Group Project 1

Machine Learning III

Group D

Alexandre Bouamama, Tarek ElNoury, Guillermo Germade, Alain Grullón, Roberto Picón, Rebecca Rosser

Summary

1. CATS & DOGS

[1 point] TODO 1. Add a convolutional layer with the specifications defined in the code.

```
# Our input feature map is 150x150x3: 150x150 for the image pixels, and 3 for
# the three color channels: R, G, and B
img_input = layers.Input(shape=(150, 150, 3))

# First convolution extracts 16 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(16, 3, activation='relu')(img_input)
x = layers.MaxPooling2D(2)(x)

# TODO 1: Create the second convolution layer followed by a MaxPooling2D layer
# Second convolution extracts 32 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)

# Third convolution extracts 64 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
```

[1 point] TODO 2. Flatten the output of the last convolutional layer to add a dense layer.

```
# TODO 2: Flatten the output of the precedent layer
# Flatten feature map to a 1-dim tensor so we can add fully connected layers
x = layers.Flatten()(x)
```

[1 point] TODO 3. Add the dense layer (the one before the last output layer) with the specifications defined in the code.

```
# TODO 3: Create the fully connected layer(dense)
# Create a fully connected layer with ReLU activation and 512 hidden units
x = layers.Dense(512, activation='relu')(x)
```

2. REGULARIZATION

[1 point] TODO 1. Data Augmentation. Explain how is working ImageDataGenerator. Specifically, explain what all the parameters, already set, and their respective values mean. One by one, from rotation_range to fill_mode.

One big issue in Machine Learning models based on images is the lengths of the datasets used to train, test and validate the models. Keras's ImageDataGenerator is a function that can help us answer this issue. Its use is called *image augmentation*. This means that it is using each image and creates

various versions of the same image, just changing some simple shape features (by rotating, shifting, flipping, etc.). This allows to add some more data to the dataset and also create robust outputs.

Here is one visual representation of how that may work:

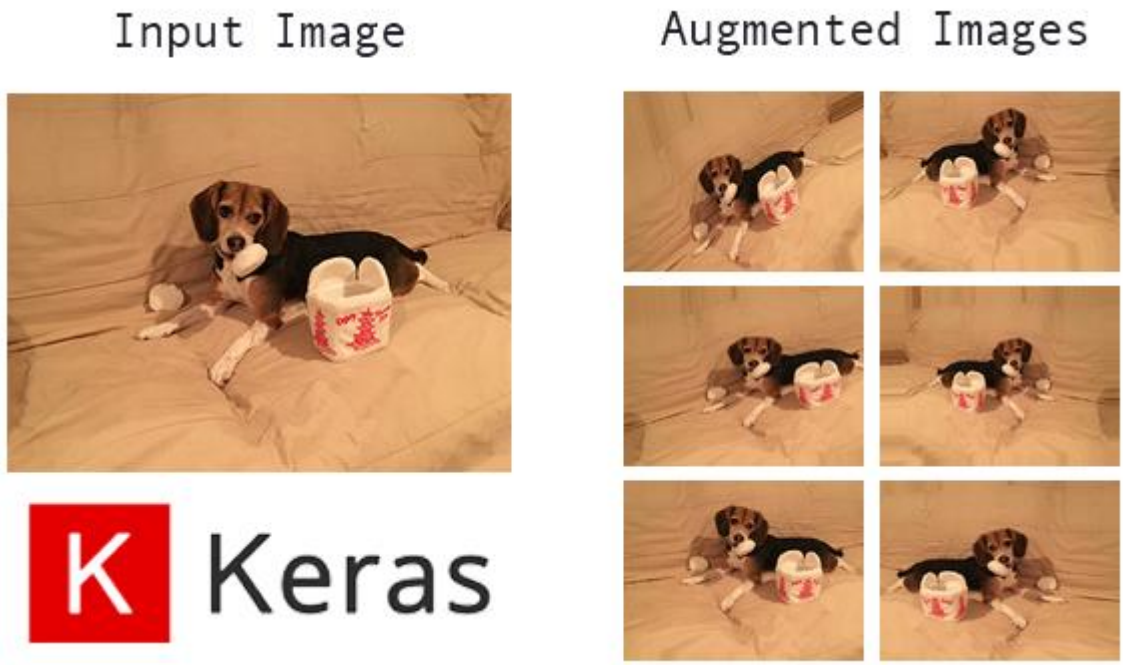

Figure 1: Visual representation of what ImageDataGenerator does[1]

In the following part of the answer we will focus on the parameters from rotation_range to fill_mode (following the order present in the Tensorflow website):

- rotation_range (Integer): This parameter can take any integer value from 0 to 360. This enables to rotate the image by any value on each side randomly by amount of degrees on the same plane.
- width_shift_range (Float): If the value is a float number, it indicates the percentage of width of the image to shift (horizontal shift). If it's an integer it will be understood as the number of pixels by which the image will be shifted randomly between each maximal value ([-x,+x]).
- height_shift_range (Float): If the value is a float number, it indicates the percentage of height of the image to shift (vertical shift). If it's an integer it will be understood as the number of pixels by which the image will be shifted randomly between each maximal value ([-x,+x]).
- brightness_range (Tuple or list of two floats, ex.: [0.5,1.2]): This parameter allows to randomly change the lightning (or brightness) of the image. Values under 1 make the image darker, while values over 1 makes it lighter. As per the same logic as the previous parameters, this allows to create a range to randomly assign changes.
- shear_range (Float): This parameter allows to change the shear angle. To make that more understandable, here is a representation of what shear is:

[1] Source: https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/

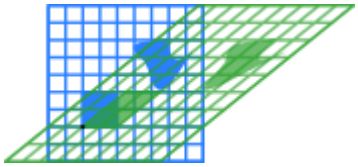

Figure 2: Representation of a shift in shear[2]

As per the same logic as the previous parameters, this allows to create a range to randomly assign changes.

- zoom_range (Float or list of two floats, ex.: [0.5,1.2]): If just an integer is given, the parameters will take the value [1-integer,1+integer]. This parameter will randomly zoom in (value under 1) or zoom out (value over 1). As per the same logic as the previous parameters, this allows to create a range to randomly assign changes.
- channel_shift_range (Float): This allows to set a range from which the images will be randomly shifted according to the channel values (it is a change in saturation). Here is an example of such change:

Figure 3: Representation of a channel shift[3]

- fill_mode (One of {"constant", "nearest", "reflect" or "wrap"}): This parameter allows to fill the image after it has been modified. Indeed, changes will lead to some parts of the image being empty and this parameter allows to fill those pixels either with a constant color, the nearest color, the reflected color or the wrapped color. Default is 'nearest'. According to the Tensorflow website, those are the ways it does the filling:
  - 'constant': kkkkkkkk|abcd|kkkkkkkk (cval=k)
  - 'nearest': aaaaaaaa|abcd|dddddddd
  - 'reflect': abcddcba|abcd|dcbaabcd
  - 'wrap': abcdabcd|abcd|abcdabcd

Figure 4: Example of fill_mode = "nearest"[4]

[2] Source: https://en.wikipedia.org/wiki/Shear_mapping
[3] Source: https://towardsdatascience.com/exploring-image-data-augmentation-with-keras-and-tensorflow-a8162d89b844
[4] Source : https://fairyonice.github.io/Learn-about-ImageDataGenerator.html

[1,5 points] TODO 2. Dropout. • Explain Dropout as a regularization technique. • Add a Dropout layer with a dropout rate of 0.2. • Try dropout of 0.9 and 0.1 rate. Explain the different behaviour.

Another issue in Machine Learning models is overfitting. We want to have a model that understands the global characteristics of the data, but we also want to avoid creating a model that focuses on reproducing every detail of the train set. One solution for that is to add a Dropout layer to our model. The amount set in the parenthesis is the rate at with the dropout occurs. When dropout occurs, it sets the input of the neurons at 0 in this particular training phase.

Here is the code used to add a Dropout layer with a rate of 0.5:

```
30 # TODO 2. Add a dropout layer with a dropout rate of 0.5
31 x = layers.Dropout(0.5)(x)
```

Since the assignment asks to add a dropout rate of 0.2 but in the code, the TODO asks for a rate of 0.5, in the following table we show the difference is accuracies and losses that we have with 5 different dropout rates:

Dropout rate	Train accuracy	Validation accuracy	Train loss	Validation loss
0	0.995	0.723	0.029	3.393
0.1	0.992	0.696	0.029	3.814
0.2	0.996	0.705	0.014	2.962
0.5	0.994	0.722	0.019	2.827
0.9	0.941	0.721	0.147	1.543

From the empirical results we got from testing different dropout rates, we see that an increase in dropout has a negative impact on the train accuracy and on the validation loss. On the contrary it has a mixed impact on the validation accuracy and the train loss. This can be interpreted understanding that adding this forgetting feature, the train set tends to overfit less, which, in return, has a positive impact on the robustness of the model (validation loss). Nonetheless, extreme values of dropout rates can have an overall negative impact (there's a trade-off between validation loss and validation accuracy when regarding Dropouts).

[0,5 points] TODO 3. Fit the final model following the specifications in the code

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size=(150, 150),  # All images will be resized to 150x150
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

# Flow validation images in batches of 20 using val_datagen generator
validation_generator = val_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')


history = model.fit(
      train_generator,
      steps_per_epoch=100,  # 2000 images = batch_size * steps
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50,  # 1000 images = batch_size * steps
      verbose=2)
```

3. STAND ON THE SHOULDERS OF GIANTS

[1 point] TODO 1. Do some research and explain the inception V3 topology. Which database do they use for training it? How was trained?

Keras's InceptionV3 is an algorithm based on the article *Rethinking the Inception Architecture for Computer Vision* published on December, 2nd 2015. First of all, InceptionV3 is a convolutional neural network. This means that the main architecture of the network is to transform an input tensor with the shape number of images x height x width x depth into what are called *convolutional* layers, which transform the image into feature maps. The different features are then used to produce the desired output. In other words, convolutional layers are layers that deal with parts of an image (for example a 5 by 5 pixels shape).

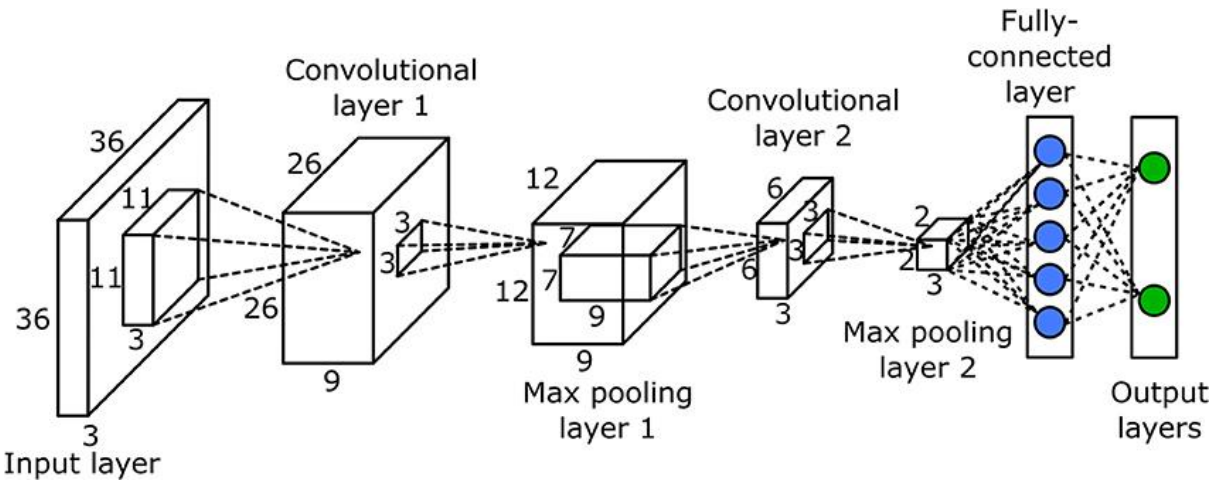

Figure 5: Example of Convolutional and Max pooling layers[5]

The main issue that arose with convolutional networks are the sizes of the images to recognise. For example, in the following set of racoon images, we can see that they vary in shapes, sizes and perspectives. This led to ask the question of how to better classify those varying features.

Figure 6: Some images of racoons[6]

The problem can be understood from three angles:

1. Because the information changes in location for every image, how can we find a constant kernel to get a steady and robust information across all items of the datasets? Small kernels are good for small information and bigger kernels are better for global information.
2. Augmenting the number of layers can create overfitting.

[5] Source: https://brilliant.org/wiki/convolutional-neural-network/
[6] Source: https://towardsdatascience.com/how-to-train-your-own-object-detector-with-tensorflows-object-detector-api-bec72ecfe1d9

3. Augmenting the number of layers costs computing power.

The solution that the Inception architecture proposes is to mix the sizes of the windows in a same layer. This can help get local and global information without creating neural networks that are too deep (it only augments the width of the layers, that is, the number of neurons per layer).

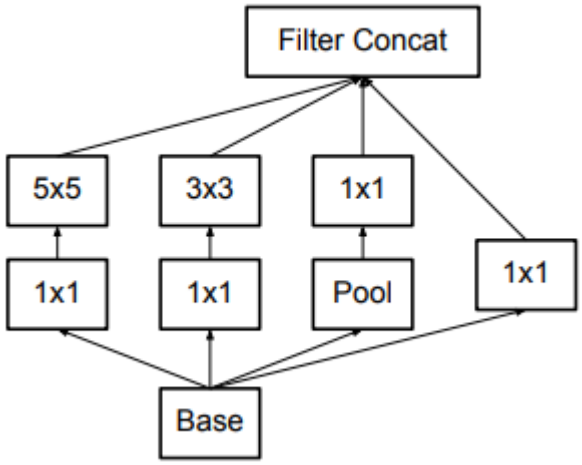

Figure 7: Original Inception Model[7]

The next generation of Inception models are just better versions in terms of computing resources and accuracy. As stated in the paper *Rethinking the Inception Architecture for Computer Vision*, InceptionV3 beats its benchmark with a modest increase in computing power compared to previous networks.

Regarding the database used for the training of Inception V3, the Keras website explains that the model was trained on ImageNet, which is a database of more than 14 million images and 20,000 categories. As for the training method, the model is already pre-trained and the weights are imported through the following code: *pre_trained_model.load_weights(local_weights_file)*.

[1 point] TODO 2. Inception V3 is set by default to admit input images of (299, 299, 3) dimensions; but we want (and we will use) inputs of (150, 150, 3). If the net is already trained, how is that even possible?

Since InceptionV3 functions based on convolutional and pooling layers that are no bigger than 7x7, the shape of the image input that be changed in some ways. Of course, one must be careful of not inputting images that are too small (Keras advises widths and heights no smaller than 75 each[8]). Even if the net is trained this is possible, because the layers that are trained are layers based on smaller parts of the images. Furthermore, the goal of the Inception architecture is to be able to locate information that can be highly changing in sizes (as per our previous example of the racoons, you can see that from one

[7] Source: *Rethinking the Inception Architecture for Computer Vision*, Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, Zbigniew Wojna; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2818-2826, p.4

[8] Source: https://keras.io/api/applications/inceptionv3/

image to another the size of their heads for example varies). Changing the input size is therefore not a problem unless the input is too small to be efficiently used (here 75x75x3).

[2 points] TODO 3. Let's change the database. Will Inception work well with different objects? Use Inception V3 to outperform the results of a small convnet in a flower database. Complete the code in 04_cnn_template.ipynb to use Inception V3 in this database in the same way we did it for cats & dogs.

You will find the code in the notebooks attached.

As we can see on the notebook, the previous convolutional neural net we used will give a validation accuracy of around 70%. When applying Inception V3, we see that we achieve to increase the accuracy to around 82%. This shows the ability Inception V3 has to outperform a small convnet. We believe that this is due to the important range of the training set of the model (database of more than 14 million images and 20,000 categories, as previously stated). Furthermore, it is important to note that we have to change the loss function to sparse categorical cross entropy because we have more than 2 classes of images to classify and that the input size has to be changed to (100,100,3) since all images were resized to 100x100.

4. Object Detection. YOLO nets [Advanced & Optional] [1 extra point]

TODO 1. Describe YOLO neural net – no matter the version. I just need to know you know what you are doing

The YOLO neural net does not stand for You Only Live Once but is a play on words on that particular expression. In Machine Learning, YOLO means You Only **Look** Once. The name of the neural net itself give us a hint on its goal: image detection. The other objective of YOLO is to detect objects in a faster(it processes images in real-time at 45 frames per second) and more reliable way, but how does it do all that?

The main answer can be found in the original paper that made the YOLO family emerge, namely *You Only Look Once: Unified, Real-Time Object Detection* (2015) by Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi[9]. Previous techniques in images detections aimed at dissecting an image into location, identifying them and scaling them back. YOLO takes another approach. Its CNN, through a single regression problem, creates *bounding boxes* and associates probabilities of objects. The idea is that the bounding box with the highest probability of fitting the object is the one with the highest score. A simple explanation is found in the article, as shown below.

[9] Source: https://arxiv.org/pdf/1506.02640.pdf

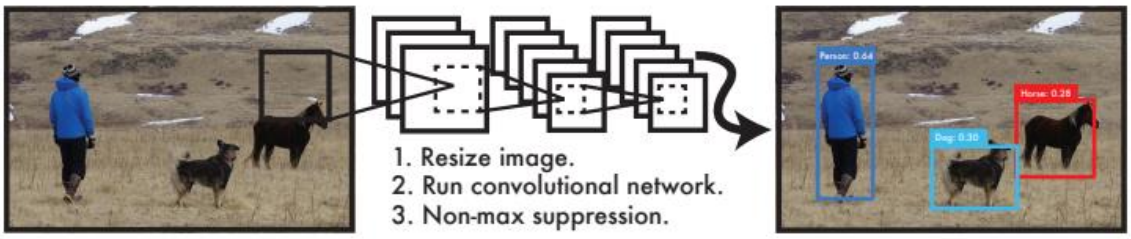

Figure 1: The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to 448 × 448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

Figure 8: Explanation of the YOLO Detection System[10]

On a more in-depth description of the model, we can see that YOLO models divide images in multiple bounding boxes (varying in shapes), this aims at finding the location of the objects. Another task is to create a map of categories (with grids shaped x by x). Each grid is then associated with a probability of being part of an object (image of a dog, or a man, or a bike, etc.). According to those probabilities, the model then aggregates the different grids and their probabilities. The bounding boxes that contains a majority of one category showcase the bounds of the object detected and its nature. The second figure present in the paper illustrates this:

[10] Ibid. p.1

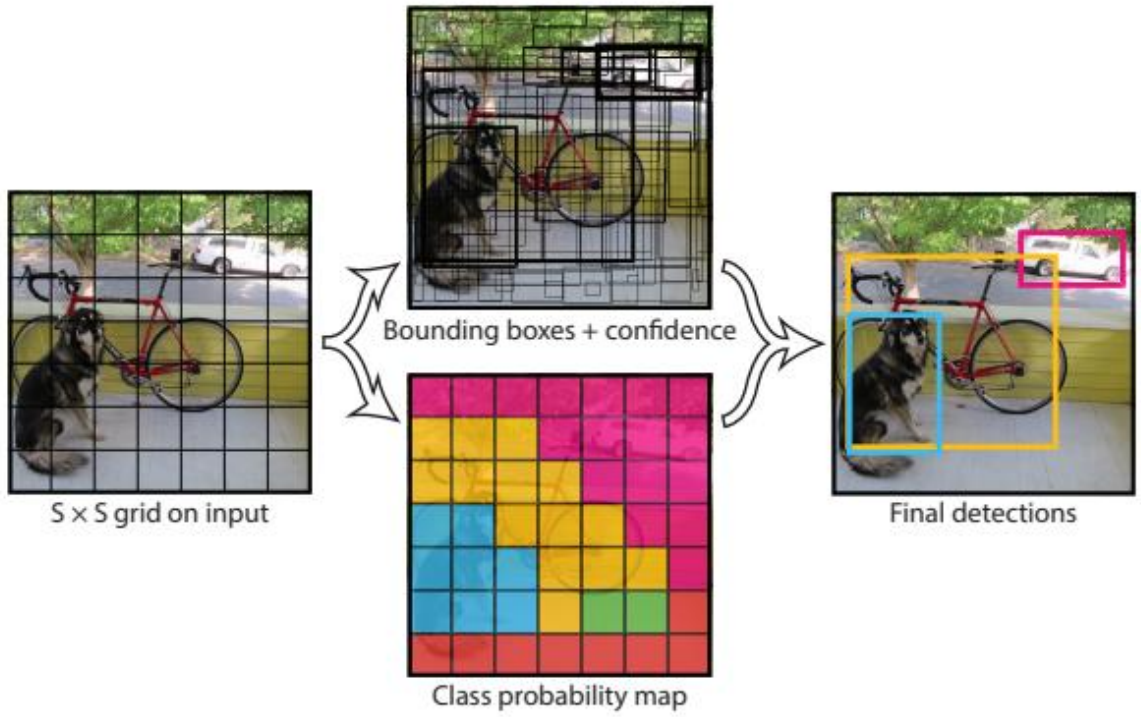

Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

Figure 9: Explanation of the Model[11]

TODO 2. Put to work YOLO_v5 in the database you want and send to me the notebooks with your comments.

You will find the code in the notebooks attached. We used a dataset of chess pieces.

The main takeaways that we understood from this part of the assignment come in two ways. First of all, we saw the efficiency of the YOLO_v5 model. This is shown by the increase in the accuracy metrics (mAP, i.e figure 10). Secondly, this task enabled us to see a visual example of how data is classified (figure 11) and also of how data is augmented (figure 12).

[11] Ibid. p.2

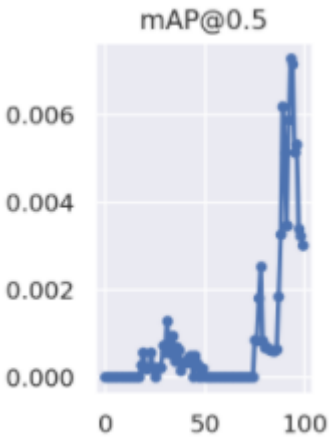

Figure 10: mAP results

Figure 11: Ground truth of the training data

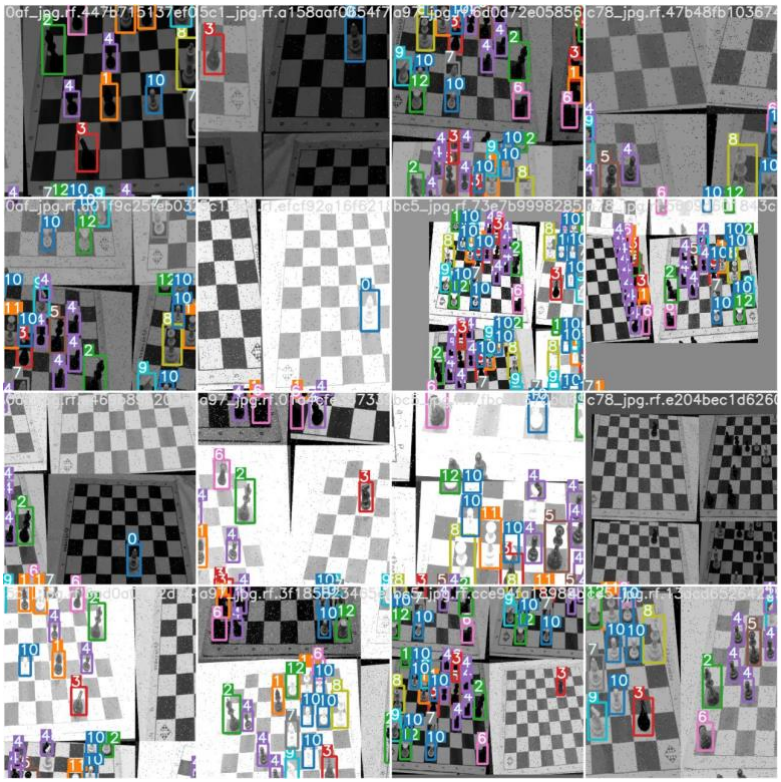

Figure 12: Ground truth of the augmented data