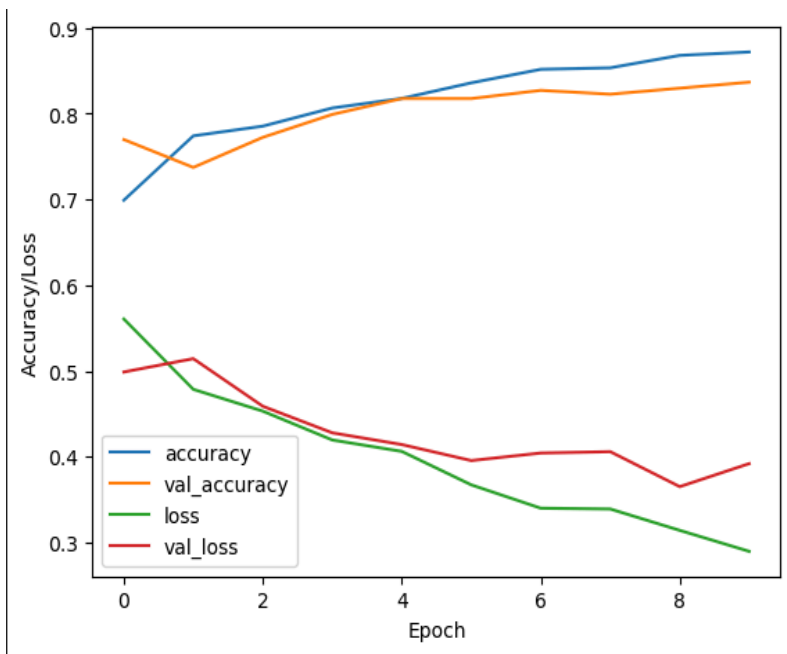
We portray the result of our model through various metrics such as accuracy, loss and comparing the predicted value of 5 pre-known and labeled images. Finally we compare our results with a more advanced model specifically designed to resolve the ISIC 2019 challenge.

## I. Learning in each epoch:

We can see that the neural network is learning over each epoch with increasing accuracy and decreasing loss over each epoch.

```
197/197 — 247s 1s/step - accuracy: 0.6170 - loss: 0.6577 - val_accuracy: 0.7081 - val_loss: 0.5489
Epoch 2/10
197/197 — 249s 1s/step - accuracy: 0.7414 - loss: 0.5164 - val_accuracy: 0.7533 - val_loss: 0.5043
Epoch 3/10
197/197 — 252s 1s/step - accuracy: 0.7836 - loss: 0.4692 - val_accuracy: 0.7604 - val_loss: 0.4767
Epoch 4/10
197/197 — 251s 1s/step - accuracy: 0.7903 - loss: 0.4454 - val_accuracy: 0.7820 - val_loss: 0.4416
Epoch 5/10
197/197 — 252s 1s/step - accuracy: 0.8099 - loss: 0.4112 - val_accuracy: 0.7980 - val_loss: 0.4322
Epoch 6/10
197/197 — 254s 1s/step - accuracy: 0.8215 - loss: 0.3891 - val_accuracy: 0.8050 - val_loss: 0.4369
Epoch 7/10
197/197 — 239s 1s/step - accuracy: 0.8249 - loss: 0.3785 - val_accuracy: 0.8094 - val_loss: 0.4069
Epoch 8/10
197/197 — 237s 1s/step - accuracy: 0.8294 - loss: 0.3707 - val_accuracy: 0.8209 - val_loss: 0.4122
Epoch 9/10
197/197 — 239s 1s/step - accuracy: 0.8376 - loss: 0.3447 - val_accuracy: 0.8196 - val_loss: 0.3920
Epoch 10/10
197/197 — 256s 1s/step - accuracy: 0.8618 - loss: 0.3131 - val_accuracy: 0.8260 - val_loss: 0.3866
```

## II. Accuracy and loss plot:



As shown the model begins with a very high loss of greater than 60% which is reduced over time by learning.

### III. Prediction of unseen images:

We pulled a few images that the model had not seen from google and asked the model to predict the outcomes for them. The model was correct around 80% of the time, which validates the above metrics.

Example, here we test the model with 4 images, 2 of each cancer.



Melanoma1.jpg



Melanoma2.jpg



Bcc1.jpg



Bcc2.jpg

For these 4 images, the model predicts the following:

```
1/1 _____ 0s 18ms/step
1/1 _____ 0s 22ms/step
1/1 _____ 0s 17ms/step
1/1 _____ 0s 18ms/step
Prediction for the melanoma1 image: [[1.]]
Prediction for the melanoma2 image: [[0.99998367]]
Prediction for the bcc1 image: [[0.54928523]]
Prediction for the bcc2 image: [[0.96522427]]
```

The first melanoma image, it certainly predicts the image of Melanoma which is correct.

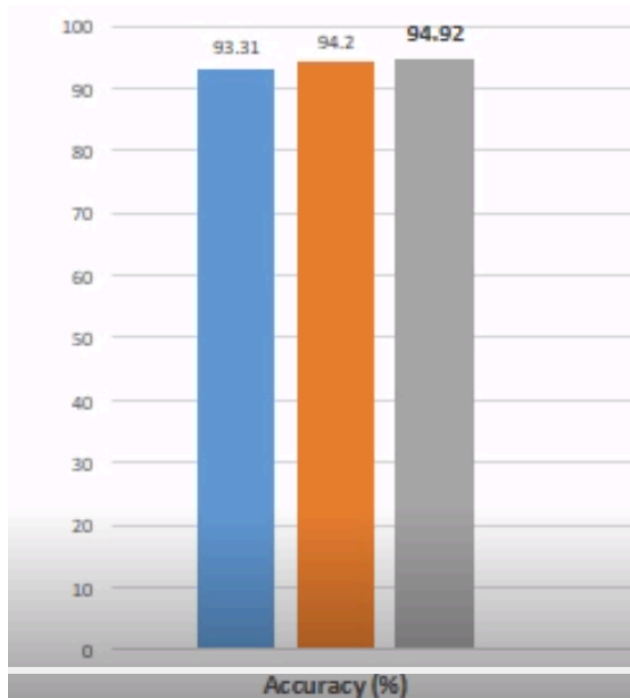
The second melanoma image, it predicts as Melanoma .

The first bcc image, it predicts correctly as bcc as the value is close to 0.5.

The second bcc image, it predicts incorrectly as melanoma while it is bcc.

#### IV. Comparing the results with an advanced model:

The model designed by combination of CNN and Transfer learning mentioned here: <https://ieeexplore.ieee.org/document/9121248> shows an accuracy of 94% which is higher than what we were able to achieve with a simple CNN model.



We understand that fine tuning and improving the final percentages of accuracy are much more difficult and require more precise optimizations.

## 4. Image segmentation

This section explores the application of image segmentation using a pre-trained Mask R-CNN (Region-based Convolutional Neural Network) model obtained from torchvision.models. With the growing exploration of Artificial Intelligence, there are several open source models that we can utilize and fine tune for our needs. Specifically, this model, Mask R-CNN with a ResNet-50 backbone and feature pyramid network (FPN), is loaded with pre-trained weights, enabling us to leverage its learned features and capabilities for segmenting images.

### I. Methodology:

1. To load the pre-trained data, we utilized the

`torchvision.models.detection.maskrcnn_resnet50_fpn` function with the `pretrained=True` argument, which automatically fetched the pre-trained weights for the model. This approach allows us to initialize the model with knowledge gained from extensive training on large-scale datasets, enhancing its segmentation performance.

2. To prepare the model for inference, we invoke the `model.eval()` function, ensuring that it operates in evaluation mode for accurate and efficient segmentation results.

```
# get the pretrained model from torchvision.models
# Note: pretrained=True will get the pretrained weights for the model.
# model.eval() to use the model for inference
model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)
model.eval()
```

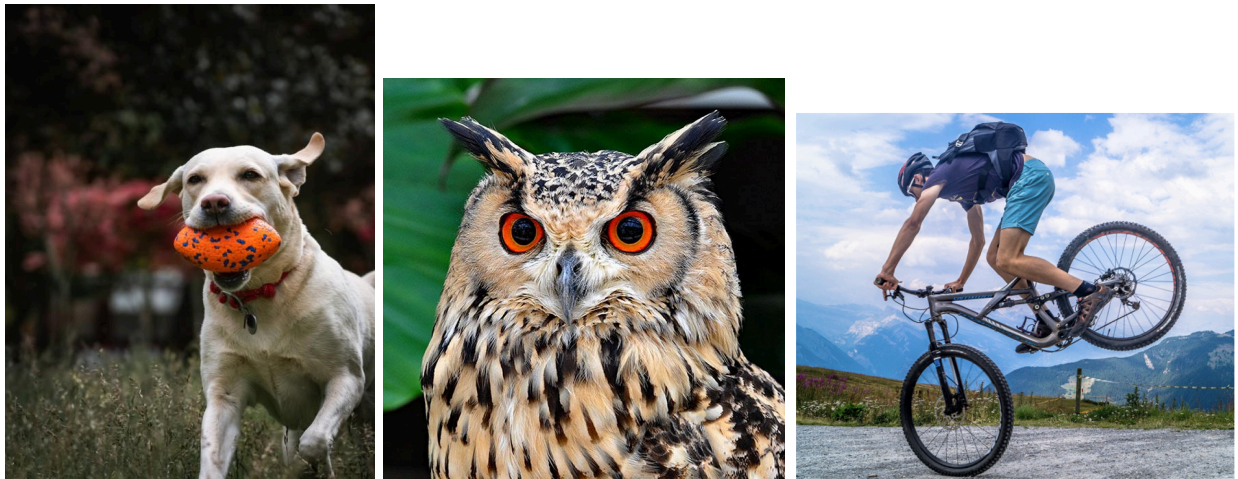
Through this report, we aim to provide insights into the process of utilizing pre-trained models for image segmentation and highlight the outcomes achieved through this approach.

3. We conducted inference on three images (dog, owl, cycle) using a pre-trained Mask R-CNN model. The images were transformed into tensors using PyTorch's transforms for prediction. This process allowed us to obtain segmentation predictions for each image, demonstrating the model's ability to detect and segment objects accurately.

```
# Running inference on the image
transform = T.Compose([T.ToTensor()])
img_tensor = transform(img)
pred = model([img_tensor])

img_tensor2 = transform(img2)
img_tensor3 = transform(img3)
pred2 = model([img_tensor2])
pred3 = model([img_tensor3])
```

4. For the images, we used these sets of images that the model had never seen before:

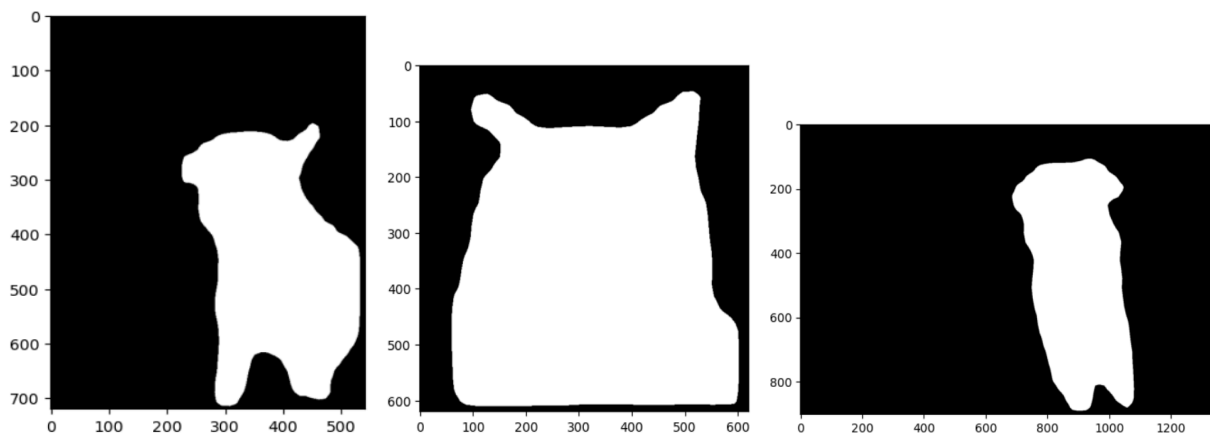


5. The code applies a threshold of 0.5 to segmentation masks derived from a pre-trained Mask R-CNN model on three images. This thresholding converts the masks into binary format, with values above 0.5 becoming 1 and others 0. Subsequently, the 'person' class masks are displayed using Matplotlib, and random colors are applied using a custom function

(random\_colour\_masks), illustrating the segmented 'person' areas with colored masks for each image.

```
# We will keep only the pixels with values greater than 0.5 as 1, and set the rest to 0.
masks = (pred[0]['masks']>0.5).squeeze().detach().cpu().numpy()
masks.shape
masks2 = (pred2[0]['masks']>0.5).squeeze().detach().cpu().numpy()
masks2.shape
masks3 = (pred3[0]['masks']>0.5).squeeze().detach().cpu().numpy()
masks3.shape
```

## II. Output:

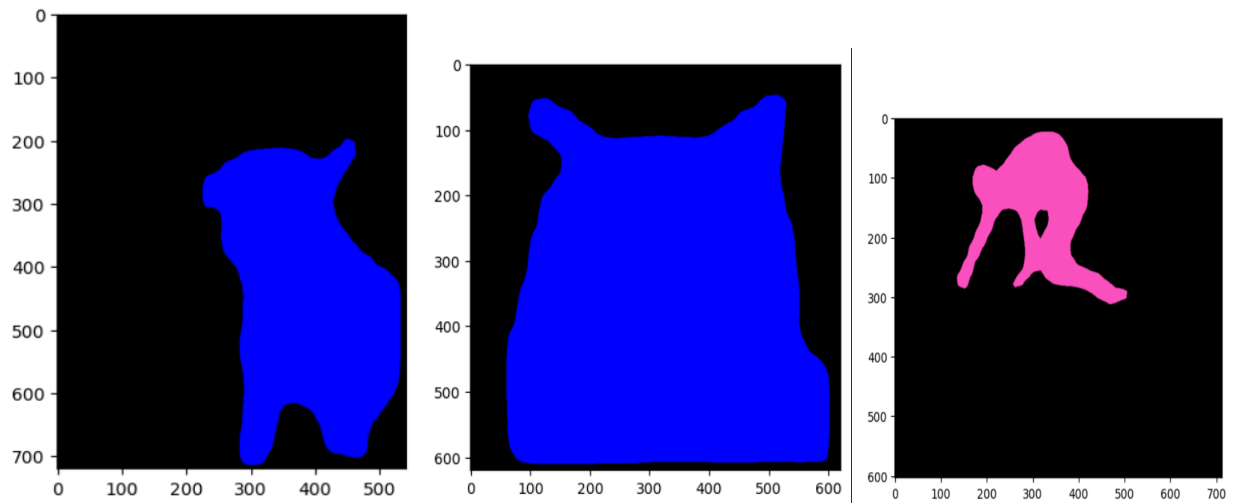


Now, We are coloring the 'person' masks using the 'random\_colour\_masks' function and visualizing them using Matplotlib's 'imshow' function, displaying the colored masks for each of the three images.

Subsequently, the 'person' class masks are displayed ,illustrating the segmented 'person' areas with colored masks for each image.

```
# Let's color the `person` mask using the `random_colour_masks` function
mask1 = random_colour_masks(masks[0])
plt.imshow(mask1)
plt.show()
mask2 = random_colour_masks(masks2[0])
plt.imshow(mask2)
plt.show()
mask3 = random_colour_masks(masks3[0])
plt.imshow(mask3)
plt.show()
```





6. We are finally blending the original images with their respective colored 'person' masks using OpenCV's `addWeighted` function to create visually enhanced images highlighting the segmented 'person' areas, and then displaying the blended images using Matplotlib.

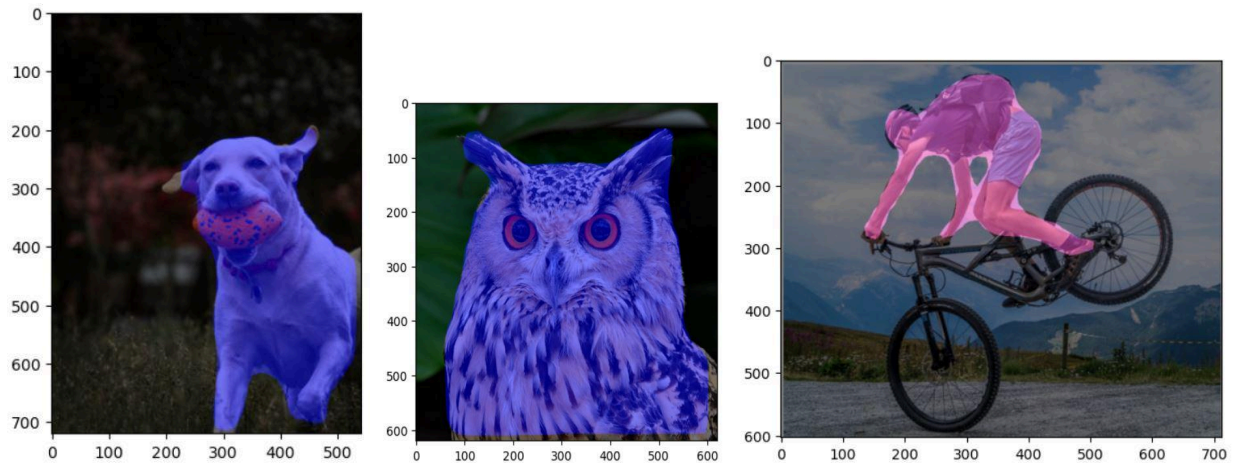
```
# Let's blend the original and the masked image and plot it.
blend_img = cv2.addWeighted(np.asarray(img), 0.5, mask1, 0.5, 0)

plt.imshow(blend_img)
plt.show()
blend_img2 = cv2.addWeighted(np.asarray(img2), 0.5, mask2, 0.5, 0)

plt.imshow(blend_img2)
plt.show()
blend_img3 = cv2.addWeighted(np.asarray(img3), 0.5, mask3, 0.5, 0)

plt.imshow(blend_img3)
plt.show()
```





### III. Results:

We obtain predictions for an input image with specified thresholds, and visualize the segmentation results by overlaying colored masks, drawing bounding boxes, and adding class labels, showcasing the model's capabilities for object detection and segmentation. One more interesting thing to notice is that since we had not removed all the pictures of cycle from the trained section of the model, the person was well segmented but it could not detect the cycle by it's own.

## 5. Conclusion

In conclusion, our journey through building a classification model from scratch using TensorFlow provided valuable insights into the inner workings of CNNs and the iterative process of model refinement. While achieving a commendable 80% accuracy, we acknowledge the potential for further optimization highlighted by the comparison with a more advanced model. This underscores the importance of continuous improvement and exploration of optimization techniques in pursuit of higher accuracy levels.

Moreover, the second part dealing with utilization of pre-trained models, exemplified by our exploration of object segmentation with PyTorch Mask R-CNN ResNet-50-FPN, unveiled the significance of both object identification and classification. Leveraging pre-trained models not only provided practical solutions but also presented vast learning opportunities in model utilization and fine-tuning.