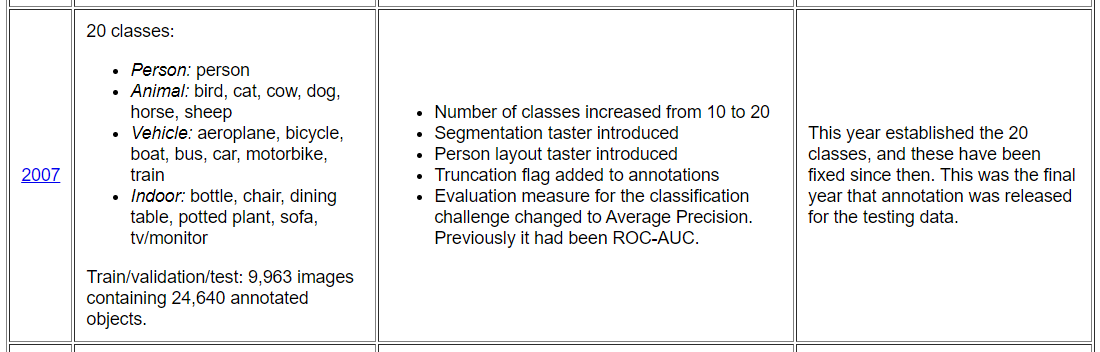
**2.10 SSD - Step 10 (training)**

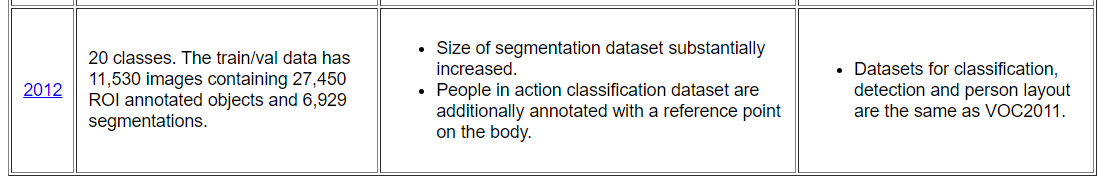
In the previous steps we're actually doing the "**Testing**" with a pre-trained SSD-NN. Now it's time to understand how we "**Train**" the SSD-NN with different data.

* We're not going to implement the whole code that train the SSD-NN because
* It's a *huge code*
* And we're going to need *specific requirements* to be able to run that training.
* Remember we'll need some *pretty advanced system*.
* So this last step is just an *explanation* about how we could *train* the *SSD*.
* We'll see how to execute the file.
* Also we'll see the data-set on which the SSD was trained.
* **Pascal VOC project:** Following webpage contains the dataset on which the SSD-NN was trained.

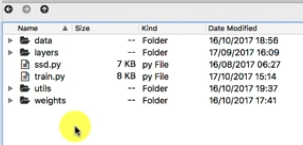
<http://host.robots.ox.ac.uk/pascal/VOC/>

* This dataset is not Image-Net but the Pascal Visual Object Classes dataset.
* Remember in the code we saw **VOC** classes to have the mapping between the classes and integers.
* **VOC** stands for **visual object classes** and this is already a huge dataset not as huge as Image-Net but a huge one that you will see contains lots and lots of images.
* The Pascal VOC project provides standardized image data sets for object recognition.
* It also provides a common set of tools for accessing the data set and annotations.
* So it's not only a data set it's also an advanced structure of data set that helps a lot for the training.
* Look at the challenges from 2005 to 2012 and you can see there's several data-sets
* Our SSD was trained on two data sets: the VOC 2007 and the VOC 2012.
* We can also have the details on the datasets: The table below gives a brief summary of the VOC 2007 and the VOC 2012





* We can see how the data sets are structured and made and what they contain exactly. That’s why our pre-trained SSD-NN can detect between 30 to 40 object.
* Folder structure: We have to make following folder structure that contains all the tools and data sets to train the SSD.
* The training is made with PyTorch.



* **cuda system:** You need to have CUDA on your machine.
* **CUDA** is like an accelerator that is connected to **NVIDIA graphics cards** and allows to speed up the training computations. It's do the parallel computation better.
* Therefore the training implementations that were made are only compatible with CUDA.
* The training implementation **without CUDA** would take several months or even years.
* Since the training of the SSD is made with thousands and thousands of images.
* You cannot train the **SSD** on your **CPU**, you have to train it with a **GPU** and the best you can find is **CUDA**.
* Train file is on the following link:
* <https://github.com/amdegroot/ssd.pytorch/blob/master/train.py>
* And for more details go to Max de Groot: <https://github.com/amdegroot/ssd.pytorch>
* Data-Set: Download the datasets VOC 2007 and VOC 2012. These dataset will contain lots of images in jpeg format e.g. chair, people, cars, trains etc.
* We'll use the data-set and "**utils**". **utils** contains the augmentation.
* Augmentations are some ways to increase the amount of images so that we can have even more material to train our SSD.
* There is also **weights** folder, with some **initialized weights**.
* These are not some weights of pre-trained model these are just some initialized weights.
* Weights are need to be initialized in some way that it should be compatible with the future updates of the weights during the training.
* train.py: perser's are determining the way of training. We can change the learning rate, the decay rate, momentum, gamma etc. These are the hyper-parameters that can be tuned if we want.
* We can also work with **IMAGE-NET** dataset.
* **xavier** & **weight\_init** will *initialize* the weights in *proper ways*.
* **train()** function will contain the *training mechanism*.

**All code at once**

* SSD – object detection:

# *--------------    Object Detection    --------------*

# *Importing the libraries*

**import** torch

**from** torch.autograd **import** Variable

**import** cv2

**from** data **import** BaseTransform, VOC\_CLASSES **as** labelmap

**from** ssd **import** build\_ssd

**import** imageio

# *Defining a function that will do the detections*

    # *We define a detect function that will take as inputs,*

        # *a frame,*

        # *a ssd neural network, and*

        # *a transformation to be applied on the images,*

    # *The function will return the frame with the detector rectangle.*

**def** **detect**(frame, net, transform):

    height, width = frame.shape[:2]     # *We get the height and the width of the frame.*

    frame\_t = **transform**(frame)[0]       # *We apply the transformation to our frame.*

    x = **torch.from\_numpy**(frame\_t).**permute**(2, 0, 1)  # *We convert the frame into a torch tensor.*

    x = **Variable**(**x.unsqueeze**(0))    # *We add a fake dimension corresponding to the batch.*

    y = **net**(x)      # *We feed the ssd-NN with the image and we get the output y.*

    detections = y.data     # *We create the detections tensor contained in the output y.*

    # *We create a tensor object of dimensions [width, height, width, height].*

    scale = **torch.Tensor**([width, height, width, height])

    # *detections = [batch, number of classes, number of occurence, (score, x0, Y0, x1, y1)]*

**for** i **in** **range**(**detections.size**(1)): # *For every class:*

        j = 0       # *We initialize the loop variable j that will correspond to the occurrences of the class.*

        # *all the 'occurrences' j of the class i that have a matching score larger than 0.6.*

**while** detections[0, i, j, 0] **>=** 0.6:

            # *coordinates of upper left and the lower right  corner of the detector rectangle.*

            pt = (detections[0, i, j, 1:] \* scale).**numpy**()

            # *We draw a rectangle around the detected object.*

**cv2.rectangle**(frame, (**int**(pt[0]), **int**(pt[1])), (**int**(pt[2]), **int**(pt[3])), (255, 0, 0), 2)

            # *We put the label of the class right above the rectangle.*

**cv2.putText**(frame, labelmap[i - 1], (**int**(pt[0]), **int**(pt[1])), cv2.FONT\_HERSHEY\_SIMPLEX, 2, (255, 255, 255), 2, cv2.LINE\_AA)

            j += 1 # *We increment j to get to the next occurrence.*

**return** frame # *We return the original frame with the detector rectangle and the label around the detected object.*

# *Creating the SSD neural network*

net = **build\_ssd**('test')     # *We create an object that is our neural network ssd.*

# *We get the weights of the neural network from another one that is pretrained (ssd300\_mAP\_77.43\_v2.pth).*

**net.load\_state\_dict**(**torch.load**('ssd300\_mAP\_77.43\_v2.pth', map\_location = **lambda** storage, loc: storage))

# *Creating the transformation*

    # *We create an object of the BaseTransform class, a class that will do the required transformations*

        # *so that the image can be the input of the neural network.*

transform = **BaseTransform**(net.size, (104/256.0, 117/256.0, 123/256.0))

# *Doing some OBJECT DETECTION on a video*

reader = **imageio.get\_reader**('funny\_dog.mp4')    # *We open the video.*

fps = **reader.get\_meta\_data**()['fps']     # *We get the 'fps' frequence (frames per second).*

writer = **imageio.get\_writer**('output.mp4', fps = fps)    # *We create an output video with this 'same fps' frequence.*

**for** i, frame **in** **enumerate**(reader):      # *We 'ITERATE' on the frames of the output video:*

    # *We call our detect() function (defined above) to detect the object on the frame.*

    frame = **detect**(frame, **net.eval**(), transform)

**writer.append\_data**(frame)   # *We add the next frame in the output video.*

**print**(i)    # *We print the number of the processed frame.*

**writer.close**()  # *We close the process that handles the creation of the output video.*

'''

# ::::::::::::::::    no comment    ::::::::::::::::::

# Object Detection

# Importing the libraries

import torch

from torch.autograd import Variable

import cv2

from data import BaseTransform, VOC\_CLASSES as labelmap

from ssd import build\_ssd

import imageio

# Defining a function that will do the detections

def detect(frame, net, transform):

    height, width = frame.shape[:2]

    frame\_t = transform(frame)[0]

    x = torch.from\_numpy(frame\_t).permute(2, 0, 1)

    x = Variable(x.unsqueeze(0))

    y = net(x)

    detections = y.data

    scale = torch.Tensor([width, height, width, height])

    # detections = [batch, number of classes, number of occurence, (score, x0, Y0, x1, y1)]

    for i in range(detections.size(1)):

        j = 0

        while detections[0, i, j, 0] >= 0.6:

            pt = (detections[0, i, j, 1:] \* scale).numpy()

            cv2.rectangle(frame, (int(pt[0]), int(pt[1])), (int(pt[2]), int(pt[3])), (255, 0, 0), 2)

            cv2.putText(frame, labelmap[i - 1], (int(pt[0]), int(pt[1])), cv2.FONT\_HERSHEY\_SIMPLEX, 2, (255, 255, 255), 2, cv2.LINE\_AA)

            j += 1

    return frame

# Creating the SSD neural network

net = build\_ssd('test')

net.load\_state\_dict(torch.load('ssd300\_mAP\_77.43\_v2.pth', map\_location = lambda storage, loc: storage))

# Creating the transformation

transform = BaseTransform(net.size, (104/256.0, 117/256.0, 123/256.0))

# Doing some Object Detection on a video

reader = imageio.get\_reader('funny\_dog.mp4')

fps = reader.get\_meta\_data()['fps']

writer = imageio.get\_writer('output.mp4', fps = fps)

for i, frame in enumerate(reader):

    frame = detect(frame, net.eval(), transform)

    writer.append\_data(frame)

    print(i)

writer.close()

'''

* SSD-NN structure (ssd.py):

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**from** torch.autograd **import** Variable

**from** layers **import** \*

**from** data **import** v2

**import** os

**class** SSD(nn.Module):

    """Single Shot Multibox Architecture

    The network is composed of a base VGG network followed by the

    added multibox conv layers.  Each multibox layer branches into

        1) conv2d for class conf scores

        2) conv2d for localization predictions

        3) associated priorbox layer to produce default bounding

           boxes specific to the layer's feature map size.

    See: https://arxiv.org/pdf/1512.02325.pdf for more details.

    Args:

        phase: (string) Can be "test" or "train"

        base: VGG16 layers for input, size of either 300 or 500

        extras: extra layers that feed to multibox loc and conf layers

        head: "multibox head" consists of loc and conf conv layers

    """

**def** **\_\_init\_\_**(self, phase, base, extras, head, num\_classes):

**super**(SSD, self).**\_\_init\_\_**()

        self.phase = phase

        self.num\_classes = num\_classes

        # *TODO: implement \_\_call\_\_ in PriorBox*

        self.priorbox = **PriorBox**(v2)

        self.priors = **Variable**(self**.priorbox.forward**(), volatile=**True**)

        self.size = 300

        # *SSD network*

        self.vgg = **nn.ModuleList**(base)

        # *Layer learns to scale the l2 normalized features from conv4\_3*

        self.L2Norm = **L2Norm**(512, 20)

        self.extras = **nn.ModuleList**(extras)

        self.loc = **nn.ModuleList**(head[0])

        self.conf = **nn.ModuleList**(head[1])

**if** phase **==** 'test':

            self.softmax = **nn.Softmax**()

            self.detect = **Detect**(num\_classes, 0, 200, 0.01, 0.45)

**def** **forward**(self, x):

        """Applies network layers and ops on input image(s) x.

        Args:

            x: input image or batch of images. Shape: [batch,3\*batch,300,300].

        Return:

            Depending on phase:

            test:

                Variable(tensor) of output class label predictions,

                confidence score, and corresponding location predictions for

                each object detected. Shape: [batch,topk,7]

            train:

                list of concat outputs from:

                    1: confidence layers, Shape: [batch\*num\_priors,num\_classes]

                    2: localization layers, Shape: [batch,num\_priors\*4]

                    3: priorbox layers, Shape: [2,num\_priors\*4]

        """

        sources = **list**()

        loc = **list**()

        conf = **list**()

        # *apply vgg up to conv4\_3 relu*

**for** k **in** **range**(23):

            x = self.vgg[k](x)

        s = self**.L2Norm**(x)

**sources.append**(s)

        # *apply vgg up to fc7*

**for** k **in** **range**(23, **len**(self.vgg)):

            x = self.vgg[k](x)

**sources.append**(x)

        # *apply extra layers and cache source layer outputs*

**for** k, v **in** **enumerate**(self.extras):

            x = **F.relu**(**v**(x), inplace=**True**)

**if** k % 2 **==** 1:

**sources.append**(x)

        # *apply multibox head to source layers*

**for** (x, l, c) **in** **zip**(sources, self.loc, self.conf):

**loc.append**(**l**(x).**permute**(0, 2, 3, 1).**contiguous**())

**conf.append**(**c**(x).**permute**(0, 2, 3, 1).**contiguous**())

        loc = **torch.cat**([**o.view**(**o.size**(0), -1) **for** o **in** loc], 1)

        conf = **torch.cat**([**o.view**(**o.size**(0), -1) **for** o **in** conf], 1)

**if** self.phase **==** "test":

            output = self**.detect**(

**loc.view**(**loc.size**(0), -1, 4),                   # *loc preds*

                self**.softmax**(**conf.view**(-1, self.num\_classes)),  # *conf preds*

                self**.priors.type**(**type**(x.data))                  # *default boxes*

            )

**else**:

            output = (

**loc.view**(**loc.size**(0), -1, 4),

**conf.view**(**conf.size**(0), -1, self.num\_classes),

                self.priors

            )

**return** output

**def** **load\_weights**(self, base\_file):

        other, ext = **os.path.splitext**(base\_file)

**if** ext **==** '.pkl' **or** '.pth':

**print**('Loading weights into state dict...')

            self**.load\_state\_dict**(**torch.load**(base\_file, map\_location=**lambda** storage, loc: storage))

**print**('Finished!')

**else**:

**print**('Sorry only .pth and .pkl files supported.')

# *This function is derived from torchvision VGG make\_layers()*

# *https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py*

**def** **vgg**(cfg, i, batch\_norm=**False**):

    layers = []

    in\_channels = i

**for** v **in** cfg:

**if** v **==** 'M':

            layers += [**nn.MaxPool2d**(kernel\_size=2, stride=2)]

**elif** v **==** 'C':

            layers += [**nn.MaxPool2d**(kernel\_size=2, stride=2, ceil\_mode=**True**)]

**else**:

            conv2d = **nn.Conv2d**(in\_channels, v, kernel\_size=3, padding=1)

**if** batch\_norm:

                layers += [conv2d, **nn.BatchNorm2d**(v), **nn.ReLU**(inplace=**True**)]

**else**:

                layers += [conv2d, **nn.ReLU**(inplace=**True**)]

            in\_channels = v

    pool5 = **nn.MaxPool2d**(kernel\_size=3, stride=1, padding=1)

    conv6 = **nn.Conv2d**(512, 1024, kernel\_size=3, padding=6, dilation=6)

    conv7 = **nn.Conv2d**(1024, 1024, kernel\_size=1)

    layers += [pool5, conv6,

**nn.ReLU**(inplace=**True**), conv7, **nn.ReLU**(inplace=**True**)]

**return** layers

**def** **add\_extras**(cfg, i, batch\_norm=**False**):

    # *Extra layers added to VGG for feature scaling*

    layers = []

    in\_channels = i

    flag = **False**

**for** k, v **in** **enumerate**(cfg):

**if** in\_channels **!=** 'S':

**if** v **==** 'S':

                layers += [**nn.Conv2d**(in\_channels, cfg[k + 1],

                           kernel\_size=(1, 3)[flag], stride=2, padding=1)]

**else**:

                layers += [**nn.Conv2d**(in\_channels, v, kernel\_size=(1, 3)[flag])]

            flag = **not** flag

        in\_channels = v

**return** layers

**def** **multibox**(vgg, extra\_layers, cfg, num\_classes):

    loc\_layers = []

    conf\_layers = []

    vgg\_source = [24, -2]

**for** k, v **in** **enumerate**(vgg\_source):

        loc\_layers += [**nn.Conv2d**(vgg[v].out\_channels,

                                 cfg[k] \* 4, kernel\_size=3, padding=1)]

        conf\_layers += [**nn.Conv2d**(vgg[v].out\_channels,

                        cfg[k] \* num\_classes, kernel\_size=3, padding=1)]

**for** k, v **in** **enumerate**(extra\_layers[1::2], 2):

        loc\_layers += [**nn.Conv2d**(v.out\_channels, cfg[k]

                                 \* 4, kernel\_size=3, padding=1)]

        conf\_layers += [**nn.Conv2d**(v.out\_channels, cfg[k]

                                  \* num\_classes, kernel\_size=3, padding=1)]

**return** vgg, extra\_layers, (loc\_layers, conf\_layers)

base = {

    '300': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'C', 512, 512, 512, 'M',

            512, 512, 512],

    '512': [],

}

extras = {

    '300': [256, 'S', 512, 128, 'S', 256, 128, 256, 128, 256],

    '512': [],

}

mbox = {

    '300': [4, 6, 6, 6, 4, 4],  # *number of boxes per feature map location*

    '512': [],

}

**def** **build\_ssd**(phase, size=300, num\_classes=21):

**if** phase **!=** "test" **and** phase **!=** "train":

**print**("Error: Phase not recognized")

**return**

**if** size **!=** 300:

**print**("Error: Sorry only SSD300 is supported currently!")

**return**

**return** **SSD**(phase, \***multibox**(**vgg**(base[**str**(size)], 3),

**add\_extras**(extras[**str**(size)], 1024),

                                mbox[**str**(size)], num\_classes), num\_classes)