Machine Learning III

Practice I: Group F

CONVOLUTIONAL NEURAL NETS

Group F:

Agathe de Drouas, Francisco Fortes, Andrea Fusetti, Milan Goetz, Eleanor Manley, Tamara Amer Nayef Qassem, Luisa Runge

1. CATS & DOGS

Refer to notebook: 01_cnn_GroupF.ipynb

1) Add a convolutional layer with the specifications defined in the code.

```
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
```

2) Flatten the output of the last convolutional layer to add a dense layer.

```
x = layers.Flatten()(x)
```

3) Add the dense layer (the one before the last output layer) with the specifications defined in the code.

```
x = layers.Dense(512, activation='relu')(x)
```

2. REGULARIZATION

Refer to notebook: 02_cnn_GroupF.ipynb

1. Data Augmentation. Explain how it 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.

Data Augmentation using ImageDataGenerator allows us to have a more generalized classifier, improving the model performance and tackling overfitting.

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

datagen = ImageDataGenerator(
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

`Rotation_range`

Data augmentation is creating new data based on modification of our existing training data. Here we are first rotating the images clockwise by a 40 degree angle. It can happen that it rotates pixels out of the image frame and leaves areas of the frame with no pixel so the nearest-neighbor fills in the empty areas. This happens since we added the option `fill_mode='nearest'` at the end of the code.

`Width_shift_range & height_shift_range`

Then we are shifting the picture both horizontally and vertically keeping the image dimensions the same. In this case again we will have some of the pixels clipped off the image and to solve this the pixel values at the edge of the image are duplicated to fill in the empty part of the image created by the shift. (`fill_mode='nearest'`)

With the function `width_shift_range` we are moving all the pictures first horizontally. We indicate with a percentage (between 0 and 1 here we chose 0.2 so 20%) of the width of the image by which it is to be randomly shifted. This shift is either left or right. It will pick a sample value between the percentage indicated and no shift for each image to execute the shift. You can also specify an array and it will select randomly from this array the shift to perform. Like the horizontal shift We do the same with a vertical shift `height_shift_range` (either up or down) specifying the percentage of the image to shift as 0.2 the height of the image.

`Shear_range`

The Shear transformation is titling the image and changing its shape. It fixes one axis and stretches the image by moving each point in this fixed direction. It stretches the image at a certain angle, the shear angle here 0.2 radians, 57.3º degrees, in counter-clockwise direction.

`Zoom_range`

This feature allows us to randomly zoom in (if the value is less than 1) or out (if the value is greater than 1) of the picture. Again some pixels will be added in case of zoom out around the image. Here the zoom range is 0.2, this will result in a 20% zoom in. There is no need for additional pixel in this zoom.

`Horizontal_flip`

The horizontal flip means reversing the columns of pixels of the image. If we would to do it vertically it would inverse the rows of pixels. It makes more sense to do it horizontally since it is unlikely to see a picture of a dog or cat upside down.

1) Explain Dropout as a regularization technique.

Another form of regularization is the dropout technique. It works by randomly "dropping out" nodes in the network during the training for a single gradient step. During the training some units are ignored (temporarily removed from the network, along with all its incoming and outgoing connection), making the network look like its layers have a different number of nodes. It can be used in most of the layers and for more than one hidden layer but it can’t be used for the output layer. This produces a network that becomes less sensitive to the specific weights of neurons. Which in turn results in a better generalization and less overfitting. When computing the dropout we have to choose a hyperparameter that specifies the probability at which outputs of the layer are dropped out. The more you drop out, the stronger the regularization. But if you select 1.0 it will drop everything and you won’t learn anything and if you pick 0.0 there will be no dropout regularization, so an intermediate value is more useful to reduce the overfitting. This method is also beneficial because it is computationally cheap and very effective.

- Add a Dropout layer with a dropout rate of 0.2.

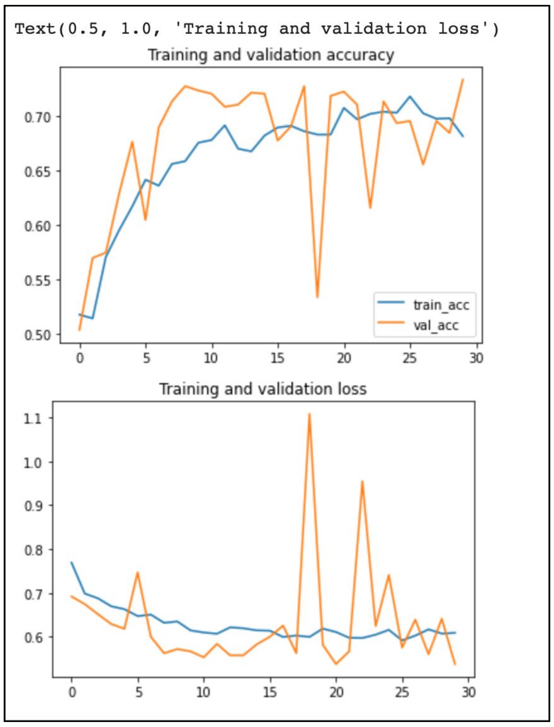
We add a dropout layer between the last hidden layer and the output layer. We try inserting different dropout rates and see the result of each.

```
x = layers.Dropout(0.2)(x)

x = layers.Dropout(0.5)(x)
```

- Try dropout of 0.9 and 0.1 rate. Explain the different behavior.

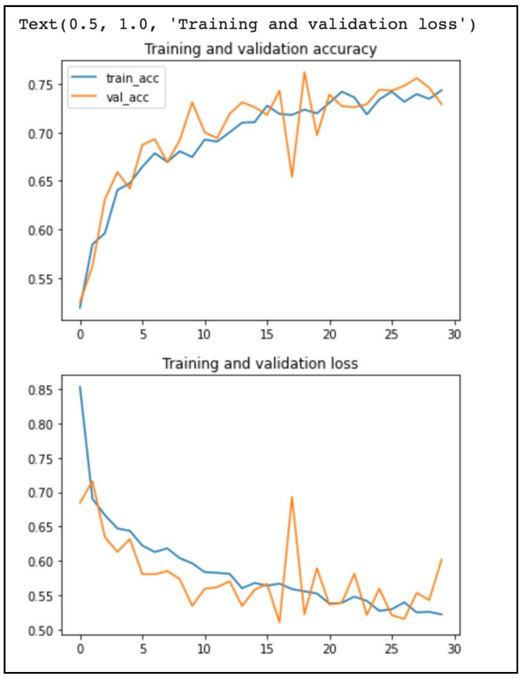
```
x = layers.Dropout(0.9)(x)

x = layers.Dropout(0.1)(x)
```

The dropout technique randomly drops out units of the neural network. Preventing overfitting because units co-adapt during training. It will make some units unreliable, therefore other units won't be able to use them to correct mistakes. The higher this dropout rate is, the more units will be dropped, so when setting the dropout rate to 0.9, we are dropping out 90% of the units. This will result in an under-learning neural net. Hence we can see when running the model that the accuracy rate is lower and the validation accuracy rate shows some noisy movements around the training accuracy which shows an unrepresentative validation dataset .



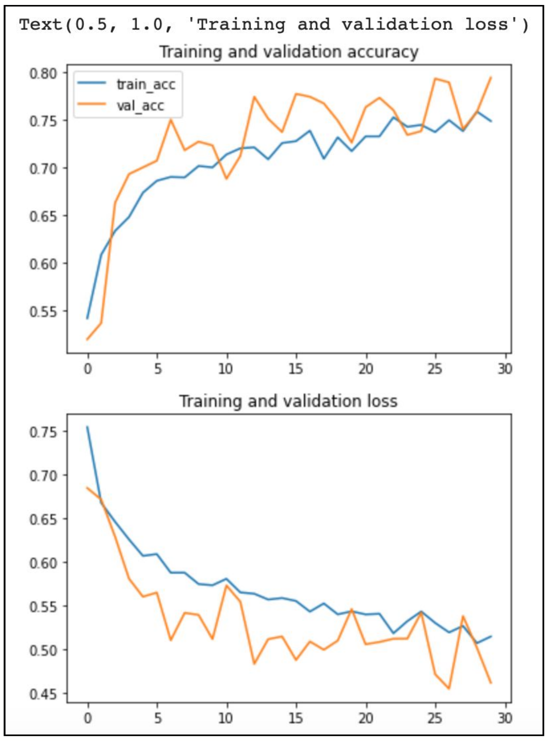
```
Text(0.5, 1.0, 'Training and validation loss')
```

On the other hand when using a dropout rate of 0.1, we are only dropping 10% of the units, this will have a minimal effect on the original model with no dropout. This is shown when running the model with a dropout rate of 0.1. The accuracy is very similar to our first model and we can observe still some overfitting in our model.



```
Text(0.5, 1.0, 'Training and validation loss')
```

This is why it is usually advised to use a dropout rate close to 0.5, in order to balance those two risks. But in our model it seems like a dropout rate of 0.5 remains very noisy in the validation. By lowering the percentage rate of the dropout to 0.2 we come to a better fit, with less noise in the validation.

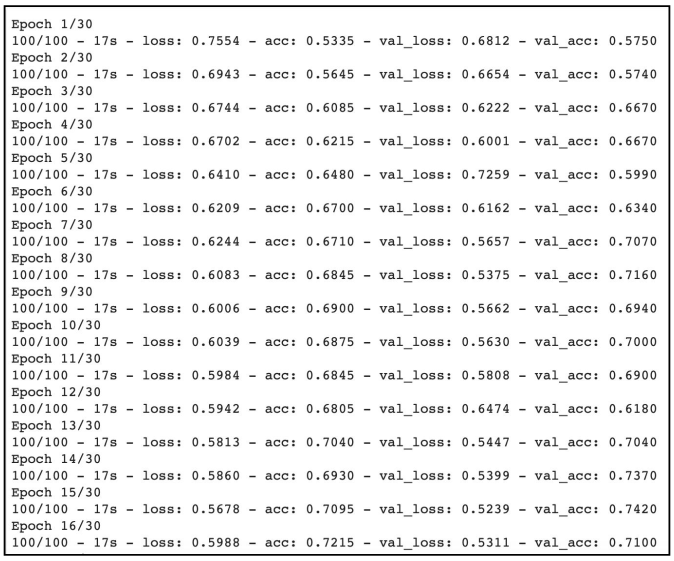


```
Text(0.5, 1.0, 'Training and validation loss')
```

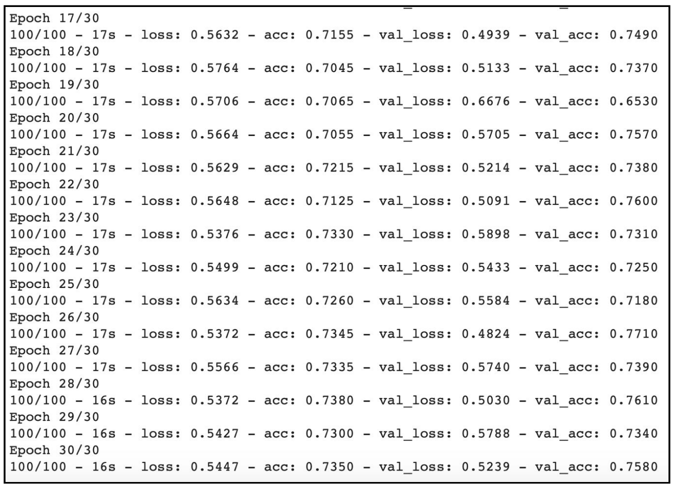
2) Fit the final model following the specifications in the code

```
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)
```

With a dropout of 0.5:

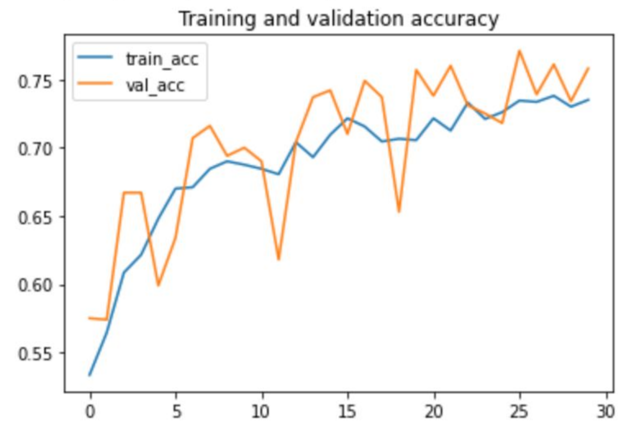


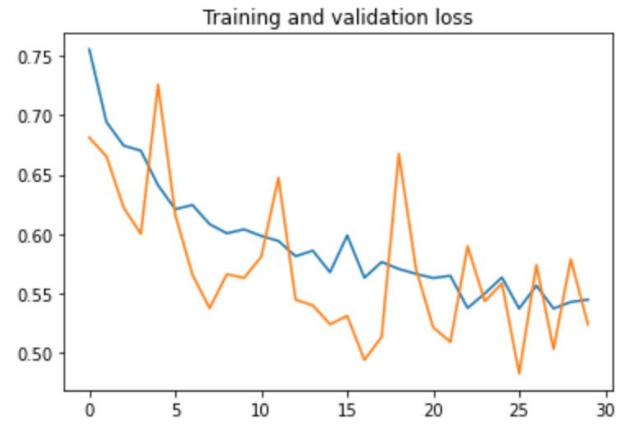
```
Epoch 1/30
100/100 - 17s - loss: 0.7554 - acc: 0.5335 - val_loss: 0.6812 - val_acc: 0.5750
Epoch 2/30
100/100 - 17s - loss: 0.6943 - acc: 0.5645 - val_loss: 0.6654 - val_acc: 0.5740
Epoch 3/30
100/100 - 17s - loss: 0.6744 - acc: 0.6085 - val_loss: 0.6222 - val_acc: 0.6670
Epoch 4/30
100/100 - 17s - loss: 0.6702 - acc: 0.6215 - val_loss: 0.6001 - val_acc: 0.6670
Epoch 5/30
100/100 - 17s - loss: 0.6410 - acc: 0.6480 - val_loss: 0.7259 - val_acc: 0.5990
Epoch 6/30
100/100 - 17s - loss: 0.6209 - acc: 0.6700 - val_loss: 0.6162 - val_acc: 0.6340
Epoch 7/30
100/100 - 17s - loss: 0.6244 - acc: 0.6710 - val_loss: 0.5657 - val_acc: 0.7070
Epoch 8/30
100/100 - 17s - loss: 0.6083 - acc: 0.6845 - val_loss: 0.5375 - val_acc: 0.7160
Epoch 9/30
100/100 - 17s - loss: 0.6006 - acc: 0.6900 - val_loss: 0.5662 - val_acc: 0.6940
Epoch 10/30
100/100 - 17s - loss: 0.6039 - acc: 0.6875 - val_loss: 0.5630 - val_acc: 0.7000
Epoch 11/30
100/100 - 17s - loss: 0.5984 - acc: 0.6845 - val_loss: 0.5808 - val_acc: 0.6900
Epoch 12/30
100/100 - 17s - loss: 0.5942 - acc: 0.6805 - val_loss: 0.6474 - val_acc: 0.6180
Epoch 13/30
100/100 - 17s - loss: 0.5813 - acc: 0.7040 - val_loss: 0.5447 - val_acc: 0.7040
Epoch 14/30
100/100 - 17s - loss: 0.5860 - acc: 0.6930 - val_loss: 0.5399 - val_acc: 0.7370
Epoch 15/30
100/100 - 17s - loss: 0.5678 - acc: 0.7095 - val_loss: 0.5239 - val_acc: 0.7420
Epoch 16/30
100/100 - 17s - loss: 0.5988 - acc: 0.7215 - val_loss: 0.5311 - val_acc: 0.7100
```



```
Epoch 17/30
100/100 - 17s - loss: 0.5632 - acc: 0.7155 - val_loss: 0.4939 - val_acc: 0.7490
Epoch 18/30
100/100 - 17s - loss: 0.5764 - acc: 0.7045 - val_loss: 0.5133 - val_acc: 0.7370
Epoch 19/30
100/100 - 17s - loss: 0.5706 - acc: 0.7065 - val_loss: 0.6676 - val_acc: 0.6530
Epoch 20/30
100/100 - 17s - loss: 0.5664 - acc: 0.7055 - val_loss: 0.5705 - val_acc: 0.7570
Epoch 21/30
100/100 - 17s - loss: 0.5629 - acc: 0.7215 - val_loss: 0.5214 - val_acc: 0.7380
Epoch 22/30
100/100 - 17s - loss: 0.5648 - acc: 0.7125 - val_loss: 0.5091 - val_acc: 0.7600
Epoch 23/30
100/100 - 17s - loss: 0.5376 - acc: 0.7330 - val_loss: 0.5898 - val_acc: 0.7310
Epoch 24/30
100/100 - 17s - loss: 0.5499 - acc: 0.7210 - val_loss: 0.5433 - val_acc: 0.7250
Epoch 25/30
100/100 - 17s - loss: 0.5634 - acc: 0.7260 - val_loss: 0.5584 - val_acc: 0.7180
Epoch 26/30
100/100 - 17s - loss: 0.5372 - acc: 0.7345 - val_loss: 0.4824 - val_acc: 0.7710
Epoch 27/30
100/100 - 17s - loss: 0.5566 - acc: 0.7335 - val_loss: 0.5740 - val_acc: 0.7390
Epoch 28/30
100/100 - 16s - loss: 0.5372 - acc: 0.7380 - val_loss: 0.5030 - val_acc: 0.7610
Epoch 29/30
100/100 - 16s - loss: 0.5427 - acc: 0.7300 - val_loss: 0.5788 - val_acc: 0.7340
Epoch 30/30
100/100 - 16s - loss: 0.5447 - acc: 0.7350 - val_loss: 0.5239 - val_acc: 0.7580
```

```
Text(0.5, 1.0, 'Training and validation loss')
```





3. STAND ON THE SHOULDERS OF GIANTS

1) **Do some research and explain the inception V3 topology. Which database do they use for training it? How was it trained?**

Our findings for Inception V3
The development of inception networks was a great breakthrough to improve accuracy and speed for CNN classifiers. Inception V3 is a convolutional neural net used to classify images trained on more than a million images from the ImageNet dataset. It has 48 layers and can classify images into 1000 object categories. So, Inception V3 has learned a lot of feature representations for diverse images. Moreover, by utilizing the final layer of the model we can utilize the knowledge that the model has learned while training in order to use it on other smaller datasets. This process can improve classification accuracy.

In order to explain the inception V3 topology, we will first describe the Inception V1 and Inception V2 as Inception V3 is an improvement of the 2 previous versions.
We will first describe the problem that had to be solved: As we know, pictures, photos can have different information and size. For example: From a selected sample of cats photos, the relevant information (cats in this scenario) can have a large variation in size. The cat can take almost the full size of the photo, only half or a small part of it.

So, it is complicated to find the right kernel size, as a large kernel would be best when the information represents a large percent of the photo and a small kernel is best when the information represents a small percentage.
To solve this problem, they designed inception networks to create wider networks instead of deeper by having filters of different sizes.

- **Inception V1:**

To do so, they implemented 3 different filter sizes (1x1, 3x3, 5x5) and also used max pooling. As deep neural networks require a lot of computation power, they added a 1x1 convolution before the 3x3, 5x5 and max pooling. Because 1x1 is a lot cheaper, by doing so, it reduces the number of inputs as well.
Next, to reduce dimensions, they added two auxiliary classifiers: they applied softmax to two inception modules and computed an auxiliary loss over these. The total loss function is a weighted sum of the auxiliary loss and the real loss. Weight value used in the paper was 0.3 for each auxiliary loss.

- **Inception V2:**

They realised that reducing the dimensions could result in a loss of information which is known as a "representational bottleneck". So they decided to use factorization methods by factoring 5x5 convolution to two 3x3 convolutions. This would improve computational speed.
Following that, they factorized convolutions of size nxn to 1xn and nx1. So for example, a 3x3 would become 1x3 and 3x1 and the 5x5 would become two 1x3 and 3x1 . This resulted in much cheaper computations.
To reduce representational bottleneck, they expanded filter banks in order to make the network wider instead of deeper which could result in too much dimensionality reduction and thus, loss of information.

- **Inception V3:**

Following the above Inceptions, they realized that the auxiliary classifiers had a significant impact only at the end of the training process.
So by including the above upgrades of Inspection V2, they implemented:

- RMSProp Optimizer, it "is an **optimizer** that utilizes the magnitude of recent gradients to normalize the gradients. We always keep a moving average over the root mean squared (hence Rms) gradients, by which we divide the current gradient." https://climin.readthedocs.io/en/latest/rmsprop.html
- Factorized 7x7 convolutions which is used to factorize the first 7×7 conv layer into three 3×3 convolutions layer.
- BatchNorm in the Auxiliary Classifiers, here it is used as regularizer
- Label Smoothing (A type of regularizing component added to the loss formula that prevents the network from becoming too confident about a class. Prevents overfitting)

For the training, the ImageNet dataset was used, it is composed of 1,331,167 images and 1000 classes which were split into training and evaluation datasets containing 1,281,167 and 50,000 images, respectively.

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?

Refer to notebook: 03_cnn_GroupF.ipynb

It is possible to change the inputs as this is a fully convolutional model and the convolutions disregard the image size. To do so, we have to specify 'include_top' = False. If so, we can change the 'input_shape' to the new shape (150, 150, 3), remembering that width and height can not be under 75. By specifying 'include_top' = False, we allow the model to leave off the fully connected layer heads, we are still loading the ImageNet weights but we are not loading the fully-connected heads.

3) **Use Inception V3 to outperform the results we obtained in the TransferLearning_flowers database with our first neural net. This is, run Inception V3 with this database.**

Refer to notebook: TransferLearning_flowers_inceptionV3_GroupF.ipynb

4. YOLO_V3 [Advanced & Optional] [3 extra points]

1) Describe YOLO_v3 neural net.

So far we have been working on classification.
We now want to work on detection. For this we use YOLO that also uses convolutional neural networks and it is one of the fastest object detection algorithms although not the most accurate (still very good when we need real-time detection without loss of too much accuracy).

First let's understand what YOLO is (You Only Look Once):
It is called YOLO because the algorithm actually looks at the image just once. It works by detecting the location of objects and predicting their class. It is able to detect multiple objects within an image all at once.
Yolo divides the image into a grid and predicts bounding boxes (each box being a potential object). It gives a confidence score to each box and a class prediction, these numbers are combined to give a final score that tells the probability that the bounding box contains a specific object.
If we divide the image into a 13 by 13 we will have 169 cells and if each cell predicts 3 bounding boxes we end up with 507 bounding boxes in total. But most of these boxes are going to have very low confidence scores. So we only keep the boxes that have a score higher than a certain threshold (which we can set). So in the end only the important boxes will remain.
Then we feed our bounding boxes to our convolutional neural network. Using traditional layers like convolutional layer, MaxPooling layer.
A Bounding box is described by 25 data elements :

- X, y , width, height for the bounding box
- Confidence score
- Probability distribution over the class

So an input image goes through the convolutional neural network in a single pass, comes out the (number of channel describing the bounding box)*(nb of cells we decided for the image) tensors describing the bounding boxes, compute the final score for the bounding boxes and decide which one to keep depending on the threshold.

YOLO v2 was lacking some complement like : no residual blocks, no skip connections and no upsampling. But YOLO v3 incorporates all of these. It uses another version of Darknet which originally has 53 layers and 53 more layers are added, giving us a 106 layer fully convolutional underlying architecture for YOLO v3. (Which makes v3 a little slower)
YOLO v3 predicts 3 bounding boxes for every cell. The detection is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network.

2) Put to work YOLO_v3 in the database you want. Note you need training images correctly labelled with the info of the location and objects in the images. In the links below you have some ideas and two different databases, for recognizing drones and weapons respectively.

Refer to notebook: Yolov3_GroupF.ipynb

Unfortunately, we had to stop our model from training due to the issue of running time being longer than we expected. However, we managed to apply YOLOv3 for our purpose, using the handgun dataset.