16 APRIL 2018 / SERIES: YOLO OBJECT DETECTOR IN PYTORCH

How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 3

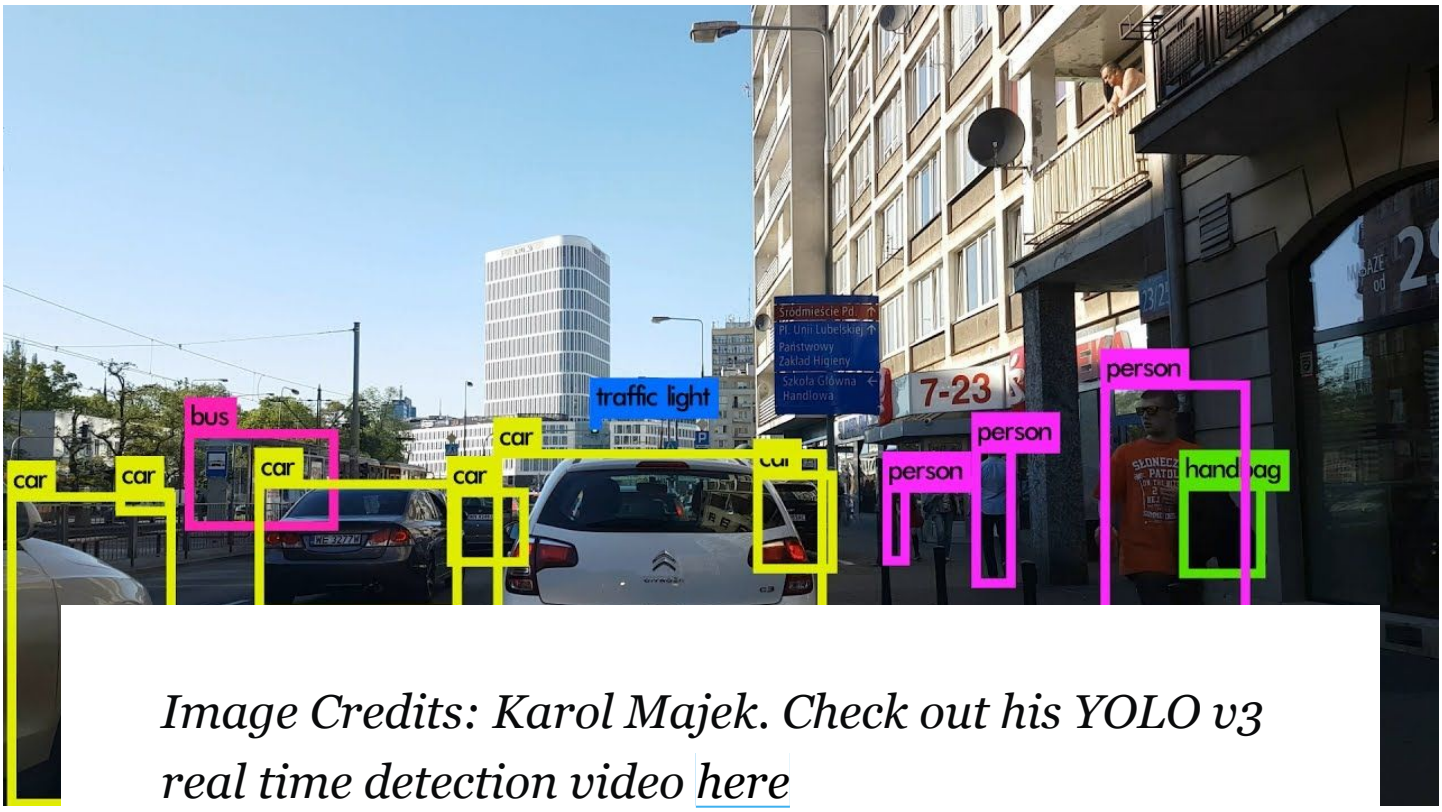


Image Credits: Karol Majek. Check out his YOLO v3 real time detection video here

This is Part 3 of the tutorial on implementing a YOLO v3 detector from scratch. In the last part, we implemented the layers used in YOLO's architecture, and in this part, we are going to implement the network architecture of YOLO in PyTorch, so that we can produce an **output** given an image.

Our objective will be to design the forward pass of the network

PyTorch **0.4**. It can be found in it's entirety at this Github repo.

This tutorial is broken into 5 parts:

1. Part 1 : Understanding How YOLO works
2. Part 2 : Creating the layers of the network architecture
3. Part 3 (This one): Implementing the the forward pass of the network
4. Part 4 : Objectness Confidence Thresholding and Non-maximum Suppression
5. Part 5 : Designing the input and the output pipelines

Prerequisites

- Part 1 and Part 2 of the tutorial.
- Basic working knowledge of PyTorch, including how to create custom architectures with `nn.Module`, `nn.Sequential` and `torch.nn.parameter` classes.
- Working with images in PyTorch

Defining The Network

As I've pointed out earlier, we use `nn.Module` class to build custom architectures in PyTorch. Let us define a network for our detector. In the `darknet.py` file, we add the following class.

```
class Darknet(nn.Module):
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.blocks
```

Here, we have subclassed the `nn.Module` class and named our class `Darknet`. We initialize the network with members, `blocks`, `net_info` and `module_list`.

Implementing the forward pass of the network

The forward pass of the network is implemented by overriding the `forward` method of the `nn.Module` class.

`forward` serves two purposes. First, to calculate the output, and second, to transform the output detection feature maps in a way that it can be processed easier (such as transforming them such that detection maps across multiple scales can be concatenated, which otherwise isn't possible as they are of different dimensions).

```
def forward(self, x, CUDA):
    modules = self.blocks[1:]
    outputs = {}   #We cache the outputs for the route layer
```

`forward` takes three arguments, `self`, the input `x` and `CUDA`, which if true, would use GPU to accelerate the forward pass.

Here, we iterate over `self.blocks[1:]` instead of `self.blocks` since the first element of `self.blocks` is a `net` block which isn't a part of the forward pass.

Since *route* and *shortcut* layers need output maps from previous layers, we cache the output feature maps of every layer in a dict `outputs`. The keys are the the indices of the layers, and the values are

`module_list` which contains the modules of the network. The thing to notice here is that the modules have been appended in the same order as they are present in the configuration file. This means, we can simply run our input through each module to get our output.

```
write = 0     #This is explained a bit later
for i, module in enumerate(modules):
    module_type = (module["type"])
```

Convolutional and Upsample Layers

If the module is a convolutional or upsample module, this is how the forward pass should work.

```
        if module_type == "convolutional" or module_type == "upsamp
            x = self.module_list[i](x)
```

Route Layer / Shortcut Layer

If you look the code for *route* layer, we have to account for two cases (as described in part 2). For the case in which we have to concatenate two feature maps we use the `torch.cat` function with the second argument as 1. This is because we want to concatenate the feature maps along the depth. (In PyTorch, input and output of a convolutional layer has the format `B X C X H X W. The depth corresponding the the channel dimension).

```
            if (layers[0]) > 0:
                layers[0] = layers[0] - i

            if len(layers) == 1:
                x = outputs[i + (layers[0])]

            else:
                if (layers[1]) > 0:
                    layers[1] = layers[1] - i

                map1 = outputs[i + layers[0]]
                map2 = outputs[i + layers[1]]

                x = torch.cat((map1, map2), 1)

        elif  module_type == "shortcut":
            from_ = int(module["from"])
            x = outputs[i-1] + outputs[i+from_]
```

YOLO (Detection Layer)

The output of YOLO is a convolutional feature map that contains the bounding box attributes along the depth of the feature map. The attributes bounding boxes predicted by a cell are stacked one by one along each other. So, if you have to access the second bounding of cell at (5,6), then you will have to index it by `map[5,6, (5+C): 2*(5+C)]`. This form is very inconvenient for output processing such as thresholding by a object confidence, adding grid offsets to centers, applying anchors etc.

Another problem is that since detections happen at three scales, the dimensions of the prediction maps will be different. Although the dimensions of the three feature maps are different, the output processing operations to be done on them are similar. It would be nice to have to do these operations on a single tensor, rather than three separate tensors.

`sform`

Transforming the output

The function `predict_transform` lives in the file `util.py` and we will import the function when we use it in `forward` of `Darknet` class.

Add the imports to the top of `util.py`

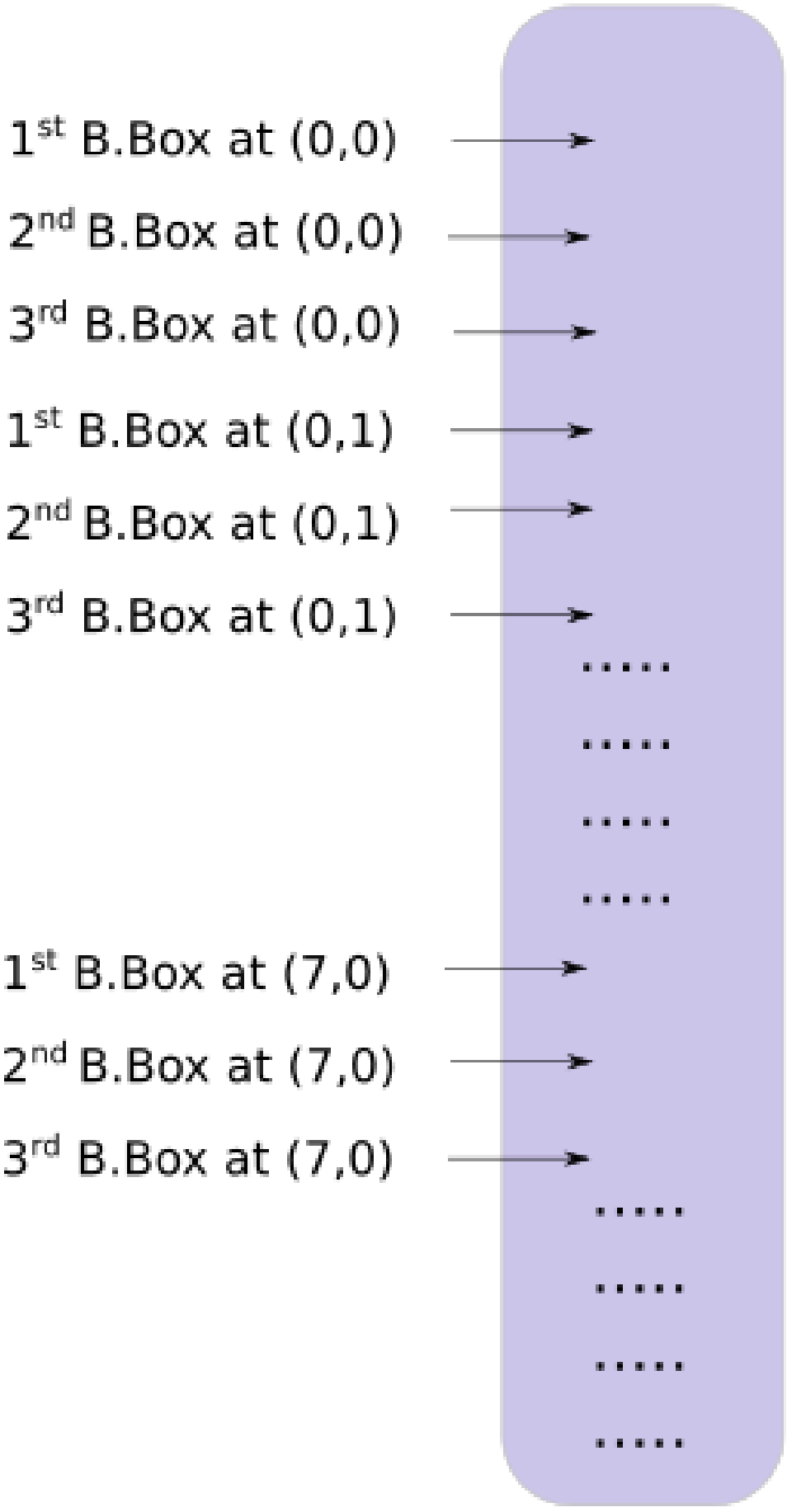
```
from __future__ import division

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
import cv2
```

`predict_transform` takes in 5 parameters; *prediction* (our output), *inp_dim* (input image dimension), *anchors*, *num_classes*, and an optional *CUDA* flag

```
def predict_transform(prediction, inp_dim, anchors, num_classes, CUI
```

`predict_transform` function takes an detection feature map and turns it into a 2-D tensor, where each row of the tensor corresponds to attributes of a bounding box, in the following order.



Here's the code to do the above transformation.

```
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    prediction = prediction.view(batch_size, bbox_attrs*num_anchors
    prediction = prediction.transpose(1,2).contiguous()
    prediction = prediction.view(batch_size, grid_size*grid_size*nu
```

The dimensions of the anchors are in accordance to the `height` and `width` attributes of the `net` block. These attributes describe the dimensions of the input image, which is larger (by a factor of *stride*) than the detection map. Therefore, we must divide the anchors by the stride of the detection feature map.

```
    anchors = [(a[0]/stride, a[1]/stride) for a in anchors]
```

Now, we need to transform our output according to the equations we discussed in Part 1.

Sigmoid the x,y coordinates and the objectness score.

```
    #Sigmoid the  centre_X, centre_Y. and object confidencce
    prediction[:,:,0] = torch.sigmoid(prediction[:,:,0])
    prediction[:,:,1] = torch.sigmoid(prediction[:,:,1])
    prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])
```

Add the grid offsets to the center cordinates prediction.

```
    x_offset = torch.FloatTensor(a).view(-1,1)
    y_offset = torch.FloatTensor(b).view(-1,1)

    if CUDA:
        x_offset = x_offset.cuda()
        y_offset = y_offset.cuda()

    x_y_offset = torch.cat((x_offset, y_offset), 1).repeat(1,num_an

    prediction[:,:,:2] += x_y_offset
```

Apply the anchors to the dimensions of the bounding box.

```
    #log space transform height and the width
    anchors = torch.FloatTensor(anchors)

    if CUDA:
        anchors = anchors.cuda()

    anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
    prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors
```

Apply sigmoid activation to the the class scores

```
    prediction[:,:,5: 5 + num_classes] = torch.sigmoid((prediction[
```

The last thing we want to do here, is to resize the detections map to the size of the input image. The bounding box attributes here are sized according to the feature map (say, 13 x 13). If the input image was 416 x 416, we multiply the attributes by 32, or the *stride*

```
prediction[:,:,:4] *= stride
```

That concludes the loop body.

Return the predictions at the end of the function.

```
    return prediction
```

Detection Layer Revisited

Now that we have transformed our output tensors, we can now concatenate the detection maps at three different scales into one big tensor. Notice this was not possible prior to our transformation, as one cannot concatenate feature maps having different spatial dimensions. But since now, our output tensor acts merely as a table with bounding boxes as it's rows, concatenation is very much possible.

An obstacle in our way is that we cannot initialize an empty tensor, and then concatenate a non-empty (of different shape) tensor to it. So, we delay the initialization of the collector (tensor that holds the detections) until we get our first detection map, and then concatenate to maps to it when we get subsequent detections.

Notice the `write = 0` line just before the loop in the function `forward`. The `write` flag is used to indicate whether we have encountered the first detection or not. If `write` is 0, it means the collector hasn't been initialized. If it is 1, it means that the

Now, that we have armed ourselves with the `predict_transform` function, we write the code for handling detection feature maps in the `forward` function.

At the top of your `darknet.py` file, add the following import.

```
from util import *
```

Then, in the `forward` function.

```
        elif module_type == 'yolo':

            anchors = self.module_list[i][0].anchors
            #Get the input dimensions
            inp_dim = int (self.net_info["height"])

            #Get the number of classes
            num_classes = int (module["classes"])

            #Transform
            x = x.data
            x = predict_transform(x, inp_dim, anchors, num_classes,
            if not write:              #if no collector has been in
                detections = x
                write = 1

            else:
                detections = torch.cat((detections, x), 1)

        outputs[i] = x
```

Now, simply return the detections.

Testing the forward pass

Here's a function that creates a dummy input. We will pass this input to our network. Before we write this function, save this image into your working directory . If you're on linux, then type.

```
wget https://github.com/ayooshkathuria/pytorch-yolo-v3/raw/master/d
```

Now, define the function at the top of your `darknet.py` file as follows:

```
def get_test_input():
    img = cv2.imread("dog-cycle-car.png")
    img = cv2.resize(img, (416,416))          #Resize to the input
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_
```

Then, we type the following code:

```
model = Darknet("cfg/yolov3.cfg")
inp = get_test_input()
pred = model(inp, torch.cuda.is_available())
```

You will see an output like:

```
(  0  ,.,.) =
   16.0962   17.0541   91.5104  ...      0.4336    0.4692    0.5279
   15.1363   15.2568  166.0840  ...      0.5561    0.5414    0.5318
   14.4763   18.5405  409.4371  ...      0.5908    0.5353    0.4979
               ⋱                ...
  411.2625  412.0660    9.0127  ...      0.5054    0.4662    0.5043
  412.1762  412.4936   16.0449  ...      0.4815    0.4979    0.4582
  412.1629  411.4338   34.9027  ...      0.4306    0.5462    0.4138
[torch.FloatTensor of size 1x10647x85]
```

The shape of this tensor is `1 x 10647 x 85` . The first dimension is the batch size which is simply 1 because we have used a single image. For each image in a batch, we have a 10647 x 85 table. The row of each of this table represents a bounding box. (4 bbox attributes, 1 objectness score, and 80 class scores)

At this point, our network has random weights, and will not produce the correct output. We need to load a weight file in our network. We'll be making use of the official weight file for this purpose.

Downloading the Pre-trained Weights

Download the weights file into your detector directory. Grab the weights file from here. Or if you're on linux,

```
wget https://pjreddie.com/media/files/yolov3.weights
```

in a serial fashion.

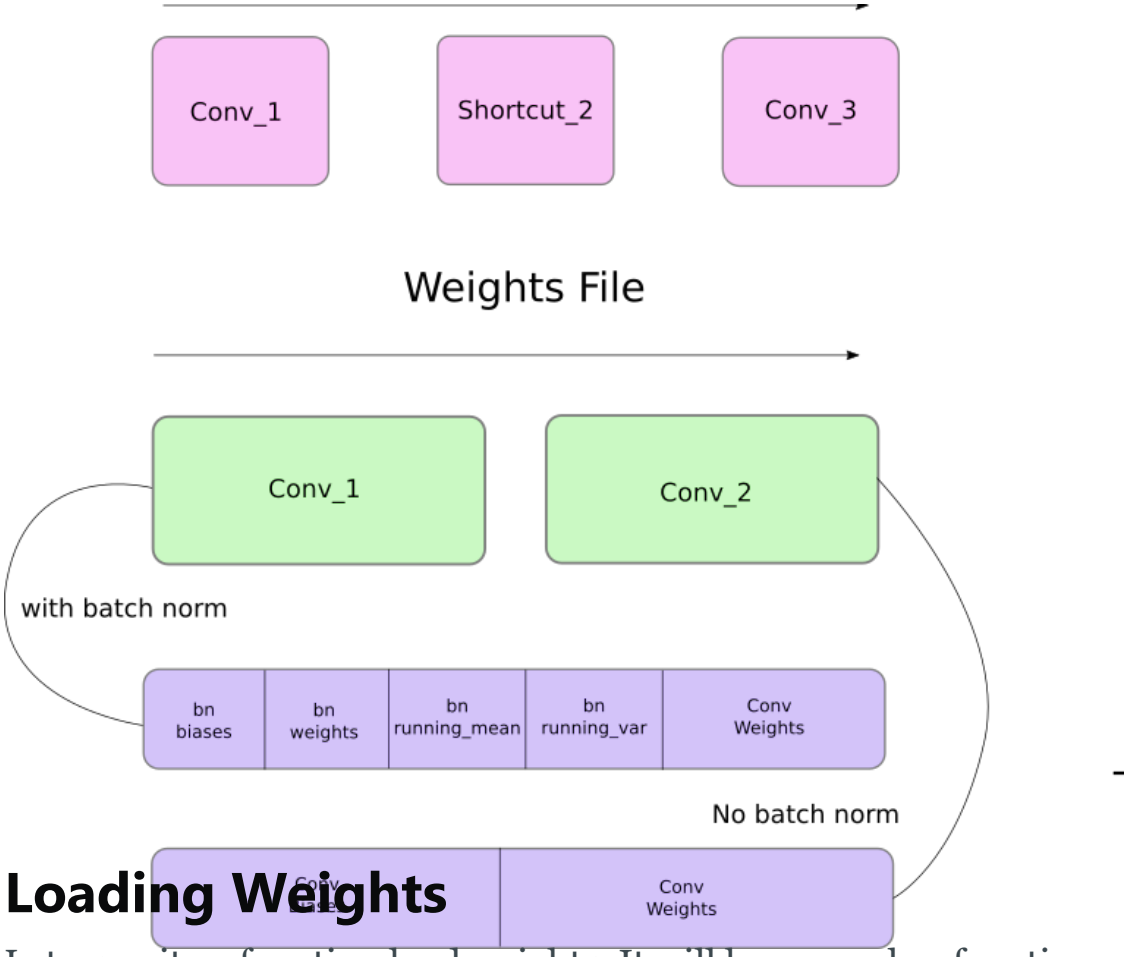
Extreme care must be taken to read the weights. The weights are just stored as floats, with nothing to guide us as to which layer do they belong to. If you screw up, there's nothing stopping you to, say, load the weights of a batch norm layer into those of a convolutional layer. Since, you're reading only floats, there's no way to discriminate between which weight belongs to which layer. Hence, we must understand how the weights are stored.

First, the weights belong to only two types of layers, either a batch norm layer or a convolutional layer.

The weights for these layers are stored exactly in the same order as they appear in the configuration file. So, if a `convolutional` is followed by a `shortcut` block, and then the `shortcut` block by another `convolutional` block, You will expect file to contain the weights of the previous `convolutional` block, followed by those of the latter.

When the batch norm layer appears in a `convolutional` block, there are no biases. However, when there's no batch norm layer, bias "weights" have to read from the file.

The following diagram sums up how the weight stores the weights.



Loading Weights

Let us write a function load weights. It will be a member function of the `Darknet` class. It'll take one argument other than `self`, the path of the weightsfile.

```
def load_weights(self, weightfile):
```

The first 160 bytes of the weights file store 5 `int32` values which constitute the header of the file.

```
    #Open the weights file
    fp = open(weightfile, "rb")

    #The first 5 values are header information
    # 1. Major version number
```

```
        self.header = torch.from_numpy(header)
        self.seen = self.header[3]
```

The rest of bits now represent the weights, in the order described above. The weights are stored as `float32` or 32-bit floats. Let's load rest of the weights in a `np.ndarray`.

```
    weights = np.fromfile(fp, dtype = np.float32)
```

Now, we iterate over the weights file, and load the weights into the modules of our network.

```
    ptr = 0
    for i in range(len(self.module_list)):
        module_type = self.blocks[i + 1]["type"]

        #If module_type is convolutional load weights
        #Otherwise ignore.
```

Into the loop, we first check whether the `convolutional` block has `batch_normalise` True or not. Based on that, we load the weights.

```
        if module_type == "convolutional":
            model = self.module_list[i]
            try:
                batch_normalize = int(self.blocks[i+1]["batch_norma
            except:
                batch_normalize = 0
```

We keep a variable called `ptr` to keep track of where we are in the weights array. Now, if `batch_normalize` is True, we load the weights as follows.

```
        if (batch_normalize):
            bn = model[1]

            #Get the number of weights of Batch Norm Layer
            num_bn_biases = bn.bias.numel()

            #Load the weights
            bn_biases = torch.from_numpy(weights[ptr:ptr + num_bn_b
            ptr += num_bn_biases

            bn_weights = torch.from_numpy(weights[ptr: ptr + num_bn_
            ptr  += num_bn_biases

            bn_running_mean = torch.from_numpy(weights[ptr: ptr + nu
            ptr  += num_bn_biases

            bn_running_var = torch.from_numpy(weights[ptr: ptr + nu
            ptr  += num_bn_biases

            #Cast the loaded weights into dims of model weights.
            bn_biases = bn_biases.view_as(bn.bias.data)
            bn_weights = bn_weights.view_as(bn.weight.data)
            bn_running_mean = bn_running_mean.view_as(bn.running_mea
            bn_running_var = bn_running_var.view_as(bn.running_var)

            #Copy the data to model
            bn.bias.data.copy_(bn_biases)
            bn.weight.data.copy_(bn_weights)
            bn.running_mean.copy_(bn_running_mean)
            bn.running_var.copy_(bn_running_var)
```

If batch_norm is not true, simply load the biases of the convolutional layer.

```
            #Number of biases
            num_biases = conv.bias.numel()

            #Load the weights
            conv_biases = torch.from_numpy(weights[ptr: ptr + num_b
            ptr = ptr + num_biases

            #reshape the loaded weights according to the dims of th
            conv_biases = conv_biases.view_as(conv.bias.data)

            #Finally copy the data
            conv.bias.data.copy_(conv_biases)
```

Finally, we load the convolutional layer's weights at last.

```
#Let us load the weights for the Convolutional layers
num_weights = conv.weight.numel()

#Do the same as above for weights
conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
ptr = ptr + num_weights

conv_weights = conv_weights.view_as(conv.weight.data)
conv.weight.data.copy_(conv_weights)
```

We're done with this function and you can now load weights in your `Darknet` object by calling the `load_weights` function on the darknet object.

```
model = Darknet("cfg/yolov3.cfg")
model.load_weights("yolov3.weights")
```

cover the use of objectness confidence thresholding and Non-maximum suppression to produce our final set of detections.

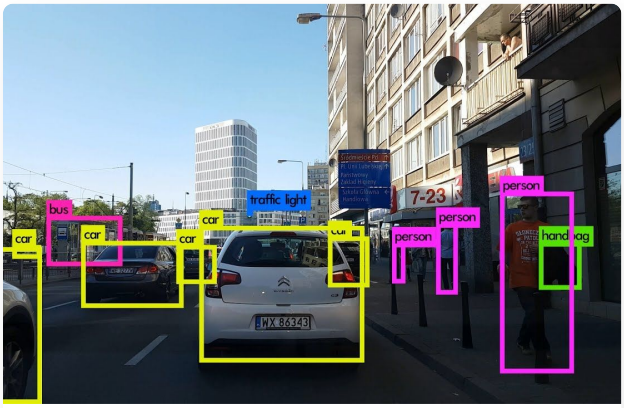
Further Reading

1. PyTorch tutorial
2. Reading binary files with NumPy
3. nn.Module, nn.Parameter classes

Ayoosh Kathuria is currently an intern at the Defense Research and Development Organization, India, where he is working on improving object detection in grainy videos. When he's not working, he's either sleeping or playing pink floyd on his guitar. You can connect with him on LinkedIn or look at more of what he does at GitHub

Ayoosh Kathuria
Deep Learning Engineer at Mathworks. Currently working on bringing GANs to MATLAB. Previously a research intern at DRDO. Passionate about computer vision and unsupervised learning.

Read More



SERIES: YOLO OBJECT DETECTOR IN PYTORCH

How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 2

How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 4

See all 4 posts →

Part 2 of the tutorial series on how to implement your own YOLO v3 object detector from scratch in PyTorch.

11 MIN READ

SERIES: YOLO OBJECT DETECTOR IN PYTORCH

How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 4

Part 4 of the tutorial series on how to implement a YOLO v3 object detector from scratch using PyTorch.

7 MIN READ