Lung cancer detection from CT scan slices using AI

Author: Marcus Connelly

Introduction

The IQ-OTH/NCCD lung cancer dataset provides a directory of lung cancer CT scan slices in jpeg format. My goal was to train an AI so that it could detect cancer in CT scans. I have achieved this using a 2D Convolutional Neural Network and cleaning both directories (normal and malignant) of CT scan slices.

The Dataset

I tried many different datasets of lung CT scans many of which used DIICOT images which are used in medical imaging as these took special software to even view these type of images were difficult to work with. I found a dataset of lung CT scan images from IQ-OTH/NCCD which used jpeg slices of CT scans.

Viewing these images, I saw there were CT scan artifacts such as ring artefacts (the circular artefact at the bottom).

Sample image:

A close-up of a ct scan

Description automatically generated

These artifacts were not apart of the lungs but were part of the equipment used in the taking of the CT scan or an issue while the image being taken such as movement. Since these were not a part of the lungs, they made it difficult for the Neural Network to train and led to worse accuracy. It is also of note that the images are grey scale but do not get saved as greyscale but rather as rgb jpegs.

Fixing the Dataset

In order to fix the dataset, I made a programme to make a mask of the important part of the image( the lung) and clean the rest of any artefact which might affect the performance of the model.

Imports:

import os  
import numpy as np  
import cv2  
from scipy import ndimage  
from skimage import morphology, measure  
from sklearn.cluster import KMeans  
import scipy.fftpack as fftpack  
import numpy as np

Primary Method:

def remove\_noise(image):  
   
 if image.shape[0] != 512 or image.shape[1] != 512:  
   
 image = cv2.resize(image, (512, 512), interpolation=cv2.INTER\_AREA)  
  
 img1 = cv2.normalize(image,None,0,255,cv2.NORM\_MINMAX)  
 clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(4, 4))  
 img2 = clahe.apply(img1.astype("uint8"))  
 central\_area = img2[100:400, 100:400]  
 kmeans = KMeans(n\_clusters=2).fit(np.reshape(central\_area, [np.prod(central\_area.shape), 1]))  
 centers = sorted(kmeans.cluster\_centers\_.flatten())  
 threshold = np.mean(centers)  
 ret, thres\_img = cv2.threshold(img2, threshold, 255, cv2.THRESH\_BINARY\_INV)  
 img4 = cv2.erode(thres\_img, kernel=np.ones([4,4]))  
 img5 = cv2.dilate(img4, kernel=np.ones([13, 13]))  
 img6 = cv2.erode(img5, kernel=np.ones([8, 8]))  
 labels = measure.label(img6)  
 regions = measure.regionprops(labels)  
 good\_labels = []  
 for prop in regions:  
 B = prop.bbox  
 if B[2] - B[0] < 500 and B[3] - B[1] < 490 and B[0] > 17 and B[2] < 495:  
 good\_labels.append(prop.label)  
  
 mask = np.zeros\_like(labels)  
 for N in good\_labels:  
 mask = mask + np.where(labels == N, 1, 0)  
  
  
 contours, hirearchy = cv2.findContours(mask, cv2.RETR\_CCOMP, cv2.CHAIN\_APPROX\_SIMPLE)  
 internal\_contours = np.zeros(mask.shape)  
 external\_contours = np.zeros(mask.shape)  
 for i in range(len(contours)):  
 if hirearchy[0][i][3] == -1:   
 area = cv2.contourArea(contours[i])  
 if area > 518.0:  
 cv2.drawContours(external\_contours, contours, i, (1, 1, 1), -1)  
 external\_contours = img = cv2.dilate(external\_contours, kernel=np.ones([4, 4]))  
  
 mask = cv2.bitwise\_not(external\_contours.astype(np.uint8))  
 mask = cv2.erode(mask, kernel=np.ones((7, 7)))  
 mask = cv2.bitwise\_not(mask)  
 mask = cv2.dilate(mask, kernel=np.ones((12, 12)))  
 mask = cv2.erode(mask, kernel=np.ones((12, 12)))  
  
 img7 = img1.astype(np.uint8)  
 mask = mask.astype(np.uint8)  
 seg = cv2.bitwise\_and(img7, img7, mask=mask)  
 return seg

Word which I’m using seems to underline random parts in red but that is code if you need it for some purpose and here is a clearer image:

A screen shot of a computer screen

Description automatically generated

A computer screen shot of a program code

Description automatically generated

A computer screen with text

Description automatically generated

This method first goes through image preprocessing. Since the jpegs are size 512 x 512 it checks if the images are the correct size and if not it resizes them. It then normalizes each image to a range of 0-255. After this it does segmentation. Using thresholding and morphological operation it binarizes the image and removes noise and CT artefacts. Finally it uses masking to make a mask of the most important area, the lung, noise from the background is removed and the mask is applied the original image to remove all irrelevant pieces of the image from the final image which is then returned.

Secondary method:

A screen shot of a computer program

Description automatically generated

The final method takes a directory and applies the remove\_noise method to each image in the folder, outputting it to a new directory of cleaned images.

Sample cleaned image (malignant):

A close-up of a ct scan

Description automatically generated

Convolutional Neural Network

Code:

import os  
import numpy as np  
import cv2  
import tensorflow as tf  
from tensorflow.keras.regularizers import l2  
from keras.src.legacy.preprocessing.image import ImageDataGenerator  
from sklearn.model\_selection import train\_test\_split  
from PIL import ImageEnhance  
  
imageHeight = 256  
imageWidth = 256  
channels = 1  
  
def loadImagesFromDirectory(directory, label):  
 images = []  
 labels = []  
 for filename in os.listdir(directory):  
 if filename.endswith(".jpg"):  
 image = cv2.imread(os.path.join(directory, filename), cv2.IMREAD\_GRAYSCALE)  
 image = cv2.resize(image, (imageWidth, imageHeight))  
 images.append(image)  
 labels.append(label)  
 return np.array(images), np.array(labels)  
  
healthyImages, healthyLabels = loadImagesFromDirectory(r'C:\Users\Marcus Connelly\Downloads\noiseremovedhealthy', 0)  
malignantImages, malignantLabels = loadImagesFromDirectory(r'C:\Users\Marcus Connelly\Downloads\noiseremovedmalignant', 1)  
  
allImages = np.concatenate((healthyImages, malignantImages))  
allLabels = np.concatenate((healthyLabels, malignantLabels))  
  
allImages = allImages.reshape(-1, imageHeight, imageWidth, channels)  
  
xTrain, xTest, yTrain, yTest = train\_test\_split(allImages, allLabels, test\_size=0.2, random\_state=42, stratify=allLabels)  
  
xTrain, xVal, yTrain, yVal = train\_test\_split(xTrain, yTrain, test\_size=0.2, random\_state=42, stratify=yTrain)  
  
xTrain = xTrain / 255.0  
xVal = xVal / 255.0  
xTest = xTest / 255.0  
  
trainDataGen = ImageDataGenerator(  
 rotation\_range=2,  
 width\_shift\_range=0.1,  
 height\_shift\_range=0.1,  
 shear\_range=0.2,  
 zoom\_range=0.05,  
 horizontal\_flip=True,  
)  
  
valDataGen = ImageDataGenerator()  
testDataGen = ImageDataGenerator()  
  
trainDataGen.fit(xTrain)  
  
trainGenerator = trainDataGen.flow(xTrain, yTrain, batch\_size=64)  
valGenerator = valDataGen.flow(xVal, yVal, batch\_size=64)  
testGenerator = testDataGen.flow(xTest, yTest, batch\_size=64, shuffle=False)  
  
inputs = tf.keras.Input(shape=(imageHeight, imageWidth, channels))  
  
y = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel\_regularizer=l2(0.01))(inputs)  
y = tf.keras.layers.MaxPooling2D((2, 2))(y)  
y = tf.keras.layers.Dropout(0.3)(y)  
  
y = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel\_regularizer=l2(0.01))(y)  
y = tf.keras.layers.MaxPooling2D((2, 2))(y)  
y = tf.keras.layers.Dropout(0.3)(y)  
  
y = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel\_regularizer=l2(0.01))(y)  
y = tf.keras.layers.MaxPooling2D((2, 2))(y)  
y = tf.keras.layers.Dropout(0.3)(y)  
  
y = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel\_regularizer=l2(0.01))(y)  
y = tf.keras.layers.MaxPooling2D((2, 2))(y)  
y = tf.keras.layers.Dropout(0.3)(y)  
  
y = tf.keras.layers.Flatten()(y)  
  
y = tf.keras.layers.Dense(128, activation='relu', kernel\_regularizer=l2(0.01))(y)  
y = tf.keras.layers.Dropout(0.3)(y)  
  
outputs = tf.keras.layers.Dense(2, activation='softmax')(y)  
  
model = tf.keras.Model(inputs=inputs, outputs=outputs)  
  
  
model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0002), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])  
  
classWeight = {0: 1.25, 1: 0.8333}  
earlyStopping = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=6, restore\_best\_weights=True)  
model.fit(trainGenerator, epochs=65, validation\_data=valGenerator, callbacks=[earlyStopping], class\_weight=classWeight)  
  
loss, accuracy = model.evaluate(testGenerator)  
print(f"Test accuracy: {accuracy:.2f}")  
  
model.save('lung\_cancer\_model.h5')  
  
newHealthyDir = r'C:\Users\Marcus Connelly\Downloads\sample'  
newHealthyImages, newHealthyLabels = loadImagesFromDirectory(newHealthyDir, 0)  
  
newHealthyImages = newHealthyImages.reshape(-1, imageHeight, imageWidth, channels) / 255.0  
  
newLoss, newAccuracy = model.evaluate(newHealthyImages, newHealthyLabels)  
print(f"Accuracy on the new healthy dataset: {newAccuracy:.2f}")

Set up

I had my model reduce the size of the images as the images were 512 x 512 and I did not have enough RAM to work with so to fix this I resized the images to half the size 256 x 256 and had it view the images in greyscale instead of RGB. Next, I made a method which loaded the directories of jpeg images with the given labels as an array.

Loading and preprocessing

My dataset was loaded from two directories healthy and malignant. Using my loadImagesFromDirectory method they were loaded into two numpy arrays, one for the images and one for the labels for the images. They were then concatenated into an array containing all images and another containing all labels then the images were reshaped to fit my requirements. Next the data was split into training and testing sets. After that I normalized the data to only be in a pixel range of 0 to 255 as my images are grey scale (0 being black and 255 being white), normalizing to this specific range (8 bit, 2^8 being 255) helps the neural network work more effectively on images. I used keras ImageDataGenerator to create three different data generators to evaluate one being for training the others for validation and testing.

Dataset augmentation

A model performs better when it has large and varied dataset to work and learn patterns on, having a dataset which is large enough allows the model to learn, get good accuracy and avoid overfitting. One way of improving my data set was Dataset augmentation. Dataset augmentation allows you to increase the quantity of training data by applying augmentations which change the data in various way. I tried many different augmentations and found more augmentation didn’t always mean better, for example changing the brightness in any way or using more rotation variation led to much lower accuracy and lower ability to generalise to a new dataset. Through experimenting i ended up using low levels of; rotation range, width shift range, height shift range, shear range, zoom range and a horizontal flip which led to best overall performance. These were all available through keras ImageDataGenerator which I was using. I will show in the results the performance I was able to get and plan on experimenting with more complex Data Augmentation to try and improve results further.

Defining the model

My model has 4 main convolutional blocks with flatten layer, a dense block and a ouput layer for classification. These 4 blocks provided enough complexity for the model to learn without making it so complex it couldn’t pick up on larger more general patterns and then fail to generalise. The input layer gave the image dimensions. The 2D convolutional layers allowed the model to extract the main features from the image. In each convolutional block I made use of a dropout layer which allowed me to avoid overfitting. Dropping 30% of neurons means that the model cant pick up on a singular feature which distinguishes the two types of data and instead has to pick up on the more general patterns. The flatten layer condensed the final convolutional layer into a 1d vector to give to the fully connected layer. Next there was an output layer which used softmax activation with a dense layer of 2 neurons which provided the probabilities. Finally, Ill explain my choice of activations- REL and softmax. I used ReLU to introduce non-linearity to my 4 convolutional blocks. I used ReLU compared to other activations like sigmoid as it is very computationally efficient, and I am working with low RAM and it is generally used for deeper networks like mine which has many layers. I used softmax as it is commonly used for binary classification, as my network is used to determine healthy vs malignant which is a binary classification task this is ideal.

Compiling saving and evaluating the model

I experimented with different learning rates and got the best performance with a rate of 0.0002 when using a batch size of 64. I was limited with batch size because I was working with a ram of 8gb and this could be increased to decrease training time. I had it training for 65 epochs but it usually didn’t get to 65 before the stabilising valuation loss stopped it early due to lack of improvement. It gave feedback on loss and accuracy and I gave the model an unseen dataset to test accuracy on to see how well it could generalise to new data. I later added early stopping as a feature which reduced training time as it stopped the model automatically when it noticed that improvement had stalled.

Output and results

Epoch 1/65

C:\Users\Marcus Connelly\PycharmProjectsSADF\pythonProjectAI4\.venv\Lib\site-packages\keras\src\trainers\data\_adapters\py\_dataset\_adapter.py:122: UserWarning: Your `PyDataset` class should call `super().\_\_init\_\_(\*\*kwargs)` in its constructor. `\*\*kwargs` can include `workers`, `use\_multiprocessing`, `max\_queue\_size`. Do not pass these arguments to `fit()`, as they will be ignored.

self.\_warn\_if\_super\_not\_called()

20/20 ━━━━━━━━━━━━━━━━━━━━ 63s 3s/step - accuracy: 0.6269 - loss: 5.7028 - val\_accuracy: 0.6518 - val\_loss: 4.3842

Epoch 2/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 48s 2s/step - accuracy: 0.6936 - loss: 4.0154 - val\_accuracy: 0.7157 - val\_loss: 3.2787

Epoch 3/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 47s 2s/step - accuracy: 0.7163 - loss: 3.0460 - val\_accuracy: 0.6901 - val\_loss: 2.6822

Epoch 4/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 47s 2s/step - accuracy: 0.6963 - loss: 2.5177 - val\_accuracy: 0.6933 - val\_loss: 2.2354

Epoch 5/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.6862 - loss: 2.1423 - val\_accuracy: 0.7061 - val\_loss: 1.8958

Epoch 6/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 39s 2s/step - accuracy: 0.7363 - loss: 1.8235 - val\_accuracy: 0.6965 - val\_loss: 1.6893

Epoch 7/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 43s 2s/step - accuracy: 0.7107 - loss: 1.5971 - val\_accuracy: 0.6805 - val\_loss: 1.4957

Epoch 8/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 41s 2s/step - accuracy: 0.7235 - loss: 1.3839 - val\_accuracy: 0.6901 - val\_loss: 1.3242

Epoch 9/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7312 - loss: 1.2323 - val\_accuracy: 0.7125 - val\_loss: 1.1753

Epoch 10/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 39s 2s/step - accuracy: 0.7125 - loss: 1.1397 - val\_accuracy: 0.7125 - val\_loss: 1.0657

Epoch 11/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7098 - loss: 1.0577 - val\_accuracy: 0.6997 - val\_loss: 0.9953

Epoch 12/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 41s 2s/step - accuracy: 0.7152 - loss: 0.9432 - val\_accuracy: 0.6518 - val\_loss: 0.9525

Epoch 13/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 39s 2s/step - accuracy: 0.7234 - loss: 0.8746 - val\_accuracy: 0.7476 - val\_loss: 0.8502

Epoch 14/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 42s 2s/step - accuracy: 0.7177 - loss: 0.8612 - val\_accuracy: 0.7157 - val\_loss: 0.8069

Epoch 15/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 42s 2s/step - accuracy: 0.7395 - loss: 0.7825 - val\_accuracy: 0.7604 - val\_loss: 0.7584

Epoch 16/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7451 - loss: 0.7376 - val\_accuracy: 0.7093 - val\_loss: 0.7533

Epoch 17/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.7099 - loss: 0.7435 - val\_accuracy: 0.7252 - val\_loss: 0.7146

Epoch 18/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.7506 - loss: 0.6919 - val\_accuracy: 0.7188 - val\_loss: 0.7103

Epoch 19/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7159 - loss: 0.7143 - val\_accuracy: 0.7220 - val\_loss: 0.6699

Epoch 20/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7331 - loss: 0.6727 - val\_accuracy: 0.7348 - val\_loss: 0.6666

Epoch 21/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.7654 - loss: 0.6258 - val\_accuracy: 0.7540 - val\_loss: 0.6220

Epoch 22/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 39s 2s/step - accuracy: 0.7323 - loss: 0.6516 - val\_accuracy: 0.7636 - val\_loss: 0.6385

Epoch 23/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 41s 2s/step - accuracy: 0.7421 - loss: 0.6332 - val\_accuracy: 0.7572 - val\_loss: 0.6186

Epoch 24/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7437 - loss: 0.6361 - val\_accuracy: 0.7508 - val\_loss: 0.6181

Epoch 25/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7588 - loss: 0.6050 - val\_accuracy: 0.7125 - val\_loss: 0.6899

Epoch 26/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7323 - loss: 0.6334 - val\_accuracy: 0.7412 - val\_loss: 0.6592

Epoch 27/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7541 - loss: 0.6030 - val\_accuracy: 0.7668 - val\_loss: 0.6018

Epoch 28/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7695 - loss: 0.5978 - val\_accuracy: 0.7891 - val\_loss: 0.5580

Epoch 29/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 35s 2s/step - accuracy: 0.7746 - loss: 0.5803 - val\_accuracy: 0.7732 - val\_loss: 0.5514

Epoch 30/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7742 - loss: 0.6050 - val\_accuracy: 0.7859 - val\_loss: 0.5389

Epoch 31/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.7776 - loss: 0.5751 - val\_accuracy: 0.7732 - val\_loss: 0.6335

Epoch 32/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.7910 - loss: 0.5709 - val\_accuracy: 0.8179 - val\_loss: 0.5100

Epoch 33/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.7811 - loss: 0.5707 - val\_accuracy: 0.8051 - val\_loss: 0.5248

Epoch 34/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7776 - loss: 0.5910 - val\_accuracy: 0.8371 - val\_loss: 0.4948

Epoch 35/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 36s 2s/step - accuracy: 0.8178 - loss: 0.5337 - val\_accuracy: 0.8339 - val\_loss: 0.5202

Epoch 36/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 42s 2s/step - accuracy: 0.8108 - loss: 0.5584 - val\_accuracy: 0.8179 - val\_loss: 0.5013

Epoch 37/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 40s 2s/step - accuracy: 0.7777 - loss: 0.5576 - val\_accuracy: 0.8083 - val\_loss: 0.5553

Epoch 38/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.7876 - loss: 0.5587 - val\_accuracy: 0.8243 - val\_loss: 0.5078

Epoch 39/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 37s 2s/step - accuracy: 0.8053 - loss: 0.5412 - val\_accuracy: 0.8147 - val\_loss: 0.5074

Epoch 40/65

20/20 ━━━━━━━━━━━━━━━━━━━━ 38s 2s/step - accuracy: 0.8032 - loss: 0.5375 - val\_accuracy: 0.8275 - val\_loss: 0.5339

7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 333ms/step - accuracy: 0.8192 - loss: 0.5027

Test accuracy: 0.83

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 178ms/step - accuracy: 0.6667 - loss: 0.7921

Accuracy on the new healthy dataset: 0.67

I got a good accuracy from the model given the complexity of the task though its ability to generalise was less good as it had an accuracy decrease of 16 percent on the new dataset. More improvements could be made such as increasing augmentation to avoid overfitting allowing the model to understand the patterns better so get greater accuracy on unseen datasets from different sources.