**AI-Powered Terrain Assessment for autonomous Landing**

# <https://www.linkedin.com/in/supriyo-ghosh-2003s/>

# <https://www.linkedin.com/in/arijit-saha-pklm/>

“““

@authors: Supriyo Ghosh [ [supriyoghosh27125@gmail.com](mailto:supriyoghosh27125@gmail.com) ], Arijit Saha [ [arijitsaha2711@gmail.com](mailto:arijitsaha2711@gmail.com) ]

© U R Rao Satellite Centre, ISRO

This code is aimed to create a powerful deep learning model to analyse incoming landing images, automatically identify safe and scientifically valuable landing zones, classify other regions of the image and guide the spacecraft to a precise touchdown, eliminating human intervention and enhancing mission autonomy.

This code was prepared after lots of study and research under URSC, ISRO internship. It gives best possible output with 80% accuracy as per our research with only 200 labeled images.

All the resources are publicly available to use. Anyone can run the code and try out by themselves. It is encouraged to implement this code and improve the model time by time for better results. It is requested to contact the authors if better results can be achieved.

Dated: 10.02.2024

”””

1. **Downloading Mars CTX images:**

Website: NASA Planetary Data System, Mars Orbital Data Explorer

Link: <https://ode.rsl.wustl.edu/mars/mapsearch>

Instruction: MRO > CTX > Raw & Derived > Tick Experimental data Record

1. **Labeling:**

Software: VIA (VGG Image Annotator)

Class names: Give the exact same attribute values to comply with the code. Attribute name should be ‘name’

Signature\_markings

Mountains

Extreme\_slopes

Crater

Crater\_floor

Crater\_wall

Crater\_central\_peak

Fluvial\_alluvial\_fans

Boulder\_fields

Texture

Landable

Link: https://www.robots.ox.ac.uk/~vgg/software/via/via\_demo.html

Instruction: Easily Available on Internet on how to label images on VIA tool. Only use the Polygon feature for easily mask generation and to comply with our code. Image name & Json name should match.

Extraction format: Extract those labelled images in Json format.

1. **Multiclass mask Generation:**

This part is for generating multiclass masks file in tif image format with those images and corresponding Json files.

Put the images in ‘images’ folder and their corresponding Json files with same names in ‘Jsons’ folder. Generated masks will be stored in ‘masks’ folder.

Python code:

# Importing required libraries:

import numpy as np

import cv2

import os

import json

import matplotlib.pyplot as plt

# Declaring image and mask folder.

imgdir="images"

maskdir="masks"

# Declaring json folder, and listing all json files within the folder

jfolder = "Jsons"

jfiles = os.listdir(jfolder)

# Generating multiclass grayscale masks in tif image format

for jfile in jfiles: #For iterating over each json files

f=open(f"Jsons/{jfile}","r") # Read each json file

data=json.load(f) # Load the json files in json format to extract data from it in future

data=data["\_via\_img\_metadata"] # All the labeling data is stored in the dictionary, so extract from it

for key,value in data.items():

filename=value["filename"] # Read the json filename

imgpath=f"{imgdir}/{filename}" # Read the corresponding image

img=cv2.imread(imgpath,cv2.IMREAD\_COLOR) # Read the ctx image

h,w, \_ =img.shape #Get the height and width of the image

regions=value["regions"] # All the labelled regions are loaded

mask=np.zeros((h,w)) # Create an empty mask with zero values of same image size

for region in regions: # Iterate through each region

class\_mask = np.zeros((h, w), dtype=np.uint8) # For each mask in a single image

region\_attributes=region["region\_attributes"] # get the details about the region

partsname=region\_attributes["name"] # Region name

#Assigning different pixel intensity for different class.

# if(partsname=="Signature\_markings"):

# intensity=4

# elif(partsname=="Mountains"):

# intensity=5

# elif(partsname=="Extreme\_slopes"):

# intensity=6

# elif(partsname=="Crater\_floor"):

# intensity=7

# elif(partsname=="Crater\_wall"):

# intensity=8

# elif(partsname=="Crater\_central\_peak"):

# intensity=9

# elif(partsname=="Fluvial\_alluvial\_fans"):

# intensity=10

# elif(partsname=="Boulder\_fields"):

# intensity=11

if(partsname=="Crater"):

intensity=1

elif(partsname=="Texture"):

intensity=2

elif(partsname=="Landable"):

intensity=3

# Get Details about each points of the polygon labels

shape\_attributes=region["shape\_attributes"]

x\_points=shape\_attributes["all\_points\_x"]

y\_points=shape\_attributes["all\_points\_y"]

contours=[]

# For saving as tif files with same name

ext='.tif'

file\_name, file\_extension = os.path.splitext(filename)

file1name=file\_name+ext

# Draw the masks

for x,y in zip(x\_points,y\_points):

contours.append((x,y))

contours=np.array(contours)

cv2.drawContours(mask,[contours],-1,intensity,thickness=cv2.FILLED)

# Saving masks in the specified format

cv2.imwrite(f"{maskdir}/{file1name}",mask)

1. **Code for training UNET model on our dataset:**

In this code the model is trained only with 4 classes: Crater, Texture, Land-able & Unlabeled/Background. So it will predict accordingly. It is basically four class classification.

Keep the images & masks folder in the same directory as the code is in.

* Save the underneath python code in unet\_model.py file within the same directory:

“””

Standard Unet Model

Model not compiled here, instead will be done externally to make it easy to test various loss functions and optimizers.

"""

from keras.models import Model

from keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate, Conv2DTranspose, BatchNormalization, Dropout, Lambda

################################################################

def multi\_unet\_model(n\_classes=4, IMG\_HEIGHT=256, IMG\_WIDTH=256, IMG\_CHANNELS=1):

#Build the model

inputs = Input((IMG\_HEIGHT, IMG\_WIDTH, IMG\_CHANNELS))

#s = Lambda(lambda x: x / 255)(inputs) #No need for this if we normalize our inputs beforehand

s = inputs

#Contraction path

c1 = Conv2D(16, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(s)

c1 = Dropout(0.1)(c1)

c1 = Conv2D(16, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c1)

p1 = MaxPooling2D((2, 2))(c1)

c2 = Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(p1)

c2 = Dropout(0.1)(c2)

c2 = Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c2)

p2 = MaxPooling2D((2, 2))(c2)

c3 = Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(p2)

c3 = Dropout(0.2)(c3)

c3 = Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c3)

p3 = MaxPooling2D((2, 2))(c3)

c4 = Conv2D(128, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(p3)

c4 = Dropout(0.2)(c4)

c4 = Conv2D(128, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c4)

p4 = MaxPooling2D(pool\_size=(2, 2))(c4)

c5 = Conv2D(256, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(p4)

c5 = Dropout(0.3)(c5)

c5 = Conv2D(256, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c5)

#Expansive path

u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)

u6 = concatenate([u6, c4])

c6 = Conv2D(128, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(u6)

c6 = Dropout(0.2)(c6)

c6 = Conv2D(128, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c6)

u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c6)

u7 = concatenate([u7, c3])

c7 = Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(u7)

c7 = Dropout(0.2)(c7)

c7 = Conv2D(64, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c7)

u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c7)

u8 = concatenate([u8, c2])

c8 = Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(u8)

c8 = Dropout(0.1)(c8)

c8 = Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c8)

u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(c8)

u9 = concatenate([u9, c1], axis=3)

c9 = Conv2D(16, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(u9)

c9 = Dropout(0.1)(c9)

c9 = Conv2D(16, (3, 3), activation='relu', kernel\_initializer='he\_normal', padding='same')(c9)

outputs = Conv2D(n\_classes, (1, 1), activation='softmax')(c9)

model = Model(inputs=[inputs], outputs=[outputs])

#NOTE: Compile the model in the main program to make it easy to test with various loss functions

#model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

#model.summary()

return model

* Now the actual code:

# Importing necessary packages

from simple\_multi\_unet\_model import multi\_unet\_model #Uses softmax

from keras.utils import normalize

import os

import cv2

import numpy as np

from matplotlib import pyplot as plt

#Resizing images, if needed

SIZE\_X = 128

SIZE\_Y = 128

n\_classes=4 # Number of classes for segmentation

# Capture training image info as a list

train\_images = []

imgfolder="images"

imgs=os.listdir(imgfolder)

imgs=sorted(imgs)

for imgg in imgs:

img\_path=f"images/{imgg}"

img = cv2.imread(img\_path,0)

img = cv2.resize(img, (SIZE\_Y, SIZE\_X))

train\_images.append(img)

# Convert list to array for machine learning processing

train\_images = np.array(train\_images)

# Capture mask/label info as a list

train\_masks = []

maskfolder="masks"

masks=os.listdir(maskfolder)

masks=sorted(masks)

for maskk in masks:

mask\_path=f"masks/{maskk}"

mask = cv2.imread(mask\_path, cv2.IMREAD\_UNCHANGED)

mask = cv2.resize(mask, (SIZE\_Y, SIZE\_X), interpolation cv2.INTER\_NEAREST) #Otherwise ground truth changes due to interpolation

train\_masks.append(mask)

#Convert list to array for machine learning processing

train\_masks = np.array(train\_masks)

#Encode labels... but multi dim array so need to flatten, encode and reshape

from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder()

n, h, w = train\_masks.shape

train\_masks\_reshaped = train\_masks.reshape(-1,1)

train\_masks\_reshaped\_encoded = labelencoder.fit\_transform(train\_masks\_reshaped)

train\_masks\_encoded\_original\_shape = train\_masks\_reshaped\_encoded.reshape(n, h, w)

np.unique(train\_masks\_encoded\_original\_shape)

# Preprocessing and normalizing the images:

train\_images = np.expand\_dims(train\_images, axis=3)

train\_images = normalize(train\_images, axis=1)

train\_masks\_input = np.expand\_dims(train\_masks\_encoded\_original\_shape, axis=3)

#Create a subset of data for quick testing

#Picking 10% for testing and remaining for training

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(train\_images, train\_masks\_input, test\_size = 0.10, random\_state = 0)

print("Class values in the dataset are ... ", np.unique(y\_train)) # 0 is the background/few unlabeled

# Covert the class labels into categorical values

from keras.utils import to\_categorical

train\_masks\_cat = to\_categorical(y\_train, num\_classes=n\_classes)

y\_train\_cat = train\_masks\_cat.reshape((y\_train.shape[0], y\_train.shape[1], y\_train.shape[2], n\_classes))

test\_masks\_cat = to\_categorical(y\_test, num\_classes=n\_classes)

y\_test\_cat = test\_masks\_cat.reshape((y\_test.shape[0], y\_test.shape[1], y\_test.shape[2], n\_classes))

# Calculating the class weights

from sklearn.utils import class\_weight

class\_weights = class\_weight.compute\_class\_weight(

class\_weight = "balanced",

classes = np.unique(train\_masks\_reshaped\_encoded),

y = train\_masks\_reshaped\_encoded

)

print("Class weights are...:", class\_weights)

# Reshaping the training images

IMG\_HEIGHT = X\_train.shape[1]

IMG\_WIDTH = X\_train.shape[2]

IMG\_CHANNELS = X\_train.shape[3]

#Defining the unet model

def get\_model():

return multi\_unet\_model(n\_classes=n\_classes, IMG\_HEIGHT=IMG\_HEIGHT, IMG\_WIDTH=IMG\_WIDTH, IMG\_CHANNELS=IMG\_CHANNELS)

# Compiling the model

model = get\_model()

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

model.summary()

# Coverting the class weights into dictionary

class\_weights\_dict = {0: class\_weights[0], 1: class\_weights[1], 2: class\_weights[2], 3: class\_weights[3]}

# Training the model

history = model.fit(X\_train, y\_train\_cat,

batch\_size = 16,

verbose=1,

epochs=800,

validation\_data=(X\_test, y\_test\_cat),

class\_weight=class\_weights\_dict,

shuffle=False)

############################################################

# Evaluate the model

\_, acc = model.evaluate(X\_test, y\_test\_cat)

print("Accuracy is = ", (acc \* 100.0), "%")

# Plot the training and validation accuracy and loss at each epoch

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'y', label='Training loss')

plt.plot(epochs, val\_loss, 'r', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

plt.plot(epochs, acc, 'y', label='Training Accuracy')

plt.plot(epochs, val\_acc, 'r', label='Validation Accuracy')

plt.title('Training and validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# IOU

from keras.metrics import MeanIoU

n\_classes = 4

IOU\_keras = MeanIoU(num\_classes=n\_classes)

IOU\_keras.update\_state(y\_test[:,:,:,0], y\_pred\_argmax)

print("Mean IoU =", IOU\_keras.result().numpy())

#######################################################################

#Predict on a few images

import random

test\_img\_number = random.randint(0, len(X\_test)-1)

test\_img = X\_test[test\_img\_number]

ground\_truth=y\_test[test\_img\_number]

test\_img\_norm=test\_img[:,:,0][:,:,None]

test\_img\_input=np.expand\_dims(test\_img\_norm, 0)

prediction = (model.predict(test\_img\_input))

predicted\_img=np.argmax(prediction, axis=3)[0,:,:]

Generating colormap for each class for visualisation

color\_map = {

0: [1, 1, 105], # Background

1: [2, 209, 250], # blue - texture

2: [135, 1, 6], # red - crater

3: [2, 191, 37], # green - landable

}

def apply\_colormap(mask, color\_map):

rgb\_mask = np.zeros((mask.shape[0], mask.shape[1], 3), dtype=np.uint8)

for class\_id, color in color\_map.items():

rgb\_mask[mask == class\_id] = color

return rgb\_mask

rgb\_predicted\_mask = apply\_colormap(predicted\_img, color\_map)

rgb\_original\_mask = apply\_colormap(ground\_truth[:,:,0], color\_map)

# Plot the images

plt.figure(figsize=(12, 8))

plt.subplot(231)

plt.title('Testing Image')

plt.imshow(test\_imgg, cmap='gray')

plt.subplot(232)

plt.title('Testing label')

plt.imshow(rgb\_original\_mask)

plt.subplot(233)

plt.title('Prediction on test image')

plt.imshow(rgb\_predicted\_mask)

plt.show()