Proietti

Stefano Rosini

December 2023

Contents

1 Introduction

In this third homework project we're going to see a project in which, given a dataset of fiches and geometric shapes that don't overlap each other, I create two machine learning programs: the first predicts the total value of the fiches in an image, and the second one predicts the number of objects, the number of objects of different shapes and colors, and the total value corresponding to the sum of all rows.

2 Let's talk about Datasets

In the introduction, we briefly described how these two datasets are composed. Now let's look inside them:

2.1 Fiches Dataset

This dataset is composed of a set of images that contain a set of non-overlapping fiches placed on different backgrounds. Here's an image sample that shows how the images are composed:

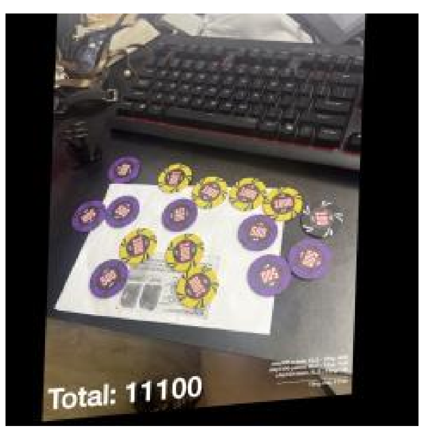


Figure 1: Image sample taken from the dataset

Each image has its own label that contains all the useful information about it and will be used by the MLP to learn information from those images and make predictions about them. The labels are composed of these columns: $< class\_id >< x\_center >< y\_center >< width >< height >$ and their values are normalized between 0 and 1. These columns represent the bounding boxes of the objects associated with a specified class. The latter is described in a specific file of the dataset, called *data.yaml*, which contains an array of the classes of the dataset. Here we show the *data.yaml* and a label file:


```
train: ../train/images
val: ../valid/images
test: ../test/images

nc: 9
names: ['10', '100', '1000', '10000', '20', '5', '50', '500', '5000']
```


(a)


```
1 0.7666666666666667 0.5 0.04375 0.05208333333333336
7 0.2875 0.37916666666666665 0.03333333333333333 0.03333333333333333
7 0.20208333333333334 0.4916666666666664 0.0416666666666664 0.0375
7 0.2833333333333333 0.4791666666666667 0.0416666666666664 0.0395833333333333
7 0.2479166666666667 0.6395833333333333 0.0479166666666667 0.04375
7 0.4229166666666666 0.4854166666666666 0.0333333333333333 0.0395833333333333
7 0.6125 0.5270833333333333 0.0458333333333333 0.04375
7 0.64375 0.6395833333333333 0.0541666666666667 0.04375
7 0.7354166666666667 0.5958333333333333 0.05 0.0541666666666667
2 0.4104166666666665 0.4 0.0333333333333333 0.04375
2 0.3229166666666667 0.5645833333333333 0.0354166666666666 0.05208333333333336
2 0.4354166666666667 0.5791666666666667 0.04375 0.0541666666666667
2 0.4645833333333335 0.6645833333333333 0.0541666666666667 0.0666666666666667
2 0.4958333333333335 0.4333333333333335 0.05 0.0416666666666664
2 0.5875 0.4375 0.0541666666666667 0.04375
2 0.675 0.4604166666666664 0.0604166666666667 0.03125
```


(b)

Figure 2: a) YAML file that contains all the classes of the dataset; b) label file that contains all the information about the bounding boxes associated with a specified class

For how the dataset is built, the MLP will learn to recognize the objects associated with a specific class, which is the **value** of the fiche.

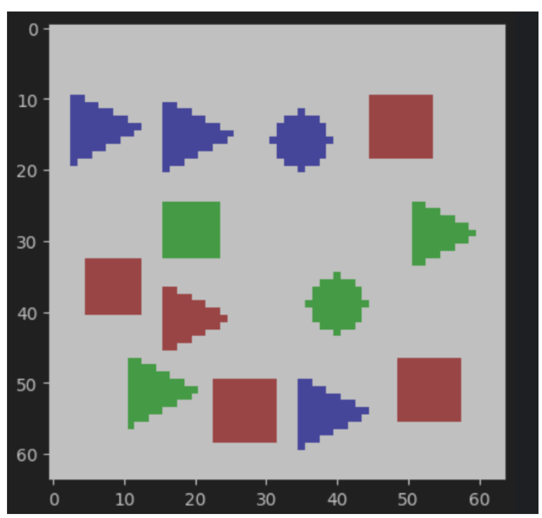
In the attached file, the name of the folder that contains the dataset is *DatasetFichesMultilabeling*.

2.2 Geometric Shapes Dataset

This dataset, on the contrary, is a synthetic dataset composed of a series of images that contain a set of geometric shapes (square, triangle, circle) with different colors (red, green, blue) and, as for the first dataset, it contains labels that describe those images. The label is an array that contains all the information about the image:

Image 0	Target
Total objects	13.00
Total squares	5.00
Red squares	4.00
Green squares	1.00
Blue squares	0.00
Total triangles	6.00
Red triangles	1.00
Green triangles	2.00
Blue triangles	3.00
Total circles	2.00
Red circles	0.00
Green circles	1.00
Blue circles	1.00
Value	33.00

(a)



(b)

Figure 3: a) Representation of a label of an image; b) Synthetic generated image

3 Let's Talk About The Models

After describing how the datasets are made, let's now focus on the models implemented for predicting values.

3.1 YoloV8

Yolov8 is an advanced version of the YOLO family (You Only Look Once) for computer vision and it was designed by Ultralytics. It is a model well suited for real-time *object detection*, *segmentation*, and *classification*. In this image we show the architecture of Yolov8:

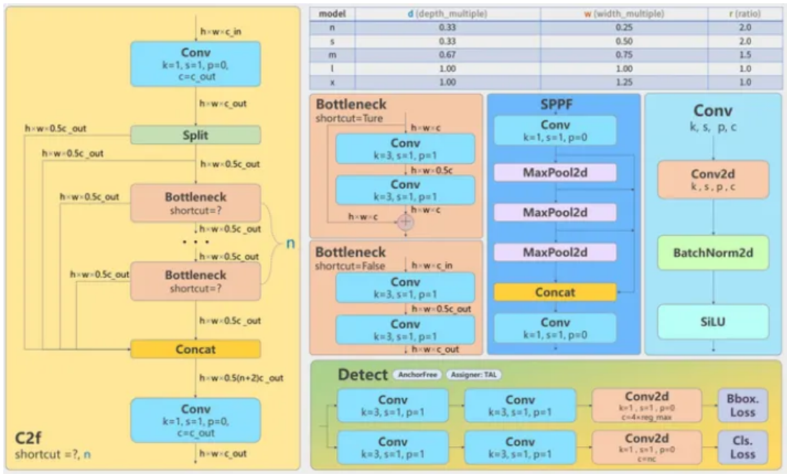


Figure 4: Architecture of the Yolov8

From the figure above, we can see that the Yolov8 architecture is divided into 4 main blocks:

- Backbone: The encoder block that extracts features from images;
- Bottleneck: An intermediate block that tries to learn deeper features;
- SPPF: Combines local and global information using convolutional layers. The output is passed to the Detect block;
- Detect: The block that produces the output of the model. It outputs the coordinates of the bounding boxes and the classes of the objects.

Since the labels and the YAML file are made in a YOLO-like format (bounding boxes and classes), I decided to use the Yolov8 model and train it on this specific dataset. These are the parameters that I used for this specific model:

- Epochs = 100;
- Image size = 640*640;
- Batch size = 16;

After training, this model returned the following validation results for all classes:

```
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  5.19it/s]
                   all         25        480      0.995      0.999      0.995      0.849
                    10         17         76      0.998          1      0.995       0.86
                   100         11         56      0.997          1      0.995      0.871
                  1000         10         62      0.997          1      0.995      0.875
                 10000          5          9      0.981          1      0.995      0.886
                    20         18         67      0.998          1      0.995      0.831
                     5         19         75          1      0.992      0.995      0.729
                    50         17         64      0.999          1      0.995       0.84
                   500         10         47      0.986          1      0.995      0.887
                  5000          5         24      0.995          1      0.995      0.862
Speed: 0.2ms preprocess, 3.0ms inference, 0.0ms loss, 1.0ms postprocess per image
Results saved to runs\detect\chip_detector
```

Figure 5: Validation Results of the model trained with this dataset. From what we can see from the results, we can say that the model has generalized our problem very well

After training, we proceeded to the testing phase, in which we took the test images and checked whether the model had really learned how to predict values. We input the images and used the ground truth labels of those images to verify whether the predictions were correct. Here we show the results produced by the model:

1	image	class_0_pred	class_1_pred	class_2_pred	class_3_pred	class_4_pred	class_5_pred	class_6_pred	class_7_pred	class_8_pred	total_value	class_0_true	class_1_true	class_2_true	class_3_true	class_4_true	class_5_true	class_6_true	class_7_true	class_8_true
2	IMG_3900_JPG_jpg.rf.e77c9af39c17e6947dd9c	7	7	2	1	1	2	3	2	0	13950	7	7	2	1	1	2	3	2	0
3	IMG_3907_JPG_jpg.rf.157b942550078aa5b8eb	5	3	6	1	5	1	2	1	6	47055	5	3	6	1	5	1	2	1	6
4	IMG_3909_JPG_jpg.rf.6f77b590fed8d6cef8ea5	8	6	6	1	5	5	3	2	6	47955	8	6	6	1	5	5	3	2	6
5	IMG_3909_JPG_jpg.rf.8c71f717f098eb1402f3e	9	6	6	1	5	5	3	2	6	47965	8	6	6	1	5	5	3	2	6
6	IMG_3910_JPG_jpg.rf.81401de6b6bbafc0885ff	8	6	6	1	5	5	3	2	6	47955	8	6	6	1	5	5	3	2	6
7	IMG_5900_JPG.rf.dbf312dc206693062ab97347	2	1	1	0	2	6	1	2	1	7240	2	1	1	0	2	6	1	2	1
8	z4517191157908_8aa21b49ee0f81c5326befe1(	0	0	0	0	0	9	0	0	0	45	0	0	0	0	0	9	0	0	0
9	z4517191307024_a6ae0cdd4872cd632906c2e3	9	1	0	0	7	4	5	0	0	600	9	1	0	0	7	4	5	0	0
10	z4517191378727_6baa2a9b191070e2a22956b	4	0	0	0	6	4	5	0	0	430	4	0	0	0	6	4	5	0	0
11	z4517191378727_6baa2a9b191070e2a22956b	4	0	0	0	6	4	5	0	0	430	4	0	0	0	6	4	5	0	0
12	z4517191461511_a985a513ce666d9ac394ab13	1	0	0	0	0	3	5	0	0	275	1	0	0	0	0	3	5	0	0
13	z4517191461511_a985a513ce666d9ac394ab13	1	0	0	0	0	3	5	0	0	275	1	0	0	0	0	3	5	0	0
14	z4517191461511_a985a513ce666d9ac394ab13	1	0	0	0	0	3	5	0	0	275	1	0	0	0	0	3	5	0	0
15	z4517191529684_04a6fb1e01cc7a07d3681760	0	0	0	0	5	0	0	0	0	100	0	0	0	0	5	0	0	0	0
16	z4517191631070_3c72ff4bef78307bc669d1104	2	0	0	0	5	2	3	0	0	280	2	0	0	0	5	2	3	0	0

Figure 6: Table comparing the ground truth (class_true) with the predicted results

```
Precision per classe:
  class_0: 0.9811
  class_1: 1.0000
  class_2: 1.0000
  class_3: 1.0000
  class_4: 1.0000
  class_5: 1.0000
  class_6: 1.0000
  class_7: 1.0000
  class_8: 1.0000
```

Figure 7: Precision Metric showing how well the model performed

The model also produced output images for the test set. Here we show just one example of those outputs:

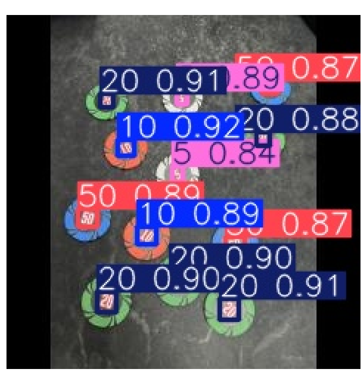


Figure 8: Example of output image of the model

3.2 Synthetic Model

The second dataset didn't have label shapes suitable for the YOLO model, so I had to build a new model suitable for this data format. For this model, I used the TensorFlow library. This model has the same structure as VGG, consisting of a decoder part composed of a series of convolutional layers and a series of fully connected layers for the output section. In this image, I provide the structure of the model:

```python
# Define the CNN model
def create_model(input_shape, output_dim):
    model = Sequential([
        # Input layer
        Input(shape=input_shape),
        # First block: Conv + BatchNorm + ReLU + MaxPool
        Conv2D(32, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        Conv2D(32, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),
        # Second block: Conv + BatchNorm + ReLU + MaxPool
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),
        # Third block: Conv + BatchNorm + ReLU + MaxPool
        Conv2D(128, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        Conv2D(128, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),
        # Fourth block: Conv + BatchNorm + ReLU + MaxPool
        Conv2D(256, (3, 3), activation='relu', padding='same'),
        BatchNormalization(),
        Conv2D(256, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),
        # Fifth block: Global Average Pooling (for reducing the
         number of parameters)
        GlobalAveragePooling2D(),
        #Flatten(),
        # Dense layer with regularization
        Dense(1024, activation='relu'),
        Dense(512, activation='relu'),
        Dense(256, activation='relu'),
        #Dropout(0.5),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        # Output layer
        Dense(output_dim, activation='linear')
    ])
    return model
```

Figure 9: Model structure

I used the Adam optimizer, the *mean_squared_error* loss, and a learning rate of 0.001 for training the model. As a validation metric, in addition to the validation loss, I used the mean absolute error.

For this model, I also implemented a callback that saves the model's weights whenever they improve during training. Then, with all this setup, I trained the model:

```
from keras.src.callbacks import ModelCheckpoint

# Filepath to save the best weights
checkpoint_filepath = 'best_model_v4.weights.h5'

# Define the ModelCheckpoint callback
checkpoint_callback = ModelCheckpoint(
    filepath=checkpoint_filepath,        # File to save weights
    monitor='val_loss',                  # Metric to monitor
    save_best_only=True,                 # Save only the best
     weights
    save_weights_only=True,              # Save only weights
     (not the entire model)
    mode='min',                          # Minimize the
     monitored metric (e.g., loss)
    verbose=1                            # Print messages when
     weights are saved
)
Executed at 2025.08.05 18:25:58 in 2ms

# Train the model
history = model.fit(train_images, train_labels,
                    validation_data=(val_images, val_labels),
                    epochs=30,
                    batch_size=32,
                    callbacks=[checkpoint_callback])
Executed at 2025.08.05 18:41:02 in 15m 3s 256ms
```

Figure 10: Code for starting the training

At the end of the training, I obtained the following values:

```
Epoch 30: val_loss did not improve from 0.02754
313/313 ━━━━━━━━━━━━━━━━━━━━ 31s 99ms/step - loss: 0.0457
 - mae: 0.1229 - val_loss: 0.0712 - val_mae: 0.1689
```

Figure 11: Values returned after the training phase. From the results we can see how well the model has generalized the problem

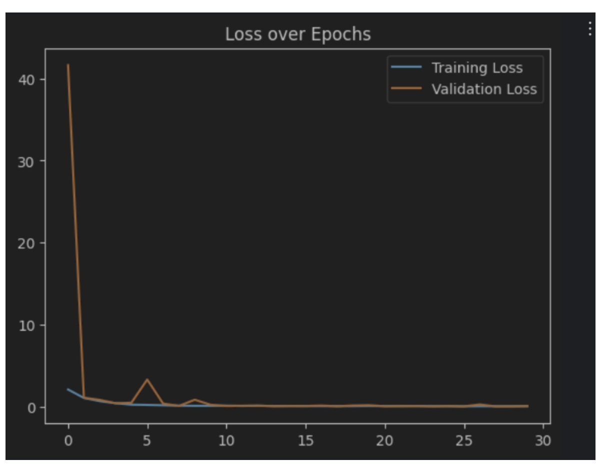


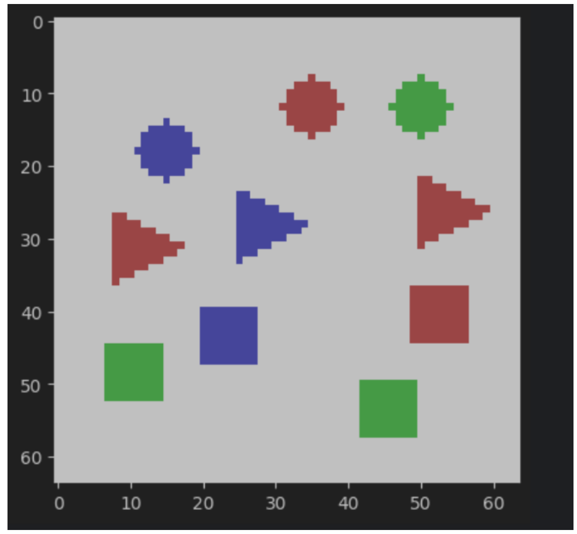
Figure 12: Plot showing how the loss and validation behaved during training

In the end, a testing phase was conducted to verify whether the model correctly predicted the images. I tested 1000 images and used their associated labels as ground truth. After testing, I obtained a **precision** of **85%** over 1000 images. Here I show an example of a well-predicted test image and another example of a poorly predicted one:

14 rows 14 rows × 3 columns Static Output ?

Image 1	Target	Prediction
Total objects	10.00	10.00
Total squares	4.00	4.00
Red squares	1.00	1.00
Green squares	2.00	2.00
Blue squares	1.00	1.00
Total triangles	3.00	3.00
Red triangles	2.00	2.00
Green triangles	0.00	-0.00
Blue triangles	1.00	1.00
Total circles	3.00	3.00
Red circles	1.00	1.00
Green circles	1.00	1.00
Blue circles	1.00	1.00
Value	25.00	25.00

(a)



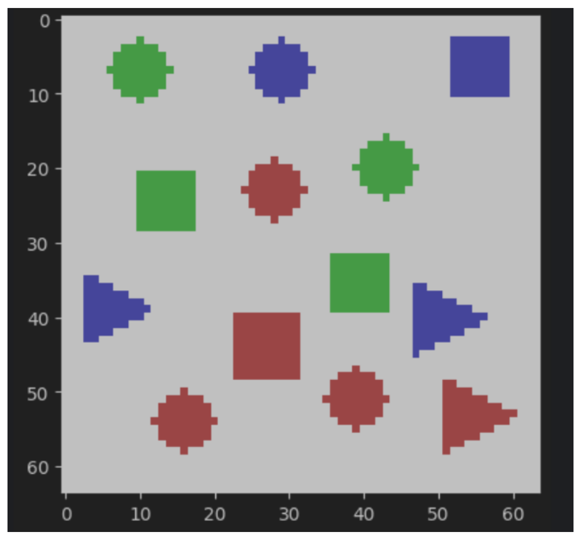
(b)

Figure 13: a) Comparison between ground truth (target) and predictions; b) the image input

14 rows 14 rows × 3 columns Static Output ?

Image 4	Target	Prediction
Total objects	13.00	13.00
Total squares	4.00	4.00
Red squares	1.00	1.00
Green squares	2.00	2.00
Blue squares	1.00	1.00
Total triangles	3.00	3.00
Red triangles	1.00	1.00
Green triangles	0.00	-0.00
Blue triangles	2.00	2.00
Total circles	6.00	6.00
Red circles	3.00	3.00
Green circles	2.00	2.00
Blue circles	1.00	1.00
Value	33.00	34.00

(a)



(b)

Figure 14: Example of a bad prediction: a) comparison between ground truth and predictions; b) test image input

Although the algorithm labeled the prediction as a mistake, we can see that the model missed the ground truth values by an offset of about (+1, -1). The metric I used labeled the prediction as a **mistake** if at least one value in the sequence was incorrect, and **good** if the entire sequence was perfectly predicted.