230521_Final_Project_XRay_Models_Multiclass

May 23, 2022

1 Python Setup

[ ]:
```
import os

import matplotlib.pyplot as plt
import numpy
import numpy as np
import scipy
import scipy.integrate
import scipy.special
import tensorflow as tf
from keras import backend as K
from keras.layers import (
    Activation,
    BatchNormalization,
    Conv2D,
    Dense,
    Dropout,
    Flatten,
    MaxPool2D,
    MaxPooling2D,
    Rescaling,
)
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.layers.core import Activation, Dense, Dropout, Flatten
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.utils.vis_utils import plot_model
from PIL import Image
from sklearn.metrics import (
    ConfusionMatrixDisplay,
    accuracy_score,
    classification_report,
    confusion_matrix,
)
from tensorflow import keras
from tensorflow.keras import applications, layers, regularizers
from tensorflow.keras.optimizers import schedules
```

2 Class Setup

[ ]:
```
class Constants_model:
    '''This class is used to define constants that will be used throughout the␣
↪project'''
    train_path = "_data/allray/train/"
    test_path = "_data/allray/test/"
    target_names = ['Gun','Knife','Nothing','Pliers','Scissors','Wrench']


class Xray_models(Constants_model):
    '''Create a xray class model which can predict on xray images'''

    def __init__(self, optimizer, img_rows, img_cols, batch_size):
        '''Initialize the parameters important for the model'''
        self.optimizer = optimizer
        self.img_rows = img_rows
        self.img_cols = img_cols
        self.batch_size = batch_size

    def create_data_gen(self):
        '''
        Create a image data generator where it is specified the split on the␣
↪training set and val set
        returns two image data generators one for training data and one for␣
↪test data as a tuple
        '''
        train_datagen = ImageDataGenerator(validation_split=0.2)
        test_datagen = ImageDataGenerator()
        return (train_datagen, test_datagen)

    def create_data_gen_aug(self):
        '''
        Create a data generator where it is specified the split and the data␣
↪augmentation
        returns two image data generators one for training data augmented and␣
↪one for test data as a tuple
        '''
        train_datagen = ImageDataGenerator(
            validation_split=0.2,
            rotation_range=45,
            horizontal_flip=True,
            vertical_flip=True,
        )
        test_datagen = ImageDataGenerator()
        return (train_datagen, test_datagen)
```

```
    def get_images_(self, augmented=False, hard=False):
        '''
        Create the train,validation and test generator.Each contains the images␣
↪and their labels
        with flow from directory the data is gathered and resized to the dim␣
↪specified then suffled
        returns three generator one for train,for validation and test in form␣
↪of a tuple
        '''
        # load data either with augmentation or in original form
        if augmented == True:
            train_datagen, test_datagen = self.create_data_gen_aug()
        else:
            train_datagen, test_datagen = self.create_data_gen()

        # create training set from allray directory
        train_generator = train_datagen.flow_from_directory(
            self.train_path,
            target_size=(self.img_rows, self.img_cols),
            batch_size=self.batch_size,
            seed=42,
            shuffle=True,
            class_mode="categorical",
            subset="training",
        )
        # create validation set from allray directory
        val_generator = train_datagen.flow_from_directory(
            self.train_path,
            target_size=(self.img_rows, self.img_cols),
            batch_size=self.batch_size,
            shuffle=True,
            seed=42,
            class_mode="categorical",
            subset="validation",
        )
        # create test set from allray directory
        test_generator = test_datagen.flow_from_directory(
            self.test_path,
            target_size=(self.img_rows, self.img_cols),
            batch_size=self.batch_size,
            shuffle=True,
            class_mode="categorical",
        )

        return (train_generator, val_generator, test_generator)

    def create_model(self, model_type):
```

```
        '''
        Create one of the models specified as a parameter,if model mentioned is␣
↪not present return an error message
        for each case we return the model specified
        '''
        if model_type == "vgg16":

            cnn2 = tf.keras.models.Sequential()
            cnn2.add(tf.keras.layers.Rescaling(scale=1 / 127.5, offset=-1))
            cnn2.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3,␣
↪activation='relu', input_shape=[self.img_rows, self.img_cols, 3]))
            cnn2.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn2.add(tf.keras.layers.Conv2D(filters=64, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.Conv2D(filters=64, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn2.add(tf.keras.layers.Conv2D(filters=128, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.Conv2D(filters=128, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.Conv2D(filters=128, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn2.add(tf.keras.layers.Conv2D(filters=256, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.Conv2D(filters=256, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.Conv2D(filters=256, kernel_size=3,␣
↪activation='relu'))
            cnn2.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn2.add(tf.keras.layers.GlobalAveragePooling2D())
            cnn2.add(tf.keras.layers.Dropout(0.5))
            cnn2.add(tf.keras.layers.Dense(4096, activation='relu'))
            cnn2.add(tf.keras.layers.Dense(4096, activation='relu'))
            cnn2.add(tf.keras.layers.Dense(1024, activation='relu'))
            cnn2.add(tf.keras.layers.Dense(6, activation='softmax'))
            # set metrics of interest to recall
            cnn2.compile(optimizer=self.optimizer,␣
↪loss="categorical_crossentropy", metrics=[tf.keras.metrics.Recall()])
            return cnn2

        elif model_type == "transfer_learning":
```

```
            base_model = applications.EfficientNetV2S(
                weights='imagenet',  # Load weights pre-trained on ImageNet
                input_shape=(128, 128, 3),
                include_top=False)

            base_model.trainable = False
            inputs = keras.Input(shape=(self.img_rows, self.img_cols, 3))
            # rescale pixels between -1 and 1, needed for transfer learning␣
↪model
            scale_layer = keras.layers.Rescaling(scale=1 / 127.5, offset=-1)
            x = scale_layer(inputs)
            x = base_model(inputs, training=False)
            x = keras.layers.GlobalAveragePooling2D()(x)
            outputs = keras.layers.Dense(6, activation="softmax")(x)
            model = keras.Model(inputs, outputs)
            model.compile(
                optimizer=keras.optimizers.Adam(learning_rate=0.001),
                loss="categorical_crossentropy",
                # set metrics of interest to recall
                metrics=[tf.keras.metrics.Recall()],
            )
            return model

        elif model_type == "own_model":

            cnn = tf.keras.models.Sequential()
            cnn.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3,␣
↪activation='relu', input_shape=[self.img_rows, self.img_cols,3]))
            cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn.add(tf.keras.layers.Conv2D(filters=64, kernel_size=3,␣
↪activation='relu'))
            cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn.add(tf.keras.layers.Conv2D(filters=64, kernel_size=3,␣
↪activation='relu'))
            cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn.add(tf.keras.layers.Conv2D(filters=128, kernel_size=3,␣
↪activation='relu'))
            cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn.add(tf.keras.layers.Conv2D(filters=256, kernel_size=3,␣
↪activation='relu'))
            cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
            cnn.add(tf.keras.layers.GlobalAveragePooling2D())
            cnn.add(tf.keras.layers.Dropout(0.5))
            cnn.add(tf.keras.layers.Dense(128, activation='relu'))
            cnn.add(tf.keras.layers.Dense(64, activation='relu'))
            cnn.add(tf.keras.layers.Dense(6, activation='softmax'))
```

```
            # set metrics of interest to recall
            cnn.compile(optimizer=self.optimizer,␣
↪loss="categorical_crossentropy", metrics=[tf.keras.metrics.Recall()])
            return cnn

        elif model_type == "svm":

            model = Sequential()
            model.add(Conv2D(filters=32, padding="same", activation="relu",␣
↪kernel_size=3, strides=2, input_shape=(self.img_rows, self.img_cols, 3)))
            model.add(MaxPool2D(pool_size=(2, 2), strides=2))
            model.add(Conv2D(filters=32, padding="same", activation="relu",␣
↪kernel_size=3))
            model.add(MaxPool2D(pool_size=(2, 2), strides=2))
            model.add(Flatten())
            model.add(Dense(128, activation="relu"))
            model.add(Dropout(0.2))
            model.add(Dense(6, kernel_regularizer=regularizers.l2(0.01),␣
↪activation="softmax"))
            model.compile(
                optimizer=self.optimizer,
                # use hinge for two class SVM
                loss = 'squared_hinge',
                # set metrics of interest to recall
                metrics=[tf.keras.metrics.Recall()])
            return model
        else:
            return 'Model not found.Please specify another model'

    def fit_model(self, model_type, epochs, augmented=False, hard_test=False):
        '''
        This function will fit the model to the data.
        It will check if the data is augmented or not and get the specific␣
↪generator.
        It will define the type of callbacks that are used
        returns the model fitted on the training set and the test generator
        '''
        my_callbacks = [
            # stop model if it does not increase after 20 epochs
            tf.keras.callbacks.EarlyStopping(patience=20),
            tf.keras.callbacks.ModelCheckpoint(
                filepath="model.{epoch:02d}-{val_loss:.2f}.h5",
                monitor="val_loss",
                mode="min",
                save_best_only=True,
                verbose=1,
            ), tf.keras.callbacks.TensorBoard(log_dir="./logs")]
```

```
        # check for augmentation
        if augmented == True:
            train_generator, val_generator, test_generator = self.
↪get_images_(augmented=True, hard=False)
        else:
            train_generator, val_generator, test_generator = self.
↪get_images_(augmented=False, hard=False)
        # get the specified model
        model = self.create_model(model_type)
        model.fit(
            x=train_generator,
            validation_data=val_generator,
            epochs=epochs,
            callbacks=my_callbacks,
        )
        return (model, test_generator)

    def evaluate_model(self, model_type, epochs, augmented=False,␣
↪hard_test=False):
        '''
        This function gets the fitted model and then generates the predictions␣
↪on the test set
        return the predicted classes, the true classes and the model
        '''
        # get fitted model and test data
        model, test_generator = self.fit_model(model_type, epochs, augmented)
        STEP_SIZE = test_generator.n // test_generator.batch_size
        prediction_classes = np.array([])
        true_classes = np.array([])

        i = 0
        # contenate batches together to allow for creation of confusion matrix
        for x, y in test_generator:
            i = i + 1
            if i > STEP_SIZE + 1:
                break
            prediction_classes = np.concatenate(
                [prediction_classes, np.argmax(model.predict(x), axis=-1)]
            )
            true_classes = np.concatenate([true_classes, np.argmax(y, axis=-1)])
        return (prediction_classes, true_classes, model)

    def class_report(self, model_type, epochs, augmented=False,␣
↪hard_test=False):
        '''
        Function that generates the classification report after getting the␣
↪prediction classes and true classes
```

```
        prints a classification report specific for the mentioned model
        '''
        prediction_classes, true_classes = self.evaluate_model(
            model_type, epochs, augmented)
        print(classification_report(true_classes, prediction_classes,␣
↪target_names=self.target_names))

    def confusion_matrix(self, model_type, epochs, augmented=False,␣
↪hard_test=False):
        '''
        Function that generates the confusion matrix after getting the␣
↪prediction classes and true classes
        prints a confusion matrix specific for the mentioned model
        '''
        prediction_classes, true_classes = self.evaluate_model(model_type,␣
↪epochs, augmented)
        fig, ax = plt.subplots(figsize=(10, 8))
        ConfusionMatrixDisplay.from_predictions(
            true_classes,
            prediction_classes,
            display_labels=traget_names,
            ax=ax)
        plt.show()
```

3 Create Models

[ ]:
```
# use Adam optimizer
opt = keras.optimizers.Adam(learning_rate=0.0002)

# set images size to 128*128 and batch size to 64
xray = Xray_models(opt, 128, 128, 64)
```

[ ]:
```
# create and train different models
model1 = xray.evaluate_model("svm", epochs=40, augmented=False)
```

[ ]:
```
model2 = xray.evaluate_model("own_model", epochs=30, augmented=False)
```

[ ]:
```
model3 = xray.evaluate_model("vgg16", epochs=40, augmented=False)
```

[ ]:
```
model4 = xray.evaluate_model("transfer_learning", epochs=40, augmented=False)
```

[ ]:
```
models = {"SVM":model1,"Own_Model":model2,"VGG16":model3,"CNN-TL":model4}
```

4 Evaluate Models

[ ]:
```
# print confusion matrix for all models
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(25, 15))
for model_name, ax in zip(models.items(), axes.flatten()):
    name, model = model_name
    prediction_classes, true_classes,model_x = model
    ConfusionMatrixDisplay.from_predictions(
        true_classes,
        prediction_classes,
        display_labels=['Gun','Knife','Nothing','Pliers','Scissors','Wrench'],
        ax=ax,
    )
    ax.title.set_text(name)

plt.tight_layout()
plt.show()
```

[ ]:
```
# print classification report for all models
target_names = ['Gun','Knife','Nothing','Pliers','Scissors','Wrench']
for name,model in models.items():
    prediction_classes,true_classes,model_x= model
    print(name)
    print(classification_report(true_classes, prediction_classes,␣
 ↪target_names=target_names))
```