Chapter – 2 (SSD implementation: part 1)

**Computer Vision**

**SSD: Object Detection**

Intro

Installation

Implementation

**2.1 SSD - Step 1 (File Structure and import Libraries)**

SSD is much more powerful than OpenCv because it is based on Deep Learning and Neural Networks. That means the computer will have a brain.

* Problem Description: We have a video of a very cute dog bouncing on the field. And our challenge will be to implement some program that will detect the dog in the video.
* In this section we're going to learn how to do some **object detection** on a video **directly**.
* Big thank to **"Max de Groot"** (<https://github.com/amdegroot> ) the creator of the **PyTorch** implementation of Single Shot Multi-Box Detector that we're going to use in this chapter.
* We've tried the **Fast R-CNN**, the **YOLO-openCV** and the **SSD**.
* We obtain the best result with the SSD.
* We're going to use **Max de Groot**'s pre-trained model which was trained to detect between ***30 and 40 objects*** including cars, dogs, horses, ships, boats, planes and more.
* We're going to learn how to use it and how to detect any object on any video.
* File Structure:
* Connect to the virtual platform & launch spider:

>>> conda env list

>>> conda activate py354

>>> spyder

1. **data:** is just a folder that contains the classes based transform that will do the *required transformations* so that the input images will be ***compatible*** with the ***neural network***.
2. **funny\_dog.mp4:** Is the video, in which we try to detect the dog.
3. **layers:** is another folder that contained some other tools for the detection and the Multi-Box part of the SSD.
4. **object\_detection\_commented.py** and **object\_detection\_nocomment.py** are the codes. We'll write these code in this chapter.

* We recommend to look at the *commented version* before start coding, so that you can expect what you need to understand.

1. **ssd.py:** contains the **architecture of the SSD model**. It's mostly about the architecture: with all the boxes, how they're defined.

* We won't implement this from scratch, because the most important for you to understand is object detection implementation.

1. **ssd300\_mAP\_77.43\_v2.pth:** Is the file we will be loading to get the pre-trained SSD model.

* This is the file that contains the weight of the SSD neural network that was already pre-trained.
* We will be loading this file with the ***torch library*** using ***load()***.
* This **torch.load()** function will open a tensor that will contain the weight of this already pre-trained Neural Network.
* Then through a mapping with a dictionary, we will transfer these weights to the model we implement.
* About the video: The video last two seconds. The model we will implement, will not only detect the dog but also humans present in the video.
* The dog is actually pretty small in the video. When we tried to detect that with OpenCV we had extremely bad results (some rectangles everywhere).
* But SSD will do a better job, even if there is not a perfect contrast between the dog and environment.
* For example: *a white environment* with a *black dog*. The dog can be confused with something else.

**2.2 SSD - Step 2 (Importing Libraries)**

We have to detect a funny dog on a two seconds video and we will do it through a **Computer Vision** based on **Deep Learning Neural** **Networks**.

* Libraries: The libraries that we're going to use.

1. ***torch:*** The **torch** library that contains **PyTorch** which is used to build NN and do some computer vision.

* Because **PyTorch** contains the ***dynamic graphs***, it helps us to compute the ***gradients*** of composition functions in ***backward*** ***propagation***.
* To update the ***weight*** through ***stochastic gradient descent*** (SGD), we have to compute the gradient of some composition functions because we have several layers in deep neural network.

* Why dynamic graphs?

For example, say *one layer* has some functions of the *previous layer* which has some functions of the *previous-previous layer* so that generates some composition functions.

We have to compute the *gradient* of these *composition functions* to *update the weights* according to how much they are *responsible* for the error between the target and the predictions.

The dynamic graphs is a highly advanced graph structure that allows to have some very fast and efficient computation of the gradients.

So that's why **torch** is our first choice.

# *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

1. ***torch.autograd*** is the module responsible for Gradient Descent. We are importing the ***Variable*** class which will be used to convert the tensors into some torch variables that will contain both the tensor and a gradient.

* And then the torch variables containing the tensor and gradient will be one element of the graph.

1. We import ***cv2*** but we're not going to implement ***OpenCV-Haar-Cascade***. We're using this library to draw some rectangles around the ***dog***.
2. From "**data**" folder we import **BaseTransform** and **VOC\_CLASSES** **as** **labelmap**.

* **data** is just a folder that contains the **BaseTransform** and **VOC\_CLASSES**.
* **BaseTransform** is a class that will do the required transformations so that the input images will be compatible with the neural network.
* **VOC\_CLASSES** is just a dictionary that will do the encoding of the classes. e.g. it encodes planes, dogs into integers.

It's just a very simple dictionary doing the mapping between the ***text fields*** of the classes and some ***integers***.

1. ***ssd*** is the library of the ***Single-Shot-Multi-Box-Detection*** model. ***build\_ssd*** is the constructor of the SSD Neural Network.

* ***ssd.py*** contains the architecture of this ***Single-Shot-Multi-Box-Detection*** model.

1. **imageio** is just the library that we'll use to process the images of the video and applying the ***detect()*** function that will implement on the images.

* At first we wanted to import **pil** which is another library but with **imageio** we have to code two or three lines to process the images of the video.

Next we're going to define the **detect()** function that will do the detections.

**2.3 SSD - Step 3 (define detect() function)**

Now we are going to define the ***detect()*** function. In previous chapter we used OpenCV, but this time we're going to use SSD-model which is based on deep-learning.

* Like previous chapter we are going to do a "frame by frame detection", i.e. the **detect()** function will work on single images (not the video).
* And then using ***imageio*** we will

1. Extract all the frames of the video,
2. Apply the **detect()** function on the frames
3. And then reassemble the whole thing to make the video with the *rectangles* detecting the dog and the humans.

* arguments: This **detect()** function is going to take three arguments.

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

1. **frame:** It is the image on which the **detect()** function is going to be applied to detect the object. Note that, in the OpenCV, along with "frame" we used "gray", the black-White version of the image.

* In SSD, we only need to put the original image, the original frame in color.

1. **net:** It is the SSD neural network.
2. **transform:** There is going to be some transformations applied to the image. To make sure that the images are compatible with the neural network,

* Because the images will be the input of the neural network and therefore they have to have a certain format.
* **What exactly is this function going to return:** It'll return some Labeled rectangles.
* It will simply return this same **frame** but with the rectangle detecting the objects the dog and the humans.
* Also there will be the label on the rectangle, e.g. we'll see some rectangles with the label "Human" and some with label "Dog".
* Height & Width of the image:

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

* ***shape*** is the attribute of ***frame***. *shape* attribute returns a vector of three elements.

1. The first one is the *height* of the *frame*.
2. The second one is the *width* of the *frame* and
3. Third one, i.e. element of index 2 is the *number of channels*.

* For a black and white image you will have one channel and
* If you have a color image you will have three channels **red**, **blue** and **green**.

But we just want the height and width. So we take a range from **0** to **2** with **2 excluded**.

**2.4 SSD - Step 4 (Transform "frame" -> "** **torch-variable ")**

* Transformations: Now we have to do several transformations to go from the original image (frame) to a ***torch variable*** that will be accepted into SSD-Neural-Network.

1. The first one is to apply the transform to make sure that the image has the right format i.e. the right dimensions and color values.
2. In second transformation, we need to convert this transformed-frame NumPy-array to a torch-tensor.
3. We'll do a third transformation that will add a fake dimension to the torch-tensor and that fake-dimension will correspond to the batch.
4. The fourth and final transformation is to convert torch-tensor into a torch-variable. This torch-variable will be the input of our SSD-neural-network.

* Remember this **Variable** class from **torch.autograd** converts a torch-tensor into a ***torch-variable*** that contains both the ***tensor*** and a ***gradient***.
* This torch-variable will then be an element of the DYNAMIC-GRAPH, which will allow us later to do some very fast and efficient computation of the Gradients during Backward Propagation.
* Let's do these four transformations:

1. The first one is to apply the transform transformation so that our input **frame** has a right dominations and the right colors.

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

We used index **0**, since we only use the first element.

1. Note that, **frame\_t** is a NumPy array. We need to convert it to a torch-tensor.

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

* Here, **torch.from\_numpy(frame\_t)** doing the transform to the torch-tensor.
* **permute(2, 0, 1)** is to re-arrange the color channel sequence Red, Blue, Green (of index 0, 1, 2) to Green, Red, Blue (index 2, 0, 1).
* Because, the SSD neural network was trained under the convention (green, red, blue).
* So we just need to rearrange the order, it's not the part of the transformation.
* Last two transformations: We're going to make the last two transformations in the same one line of code.

1. Add a fake dimension (corresponding to the batch) to our tensor. Because the *NN cannot accept single inputs* like a single input vector or a *single input image*, it only accepts them in to some *batches*.

* We have to create a structure with the first-dimension corresponds to the batch and the other dimension corresponding to the input.
* We will always add a fake-dimension with the **unsqueeze()** function just before feeding the input to the neural network.

**x.unsqueeze**(0)

* This **unsqueeze()** function takes one argument which is the *dimension of the batch* and the *batch* should always be the *first dimension* i.e. ***0 index***.

1. We'll do the last transformation in that same line. After this transformation we'll be ready to feed the NN with the input image.

* We convert this "batch of torch-tensor of input" into a torch-viable. This *torch-variable* contains both a *tensor* and a *gradient*.
* This torch-viable will become an element of the dynamic graph which will compute very efficiently the gradients of any composition functions during backward propagation.

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

* We use Variable class to create torch-viable.
* We'll overwrite the previous x.
* Next we will feed this torch-viable to the already pre-trained Neural Network SSD. Which will detect between 30 to 40 objects including planes, horses, dogs, cars, boats a lot more.
* We're not going to do the whole training.

**2.5 SSD - Step 5 (the output tensor)**

We've done our four transformations and the input is ready to be feed into the neural network. In this step we'll feed it into the SSD-NN.

* Since we already have our pre-trained model SSD, we will load the weight at the end.
* Right now "**net**" parameter in **detect(frame, net, transform)** is just a variable. We'll just use the variable for now, but at the end of this implementation we will load the weights to get our pre-trained model.
* To feed **x** (*torch-variable* containing a *tensor* and a *gradient*) to the NN we use the "**net**" parameter, to get the output (**y**).

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

* **y** *doesn't contain* is the result of the ***detection*** directly (whether we have a dog or a human in the input frame).
* To get the *specific information* that we're interested in, we need to take the "**data**" *attribute* from **y**.
* We'll create a new tensor that takes the detections (the values we're interested in) contained in the output **y**. We call it "**detection**":

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

Now we get the values of the output

* **scale:** The position of the detected objects inside the image has to be normalized between **0** and **1**. To do this normalization we'll need this ***scale*** tensor.
* So we need to create a new tensor object called "**scale**" which will have the dimensions: (width, height, width, height)

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

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

The **first** two **(with, height)** correspond to the *scale of values* of the **upper left corner** of the *detector-rectangle*.

The **second** two **(with, height)** correspond to the *scale of values* of the **lower right corner** of this *same detector-rectangle*.

**All code we've done**

# *--------------    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])

color code RGB\_(229, 255, 299)

**2.6 SSD - Step 6 (the for loop)**

Now we have the output of the NN-SSD which is **y = net(x)**.

    y = **net**(x)

    detections = y.data

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

* From y we've extracted **detections** which is the tensor part of the torch variable. (The torch variable composed of two elements a torch tensor and a gradient.)
* And by taking **y.data** we get the first element of this torch variable a *torch tensor*.
* Components of the extracted tensors: Let's see what this ***detections*** tensor contains exactly.
* This ***detections*** tensor right now contains four elements.

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

1. **batch:** It is the fake dimension of the batch which we created using **unsqueeze()**. Note that we've used **unsqueeze()** for the inputs **x**, and that’s why we also need it for the output **y**.

* We'll have batch of outputs associated to the same batch of inputs.

1. **number of classes:** It is the number of objects that was detected in the input image.

* e.g. one class for the dog, another class for the plane, other class will be the car.
* So each class corresponds to each object that can be detected.

1. **number of occurrence:** It is the number of occurrence of a single class.

* For example, let's say a class "class2" represents the dog, then there can be multiple dogs present in a single frame. The number of occurence represents the number of dogs here.
* We would have two numbers of occurrence **0** and occurrence **1**, if we had two dogs in the video. **0** for the **first dog** on the video and **1** for the **second dog**.
* In our case, we'll detect a single dog, so we will only have one number of occurence.

1. **fourth element:** The fourth element will be a touple of five elements: **(score, x0, y0, x1, y1)**

* For each *occurrence* of each class in the batch we will get a ***score*** for this *occurrence*.
* And the coordinates of the ***upper left corner*** of the rectangle detecting the occurance and the ***lower*** ***right corner***.
* **scores:** ***scores*** are going from low to high. It is like a **threshold**. For each occurrence of each class, we'll get a score.

We'll have a score for each occurrence of each class such that: if the score is lower than **0.6** then the occurrence of the class won't be found in the image if it is higher than **0.6** then it will consider it to be found and the coordinates of the corners will be returned.

* For-loop: We have to *iterate through all the classes* and then through all the occurrence of each classes.
* We're going to look for a certain number of occurrence for each class.
* We're going to get the scores for each of these occurrences and
* We'll make an IF-condition to say that if **scores > 0.6** we keep the occurrences otherwise we reject the occurrence.

    # *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       # *initialize the loop variable j that correspond to 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)

* **detections.size(1)** is exactly the *number of classes*.
* **j** represents the *occurrence*
* So **i** is the class and **j** will be the *occurrence* of the class.
* while loop: we're going to put the score-condition into this loop.
* since we only want to keep the *occurrence* for which the score is higher than 0.6.
* Well we simply need to take the **detections**
* of the *occurrence* **j** of the class **i**.
* 0 represents the batch
* 4th element **0** gets the score, which is the *first element (index of the score)* of the touple.
* Here detections[0, i, j, 0] represents the score, (it’s the first element of the tuple (score, x0, y0, x1, y1)), if this is ***larger*** than ***0.6*** than what are we going keep the ***occurrence***

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

**0, 1, 2, 3, 4**

Scroe = detections[batch, number of classes, number of occurrence, 0]

X0 = detections[batch, number of classes, number of occurrence, 1]

Y0 = detections[batch, number of classes, number of occurrence, 2]

X1 = detections[batch, number of classes, number of occurrence, 3]

Y1 = detections[batch, number of classes, number of occurrence, 4]

* Grab the occurrence: We're going to keep the *occurrence* by keeping the points i.e. the coordinates of the upper left corner and the lower right corner of the detecting rectangle.

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

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

* detections[0, i, j, 1:] gets the coordinates **x0, y0, x1, y1** the last four elements of the tuple **(score, x0, y0, x1, y1)**. Notice, we applied the **range** "1:".
* We multiply it with ***scale*** tensor to do this **normalization** of the **coordinates** between **0** and **1**.
* Finally we convert the tensor to a numpy-array, because we are going to use OpenCV to draw the *detector-rectangle*. OpenCV works with NumPy array and not with torch tensors.
* We're going to use the **rectangle()** function from **OpenCV** as we did in the previous face-detection app.
* Draw the rectangle: Now we draw the rectangle using the **rectangle()** function from **OpenCV** :

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

* Here **frame** is the image, *upper-left* corner is **(int(pt[0]), int(pt[1]))** and *lower-right* corner is **(int(pt[2]), int(pt[3]))** and **(255, 0, 0), 2** are the color & thickness of the detector-rectangle.
* Notice, for safety, we converted the coordinates into integers.
* Print The Label: We'll print the label onto the rectangle, because we'll detect several objects.
* We want to print the label "DOG" onto the detector-rectangle of dog and then the label "PERSON" onto the rectangle that is detecting the person.
* To do this, we'll use OpenCV again. We'll use another function called **putText()**

# *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)

* Argumnets:

1. **frame** : It is the image.
2. **labelmap[i - 1]** : This is for the text to display. We need to use the **labelmap** which is the shortcut name we gave to the **VOC\_CLASSES**.

* **VOC\_CLASSES** is the dictionary that maps the names of the classes with numbers.
* Using right index we can get the label of the text we're interested in. Inside the for-loop we're dealing with the class **i**.
* We'll use the index "**i-1**" because index starts from **0** and "**i-1**" means the **i**'th class.

1. **(int(pt[0]), int(pt[1]))** : It is the position of the text where we want to display the label. We're going to display it at the upper left corner of the rectangle.
2. **cv2.FONT\_HERSHEY\_SIMPLEX** : It's just the Font for the text.
3. **2** : Is the size of the text.
4. **(255, 255, 255)** : Is the color of the text.
5. **2** : Is the thickness of the text.
6. **cv2.LINE\_AA** : Makes our text to be displayed continuously without little dots.

* Increment the occurrence **j**: Since we need to deal with the *next occurrence* of the class **i**, we need to increment the occurrence variable **j**.
* Finally we'll ***return*** the ***frame*** with the detector rectangles on any object (our model is trained for 30-40 object) with labels. We do this outside the for-loop.

**All codes we've done up to this step**

# *--------------    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.*

**2.7 SSD - Step 7 (SSD NN)**

Now we're going to create the SSD neural network.

* We already have the weights of a pre-trained SSD neural network. Now we simply create an object that will represent the neural network itself using the **build\_ssd()**function from this **ssd.py** (NN-architecture of SSD) file.
* After we create this neural network object we will get the weights by loading the pre-trained model from "ssd300\_mAP\_77.43\_v2.pth" file using **torch.load()**, which is a function of **torch** and that will open a tensor that will contain these weights.
* Then using another function **load\_state\_dict()** we'll **attribute** these loaded **weights** to our instance object of **NN**.

# *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 a pre-trained model (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 SSD-NN:
* **build\_ssd()** takes only one argument. This argument is the phase. We have two possible phases.

1. **train phase** and
2. **test phase**.

* Since we're going to use an already pre-trained model by loading its weights, we're not going to train anything.
* We're actually going to test the SSD model on our Video.
* So the phase we have to choose is **test** as below

net = **build\_ssd**('test')

Our SSD neural network is created with this single line of code.

* Load the weight: The file 'ssd300\_mAP\_77.43\_v2.pth' contains the weights of an already pre-trained SSD neural network. It's a powerful one. It was pre-trained to detect between **30** to **40** **objects**.
* It is actually more powerful than some other great object detection model's like **Fast R-CNN** or the **YOLO-openCV**.

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

* We take our **net** object and then we're going to use the **load\_state\_dict()** to load the weights, but inside this function we have to use **torch.load()** because we're dealing with tensors, and it loads the tensor that contains the weights.
* Inside **torch.load()** we input following arguments
* **'ssd300\_mAP\_77.43\_v2.pth'** is the file contains the weight of pre-trained SSD-NN.
* map\_location = lambda storage and loc: storage specifies how to load the weight.

Now the **SSD-NN** is created, i.e. we have the frames coming from the video and we have our neural network.

* But, to apply the **detect()** function on the frames, we *not only need* the ***frames*** and the ***NN*** but we also need the ***transform*** transformation. We'll create this transformation in the next step.
* This transformation will make sure that the input frames coming from the video will be compatible with our **SSD-NN**.