ME 249 - Final Project

Jack Leckert

December 2023

Contents

1 Work division

During this final project, I worked in full autonomy.

2 Introduction

Wildfires are increasingly becoming a source of economic and environmental damage in California. According to a recent study from the Energy Policy Institute at the University of Chicago, wildfires were responsible alone in 2020 for killing 30 people and causing more than $19 billion in economic losses [1]. In addition, wildfires made up 30 percent of California's greenhouse gas emissions on that year and became the second biggest source of carbon emissions, right after transportation.
To counteract this trend, a faster emergency response is needed from the CALFIRE department using cutting-edge technology. In particular, recent autonomous drone technology developments enable to spot and map early on the start of a wildfire. Onboard image processing such as computer vision algorithms is notably used to detect wildfires by identifying smoke or flames.
The purpose of this project is to train several neural networks with different architecture enabling to detect autonomously a wildfire on a drone image with an accuracy as close as possible to 100%. As time is an essential parameter during a fire emergency response, the image processing time of the neural networks will also be analyzed.

3 Data

The data used in this project corresponds to RGB and Thermal (infrared) images. Two different datasets have been found online and presents each advantages and drawbacks.

- **Dataset 1:** A first dataset was found on Kaggle [2] giving access to 1900 250p×250p RGB images, classified in two categories: 950 images labelled "fire" and the other 950 images labelled "non-fire". The advantage of this dataset is that it includes a strong diversity of cases. On the following pictures, we can see situations of wildfires taken from the ground or the sky, during the day or at night, including smoke or not. The dataset presents "non-fire" situation which could especially be mistaken for wildfires, such as a sunset for flames or a waterfall or clouds for some smoke.
- **Dataset 2:** The second dataset used in this study is a folder of 53,451 pairs of 254p×254p RGB - Thermal images found online at IEEE Xplore [3]. It includes 3 different labels: "Fire-Smoke","Fire-No Smoke" and "No Fire, No Smoke". "Fire" indicates whether or not there is fire visible in 254p RGB and/or 254p Thermal frame, while "Smoke" indicates whether smoke fills >= 50% of the RGB frame (visual estimate). The particularity of this dataset is too have access to pairs of RGB-Thermal images, enabling to gain insights on the best sensor to use. Later in the study, the performance of using either RGB images, Thermal images or a combination of both will be analyzed. However, this dataset presents a low diversity of cases. As shown on the examples below, most of the "No Fire No Smoke" cases correspond to a huge plain while "Fire" includes mostly a forest showing a strong smoke and little flames. With such a little diversity, and a lack of computing resources, only 10% of the initial dataset (5,345 pairs) will be used. The first dataset will still be needed to validate or not the accuracy obtained with the second dataset.

(a) Forest fire

(b) Aerial view at night

(c) Photo taken by human

(d) Huge smoke

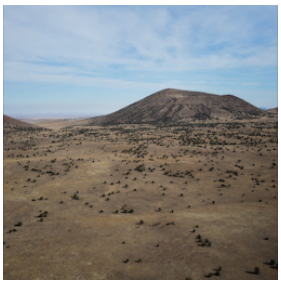
(e) Waterfall

(f) Sunset

(g) Fall colours

(h) Clouds and snow

Figure 1: Dataset 1 of 1900 RGB images



(a) No Fire 1

(b) No Fire 2

(c) Fire - No Smoke

(d) Fire and Smoke

Figure 2: Dataset 2 of 5345 RGB -Thermal pairs of images

4 Methods

4.1 Data preparation and labelling

Every single image needs first to be converted to a 3D array (height×width×3) of values between 0 and 255. Using the library cv2, the images can be read and converted to an array. Each image is also associated to a true output value y. For dataset 1, the images are already split into two folders "fire" and "no fire". Category "fire" is converted to $[1, 0]$ while "non fire" is converted to $[0, 1]$. For dataset 2, each image name contains an ID which falls into one of the categories "Fire - Smoke", "Fire - No Smoke" and "No Fire - No Smoke", respectively converted to $[0, 1, 0]$, $[0, 0, 1]$ and $[1, 0, 0]$.
Having done this first step, the arrays are than saved in a folder so that they can be directly loaded at the next run (and not the images). Finally, the data is normalized by dividing each array by 255 and split into train, validation and test set.
No data augmentation has been applied as the original dataset were already large enough.

4.2 Models

In this project, two different models are trained and tested, all based on the convolutional neural network architecture. Given an image, this type of neural network learns first to detect small to bigger features, which corresponds to observable shapes in the image. Concretely, it applies filters to detect any strong variation in pixel values. After feature learning, the data is flatten into a bidimensional array to train a classic fully connected neural network. The following illustrates the architecture of a CNN:

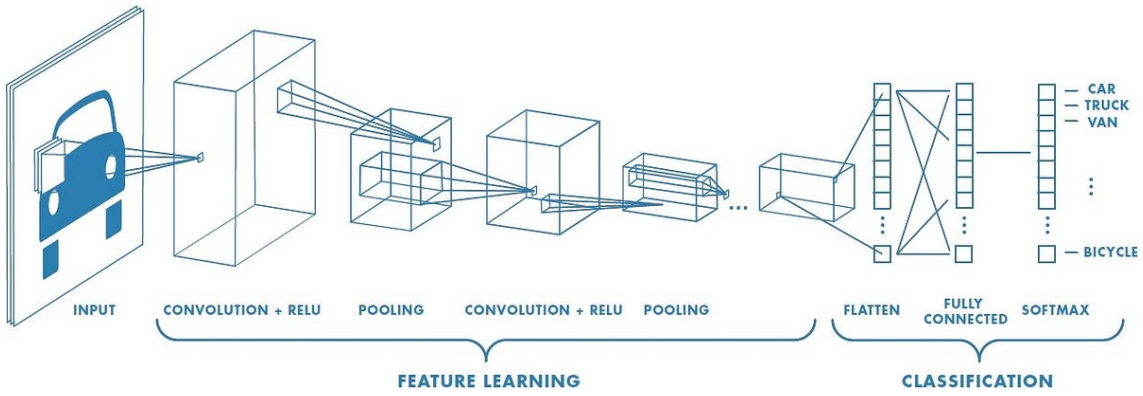


Figure 3: The convolutional neural network architecture. Source: towardsdatascience.com

After a research done in the available online literature, two studies [3][4] mentioned a very high classification accuracy using a residual neural network, a type of convolutional neural network which uses the output of deeper layers as with actual layers by establishing residual connections. Those type of neural network have major advantages over CNN:

- Residual neural network have a training loss which continually decreases, whereas classic CNN loss increases again after certain number of steps
- It can represent much more complex functions, meaning an enhanced ability to learn complex features
- It finally allows the training of much deeper neural networks without overfitting

The following picture gives an overview of the residual connection. Sometimes, the connection not only apply the identity function but includes more complex function such as a convolutional layer.

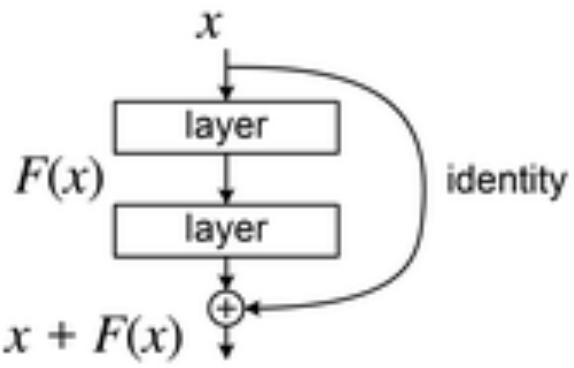


Figure 4: A residual connection. Source: wikipedia.com

4.3 Training

The two type of models trained in this project are a ResNet18 (with 18 layers) and a fine-tuned CNN. Each of the models include some feature layers followed by a fully connected neural network. The feature learning layers are summarized in the following tables:

Layers	**Channels**	**Kernel**	**Strides**	**Padding**	**Activation**
Conv2D	96	11	4	valid	relu
BatchNormalization					
MaxPooling2D		3	2	valid	
Conv2D	256	5	1	same	relu
BatchNormalization					
MaxPooling2D		3	2	valid	
Conv2D	384	3	1	same	relu
BatchNormalization					
Conv2D	384	3	1	same	relu
BatchNormalization					
Conv2D	256	3	1	same	relu
BatchNormalization					
MaxPooling2D		3	2	valid	
Flatten					

Table 1: CNN feature learning blocs

	Layers	Channels	Kernel	Strides	Padding	Activation
Initial Bloc	Conv2D	64	7	2	same	None
	BatchNormalization					
	ReLU					relu
	MaxPool2D		2	2	same	
ResNetBloc1	Conv2D	64	3	2	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	64	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc2	Conv2D	64	3	2	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	64	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc3	Conv2D	128	3	1	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	128	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc4	Conv2D	128	3	2	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	128	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc5	Conv2D	256	3	1	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	256	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc6	Conv2D	256	3	2	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	256	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc7	Conv2D	512	3	1	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	512	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
ResNetBloc8	Conv2D	512	3	2	same	
	BatchNormalization					
	ReLU					relu
	Conv2D	512	3	1	same	
	BatchNormalization					
	Add([residue,output])					
	ReLU					relu
Final Bloc	GlobalAveragePooling2D					
	Flatten					

Table 2: ResNet18 feature learning architecture

Each bloc of the ResNet18 include a residual connection, which can be either the identity of the input or a convolutional layer. These are added together with the main blocs when calling the function Add().

Blocs	Residues	Channels	Kernel	Strides	Padding
ResNetBloc1	Identity				
ResNetBloc2	Identity				
ResNetBloc3	Conv2D	128	1	2	same
	BatchNormalization				
ResNetBloc4	Identity				
ResNetBloc5	Conv2D	256	1	2	same
	BatchNormalization				
ResNetBloc6	Identity				
ResNetBloc7	Conv2D	512	1	2	same
	BatchNormalization				
ResNetBloc8	Identity				

Table 3: Residues blocs for each corresponding main blocs

Finally, either for the CNN or the ResNet18, the feature learning bloc is followed by a fully connected as follows:

FC	Neurons / Rate	Activation
Dense	4096	relu
Dropout	0.5	
Dense	4096	relu
Dropout	0.5	
Dense	2 or 3	softmax

Table 4: Fully connected layers

Each of the two models are trained 4 different times, using different data inputs.

1. Run 1: Dataset 1
2. Run 2: Dataset 2 - RGB images only
3. Run 3: Dataset 2 - Thermal images only
4. Run 4: Dataset 2 - combination of RGB and Thermal images

Run 4 requires a modification of the neural network architecture. In fact, two feature learning blocs needs to be built, one for the RGB dataset and one for the Thermal dataset. The features are then being concatenated into one single array right before inputting it in the fully connected layer. No modification are done on the fully connected layer in this case. One final model will be proposed as a combination of the CNN and ResNet18 models.

The models trained during this project are large in comparison to the available computing ressources. They include several million of parameters for a Intel Core i5 processor of 8GB RAM. 1 run takes between 2 and 9 hours to complete.

Model	**Parameters**
CNN	58.3M
2 CNNs combined	99.8M
ResNet18	30.1M
2 ResNets18 combined	43.4M
1 CNN + 1 ResNet combined	71.6M

Table 5: Model parameters count

For any run, epochs are fixed to 200, with an early stop patience of 10 and the batch size is set to 8 images (other batch sizes such as 3, 16 and 32 have been tested but without any improvement). The loss used in both runs with dataset 1 or dataset 2 is the categorical cross-entropy. A run stops based on the improvement of the validation accuracy (not the training accuracy).

5 Results and analysis

In this section, two main results will be analyzed to compare the performance of each model: the accuracy and the processing time. While accuracy won't change for a given trained neural network, the processing time could however be faster or slower on a drone processor. It gives however a first approximation and enables comparison between the models.

5.1 Fine-tuned CNN

5.1.1 Dataset 1

The proposed convolutional neural network was first trained using only the dataset 1 of 1900 RGB images presenting a strong diversity of situations. The following training curve showing the evolution of the training and validation accuracy was obtained:

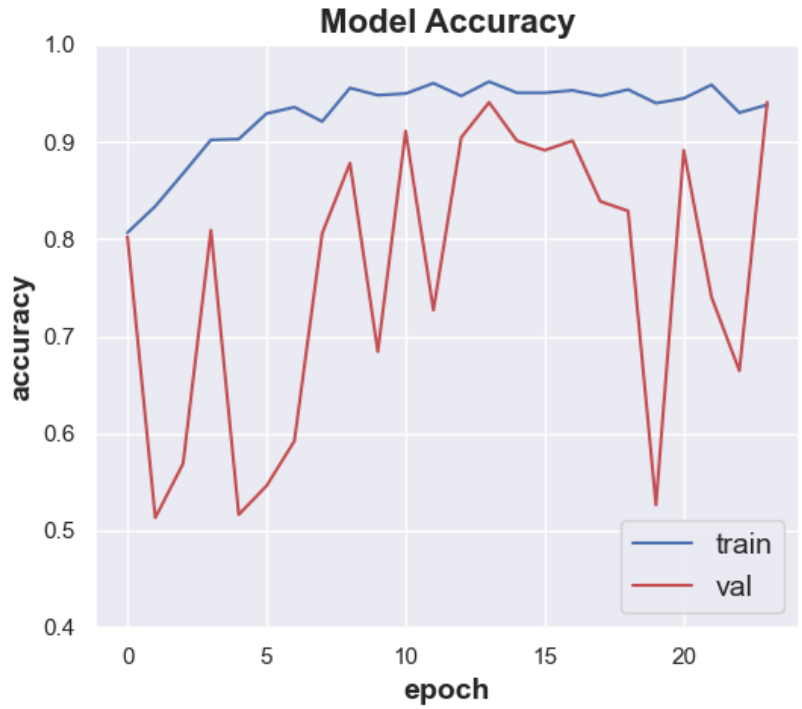
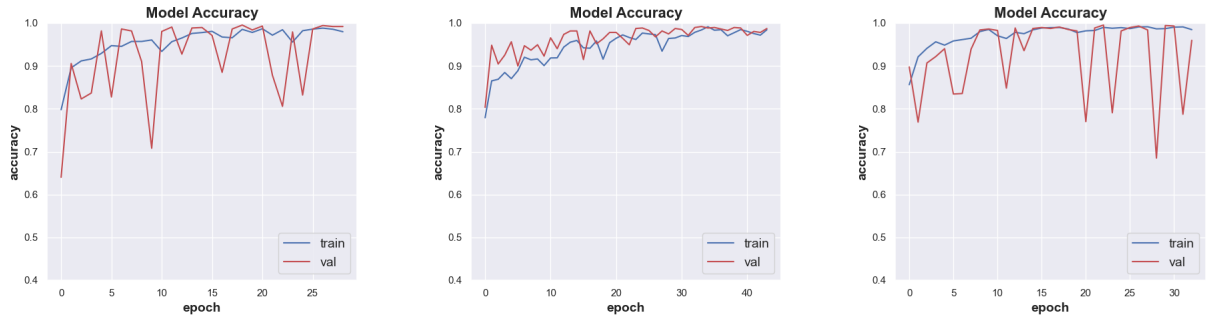


Figure 5: CNN training curve - dataset 1

It is first interesting to notice the few amount of epochs required for the model to be optimally trained, only 25! This is due to the type of input data used. Images have each $250 \times 250 \times 3 = 187,500$values which, combined, can each reveal the presence or absence of fire.
It is however hard to conclude whether the model is underfitting or overfitting given the high variability of the validation curve. It still enables to set a stopping point for the training when the validation reaches a maximum value. The accuracy obtained on the test set is 95.50% which is already very satisfying, given the very diverse dataset. The processing time per image is 18.1ms which can constitute a reference value for the next runs.

5.1.2 Dataset 2

Using now the dataset 2, the convolutional neural network is trained 3 times and gives following results:



(a) RGB images training curve (b) Thermal image training curve (c) RGB-IR images training curve

Figure 6: CNN training curves - dataset 2

	Accuracy (%)	**Processing time (ms)**
RGB	99.36	20.4
IR	98.07	22.0
RGB - IR	99.45	47.6

Table 6: CNN - Accuracy and Processing time

The convolutional achieves an outstanding performance on the dataset 2, especially when combining the the RGB and thermal images, reaching a test accuracy of 99.45%. It is however important to notice that the combination of two CNN feature learning layers increases the processing time by 2.3 when the number of parameters only increased by 1.7. It is therefore more relevant to choose the CNN trained over the RGB images only, as the accuracy is still of 99.36% but with a much lower processing time per image of 20.4ms.

5.2 ResNet18

5.2.1 Dataset 1

ResNet18 globally achieved poorer results than the CNN. Let's look at its training curve over the dataset 1:

Figure 7: ResNet18 training curve - dataset 1

Whereas the training accuracy continually improves, the validation curve struggles to keep high values and reaches its maximum after only 10 steps. The chosen batch size of 8 might too low so that the model generalizes well on new data. The final accuracy reached on the dataset 1 is 93.95%, which is just a little bit lower than the CNN. The processing time is 34.25ms, almost 2× slower than the CNN! The CNN performs therefore better than the ResNet18 for any relevant metric. This contradicts the results of the study [3].

5.2.2 Dataset 2

On dataset 2, following training curves, accuracies and processing times are obtained:

(a) RGB images training curve (b) Thermal image training curve (c) RGB-IR images training curve

Figure 8: ResNet18 training curves - dataset 2

	Accuracy (%)	Processing time (ms)
RGB	94.75	47.3
IR	98.80	39.3
RGB - IR	98.80	79.5

Table 7: ResNet18 - Accuracy and Processing time

Globally, the ResNet18 performs worse than the CNN, with a lower accuracy and around 2× the same processing time. The only exception is the ResNet18 results when trained over the IR images. While no other hyperparameters have been modified, the validation curve follows the training curve with much less perturbations and reach a final test accuracy of 98.80%. This accuracy is higher than the one obtained with the CNN but only by +0.73%.

Either with the CNN or the ResNet18, the accuracy reached when combining the RGB and Thermal images is higher than the RGB or Thermal images alone. This still proves the relevance of combining both type of data if the processing time is not taken into account.

5.3 Final model

The most accurate model on the RGB images alone is the CNN (99.36%) while the most accurate model on the thermal images is the ResNet18 (98.80%). The final model proposed here is to combine the CNN feature learning layers for RGB images and the ResNet18 feature learning layers for thermal images and verify if it is able to achieve a better accuracy than any previous model, i.e higher than 99.45%.
Following training curve is observed:

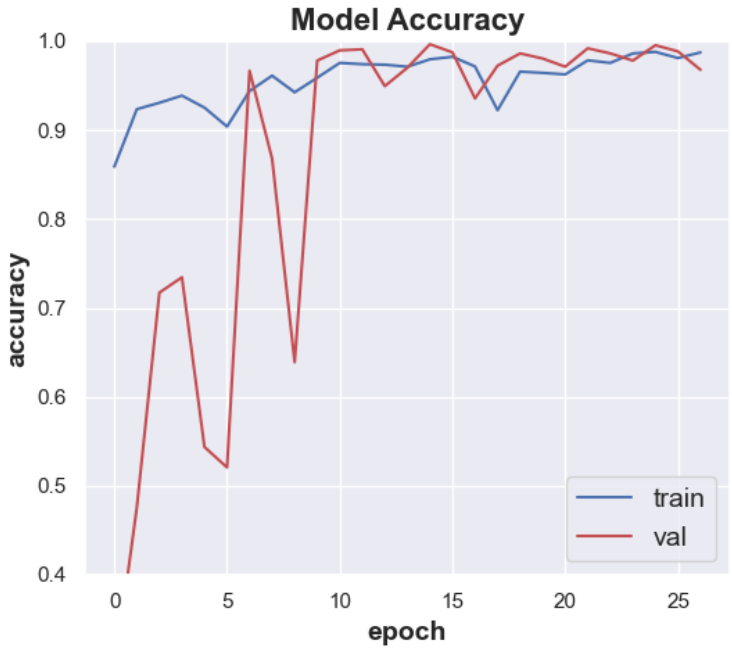


Figure 9: ResNet18 training curve - dataset 1

The final accuracy reached is of 99.08% with a processing time of 59ms. This model

therefore performs worse than the CNN model on RGB images alone.

6 Conclusion and future work

To conclude this work, large convolutional neural network onboard of a drone can help to accurately detect wildfires in an emergency situation. The model with best performance developed in this study is a deep convolutional neural network with 5 convolutional layers. It enables to reach an outstanding accuracy of 99.36% for a low processing time per image of 20.4ms. If we consider using a high quality 30 frames per second camera, this means the drone would need 0.6 second to process every 1 second recorded. This shows how drones combined with novel machine learning models can become game-changing in informing the CALFIRE department and therefore enabling faster emergency response.

7 References

References

[1] M. Jerrett, "Up in smoke: California's greenhouse gas reductions could be wiped out by 2020 wildfires," *Environmental Pollution*, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0269749122011022

[2] B. Dincer, "Wildfire detection image data," *Kaggle*, 2021. [Online]. Available: https://www.kaggle.com/datasets/brsdincer/wildfire-detection-image-data/data

[3] X. Chen, "Wildland fire detection and monitoring using a drone-collected rgb/ir image dataset," *IEEE Xplore*, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9953997

[4] A. Bouguettaya, "A review on early wildfire detection from unmanned aerial vehicles using deep learning-based computer vision algorithms," *ScienceDirect*, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165168421003467

8 Code

Train

December 15, 2023

[ ]:
```
"""
Import libraries
"""

import pandas as pd
import math
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import (
    BatchNormalization, Conv2D, MaxPooling2D, Flatten, Dropout, Dense,Input, ␣
 ↪MaxPool2D, GlobalAveragePooling2D, Layer, Add
)
import matplotlib.pyplot as plt
import numpy as np
import os
import re
import glob
from tqdm import tqdm
import cv2
import sklearn
import skimage
from sklearn.model_selection import train_test_split
from skimage.transform import resize
import random
from keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import accuracy_score
import seaborn as sns
sns.set()
```

[ ]:
```
"""
Choose dataset and model to train
"""

#Dataset to load or build
LOAD_DATASET = True
```

```
SMALL_DATASET = False
#RGB and IR images. In order to enable RGB, IR or combination, SMALL_DATASET␣
 ↪must be set to False
RGB = False
IR = False
COMBI = True

if SMALL_DATASET:
    num_classes=2
else:
    num_classes=3

#Model to train: choose 'CNN','ResNet18' or 'ResNet_CNN'
MODEL = 'ResNet_CNN'

MODEL_FILE_NAME = 'largedata_ResNet_CNN'
#Hyperparameters
BATCH_SIZE = 8
EPOCHS = 200

#Paths
RGB_data = '../Data/254p RGB Images/'
IR_data = '../Data/254p Thermal Images/'
small = "../Data/forest_fire/All"
```

[ ]:
```
"""
Function definition: load or build dataset
"""
#Fire (Y/N) indicates whether or not there is fire visible in 254p RGB and/or␣
 ↪254p Thermal frame
#Smoke (Y/N) indicates whether smoke fills >= 50% of the 254p RGB frame (visual␣
 ↪estimate)

#Functions definition
def get_large_data(path):
    y = []
    x = []
    IDs = []
    files = os.listdir(path)
    for i, file in tqdm(enumerate(files)):
        FileName = os.path.join(path, file)
        ID =  re.findall(r"\d+",FileName[-10:])[0]
        if len(ID)>=3 and int(ID[-1])!=0:
            if path==RGB_data:
                os.rename(path + "/" +file,'../Data/RGB duplicates/'+file)
            if path == IR_data:
                os.rename(path + "/" +file,'../Data/Thermal duplicates/'+file)
```

```
        else:
            ID = int(ID)
            IDs.append(ID)
            if ID in range(1,13701):
                y.append([1,0,0]) #NN = No fire, no smoke
            if ID in range(13701,14700) or ID in range(15981,19803) or ID in␣
↪range(19900,27184) or ID in range(27515,31295) or ID in range(31510,33598) or␣
↪ID in range(33930,36551) or ID in range(38031,38154) or ID in␣
↪range(41642,45280) or ID in range(51207,52287):
                y.append([0,1,0]) # YY = Yes fire, Yes smoke
            if ID in range(14700,15981) or ID in range(19803,19900) or ID in␣
↪range(27184,27515) or ID in range(31295,31510) or ID in range(33598,33930) or␣
↪ID in range(36551,38031) or ID in range(38154,41642) or ID in␣
↪range(45280,51207) or ID in range(52287,53452):
                y.append([0,0,1]) #YN = Yes fire, no smoke.

        img_file = cv2.imread(path + "/" +file)
        if img_file is not None:
            img_arr = np.asarray(img_file)
            x.append(img_arr)
    x = np.asarray(x)
    y = np.asarray(y)
    return x,y,IDs


def get_small_data(folder):
    x = []
    y = []
    for folderName in os.listdir(folder):
        if not folderName.startswith("."):
            if folderName in ["nofire"]:
                label = [0,1]
            elif folderName in ["fire"]:
                label = [1,0]
            for image_filename in tqdm(os.listdir(folder +"/" +folderName+"/")):
                img_file = cv2.imread(folder + "/" +folderName + "/" +␣
↪image_filename)
                if img_file is not None:
                    #img_file = skimage.transform.resize(img_file,(227,227,3),␣
↪mode = "constant",anti_aliasing=True)
                    img_arr = np.asarray(img_file)
                    x.append(img_arr)
                    y.append(label)
    x = np.asarray(x)
    y = np.asarray(y)
    return x,y
```

[ ]:
```
"""
Load or build dataset
"""

if SMALL_DATASET:
    if LOAD_DATASET:
        X = np.load("../Data/X_small.npy")
        y = np.load("../Data/y_small.npy")

    else:
        X,y = get_small_data(small)

        np.save("../Data/X_small.npy",X)
        np.save("../Data/y_small.npy",y)

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.
 ↪2,shuffle=True)
    X_train, X_valid, y_train, y_valid =␣
 ↪train_test_split(X_train,y_train,test_size=0.2,shuffle=True)
    # Image Normalization
    X_train, X_valid, X_test = X_train / 255.0, X_valid / 255.0, X_test / 255.0
    n=X_train.shape[1]
    inputshape = (None,n,n,3)
else:
    if LOAD_DATASET:
        X_RGB = np.load("../Data/X_RGB.npy")
        Y_RGB = np.load("../Data/Y_RGB.npy")
        IDs_RGB = np.load("../Data/IDs_RGB.npy")
        X_IR = np.load("../Data/X_IR.npy")
        Y_IR = np.load("../Data/Y_IR.npy")
        IDs_IR = np.load("../Data/IDs_IR.npy")

    else:
        X_RGB,Y_RGB,IDs_RGB = get_large_data(RGB_data)
        X_IR,Y_IR,IDs_IR = get_large_data(IR_data)

        np.save("../Data/X_RGB.npy",X_RGB)
        np.save("../Data/Y_RGB.npy",Y_RGB)
        np.save("../Data/IDs_RGB.npy",IDs_RGB)
        np.save("../Data/X_IR.npy",X_IR)
        np.save("../Data/Y_IR.npy",Y_IR)
        np.save("../Data/IDs_IR.npy",IDs_IR)

    # Split the data
    X_RGB_train, X_RGB_test, y_RGB_train, y_RGB_test =␣
 ↪train_test_split(X_RGB,Y_RGB,test_size=0.2,shuffle=True)
```

```
    X_RGB_train, X_RGB_valid, y_RGB_train, y_RGB_valid =␣
↪train_test_split(X_RGB_train,y_RGB_train,test_size=0.2,shuffle=True)
    # Image Normalization
    X_RGB_train, X_RGB_valid, X_RGB_test = X_RGB_train / 255.0, X_RGB_valid /␣
↪255.0, X_RGB_test / 255.0

    # Split the data
    X_IR_train, X_IR_test, y_IR_train, y_IR_test =␣
↪train_test_split(X_IR,Y_IR,test_size=0.2,shuffle=True)
    X_IR_train, X_IR_valid, y_IR_train, y_IR_valid =␣
↪train_test_split(X_IR_train,y_IR_train,test_size=0.2,shuffle=True)
    # Image Normalization
    X_IR_train, X_IR_valid, X_IR_test = X_IR_train / 255.0, X_IR_valid / 255.0,␣
↪X_IR_test / 255.0
    n=X_IR_train.shape[1]
    inputshape = (None,n,n,3)
if COMBI:
    #reorder the data for combi CNN
    X_RGBs=np.zeros_like(X_RGB)
    for i in range(len(IDs_RGB)):
        j=np.argwhere(np.array(IDs_RGB)==IDs_IR[i])[0][0]
        X_RGBs[i]=X_RGB[j]

    # Split the data
    X_RGB_train,X_RGB_test,X_IR_train, X_IR_test, y_train, y_test =␣
↪train_test_split(X_RGBs,X_IR,Y_IR,test_size=0.2,shuffle=True)
    X_RGB_train,X_RGB_valid,X_IR_train, X_IR_valid, y_train, y_valid =␣
↪train_test_split(X_RGB_train,X_IR_train,y_train,test_size=0.2,shuffle=True)
    # Image Normalization
    X_RGB_train, X_RGB_valid, X_RGB_test = X_RGB_train / 255.0, X_RGB_valid /␣
↪255.0, X_RGB_test / 255.0
    X_IR_train, X_IR_valid, X_IR_test = X_IR_train / 255.0, X_IR_valid / 255.0,␣
↪X_IR_test / 255.0
    n=X_IR_train.shape[1]
    inputshape = [(None,n,n,3),(None,n,n,3)]
```

[ ]:
```
"""
Models definition using classes: CNN, CNN_combi, ResNet18, Resnet18_combi and␣
↪ResNet_CNN
"""
class CNN(Model):
    def __init__(self, channels: int, **kwargs):
        super().__init__(**kwargs)
        self.cnn_conv1=Conv2D(96,(11,11),strides=(4, 4),activation="relu")
        self.cnn_bn1=BatchNormalization()
        self.cnn_pool1=MaxPooling2D((3,3), strides=(2,2))
```

```
        self.cnn_conv2=Conv2D(256,(5,5),activation="relu",padding="same")
        self.cnn_bn2=BatchNormalization()
        self.cnn_pool2=MaxPooling2D((3,3), strides=(2,2))
        self.cnn_conv3=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn_bn3=BatchNormalization()
        self.cnn_conv4=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn_bn4=BatchNormalization()
        self.cnn_conv5=Conv2D(256,(3,3),activation="relu",padding="same")
        self.cnn_bn5=BatchNormalization()
        self.cnn_pool3=MaxPooling2D((3,3), strides=(2,2))
        self.cnn_flat=Flatten()

        # Fully connected
        self.fc1=Dense(4096,activation="relu")
        self.drop1=Dropout(0.5)

        self.fc2=Dense(4096,activation="relu")
        self.drop2=Dropout(0.5)
        self.fc = Dense(num_classes, activation="softmax")

    def call(self, inputs):

        #cnn bloc
        out0=self.cnn_conv1(inputs)
        out0=self.cnn_bn1(out0)
        out0=self.cnn_pool1(out0)
        out0=self.cnn_conv2(out0)
        out0=self.cnn_bn2(out0)
        out0=self.cnn_pool2(out0)
        out0=self.cnn_conv3(out0)
        out0=self.cnn_bn3(out0)
        out0=self.cnn_conv4(out0)
        out0=self.cnn_bn4(out0)
        out0=self.cnn_conv5(out0)
        out0=self.cnn_bn5(out0)
        out0=self.cnn_pool3(out0)
        out0=self.cnn_flat(out0)


        # Fully connected
        out=self.fc1(out0)
        out=self.drop1(out)

        out=self.fc2(out)
        out=self.drop2(out)
        out=self.fc(out)
        return out
```

```
class CNN_combi(Model):
    def __init__(self, channels: int, **kwargs):
        super().__init__(**kwargs)
        self.cnn0_conv1=Conv2D(96,(11,11),strides=(4, 4),activation="relu")
        self.cnn0_bn1=BatchNormalization()
        self.cnn0_pool1=MaxPooling2D((3,3), strides=(2,2))
        self.cnn0_conv2=Conv2D(256,(5,5),activation="relu",padding="same")
        self.cnn0_bn2=BatchNormalization()
        self.cnn0_pool2=MaxPooling2D((3,3), strides=(2,2))
        self.cnn0_conv3=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn0_bn3=BatchNormalization()
        self.cnn0_conv4=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn0_bn4=BatchNormalization()
        self.cnn0_conv5=Conv2D(256,(3,3),activation="relu",padding="same")
        self.cnn0_bn5=BatchNormalization()
        self.cnn0_pool3=MaxPooling2D((3,3), strides=(2,2))
        self.cnn0_flat=Flatten()

        self.cnn1_conv1=Conv2D(96,(11,11),strides=(4, 4),activation="relu")
        self.cnn1_bn1=BatchNormalization()
        self.cnn1_pool1=MaxPooling2D((3,3), strides=(2,2))
        self.cnn1_conv2=Conv2D(256,(5,5),activation="relu",padding="same")
        self.cnn1_bn2=BatchNormalization()
        self.cnn1_pool2=MaxPooling2D((3,3), strides=(2,2))
        self.cnn1_conv3=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn1_bn3=BatchNormalization()
        self.cnn1_conv4=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn1_bn4=BatchNormalization()
        self.cnn1_conv5=Conv2D(256,(3,3),activation="relu",padding="same")
        self.cnn1_bn5=BatchNormalization()
        self.cnn1_pool3=MaxPooling2D((3,3), strides=(2,2))
        self.cnn1_flat=Flatten()


        # Fully connected
        self.fc1=Dense(4096,activation="relu")
        self.drop1=Dropout(0.5)

        self.fc2=Dense(4096,activation="relu")
        self.drop2=Dropout(0.5)
        self.fc = Dense(num_classes, activation="softmax")

    def call(self, inputs):

        #cnn bloc
        out0=self.cnn0_conv1(inputs[0])
```

```
        out0=self.cnn0_bn1(out0)
        out0=self.cnn0_pool1(out0)
        out0=self.cnn0_conv2(out0)
        out0=self.cnn0_bn2(out0)
        out0=self.cnn0_pool2(out0)
        out0=self.cnn0_conv3(out0)
        out0=self.cnn0_bn3(out0)
        out0=self.cnn0_conv4(out0)
        out0=self.cnn0_bn4(out0)
        out0=self.cnn0_conv5(out0)
        out0=self.cnn0_bn5(out0)
        out0=self.cnn0_pool3(out0)
        out0=self.cnn0_flat(out0)

        out1=self.cnn1_conv1(inputs[1])
        out1=self.cnn1_bn1(out1)
        out1=self.cnn1_pool1(out1)
        out1=self.cnn1_conv2(out1)
        out1=self.cnn1_bn2(out1)
        out1=self.cnn1_pool2(out1)
        out1=self.cnn1_conv3(out1)
        out1=self.cnn1_bn3(out1)
        out1=self.cnn1_conv4(out1)
        out1=self.cnn1_bn4(out1)
        out1=self.cnn1_conv5(out1)
        out1=self.cnn1_bn5(out1)
        out1=self.cnn1_pool3(out1)
        out1=self.cnn1_flat(out1)

        concat = tf.keras.layers.concatenate([out0, out1], name='Concatenate')

        # Fully connected
        out=self.fc1(concat)
        out=self.drop1(out)

        out=self.fc2(out)
        out=self.drop2(out)
        out=self.fc(out)
        return out



    """
    A standard resnet block.
    """
class ResnetBlock(Model):
```

```
    def __init__(self, channels: int, down_sample=False):

        super().__init__()

        self.__channels = channels
        self.__down_sample = down_sample
        self.__strides = [2, 1] if down_sample else [1, 1]

        KERNEL_SIZE = (3, 3)
        # use He initialization, instead of Xavier (a.k.a 'glorot_uniform' in␣
↪Keras), as suggested in [2]
        INIT_SCHEME = "he_normal"

        self.conv_1 = Conv2D(self.__channels, strides=self.__strides[0],
                             kernel_size=KERNEL_SIZE, padding="same",␣
↪kernel_initializer=INIT_SCHEME)
        self.bn_1 = BatchNormalization()
        self.conv_2 = Conv2D(self.__channels, strides=self.__strides[1],
                             kernel_size=KERNEL_SIZE, padding="same",␣
↪kernel_initializer=INIT_SCHEME)
        self.bn_2 = BatchNormalization()
        self.merge = Add()

        if self.__down_sample:
            # perform down sampling using stride of 2, according to [1].
            self.res_conv = Conv2D(
                self.__channels, strides=2, kernel_size=(1, 1),␣
↪kernel_initializer=INIT_SCHEME, padding="same")
            self.res_bn = BatchNormalization()

    def call(self, inputs):
        res = inputs

        x = self.conv_1(inputs)
        x = self.bn_1(x)
        x = tf.nn.relu(x)
        x = self.conv_2(x)
        x = self.bn_2(x)

        if self.__down_sample:
            res = self.res_conv(res)
            res = self.res_bn(res)

        # if not perform down sample, then add a shortcut directly
        x = self.merge([x, res])
        out = tf.nn.relu(x)
```

```
        return out



"""
A ResNet18 model
"""

class ResNet18(Model):

    def __init__(self, num_classes, **kwargs):
        """
            num_classes: number of classes in specific classification task.
        """
        super().__init__(**kwargs)
        self.conv_1 = Conv2D(64, (7, 7), strides=2,
                             padding="same", kernel_initializer="he_normal")
        self.init_bn = BatchNormalization()
        self.pool_2 = MaxPool2D(pool_size=(2, 2), strides=2, padding="same")
        self.res_1_1 = ResnetBlock(64)
        self.res_1_2 = ResnetBlock(64)
        self.res_2_1 = ResnetBlock(128, down_sample=True)
        self.res_2_2 = ResnetBlock(128)
        self.res_3_1 = ResnetBlock(256, down_sample=True)
        self.res_3_2 = ResnetBlock(256)
        self.res_4_1 = ResnetBlock(512, down_sample=True)
        self.res_4_2 = ResnetBlock(512)
        self.avg_pool = GlobalAveragePooling2D()
        self.flat = Flatten()
         # Fully connected
        self.fc1=Dense(4096,activation="relu")
        self.drop1=Dropout(0.5)

        self.fc2=Dense(4096,activation="relu")
        self.drop2=Dropout(0.5)
        self.fc = Dense(num_classes, activation="softmax")

    def call(self, inputs):

        out = self.conv_1(inputs)
        out = self.init_bn(out)
        out = tf.nn.relu(out)
        out = self.pool_2(out)
        for res_block in [self.res_1_1, self.res_1_2, self.res_2_1, self.
↪res_2_2, self.res_3_1, self.res_3_2, self.res_4_1, self.res_4_2]:
            out = res_block(out)
        out = self.avg_pool(out)
```

```
        out = self.flat(out)

        # Fully connected
        out = self.fc1(out)
        out = self.drop1(out)
        out = self.fc2(out)
        out = self.drop2(out)
        out = self.fc(out)
        return out


"""
combination of two ResNets to use RGB and Thermal images at the same time
"""
class ResNet18_combi(Model):
    def __init__(self, num_classes, **kwargs):
        """
            num_classes: number of classes in specific classification task.
        """
        super().__init__(**kwargs)
        self.conv_1 = Conv2D(64, (7, 7), strides=2,
                             padding="same", kernel_initializer="he_normal")
        self.init_bn = BatchNormalization()
        self.pool_2 = MaxPool2D(pool_size=(2, 2), strides=2, padding="same")
        self.res_1_1 = ResnetBlock(64)
        self.res_1_2 = ResnetBlock(64)
        self.res_2_1 = ResnetBlock(128, down_sample=True)
        self.res_2_2 = ResnetBlock(128)
        self.res_3_1 = ResnetBlock(256, down_sample=True)
        self.res_3_2 = ResnetBlock(256)
        self.res_4_1 = ResnetBlock(512, down_sample=True)
        self.res_4_2 = ResnetBlock(512)
        self.avg_pool = GlobalAveragePooling2D()
        self.flat = Flatten()

        self.conv2_1 = Conv2D(64, (7, 7), strides=2,
                             padding="same", kernel_initializer="he_normal")
        self.init2_bn = BatchNormalization()
        self.pool2_2 = MaxPool2D(pool_size=(2, 2), strides=2, padding="same")
        self.res2_1_1 = ResnetBlock(64)
        self.res2_1_2 = ResnetBlock(64)
        self.res2_2_1 = ResnetBlock(128, down_sample=True)
        self.res2_2_2 = ResnetBlock(128)
        self.res2_3_1 = ResnetBlock(256, down_sample=True)
        self.res2_3_2 = ResnetBlock(256)
        self.res2_4_1 = ResnetBlock(512, down_sample=True)
        self.res2_4_2 = ResnetBlock(512)
```

```
        self.avg2_pool = GlobalAveragePooling2D()
        self.flat2 = Flatten()

         # Fully connected
        self.fc1=Dense(4096,activation="relu")
        self.drop1=Dropout(0.5)

        self.fc2=Dense(4096,activation="relu")
        self.drop2=Dropout(0.5)
        self.fc = Dense(num_classes, activation="softmax")

    def call(self,inputs):
        #in1 = Input(shape=(n,n,3))

        out1 = self.conv_1(inputs[0])
        out1 = self.init_bn(out1)
        out1 = tf.nn.relu(out1)
        out1 = self.pool_2(out1)
        for res_block in [self.res_1_1, self.res_1_2, self.res_2_1, self.
↪res_2_2, self.res_3_1, self.res_3_2, self.res_4_1, self.res_4_2]:
            out1 = res_block(out1)
        out1 = self.avg_pool(out1)
        out1 = self.flat(out1)
        #model1 = Model(inputs=in1, outputs=out1)

        #in2 = Input(shape=(n,n,3))
        out2 = self.conv2_1(inputs[1])
        out2 = self.init2_bn(out2)
        out2 = tf.nn.relu(out2)
        out2 = self.pool2_2(out2)
        for res_block in [self.res2_1_1, self.res2_1_2, self.res2_2_1, self.
↪res2_2_2, self.res2_3_1, self.res2_3_2, self.res2_4_1, self.res2_4_2]:
            out2 = res_block(out2)
        out2 = self.avg2_pool(out2)
        out2 = self.flat2(out2)
        #model2 = Model(inputs=in2, outputs=out2)

        concat = tf.keras.layers.concatenate([out1, out2], name='Concatenate')

        # Fully connected
        out = self.fc1(concat)
        out = self.drop1(out)
        out = self.fc2(out)
        out = self.drop2(out)
        out = self.fc(out)

        #final_model = Model(inputs=[out1.input, out2.input], outputs=out,
```

```
        #             name='Final_output')
        #final_model.compile(optimizer='adam',␣
↪loss='categorical_crossentropy',metrics=["accuracy"])
        return out

"""
Final model: combination of CNN for RGB images and ResNet for Thermal images
"""
class ResNet_CNN(Model):
    def __init__(self, num_classes, **kwargs):
        """
            num_classes: number of classes in specific classification task.
        """
        #bloc1
        super().__init__(**kwargs)
        self.conv_1 = Conv2D(64, (7, 7), strides=2,
                             padding="same", kernel_initializer="he_normal")
        self.init_bn = BatchNormalization()
        self.pool_2 = MaxPool2D(pool_size=(2, 2), strides=2, padding="same")
        self.res_1_1 = ResnetBlock(64)
        self.res_1_2 = ResnetBlock(64)
        self.res_2_1 = ResnetBlock(128, down_sample=True)
        self.res_2_2 = ResnetBlock(128)
        self.res_3_1 = ResnetBlock(256, down_sample=True)
        self.res_3_2 = ResnetBlock(256)
        self.res_4_1 = ResnetBlock(512, down_sample=True)
        self.res_4_2 = ResnetBlock(512)
        self.avg_pool = GlobalAveragePooling2D()
        self.flat = Flatten()

        #bloc2
        self.cnn_conv1=Conv2D(96,(11,11),strides=(4, 4),activation="relu")
        self.cnn_bn1=BatchNormalization()
        self.cnn_pool1=MaxPooling2D((3,3), strides=(2,2))
        self.cnn_conv2=Conv2D(256,(5,5),activation="relu",padding="same")
        self.cnn_bn2=BatchNormalization()
        self.cnn_pool2=MaxPooling2D((3,3), strides=(2,2))
        self.cnn_conv3=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn_bn3=BatchNormalization()
        self.cnn_conv4=Conv2D(384,(3,3),activation="relu",padding="same")
        self.cnn_bn4=BatchNormalization()
        self.cnn_conv5=Conv2D(256,(3,3),activation="relu",padding="same")
        self.cnn_bn5=BatchNormalization()
        self.cnn_pool3=MaxPooling2D((3,3), strides=(2,2))
        self.cnn_flat=Flatten()

        # Fully connected
```

```
        self.fc1=Dense(4096,activation="relu")
        self.drop1=Dropout(0.5)

        self.fc2=Dense(4096,activation="relu")
        self.drop2=Dropout(0.5)
        self.fc = Dense(num_classes, activation="softmax")

    def call(self,inputs):


        #cnn bloc
        out0=self.cnn_conv1(inputs[0])
        out0=self.cnn_bn1(out0)
        out0=self.cnn_pool1(out0)
        out0=self.cnn_conv2(out0)
        out0=self.cnn_bn2(out0)
        out0=self.cnn_pool2(out0)
        out0=self.cnn_conv3(out0)
        out0=self.cnn_bn3(out0)
        out0=self.cnn_conv4(out0)
        out0=self.cnn_bn4(out0)
        out0=self.cnn_conv5(out0)
        out0=self.cnn_bn5(out0)
        out0=self.cnn_pool3(out0)
        out0=self.cnn_flat(out0)

        #resnet bloc
        out1 = self.conv_1(inputs[1])
        out1 = self.init_bn(out1)
        out1 = tf.nn.relu(out1)
        out1 = self.pool_2(out1)
        for res_block in [self.res_1_1, self.res_1_2, self.res_2_1, self.
↪res_2_2, self.res_3_1, self.res_3_2, self.res_4_1, self.res_4_2]:
            out1 = res_block(out1)
        out1 = self.avg_pool(out1)
        out1 = self.flat(out1)

        concat = tf.keras.layers.concatenate([out0, out1], name='Concatenate')
        # Fully connected
        out=self.fc1(concat)
        out=self.drop1(out)

        out=self.fc2(out)
        out=self.drop2(out)
        out=self.fc(out)
        return out
```

[ ]:
```
if MODEL == 'CNN':
    if COMBI:
        model = CNN_combi(num_classes)
    else:
        model=CNN(num_classes)
if MODEL == 'ResNet18':
    if COMBI:
        model = ResNet18_combi(num_classes)
    else:
        model = ResNet18(num_classes)
if MODEL == 'ResNet_CNN':
    model = ResNet_CNN(num_classes)




model.build(input_shape = inputshape)
model.compile(optimizer = "adam",loss='categorical_crossentropy',␣
 ↪metrics=["accuracy"])
model.summary()
```

[ ]:
```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor="val_accuracy",
    patience=10,
    restore_best_weights=True)
# Add a checkpoint where val accuracy is max, and save that model
mc = tf.keras.callbacks.ModelCheckpoint(MODEL_FILE_NAME+'.SB',␣
 ↪monitor='val_accuracy',
                     mode='max',  verbose=1, save_best_only=True)
```

[ ]:
```
batch_size=BATCH_SIZE
epochs=EPOCHS
if COMBI:
    history =  model.
 ↪fit([X_RGB_train,X_IR_train],y_train,validation_data=([X_RGB_valid,X_IR_valid],y_valid),batch
                    epochs=epochs,verbose=1,callbacks=[early_stopping])
else:
    if RGB:
        history =  model.
 ↪fit(X_RGB_train,y_RGB_train,validation_data=(X_RGB_valid,y_RGB_valid),batch_size=batch_size,
                    epochs=epochs,verbose=1,callbacks=[early_stopping])
    if IR:
         history =  model.
 ↪fit(X_IR_train,y_IR_train,validation_data=(X_IR_valid,y_IR_valid),batch_size=batch_size,
                    epochs=epochs,verbose=1,callbacks=[early_stopping])
```

```
    if SMALL_DATASET:
        history =  model.
 ↪fit(X_train,y_train,validation_data=(X_valid,y_valid),batch_size=batch_size,
                    epochs=epochs,verbose=1,callbacks=[early_stopping])



model.save('./'+MODEL_FILE_NAME)
```

[ ]:
```
model = tf.keras.models.load_model(MODEL_FILE_NAME)
if COMBI:
    score = model.evaluate([X_RGB_test,X_IR_test], y_test,␣
 ↪batch_size=batch_size, verbose=1)
else:
    if RGB:
        score = model.evaluate(X_RGB_test, y_RGB_test, batch_size=batch_size,␣
 ↪verbose=1)
    if IR:
        score = model.evaluate(X_IR_test, y_IR_test, batch_size=batch_size,␣
 ↪verbose=1)
    if SMALL_DATASET:
        score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=1)


print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

[ ]:
```
from matplotlib import pyplot as plt
%matplotlib inline

plt.figure(figsize=(6, 5))
plt.plot(history.history['accuracy'], color='b')
plt.plot(history.history['val_accuracy'], color='r')
plt.title('Model Accuracy', weight='bold', fontsize=16)
plt.ylabel('accuracy', weight='bold', fontsize=14)
plt.xlabel('epoch', weight='bold', fontsize=14)
plt.ylim(0.4, 1.0)
#plt.xticks(weight='bold', fontsize=12)
#plt.yticks(weight='bold', fontsize=12)
plt.legend(['train', 'val'], loc='lower right', prop={'size': 14})
#plt.grid(color = 'y', linewidth='0.5')
plt.savefig('training_curve_'+MODEL_FILE_NAME+'.png')
```