Assignment 1

Ahmed Sharshar
22010334
ahmed.sharshar@mbzuai.ac.ae

1. Introduction

The importance of the medical imaging field lies in analyzing images and extracting useful information from them to identify diseases, treatment, or follow-up. Despite the huge development of deep learning today and its ability to address many natural problems, some traditional methods are still effective.

In this assignment, we will use both methods, hand-crafted algorithms and deep learning models, for different purposes as:

1. Use hand-crafted algorithms to separate the spin vertebra to make it clearer, then identify each vertebra by putting a box around it in 2d and 3d.
2. Use deep learning models to classify x-rays images of lungs to healthy and tuberculosis.

2. Methodology

This assignment is divided into Two main tasks as following:

2.1. Task1

In this task, we highlighted the importance of using non-machine learning models for segmentation.

2.1.1 Task 1.1

First, we find the optimal intensity window for the spine vertebrae by visualizing the result when choosing the best range. We first loaded the file and got the middle slice from each orientation (Sagittal, Coronal, Axial).

To get the best possible values, we first used Slicer to help in visualizing the best window threshold. Then we applied a simple threshold from the upper and lower bounds for binary segmentation so that the pixel that falls in this range takes a value of 1 and the remainder is 0. Also, the resulting images must be rotated 90 degrees counterclockwise to be well visualized.

The code for separate the spine and visualizing it in the mid-sagittal view as following:

```
#load data
data, header = nrrd.read('CTA-Abdomen.nrrd')

# get the index of the mid-slice of each view
mid_sagittal = data.shape[0] // 2
mid_coronal = data.shape[1] // 2
mid_axial = data.shape[2] // 2

# get the data of the mid-slice of each view
mid_sagittal_img = data[mid_sagittal , :, : ]
mid_coronal_img = data[:,mid_coronal , :]
mid_axial_img = data[: :, mid_axial]

#put a window threshold
img_threshold_high = 200
img_threshold_low = 90
image =  data[mid_sagittal , :, : ]
img = np.zeros(image.shape)
img[(image <= img_threshold_high) & (image >
    img_threshold_low)] = 1
plt.imshow(np.rot90(img), cmap='gray')
plt.axis('off')
```

2.1.2 Task 1.2

There are many different methods that can be used to allocate objects inside an image using hand-crafted algorithms that do not depend on large data at all. From these methods, Morphological operations and Contour detection.

Morphological operations are a set of image-processing techniques that operate on the shape and structure of an image [5]. These operations are based on the shape, size, and orientation of the structuring element or kernel, which is used to probe the input image. Morphological operations are widely used in various applications, including image segmentation, object detection, and feature extraction.

Some of the commonly used morphological operations include:

1. Erosion: This operation removes the pixels at the edges of an object in an image. It is used to remove small objects from an image, separate two connected objects, and to reduce the thickness of lines.
2. Dilation: This operation adds pixels at the edges of an object in an image. It is used to fill gaps, connect broken lines, and to thicken lines.

3. Opening: This operation is a combination of erosion followed by dilation. It is used to remove small objects and smooth larger objects' boundaries.
4. Closing: This operation is a combination of a dilation followed by erosion. It is used to fill gaps and to connect broken lines.

Contour detection is a non-ML method used in image processing and computer vision to identify and extract the boundaries of objects in an image. Contours are essentially the boundaries of objects that have a continuous area of the same intensity or color in an image. The contour detection algorithm works by identifying the changes in intensity or color of adjacent pixels and grouping them together to form the outline of an object.

The Suzuki algorithm, also known as the connected-component labeling algorithm, is a contour detection algorithm implemented in OpenCV [4].It is a two-pass algorithm that works by assigning labels to connected pixels based on their adjacency relationships. The algorithm first scans the image to determine the connectivity between pixels and assigns provisional labels to the connected components. In the second pass, the algorithm refines the provisional labels to eliminate any ambiguity that may arise from overlapping components. The Suzuki algorithm is a fast and efficient way of detecting contours in digital images.

To implement this, first, we needed to normalize the image to be between 0 and 255 and cast it into uint8.

```
min_val = np.min(mid_sagittal_img)
max_val = np.max(mid_sagittal_img)

# Shift the pixel values so that the minimum
    value becomes zero
img_shifted = mid_sagittal_img - min_val

# Scale the pixel values so that the maximum
    value becomes 255
img_scaled = img_shifted * 255.0 / (max_val -
    min_val)

# Convert the pixel values to uint8 type
image = np.uint8(img_scaled)
image =  cv2.rotate(image, cv2.
    ROTATE_90_COUNTERCLOCKWISE)
```

To enhance the contrast and makes the image easier to analyze, we performed histogram equalization on the input grayscale image. We used a threshold to highlight the spine and opened and closed morphological operations with kernel = (2,2). We got contours from the resulting image and set a threshold for the size of the contour to eliminate any wrong contours. Then we drew a box that used contour points around the spine and masked anything else to 255. The corresponding code is as follows:

```
# Define a function called image_mask that takes
    an input image as argument
def image_mask(image, threshold_func):

    # Declare global variables for threshold,
    output, and mask
    global thresh, output, mask

    # Apply binary thresholding to the input
    image using a separate function called
    binary_threshold
    thresh = threshold_func

    # Apply morphological operations to the
    binary image to remove small objects and fill
     in gaps
    kernel = cv2.getStructuringElement(cv2.
    MORPH_ELLIPSE, (2, 2))
    opened = cv2.morphologyEx(thresh, cv2.
    MORPH_OPEN, kernel)
    closed = cv2.morphologyEx(opened, cv2.
    MORPH_CLOSE, kernel)

    # Find contours in the binary image using cv2
    .findContours
    contours, hierarchy = cv2.findContours(closed
    , cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Draw the contours on the original image and
     create a mask with the same shape as the
    input image
    for i in range(len(contours)):
        # Draw the maximum contour only if its
    area is greater than 300 pixels
        if cv2.contourArea(contours[i]) > 300:
            rect = cv2.minAreaRect(contours[i])
            box = cv2.boxPoints(rect)
            box = np.int0(box)
            mask = 255 * np.ones(image.shape, np.
    uint8)
            mask = cv2.drawContours(mask,
    contours[i], -1, (0, 0, 255), 2)
            [mm, mn] = image.shape
            for rr in range(mm):
                for cc in range(mn):
                    if cv2.pointPolygonTest(
    contours[i], (rr, cc), False) >= 0:
                        mask[cc][rr] = image[cc][
    rr]

    # Return the resulting mask
    return mask
```

Then we need to draw a box around each vertebra. for that, we used the same method that we used previously to separate the spine. We got the resulting image (spine) and filtered by Laplacian filter, then used the Sobel filter in the y direction to highlight the horizontal edges. We then used contours to get the contours around each vertebra and drew a box around each one. We used the contour area to control the sizes of the displayed boxes and eliminate smaller and larger boxes. The corresponding code is as follows:

```
def get_contours (mask):

    global out, sobely , laplacian
    # Apply the Laplacian filter to the mask
    image
    laplacian = cv2.Laplacian(mask,cv2.CV_64F)
```

```
    # Apply the Sobel filter to the mask image to find edges in the x-direction
    sobely = cv2.Sobel(mask,cv2.CV_8UC1,0,1,ksize=3)

    # Find contours in the binary image using the Sobel edge detection
    contours, hierarchy = cv2.findContours(sobely, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Create a copy of the original mask image to draw contours on
    out=mask.copy()

    # Loop through each contour and draw a bounding box around it if it meets certain criteria
    counter=0
    for i in range(len(contours)):
        # Check if the contour area is between 150 and 1500 pixels
        if cv2.contourArea(contours[i])>150 and cv2.contourArea(contours[i])<1500  :
            # Get the minimum area rectangle that encloses the contour
            rect = cv2.minAreaRect(contours[i])
            box = cv2.boxPoints(rect)
            box = np.int0(box)
            # Draw the bounding box on the copy of the mask image
            cv2.drawContours(out,[box],0,(0,0,255),2)
            counter+=1

    print("The number of boxes is:" , counter)
    return out
```

Finally, we tried to make a 3d box around each vertebra; even if it did not work well, we would explain our methodology, the results, and why it did not produce the desired output.

To start easier, instead of getting all 21 slices (mid slice + 10 before + 10 after), we started with only three (mid slice + one before + one after); we normalized each slice to be from 0 to 255 and then apply the same two functions $image - mask(image)$, $get - contours(mask)$ to apply segmentation for each slice, then we save slices locally to load them later.

We load imread collection from io library to load the image as a 3d collection and convert the images to gray scale. We then used a function to try to merge these slices into a 3d volume; the function is as follows:

```
# Use marching cubes algorithm for volumetric rendering
def plot_3d(threshold, elev, azim):
    # Use marching cubes algorithm for volumetric rendering
    verts, faces, normals, values = measure.marching_cubes(gray_images, threshold)

    # Visualize the 3D image using Matplotlib
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection="3d")
    ax.plot_trisurf(verts[:, 2], verts[:, 0], faces, verts[:, 1], shade=True, color="none", alpha=0.3, edgecolor="none")
    ax.view_init(elev=elev, azim=azim) #Adjust the viewing angle
    plt.show()
```

2.2. Task2

In this task, We try to build a model for the binary classification of health and tuberculosis(TB) lungs using TBX11 dataset.

The TBX11K dataset comprises of 11200 X-ray images which have been annotated with bounding boxes to indicate areas affected by tuberculosis (TB). All the images are 512x512 in size. This dataset includes five categories: Healthy, Sick but Non-TB, Active TB, Latent TB, and Uncertain TB. The dataset is divided into three sets: training, validation, and testing, which consist of 6600, 1800, and 2800 X-ray images, respectively. [3]

Here we will only work with two classes (Healthy and TB) for binary classification. There are 3800 healthy x-ray images and 800 TB x-rays.

2.2.1 Task 2.1

In this task, we need to split the data into training and testing based on the ID of the patients in each class. First, we loaded the two classes and separated them into two data frames, one for each class, where the data frame contains the file path to each image and the label of the image. These file paths are the same except for the last part of it, which contains the image name. Each image is named by the patient id that we need to sort. For that, we used the file path column to sort the data frames in ascending order. We got the first 20% indices of each dataframe and chose them from each class as testing and the rest as training. Table 1 summarizes the range of each csv file.

Table 1. The Range and Number of Samples for Each File

File	Start	End	# of Samples
Healthy Testing	h0001	h0993	760
Healthy Training	h0995	h5000	3040
TB Testing	tb0003	tb0248	160
TB Training	tb0250	tb1199	640

2.2.2 Task 2.2

In this part, we tried to design a deep learning model to classify the data we got in task 2.1. First, we concatenated the training two data frames together so the testing data. Then, we used them to load the images in tensor form using pytorch.

After preparing the data, we chose the model. As the data is not complicated and the number of images is not too much, then to avoid overfitting, we chose a simple model that can achieve good performance regarding accuracy and speed. To respect the year of sustainability and afford a lightweight model that does cont consume much power for an easy task , we selected EfficientNet-Lite0 . [8]

EfficientNet-Lite0 is designed to be a smaller and more efficient version of the original EfficientNet architecture, with fewer parameters and faster inference times, making it suitable for deployment on resource-constrained devices such as mobile phones and embedded systems.

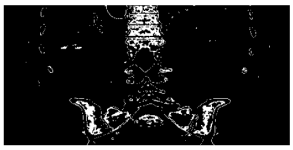
EfficientNet-Lite0 achieves this efficiency by using a combination of techniques such as depthwise separable convolutions, squeeze-and-excitation modules, and mobile inverted bottleneck (MBConv) blocks. It has about 3.7 million parameters, which is much smaller than the original EfficientNet but still achieves high accuracy on image classification tasks.

As the dataset is relatively small, we choose the pre-trained model and fine-tune it with our dataset. We also try to fix the seed for all experiments for reproducibility. We used batch size = 32 and Adam optimizer with learning rate = 1e-5, which is low to suit the fine-tuning task.

We did some pre-processing of the images to increase their quality. From these pre-processing methods are data augmentations, a technique used to increase the size of a dataset by applying transformations to the original data. These transformations can be various types of modifications to the input data, such as cropping, rotating, flipping, adding noise, adjusting brightness or contrast, and more. The goal of augmentations is to improve the robustness and generalization ability of the trained models by exposing them to a wider variety of variations and distortions of the original data. [1]

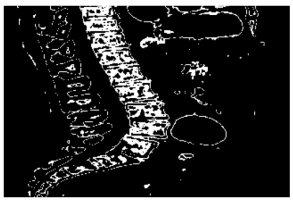
In data augmentations, we need to be careful in choosing the suitable set of augmentations depending on the type of the dataset. Given the nature of the chest xrays data here, some augmentation choices may be better than others; for example, the orientation of the image here is not an important factor in determining the disease, so the addition of rotation and flipping may be useful for generalization. For that, we choose RandomHorizontalFlip(p=0.1) and RandomRotation(degrees=30).

Also, improving the quality of the image through brightness, contrast, and saturation may be effective in showing more details about the image, as well as training the model to recognize the disease, even with the difference of these factors, which are very likely to differ between images and some of them due to the nature of the x-ray images.Thus we added ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2). We did not choose high values to avoid destroying the images or ruining their features.



(a) Threshold of Axial View

(b) Threshold of Coronal View



(c) Threshold of Sagittal View

Figure 1. Threshold Output of Different Views

However, theoretically, shearing and cropping can hurt the model. Cause shearing can cause in changing The size and shape of the affected part, which can be an important feature for it. Cropping can remove an important part of the image, which can be affected, leading to unrealistic results. Thus we did not use such methods.

We used Focal loss, a loss function used in machine learning designed to address class imbalance problems; the idea behind the focal loss is to down weight the contribution of well-classified examples during training to focus more on the harder-to-classify examples [2]. This is achieved by introducing a modulating factor to the cross-entropy loss, which reduces the contribution of easy examples and increases the contribution of harder examples.

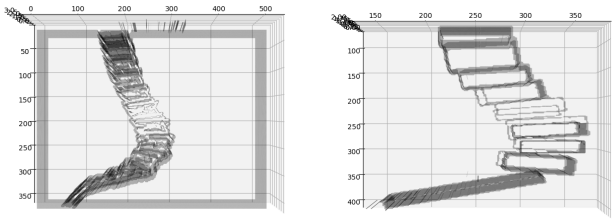
Beside focal loss, we also used Weighted Cross-Entropy [6]. It is a variant of the Cross-Entropy Loss function used in classification tasks, particularly when dealing with imbalanced datasets where some classes have fewer samples than others. This variant assigns different weights to each class based on their frequency or importance, allowing the model to pay more attention to the minority classes during training. By increasing the weight for the minority classes, the loss function can give them more importance and encourage the model to learn them better. The choice of weighting strategy depends on the specific problem and goals of the model.

3. Results

3.1. Task 1

The first part of the task, windowing, is an empirical solution as we do not need to use an algorithm to perfectly separate spine vertebrae at this step. Therefore we tried our best to get the best results. The best window value is from 90 to 120. These values are for the original images, which range from almost -1400 to 1400. Figure 1 shows the output of the mid-slice of each view for the same threshold.

Figure 2. Separated Spine



(a) 3d Spine (b) 3D Vertebrae Boxes

Figure 3. 3D Simulation of Spine and Boxes Around Vertebrae

Next, we used the contours to separate the spine alone, and the resulting Figure is shown in 2. It shows the spine separated well with a masked background.

We used the result in separated spine in 2 as an input to the contour algorithm. Figure 3 shows the resulting boxes around each vertebra. It shows a good performance for the upper nine vertebrae; however, the lumbar shows a bad performance as the algorithm can not get its vertebrae corrected due to its lack of clarity in the original picture. The number of boxes = 10, 8 out of 9 for the large vertebrae and two for all lumbar, as there should be five vertebrae in this place (by eyes inspection).

Then we moved to 3d boxes. Figure 3 shows the 3d boxes and spine for the three slices we selected before (mid-slice, one before, one after). The results show inconsistency and shades. The same boxes in the three slices do not overlap because the small difference between each slice position leads to a difference in box dimensions and position.

Also, hand-crafted algorithms have many parameters that highly affect the performance of the code. These parameters can differ from one slice to another depending on the light intensity, position, and other factors that demand a fine-tuning of these parameters. This means that the hand-crafted algorithm may not be able to generalize well, which leads to inconsistency in the number of boxes and differences in dimension for the same box in different slices.

In Figure 3, there are two separated images, (a) showing the 3d image of the spine. It is not a perfect 3d but has a depth of 3 (number of slices). Figure 3 (b) represents 3D boxes of the vertebrae; this image has a shadow that shows the differences in the dimensions of the boxes.

Figure 3 are two separate images instead of one 3d image of the spine and boxes around each vertebra because the method we used accepts only gray images. The white part does not show in the 3d image, so if the spin is black and the box is white, only the spine appears and vice versa. If both are black, the boxes do not appear well. We also tried to make the method accept RBG, but it did not work.

Many limitations prevented us from achieving the task in a good way. Some are:

1. Lack of knowledge of 3D representation: We do not know how well we can use a number of images to make a good 3D model, and despite the attempt, there are better methods that should have been used, but we could not like Mesh.

2. Inconsistency: hand-crafted algorithms sometimes do not give good results in all images. Because we used the same code on all slices with the same parameters as the threshold, some slides showed more or fewer boxes or even in different positions.

3. Boxes Matching: As there is inconsistency in box generations and dimensions, we need to unify the dimensions of all boxes. The simplest way to accomplish this is by matching each box to its nearest box from other slices, starting from the middle one. Then reshape the box dimension to match the middle slice box and so on. If there are any extra boxes, remove them. More accurate approaches could be done (taking the mean of the same box dimensions and re-scale on it and so on). Unfortunately, even the simplest algorithm could not work well with us.

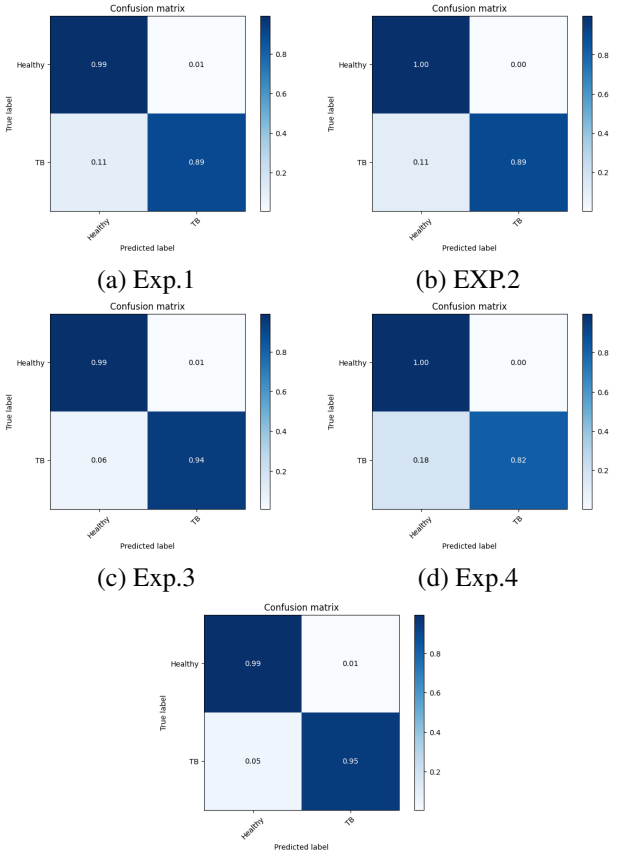
3.2. Task 2

Table 2 shows all five experiments we did, the details of each experiment, and the results. Here we will explain each experiment along with its result. We normalized the training and testing datasets for all the experiments to 224*224 to reduce the noise. All experiments ran for 28 epochs for a fair comparison.

The first experiment is the base one; we imported the pre-trained model and ran it without any dropout or data augmentations. The Model overfitted rapidly; the accuracy was good; however, the F1 score for both classes was not fair. That is because of the data imbalance, which made the model learn better for the major class (Healthy class). Figure 4 shows the confusion matrices for all experiments and explains each class's model performance. Because of these results, there was still room for improvement.

Table 2. The Range and Number of Samples for Each File

Experiment	Augmentations	Dropout	Loss Function	GFLOPS	Accuracy	Healthy F1-Score	TB F1-Score
Experiment 1	None	None	Cross-Entropy	0.354	97.60%	98.56%	92.86%
Experiment 2	None	0.6	Cross-Entropy	0.353	98.04%	98.83%	93.08%
Experiment 3	Both Classes	0.6	Cross-Entropy	0.353	98.36%	99.02%	95.27%
Experiment 4	TB Class	0.6	Cross-Entropy	0.353	96.63%	98.00%	89.42%
Experiment 5	Both Classes	0.7	Weighted Cross-Entropy + Focal Loss	0.353	98.69%	99.21%	96.20%



(a) Exp.1 (b) EXP.2

(c) Exp.3 (d) Exp.4

(e) Exp.5

Figure 4. Confusion Matrices of All Experiments

In experiment 2, We tried to avoid overfitting; [7] simply started by adding a dropout with 0.6; the dropout value is relatively high because the model was greatly overfitting. The results table shows some improvement in the model accuracy and a little in each class; however, it did not solve the imbalance problem; both F1 score and Figure 4 (b) shows that the results of each are not balanced and still the model perform badly on TB class. The FLOPs showed to be also mentioned, as from this experience onwards, Giga-FLOPs (GLOPS) has decreased a little from the original model. The reason is adding dropout that decreases the number of computations a bit.

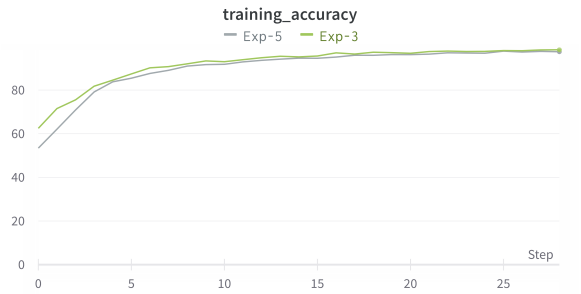
From experiment 3 onward, we will try to tackle the problem of imbalance results. Therefore we applied data augmentations with the parameters discussed earlier to both classes during learning. This shows a good improvement in both classes, especially TB class. The reason is that adding augmentations increases the number of images and variety, preventing the model from overfitting. Figure 4 (c) shows that the minor class (TB) increases by 5% from the last experiment.

In experiment 4, we tried to answer a question, is adding data augmentations to the minor class only increase its chance of getting better? The answer is no; after applying augmentations to the minority class only, the results became bad again and were worth it for the target class. It can be shown from F1 score of TB class that dropped around 6% from the previous experiment and also the confusion matrix in Figure 4 (d), which reached its worst value for this experiment among all the experiments. The reason is that when the model got only augmented images from the minor class, the images differed. Because they were few, the model did not have the ability to learn its feature, then became much easier for it to recognize them as the major class, which happened in Figure 4 (d).

There are still a lot of adjustments that can be made, but since we were controlled by only five experiments, we made two adjustments together in the fifth experiment. We used focal loss and class weights to address the imbalanced data problem. We set the class weight into cross-entropy functions with ratios inverse proportional to the number of samples in each class. As the healthy class contains 3800 samples and TB contains only 800, we set the class weight ratios to be 0.2, and 1 for the healthy and TB classes, respectively. We used data augmentations for both classes and increased the dropout to 0.7 for more regularization.

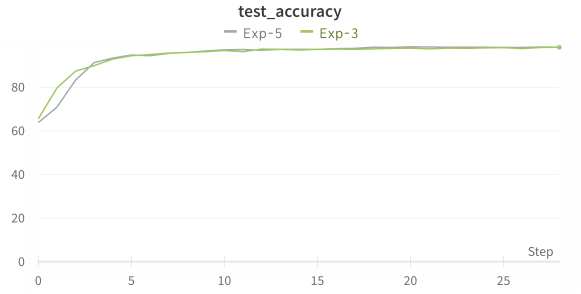
The results of experiment 5 are impressive. It can be considered the best experience. The total accuracy increased to its maximum thanks to the increase in F1 score in both classes, especially TB class, which is 1% higher than its nearest competitor. Figure 4 (e) shows both classes' outstanding and fair performance.

Figure 5 shows the accuracy and loss for training and testing during 28 epochs for our two best experiments, ex-



(a) Train Accuracy

(b) Train Loss



(c) Test Accuracy

(d) Test Loss

Figure 5. Learning Curves For Training & Testing

periments 3 and 5. The loss curves show how focal loss helped in more stable learning and lower loss in training and testing. For the accuracy curves, as the differences between both experiments are slightly small, they won't be recognizable in the graph. However, both curves show a table learning process. It should be mentioned that the test can start with higher accuracy than training. That is because the model is pre-trained. However, the training accuracy eventually caught and became higher than the testing.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, 2012. 4

[2] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. pages 2999–3007, 10 2017. 4

[3] Yun Liu, Yu-Huan Wu, Yunfeng Ban, Huifang Wang, and Ming-Ming Cheng. Rethinking computer-aided tuberculosis diagnosis. pages 2646–2655, 2020. 3

[4] OpenCV. Contour features. Accessed 2023. 2

[5] Jean Serra. Morphological operations and image filtering. *Journal of Visual Communication and Image Representation*, 3(3):210–217, 1982. 1

[6] Kanika Sharma and Rajeev Srivastava. Weighted cross-entropy based imbalance compensation for handwritten digit recognition. *Expert Systems with Applications*, 42(7):3666–3673, 2015. 4

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. 6

[8] Mingxing Tan and Quoc V Le. Efficientnet-lite: A compact deep neural network for mobile devices. pages 6638–6646, 2020. 4