ScaledYOLOv4笔记

- Paper: Scaled-YOLOv4: Scaling Cross Stage Partial Network
- Code: ScaledYOLOv4
- Blog: Scaled YOLO v4 is the best neural network for object detection on MS COCO dataset

0. Abstract

Improvements in Scaled YOLOv4 over YOLOv4

- Scaled YOLOv4 used optimal network scaling techniques to get YOLOv4-CSP -> P5 -> P6 -> P7 networks
- Improved network architecture: Backbone is optimized and Neck (PAN) uses Cross-stage-partial (CSP) connections and Mish activation
- Exponential Moving Average (EMA) is used during training — this is a special case of SWA: https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging/
- For each resolution of the network, a separate neural network is trained (in YOLOv4, only one neural network was trained for all resolutions)
- Improved objectness normalizers in [yolo] layers
- Changed activations for Width and Height, which allows faster network training
- The [net] letter_box = 1 parameter (to keep the aspect ratio of the input image) is used for high resolution networks (for all networks except yolov4-tiny.cfg)

different Losses

There are different Losses in YOLOv3, YOLOv4 and Scaled-YOLOv4:

YOLOv3	YOLOv4	Scaled-YOLOv4
$b_x = \sigma(t_x) + c_x$	$b_x = \sigma(t_x) * 1.1 - 0.05 + c_x$	$b_x = \sigma(t_x) * 2 - 0.5 + c_x$
$b_y = \sigma(t_y) + c_y$	$b_y = \sigma(t_y) * 1.1 - 0.05 + c_y$	$b_y = \sigma(t_y) * 2 - 0.5 + c_y$
$b_w = p_w e^{t_w}$	$b_w = p_w e^{t_w}$	$b_w = (\sigma(t_w) * 2)^2 * p_w$
$b_h = p_h e^{t_h}$	$b_h = p_h e^{t_h}$	$b_h = (\sigma(t_h) * 2)^2 * p_h$

Loss for YOLOv3, YOLOv4 and Scaled-YOLOv4

- for bx and by — this eliminates grid sensitivity in the same way as in YOLOv4, but more aggressively
- for bw and bh — this limits the size of the bounded-box to 4*Anchor_size

network architecture

Changes to the network architecture (CSP in the Neck and Mish-activation for all layers) then eliminate flaws of Pytorch implementation, so CSP+Mish improves both AP, AP50 and FPS:

- YOLOv4-CSP — 608x608–75FPS — **47.5%** AP — **66.1%** AP50
- YOLOv4-CSP — 640x640–70FPS — **47.5%** AP — **66.2%** AP50

CSP connection is extremely efficient, simple, and can be applied to any neural network. The idea is:

- half of the output signal goes along the main path (generates more semantic information with a large receiving field)
- and the other half of the signal goes bypass (preserves more spatial information with a small perceiving field)

1. 模型结构

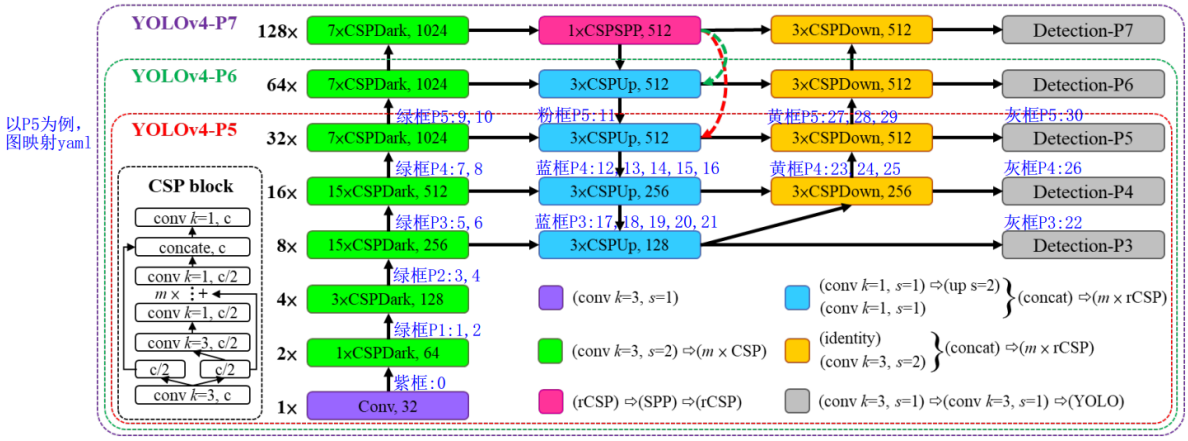

Figure 4: Architecture of YOLOv4-large, including YOLOv4-P5, YOLOv4-P6, and YOLOv4-P7. The dashed arrow means replace the corresponding CSPUp block by CSPSPP block.

- 以yolov4-p5.yaml为例

```
# parameters
nc: 80  # number of classes
depth_multiple: 1.0  # model depth multiple
width_multiple: 1.0  # layer channel multiple

# anchors
anchors:
  - [13,17,  31,25,  24,51, 61,45]  # P3/8  (img缩小8倍)
  - [48,102,  119,96, 97,189, 217,184]  # P4/16  (img缩小16倍)
  - [171,384, 324,451, 616,618, 800,800]  # P5/32  (img缩小32倍)

# csp-p5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [32, 3, 1]],  # 0  (在list中的index为0)  对应图中紫框

   [-1, 1, Conv, [64, 3, 2]],  # 1-P1/2  (img缩小2倍)  对应图中绿框P1中的conv
k=3,s=2
   [-1, 1, BottleneckCSP, [64]],  # 2  对应图中绿框P1中的1 x CSP

   [-1, 1, Conv, [128, 3, 2]],  # 3-P2/4  (img缩小4倍)  对应图中绿框P2中的conv
k=3,s=2
   [-1, 3, BottleneckCSP, [128]],  # 4  对应图中绿框P2中的3 x CSP

   [-1, 1, Conv, [256, 3, 2]],  # 5-P3/8  (img缩小8倍)  对应图中绿框P3中的conv
k=3,s=2
   [-1, 15, BottleneckCSP, [256]],  # 6  对应图中绿框P3中的15 x CSP
```

```
   [-1, 1, Conv, [512, 3, 2]],  # 7-P4/16  (img缩小16倍)  对应图中绿框P4中的conv
k=3,s=2
   [-1, 15, BottleneckCSP, [512]],  # 8  对应图中绿框P4中的15 x CSP

   [-1, 1, Conv, [1024, 3, 2]],  # 9-P5/32  (img缩小32倍)  对应图中绿框P5中的conv
k=3,s=2
   [-1, 7, BottleneckCSP, [1024]],  # 10  对应图中绿框P5中的7 x CSP
  ]

# yolov4-p5 head
# na = len(anchors[0])  # 每一个cell多少个anchor
head:
  [[-1, 1, SPPCSP, [512]],  # 11  对应图中粉框P5

   [-1, 1, Conv, [256, 1, 1]],  # 12  对应图中蓝框P4中的conv k=1,s=1
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],  # 13  对应图中蓝框P4中的up s=2
   [8, 1, Conv, [256, 1, 1]],  # 14  route backbone P4  对应图中蓝框P4中的conv
k=1,s=1
   [[-1, -2], 1, Concat, [1]],  # 15  对应图中蓝框P4中的concat，将绿框P4卷积后和上面up
的特征融合
   [-1, 3, BottleneckCSP2, [256]], # 16  对应图中蓝框P4中的3 x rCSP

   [-1, 1, Conv, [128, 1, 1]],  # 17  对应图中蓝框P3中的conv k=1,s=1
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],  # 18  对应图中蓝框P3中的up s=2
   [6, 1, Conv, [128, 1, 1]],  # 19  route backbone P3  对应图中蓝框P3中的conv
k=1,s=1
   [[-1, -2], 1, Concat, [1]],  # 20  对应图中蓝框P3中的concat，将绿框P3卷积后和上面up
的特征融合
   [-1, 3, BottleneckCSP2, [128]],  # 21  对应图中蓝框P3中的3 x rCSP

   [-1, 1, Conv, [256, 3, 1]],  # 22  对应图中灰框P3的conv k=3,s=1

   [-2, 1, Conv, [256, 3, 2]],  # 23  对应图中黄框P4中的conv k=3,s=2
   [[-1, 16], 1, Concat, [1]],  # 24  对应图中黄框P4中的concat，将蓝框P4和上面的特征融
合
   [-1, 3, BottleneckCSP2, [256]], # 25  对应图中黄框P4中的3 x rCSP

   [-1, 1, Conv, [512, 3, 1]],  # 26  对应图中灰框P4的conv k=3,s=1

   [-2, 1, Conv, [512, 3, 2]],  # 27  对应图中黄框P5中的conv k=3,s=2
   [[-1, 11], 1, Concat, [1]],  # 28  对应图中黄框P5中的concat，将蓝框P5和上面的特征融
合
   [-1, 3, BottleneckCSP2, [512]],  # 29  对应图中黄框P5中的3 x rCSP

   [-1, 1, Conv, [1024, 3, 1]],  # 30  对应图中灰框P5的conv k=3,s=1

    ##在网络结构list中的index为22、26、30的layer
   [[22,26,30], 1, Detect, [nc, anchors]],   # Detect(P3, P4, P5)  对应图中灰色框
  ]
```

- 模型主文件：models/yolo.py

2. Train Custom Data

2.1 Create dataset.yaml

```yaml
# train and val datasets (image directory or *.txt file with image paths)
train: ../person/images/train  # 118k images
val: ../person/images/val  # 5k images

# number of classes
nc: 1

# class names
names: ['person']
```

2.2 Create Labels

one `*.txt` file per image (if no objects in image, no `*.txt` file is required). The `*.txt` file specifications are:

- One row per object
- Each row is `class x_center y_center width height` format.
- Box coordinates must be in **normalized xywh** format (from 0 - 1). If your boxes are in pixels, divide `x_center` and `width` by image width, and `y_center` and `height` by image height.
- Class numbers are zero-indexed (start from 0).

2.3 Organize Directories

```
/person
    /images
        /train
            001.jpg
        /val
            007.jpg
/person
    /labels
        /train
            001.txt
        /val
            007.txt
```

2.4 修改模型配置文件

- **以yolov4-p5.yaml为例**:

```yaml
# parameters
  nc: 1  # number of classes
  depth_multiple: 1.0  # model depth multiple
  width_multiple: 1.0  # layer channel multiple
```

注意，其实不修改也行，因为作者在代码中进行了处理:

```
    ##opt.data，数据文件(coco.yaml)
    with open(opt.data) as f:
        data_dict = yaml.load(f, Loader=yaml.FullLoader)  # model dict
    train_path = data_dict['train']
    test_path = data_dict['val']
    ##nc是以数据文件(coco.yaml)中的nc为准，在后续处理中会将模型结构(例如yolov4-p5.yaml)中的nc设置为数据文件(coco.yaml)中的nc
    nc, names = (1, ['item']) if opt.single_cls else (int(data_dict['nc']), data_dict['names'])  # number classes, names
    assert len(names) == nc, '%g names found for nc=%g dataset in %s' % (len(names), nc, opt.data)  # check
```

```
model = Model(opt.cfg or ckpt['model'].yaml, ch=3, nc=nc).to(device)  # create
```

```
##模型主类
class Model(nn.Module):
    def __init__(self, cfg='yolov4-p5.yaml', ch=3, nc=None):  # model, input channels, number of classes
        super(Model, self).__init__()
        # Define model
        if nc and nc != self.yaml['nc']:
            print('Overriding %s nc=%g with nc=%g' % (cfg, self.yaml['nc'], nc))
            ##nc是以数据文件(coco.yaml)中的nc为准，在这里将模型结构(例如yolov4-p5.yaml)中的nc设置为数据文件(coco.yaml)中的nc
            self.yaml['nc'] = nc  # override yaml value
```

问雪更新于20210112