本教程分为5个部分：

第1部分：理解 YOLO 的原理

第2部分（本文）：创建网络结构

第3部分：实现网络的前向传递

第4部分：目标分阈值和非极大值抑制

第5部分：网络的输入和输出

文章目录 [隐藏]

开始

首先创建一个存放检测器代码的文件夹，然后再创建 Python 文件 darknet.py。Darknet 是构建 YOLO 底层架构的环境，这个文件将包含实现 YOLO 网络的所有代码。同样我们还需要补充一个名为 util.py 的文件，它会包含多种需要调用的函数。在将所有这些文件保存在检测器文件夹下后，我们就能使用 git 追踪它们的改变。

配置文件

官方代码（authored in C）使用一个配置文件来构建网络，即 cfg 文件一块块地描述了网络架构。如果你使用过 caffe 后端，那么它就相当于描述网络的.protxt 文件。

我们将使用官方的 cfg 文件构建网络，它是由 YOLO 的作者发布的。我们可以在以下地址下载，并将其放在检测器目录下的 cfg 文件夹下。

配置文件下载：https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg

当然，如果你使用 Linux，那么就可以先 cd 到检测器网络的目录，然后运行以下命令行。

```
mkdir cfg
cd cfg
wget https://raw.githubusercontent.com/pjreddie/darknet/master/cfg/yolov3.cfg
```

如果你打开配置文件，你将看到如下一些网络架构：

```
[convolutional]
batch_normalize=1
filters=64
```

```
size=3
stride=2
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=32
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[shortcut]
from=-3
activation=linear
```

我们看到上面有四块配置，其中 3 个描述了卷积层，最后描述了 ResNet 中常用的捷径层或跳过连接。下面是 YOLO 中使用的 5 种层级：

1. 卷积层

```
[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky
```

2. 跳过连接

```
[shortcut]
from=-3
activation=linear
```

跳过连接与残差网络中使用的结构相似，参数 from 为-3 表示捷径层的输出会通过将之前层的和之前第三个层的输出的特征图与模块的输入相加而得出。

3.上采样

```
[upsample]
stride=2
```

通过参数 stride 在前面层级中双线性上采样特征图。

4.路由层（Route）

```
[route]
layers = -4

[route]
layers = -1, 61
```

路由层需要一些解释，它的参数 layers 有一个或两个值。当只有一个值时，它输出这一层通过该值索引的特征图。在我们的实验中设置为了-4，所以层级将输出路由层之前第四个层的特征图。

当层级有两个值时，它将返回由这两个值索引的拼接特征图。在我们的实验中为-1 和 61，因此该层级将输出从前一层级（-1）到第 61 层的特征图，并将它们按深度拼接。

5.YOLO

```
[yolo]
mask = 0,1,2
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326
classes=80
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=1
```

YOLO 层级对应于上文所描述的检测层级。参数 anchors 定义了 9 组锚点，但是它们只是由 mask 标签使用的属性所索引的锚点。这里，mask 的值为 0、1、2 表示了第一个、第二个和第三个使用的锚点。而掩码表示检测层中的每一个单元预测三个框。总而言之，我们检测层的规模为 3，并装配总共 9 个锚点。

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=16
width= 320
height = 320
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

配置文件中存在另一种块 net，不过我不认为它是层，因为它只描述网络输入和训练参数的相关信息，并未用于 YOLO 的前向传播。但是，它为我们提供了网络输入大小等信息，可用于调整前向传播中的锚点。

解析配置文件

在开始之前，我们先在 darknet.py 文件顶部添加必要的导入项。

```
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
```

我们定义一个函数 parse_cfg，该函数使用配置文件的路径作为输入。

```
def parse_cfg(cfgfile):
    """
    Takes a configuration file

    Returns a list of blocks. Each blocks describes a block in the neural
    network to be built. Block is represented as a dictionary in the list

    """
```

这里的思路是解析 cfg，将每个块存储为词典。这些块的属性和值都以键值对的形式存储在词典中。解析过程中，我们将这些词典（由代码中的变量 block 表示）添加到列表 blocks 中。我们的函数将返回该 block。

我们首先将配置文件内容保存在字符串列表中。下面的代码对该列表执行预处理：

```
file = open(cfgfile, 'r')
lines = file.read().split('\n')                        # store the lines in a list
lines = [x for x in lines if len(x) > 0]               # get read of the empty lines
lines = [x for x in lines if x[0] != '#']              # get rid of comments
lines = [x.rstrip().lstrip() for x in lines]           # get rid of fringe whitespaces
```

然后，我们遍历预处理后的列表，得到块。

```
block = {}
blocks = []

for line in lines:
    if line[0] == "[":               # This marks the start of a new block
        if len(block) != 0:          # If block is not empty, implies it is storing values of pr
            blocks.append(block)     # add it the blocks list
            block = {}               # re-init the block
        block["type"] = line[1:-1].rstrip()
    else:
        key,value = line.split("=")
        block[key.rstrip()] = value.lstrip()
blocks.append(block)

return blocks
```

创建构建块

现在我们将使用上面 parse_cfg 返回的列表来构建 PyTorch 模块，作为配置文件中的构建块。

列表中有 5 种类型的层。PyTorch 为 convolutional 和 upsample 提供预置层。我们将通过扩展 nn.Module 类为其余层写自己的模块。

create_modules 函数使用 parse_cfg 函数返回的 blocks 列表：

```
def create_modules(blocks):
    net_info = blocks[0]     #Captures the information about the input and pre-processing
    module_list = nn.ModuleList()
    prev_filters = 3
    output_filters = []
```

在迭代该列表之前，我们先定义变量 net_info，来存储该网络的信息。

```
nn.ModuleList
```

我们的函数将会返回一个 nn.ModuleList。这个类几乎等同于一个包含 nn.Module 对象的普通列表。然而，当添加 nn.ModuleList 作为 nn.Module 对象的一个成员时（即当我们添加模块到我们的网络时），所有 nn.ModuleList 内部的 nn.Module 对象（模块）的 parameter 也被添加作为 nn.Module 对象（即我们的网络，添加 nn.ModuleList 作为其成员）的 parameter。

当我们定义一个新的卷积层时，我们必须定义它的卷积核维度。虽然卷积核的高度和宽度由 cfg 文件提供，但卷积核的深度是由上一层的卷积核数量（或特征图深度）决定的。这意味着我们需要持续追踪被应用卷积层的卷积核数量。我们使用变量 prev_filter 来做这件事。我们将其初始化为 3，因为图像有对应 RGB 通道的 3 个通道。

路由层（route layer）从前面层得到特征图（可能是拼接的）。如果在路由层之后有一个卷积层，那么卷积核将被应用到前面层的特征图上，精确来说是路由层得到的特征图。因此，我们不仅需要追踪前一层的卷积核数量，还需要追踪之前每个层。随着不断地迭代，我们将每个模块的输出卷积核数量添加到 output_filters 列表上。

现在，我们的思路是迭代模块的列表，并为每个模块创建一个 PyTorch 模块。

```
    for index, x in enumerate(blocks[1:]):
        module = nn.Sequential()

        #check the type of block
        #create a new module for the block
        #append to module_list
```

nn.Sequential 类被用于按顺序地执行 nn.Module 对象的一个数字。如果你查看 cfg 文件，你会发现，一个模块可能包含多于一个层。例如，一个 convolutional 类型的模块有一个批量归一化层、一个 leaky ReLU 激活层以及一个卷积层。我们使用 nn.Sequential 将这些层串联起来，得到 add_module 函数。例如，以下展示了我们如何创建卷积层和上采样层的例子：

```
        if (x["type"] == "convolutional"):
            #Get the info about the layer
            activation = x["activation"]
            try:
                batch_normalize = int(x["batch_normalize"])
```

```
            bias = False
        except:
            batch_normalize = 0
            bias = True

        filters= int(x["filters"])
        padding = int(x["pad"])
        kernel_size = int(x["size"])
        stride = int(x["stride"])

        if padding:
            pad = (kernel_size - 1) // 2
        else:
            pad = 0

        #Add the convolutional layer
        conv = nn.Conv2d(prev_filters, filters, kernel_size, stride, pad, bias = bias)
        module.add_module("conv_{0}".format(index), conv)

        #Add the Batch Norm Layer
        if batch_normalize:
            bn = nn.BatchNorm2d(filters)
            module.add_module("batch_norm_{0}".format(index), bn)

        #Check the activation.
        #It is either Linear or a Leaky ReLU for YOLO
        if activation == "leaky":
            activn = nn.LeakyReLU(0.1, inplace = True)
            module.add_module("leaky_{0}".format(index), activn)

    #If it's an upsampling layer
    #We use Bilinear2dUpsampling
    elif (x["type"] == "upsample"):
        stride = int(x["stride"])
        upsample = nn.Upsample(scale_factor = 2, mode = "bilinear")
        module.add_module("upsample_{}".format(index), upsample)
```

路由层/捷径层

接下来，我们来写创建路由层（Route Layer）和捷径层（Shortcut Layer）的代码：

```
    #If it is a route layer
    elif (x["type"] == "route"):
        x["layers"] = x["layers"].split(',')
        #Start  of a route
        start = int(x["layers"][0])
        #end, if there exists one.
        try:
            end = int(x["layers"][1])
        except:
            end = 0
        #Positive anotation
        if start > 0:
            start = start - index
        if end > 0:
            end = end - index
        route = EmptyLayer()
        module.add_module("route_{0}".format(index), route)
        if end < 0:
            filters = output_filters[index + start] + output_filters[index + end]
```

```
            else:
                filters= output_filters[index + start]

        #shortcut corresponds to skip connection
        elif x["type"] == "shortcut":
            shortcut = EmptyLayer()
            module.add_module("shortcut_{}".format(index), shortcut)
```

创建路由层的代码需要做一些解释。首先，我们提取关于层属性的值，将其表示为一个整数，并保存在一个列表中。

然后我们得到一个新的称为 EmptyLayer 的层，顾名思义，就是空的层。

```
route = EmptyLayer()
```

其定义如下：

```
class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()
```

等等，一个空的层?

现在，一个空的层可能会令人困惑，因为它没有做任何事情。而 Route Layer 正如其它层将执行某种操作（获取之前层的拼接）。在 PyTorch 中，当我们定义了一个新的层，我们在子类 nn.Module 中写入层在 nn.Module 对象的 forward 函数的运算。

对于在 Route 模块中设计一个层，我们必须建立一个 nn.Module 对象，其作为 layers 的成员被初始化。然后，我们可以写下代码，将 forward 函数中的特征图拼接起来并向前馈送。最后，我们执行网络的某个 forward 函数的这个层。

但拼接操作的代码相当地短和简单（在特征图上调用 torch.cat），像上述过程那样设计一个层将导致不必要的抽象，增加样板代码。取而代之，我们可以将一个假的层置于之前提出的路由层的位置上，然后直接在代表 darknet 的 nn.Module 对象的 forward 函数中执行拼接运算。（如果感到困惑，我建议你读一下 nn.Module 类在 PyTorch 中的使用）。

在路由层之后的卷积层会把它的卷积核应用到之前层的特征图（可能是拼接的）上。以下的代码更新了 filters 变量以保存路由层输出的卷积核数量。

```
if end < 0:
    #If we are concatenating maps
    filters = output_filters[index + start] + output_filters[index + end]
else:
    filters= output_filters[index + start]
```

捷径层也使用空的层，因为它还要执行一个非常简单的操作（加）。没必要更新 filters 变量，因为它只是将前一层的特征图添加到后面的层上而已。

YOLO 层

最后，我们将编写创建 YOLO 层的代码：

```
        #Yolo is the detection layer
        elif x["type"] == "yolo":
            mask = x["mask"].split(",")
            mask = [int(x) for x in mask]

            anchors = x["anchors"].split(",")
            anchors = [int(a) for a in anchors]
            anchors = [(anchors[i], anchors[i+1]) for i in range(0, len(anchors),2)]
            anchors = [anchors[i] for i in mask]

            detection = DetectionLayer(anchors)
            module.add_module("Detection_{}".format(index), detection)
```

我们定义一个新的层 DetectionLayer 保存用于检测边界框的锚点。

检测层的定义如下：

```
class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors
```

在这个回路结束时，我们做了一些统计（bookkeeping.）。

```
        module_list.append(module)
        prev_filters = filters
        output_filters.append(filters)
```

这总结了此回路的主体。在 create_modules 函数后，我们获得了包含 net_info 和 module_list 的元组。

```
return (net_info, module_list)
```

测试代码

你可以在 darknet.py 后通过输入以下命令行测试代码，运行文件。

```
blocks = parse_cfg("cfg/yolov3.cfg")
print(create_modules(blocks))
```

你会看到一个长列表（确切来说包含 106 条），其中元素看起来如下所示：

```
.
.

  (9): Sequential(
    (conv_9): Conv2d (128, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (batch_norm_9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
```

```
    (leaky_9): LeakyReLU(0.1, inplace)
  )
  (10): Sequential(
    (conv_10): Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (batch_norm_10): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
    (leaky_10): LeakyReLU(0.1, inplace)
  )
  (11): Sequential(
    (shortcut_11): EmptyLayer(
    )
  )
.
.
.
```

该教程的第二部分到此结束。