Appendix B: complete program

Main.py

```
import os
import time
import subprocess
import webbrowser
import preprocessing_data as prep
import models_training_and_optimization as train
import plotting_learning_curves as plot
import GUI_Tkinker as GUItk

def main():

    os.chdir(prep.base_path)

    # preprocess and pack the dataset to X.picle and y.pickle

    prep.Create_images_array()
    training_data = prep.Create_training_data()
    prep.Pack_data(training_data)

    x_train, y_train = prep.Load_data()
    x_train = prep.Normalize_data(x_train=x_train)

    # train the models for different parameters

    histories = {}
    colors = {}

    model_num = 1

    for dense_layers_num in train.dense_layers_nums:
        for neurons_num in train.neurons_per_layer:
            for conv_layers_num in train.conv_layers_nums:
                for kernel_size in train.kernel_sizes:

                    NAME = train.Set_NAME(neurons_num, dense_layers_num, conv_layers_num, kernel_size)

                    if os.path.exists("models/{}.model".format(NAME)) == False:
                        MyModel = train.CNN_Model(x_train, len(prep.CATEGORIES), neurons_num, dense_layers_num, conv_layers_num, kernel_size)
                        tensorboard = train.Save_TensorBoard_logs(NAME)
                        history = MyModel.Train_model(x_train, y_train, tensorboard)
                        MyModel.Save_model(NAME)
```

```
                        print("Saved model {}/{}".format(model_num,
train.number_of_models))

                        # Store this model's history and color
                        histories[NAME] = history.history
                        colors[NAME] = train.random_color()

                        model_num = model_num + 1

                    else:
                        GUItk.run_output_box("An old model in this folder
with this name already exists - it will not be trained again. If you want
to train a new one, please delete the old models first.")
                        pass

    train.save_histories_and_colors(histories, colors)

    # plot learning curve graphs for the trained models

    train.check_and_create_dir(train.plots_dir)
    histories, colors = plot.load_saved_histories_and_colors(histories,
colors)
    plot.plot_learning_curves(histories, colors)


    # Run the tensorboard command and open logs online

    tensorboard_command = 'tensorboard --logdir logs'
    subprocess.Popen(tensorboard_command, shell=True)
    time.sleep(60)
    #print('opening website')
    os.system('start chrome http://localhost:6006/')
    time.sleep(10)
    GUItk.run_output_box("Your models are trained and their performances
ready to evaluate.")


if __name__ == "__main__":
    main()
```

preprocessing_data.py

```
1  import os
2  import numpy as np
3  import cv2
4  import random
5  import matplotlib.pyplot as plt
6  import pickle
7  import GUI_Tkinker as GUItk
8
9  base_path = GUItk.run_input_box("Input the path to directory containing your
   dataset folder: ")
10 dataset_folder_name = GUItk.run_input_box("Input the name of your dataset folder:
   ")
11
12 dataset_dir = os.path.join(base_path, dataset_folder_name)
13
14 IMG_SIZE = int(GUItk.run_input_box("Input the target image size: "))
15
16 CATEGORIES = []
17
18 for category_name in os.listdir(dataset_dir):
19      CATEGORIES.append(category_name)
20
21 def Create_images_array():
22     for category in CATEGORIES:
23         img_path = os.path.join(dataset_dir, category)
24         for img in os.listdir(img_path):
25             img_array = cv2.imread(os.path.join(img_path,img),
   cv2.IMREAD_GRAYSCALE) #convert images into a grayscale array
26             #print(img_array)
27             #print(img_array.shape) #prints the original size of an image example
28             #plt.imshow(img_array, cmap="gray") #shows the example of an image
   before resizing
29             #plt.show()
30
31             new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE)) #resize images
   to 50x50 pixels
32             #print(new_array)
33             #print(new_array.shape) #prints the size of the image example after
   resizing to 50x50 pixels
34             #plt.imshow(new_array, cmap="gray") #shows the example of an image
   after resizing to 50x50 pixels
35             #plt.show()
36
37             break
38         break
39
40 def Create_training_data():
41
42     training_data = []
43
```

```
    for category in CATEGORIES:
        path = os.path.join(dataset_dir, category)
        class_num = CATEGORIES.index(category) #each category prescribed an index number
        for img in os.listdir(path):
            try:
                img_array = cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE) #convert images into a grayscale array
                new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
                training_data.append([new_array, class_num])
            except Exception as e:
                pass

    random.shuffle(training_data)
    return training_data

def Pack_data(training_data):
    x_train = [] #features set
    y_train = [] #labels

    for features, label in training_data:
        x_train.append(features)
        y_train.append(label)

    x_train = np.array(x_train).reshape(-1, IMG_SIZE, IMG_SIZE, 1) # -1 = how many features, 1 = grayscale, 3 = RGB
    y_train = np.array(y_train)

    # saving

    pickle_out = open("X.pickle", "wb")
    pickle.dump(x_train, pickle_out)
    pickle_out.close()

    pickle_out = open("y.pickle", "wb")
    pickle.dump(y_train, pickle_out)
    pickle_out.close()

def Load_data():
    x_train = pickle.load(open("X.pickle", "rb"))
    y_train = pickle.load(open("y.pickle", "rb"))
    return x_train, y_train

def Normalize_data(x_train):
    x_train = x_train/255.0
    return x_train
```

models_training_and_optimization.py

```
import os
import random
import tensorflow as tf
import pickle
import preprocessing_data as prep
import GUI_Tkinker as GUItk

plots_dir = os.path.join(prep.base_path, "plots")
histories_dir = os.path.join(plots_dir, "histories")
colors_dir = os.path.join(plots_dir, "colors")


dense_layers_nums, neurons_per_layer, conv_layers_nums, kernel_sizes = GUItk.run_input_box_with_parameters()

number_of_models = len(dense_layers_nums) * len(neurons_per_layer) * len(conv_layers_nums) * len(kernel_sizes)
model_num = 1

def check_and_create_dir(dir_path):
    if not os.path.exists(dir_path):
        os.makedirs(dir_path)

def Set_NAME(neurons_num, dense_layers_num, conv_layers_num, kernel_size):
    NAME = "{}-neurons-{}-dense-{}-conv-{}-kernel".format(neurons_num, dense_layers_num, conv_layers_num, kernel_size)
    return NAME

class CNN_Model:

    def __init__(self, x_train, CATEGORIES_num, neurons_num, dense_layers_num, conv_layers_num, kernel_size):
        self.model = self.Define_model_architecture(x_train, CATEGORIES_num, neurons_num, dense_layers_num, conv_layers_num, kernel_size)

    def Define_model_architecture(self, x_train, CATEGORIES_num, neurons_num, dense_layers_num, conv_layers_num, kernel_size):

        model = tf.keras.models.Sequential()

        model.add(tf.keras.layers.Conv2D(neurons_num, kernel_size, input_shape = x_train.shape[1:])) # 3x3 pixels window, shape dynamically defined of X, here img_size = 50x50
        model.add(tf.keras.layers.Activation("relu"))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2,2)))

        for _ in range(conv_layers_num-1):

```

```
            model.add(tf.keras.layers.Conv2D(neurons_num, kernel_size,
  input_shape = x_train.shape[1:])) # 3x3 pixels window, shape dynamically
42 defined of X, here img_size = 50x50
43             model.add(tf.keras.layers.Activation("relu"))
44             model.add(tf.keras.layers.MaxPooling2D(pool_size=(2,2)))
45 
46         model.add(tf.keras.layers.Flatten())
47 
48         for _ in range(dense_layers_num):
49             model.add(tf.keras.layers.Dense(64))
50             model.add(tf.keras.layers.Activation("relu"))
51 
           model.add(tf.keras.layers.Dense(len(prep.CATEGORIES))) #for each
52 category
53         model.add(tf.keras.layers.Activation("softmax"))
54 
55         model.compile(optimizer='adam',
56                       loss='sparse_categorical_crossentropy',
57                       metrics=['accuracy']) # what to track
58         return model
59 
60     def Train_model(self, x_train, y_train, tensorboard):
           history = self.model.fit(x_train, y_train, epochs=10,
61 batch_size=32, validation_split=0.3, callbacks=[tensorboard])
62         return history
63 
64     def Save_model(self, NAME):
65         self.model.save('models/{}.model'.format(NAME))
66 
67 def random_color():
       return "#{:02x}{:02x}{:02x}".format(random.randint(0, 255),
68 random.randint(0, 255), random.randint(0, 255))
69 
70 def save_histories_and_colors(histories, colors):
71     check_and_create_dir(histories_dir)
72     with open("{}/histories.pkl".format(histories_dir), "wb") as f:
73         pickle.dump(histories, f)
74 
75     check_and_create_dir(colors_dir)
76     with open("{}/colors.pkl".format(colors_dir), "wb") as f:
77         pickle.dump(colors, f)
78 
79 def Save_TensorBoard_logs(NAME):
       tensorboard =
80 tf.keras.callbacks.TensorBoard(log_dir="logs/{}".format(NAME))
       return tensorboard
```

plotting_learning_curves.py

```python
import os
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import pickle
import models_training_and_optimization as train

def load_saved_histories_and_colors(histories, colors):
    with open(os.path.join(train.histories_dir, "histories.pkl"), 'rb') as f:
        histories = pickle.load(f)

    with open(os.path.join(train.colors_dir, "colors.pkl"), 'rb') as f:
        colors = pickle.load(f)
    print("Loaded Histories:", histories)

    return histories, colors

def plot_learning_curves(histories, colors):
    print("Loaded Histories:", histories)
    metrics = ['loss', 'val_loss', 'accuracy', 'val_accuracy']

    for metric in metrics:

        plt.figure(figsize=(20,30))
        plt.grid(True, color='lightgrey')


        reverse_order = True if 'loss' or 'val_loss' == metric else False
#Sort the histories based on their final metric value
        sorted_names = sorted(histories.keys(), key=lambda x: histories[x][metric][-1], reverse=reverse_order)

        for NAME in sorted_names:
            color = colors[NAME]
            plt.plot(histories[NAME][metric], label=NAME, color=color)


        plt.xlabel('Epoch')
        plt.ylabel(metric.capitalize())
        plt.grid(color='lightgrey')
        plt.title(f'{metric.capitalize()} Over Epochs')

        plt.savefig(os.path.join(train.plots_dir, f"{metric}_Graph.png"))
        plt.close()


        legend_labels = [mpatches.Patch(color=colors[NAME], label=f"{NAME} ({histories[NAME][metric][-1]:.4f})") for NAME in sorted_names] #Create a legend figure
        fig_leg = plt.figure(figsize=(5, 5))
```

```
        ax_leg = fig_leg.add_subplot(111)
        ax_leg.legend(handles=legend_labels)
        ax_leg.axis('off')

        fig_leg.savefig(os.path.join(train.plots_dir,
f"{metric}_Legend.png"), bbox_inches='tight')
        plt.close()


        sorted_names_by_val_acc = sorted(histories.keys(), key=lambda x:
histories[x]['val_accuracy'][-1], reverse=True)#Identify the best model
with the highest final validation accuracy
        best_model_name = sorted_names_by_val_acc[0]


        #Best model's accuracy graph
        plt.figure(figsize=(10, 10))
        plt.plot(histories[best_model_name]['accuracy'], label='Training
Accuracy', color="lightblue")
        plt.plot(histories[best_model_name]['val_accuracy'],
label='Validation Accuracy', color="blue")
        plt.title(f"Accuracy curves for best model: {best_model_name}")
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.savefig(os.path.join(train.plots_dir,
f"Best_Model_Accuracy.png"))
        plt.close()

        #Best model's loss graph
        plt.figure(figsize=(10, 10))
        plt.plot(histories[best_model_name]['loss'], label='Training
Loss', color="lightblue")
        plt.plot(histories[best_model_name]['val_loss'], label='Validation
Loss', color="blue")
        plt.title(f"Loss curves for best model: {best_model_name}")
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.savefig(os.path.join(train.plots_dir, f"Best_Model_Loss.png"))
        plt.close()
```

GUI_Tkinker.py

```
import tkinter as tk

def run_input_box(prompt):
    def on_submit(event=None):

        user_input[0] = input_box.get()
        root.destroy() #Closes the input box after the user clicks the submit button or the Enter key on the keyboard

    root = tk.Tk()#Initializing the box
    root.title("Input Box")
    label = tk.Label(root, text=prompt)
    label.pack()

    input_box = tk.Entry(root, width=50)#Creates the box
    input_box.pack()
    input_box.focus() #Focus the input box

    root.bind("<Return>", on_submit)
    input_box.bind("<Return>", on_submit)
    submit_button = tk.Button(root, text="Enter", command=on_submit)#submit button
    submit_button.pack()

    user_input = [None]#a list stores the input
    root.mainloop()

    return user_input[0]


def run_input_box_with_parameters():
    def on_submit():

        try:

            results['dense_layers_nums'] = [int(x) for x in dense_layers_input.get().split(',')]
            results['neurons_per_layer'] = [int(x) for x in neurons_input.get().split(',')]
            results['conv_layers_nums'] = [int(x) for x in conv_layers_input.get().split(',')]

            input_string = kernel_sizes_input.get().replace(" ", "")
            tuple_strings = input_string[1:-1].split("),(") if input_string.startswith("(") else input_string.split("),(")
            results['kernel_sizes'] = [tuple(map(int, x.split(','))) for x in tuple_strings]

        finally:
            root.destroy()
```

```
44
45      root = tk.Tk()
46      root.title("Configuration Inputs")
47
48      #Dense layers input
49      tk.Label(root, text="Input the numbers of dense layers (lowest
    possible nuber: 0) in the format 0,1,2...").pack()
50      dense_layers_input = tk.Entry(root)
51      dense_layers_input.pack()
52
53      #Neurons per layer input
54      tk.Label(root, text="Input the numbers of neurons per layer (in powers
    of 2) in the format 32,64,128...").pack()
55      neurons_input = tk.Entry(root)
56      neurons_input.pack()
57
58      #Convolutional layers input
59      tk.Label(root, text="Input the numbers of convolutional layers (lowest
    possible nuber: 1) in the format 1,2,3...:").pack()
60      conv_layers_input = tk.Entry(root)
61      conv_layers_input.pack()
62
63      #Kernel sizes input
64      tk.Label(root, text="Input the kernel matrice sizes (lowest possible
    size: (1,1)) in the format (1,1),(2,2),...:").pack()
65      kernel_sizes_input = tk.Entry(root)
66      kernel_sizes_input.pack()
67
68      submit_button = tk.Button(root, text="Submit", command=on_submit)
69      submit_button.pack()
70
71      results = {}
72      root.mainloop()
73
74      return (results.get('dense_layers_nums', []),
75              results.get('neurons_per_layer', []),
76              results.get('conv_layers_nums', []),
77              results.get('kernel_sizes', []))
78
79
80  def run_output_box(prompt):
81      def on_submit():
82          root.destroy()
83
84      root = tk.Tk()
85      root.title("Output Box")
86      label = tk.Label(root, text=prompt)
87      label.pack()
88      submit_button = tk.Button(root, text="Enter", command=on_submit)
89      submit_button.pack()
90
91      root.mainloop()
```