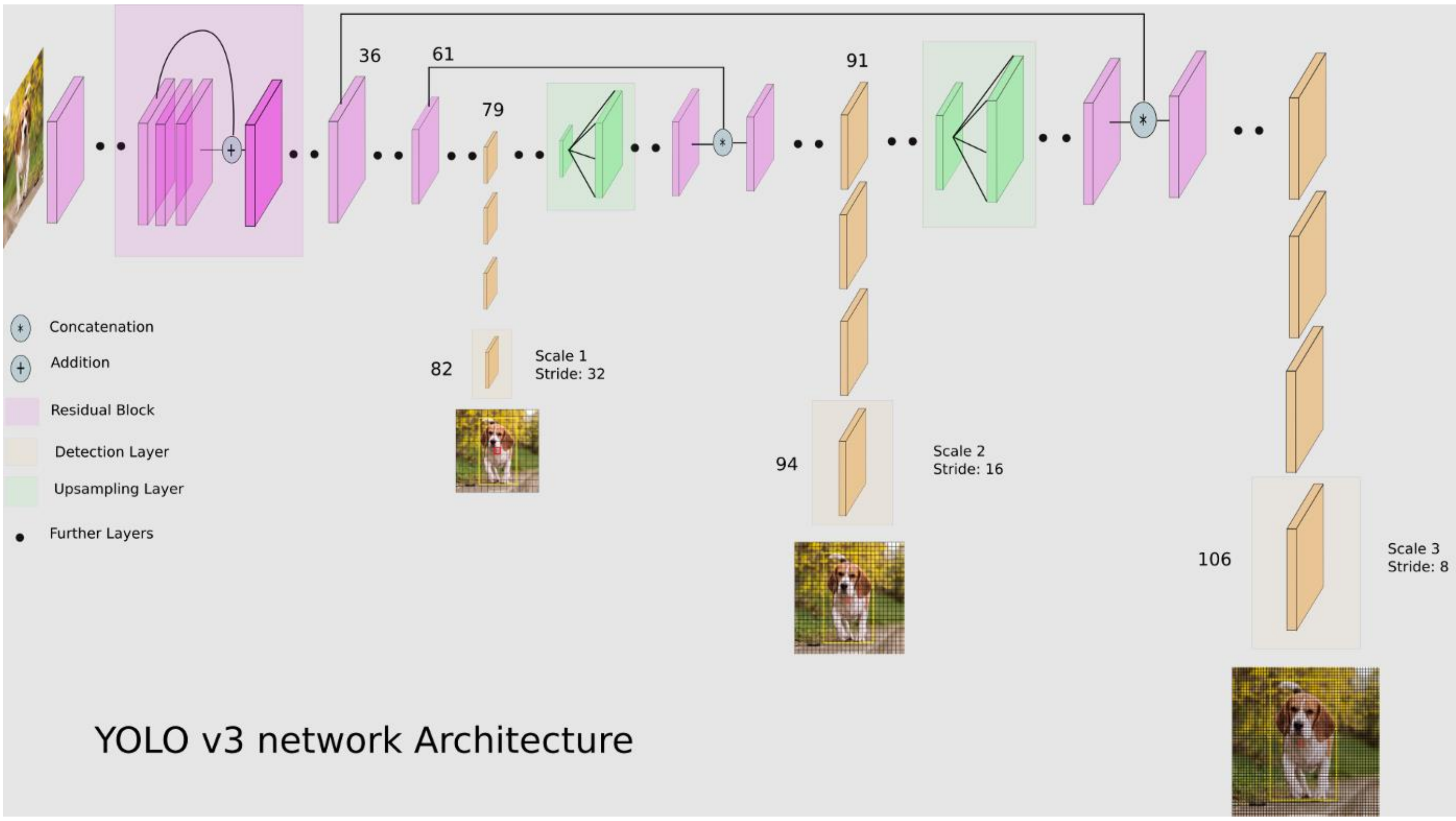
YOLO Reproduction-4

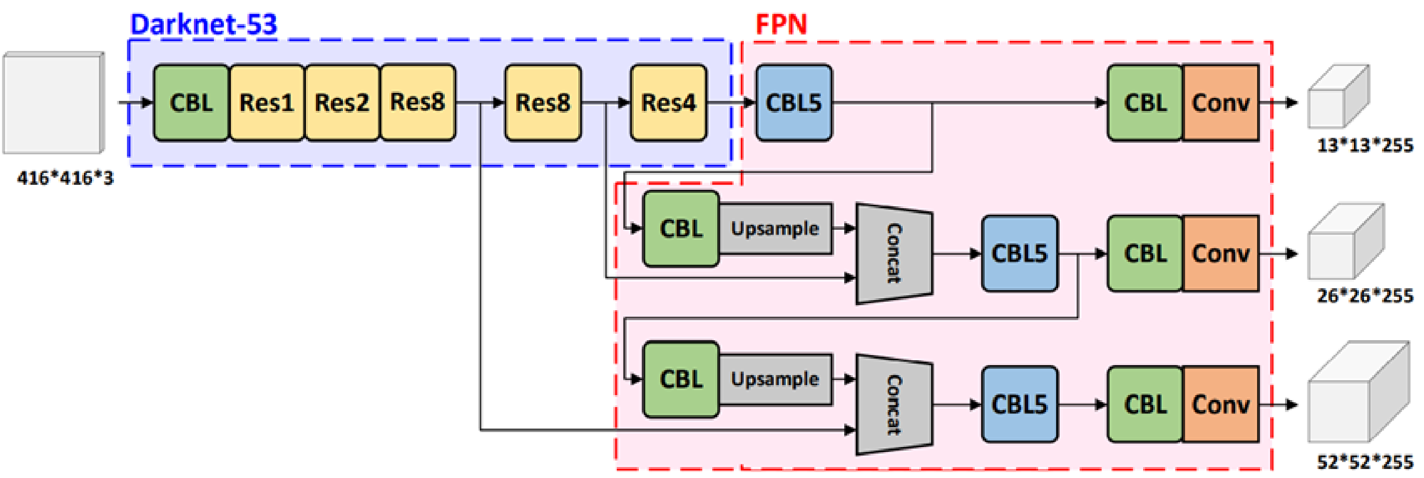
Advisor: Dr. Chih-Yu Wang
Presenter: Shao-Heng Chen
Date: July, 27, 2022

1. YOLOv3 problems

(1) currently still untrainable

(2) original YOLOv3 network architecture





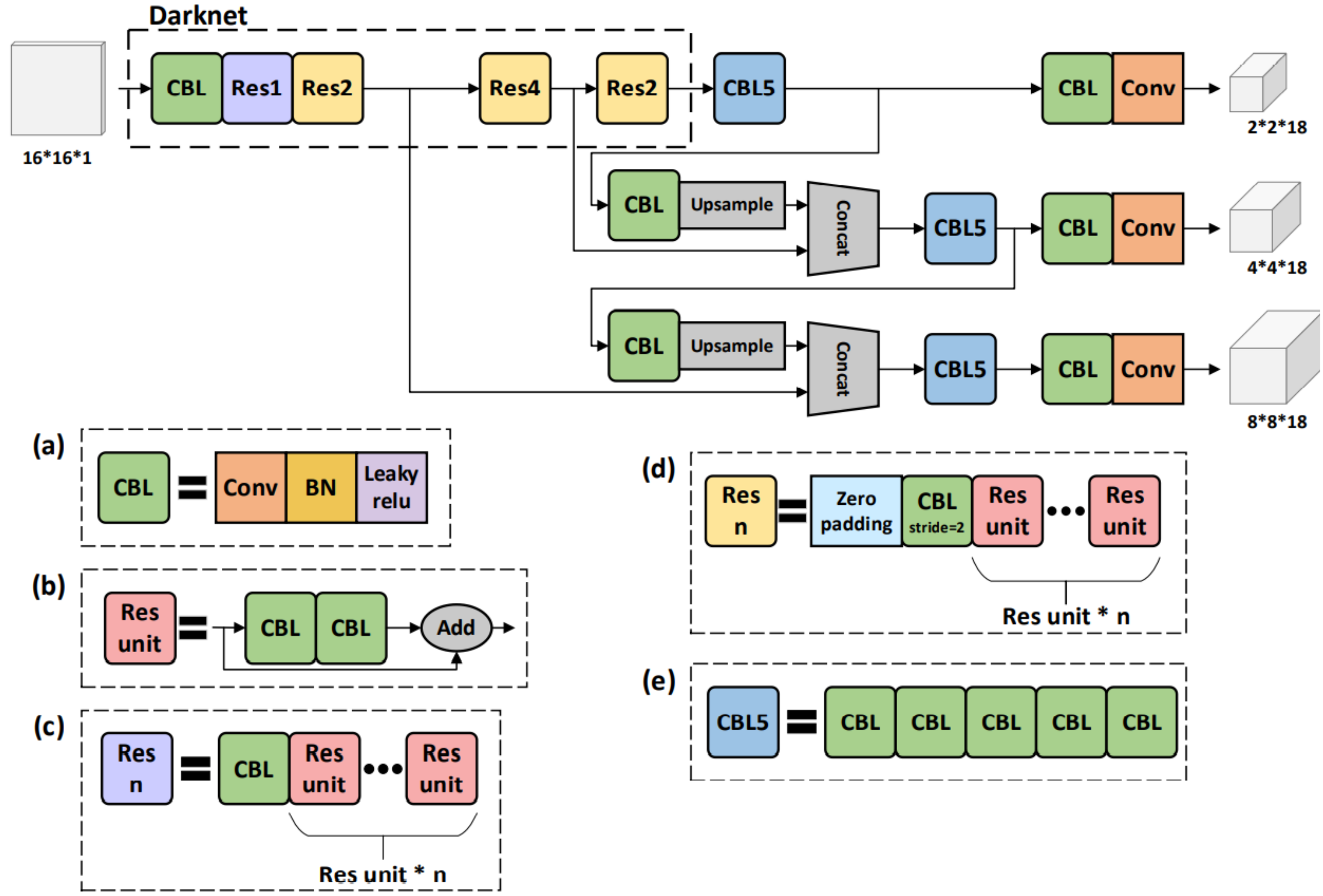
(3) YOLO-CFAR network architecture

- YOLO-CFAR vs. Keras YOLOv3 model comparison

(https://github.com/paulchen2713/YOLO_project/commit/ae46523c274b97774db01dd9af90bc8c48dc174f)

- YOLOv3-PyTorch model

(https://github.com/paulchen2713/YOLO_project/commit/05fe39a7036da9ff71c32b6f027ab93d8490379b)




```
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 18 17:04:43 2022

@author: Paul
@file: model.py
@dependencies:
    env pt3.7
    python 3.7.13
    torch >= 1.7.1
    torchvision >= 0.8.2

@references:
    Redmon, Joseph and Farhadi, Ali, YOLOv3: An Incremental Improvement, April 8, 2018.
(https://doi.org/10.48550/arXiv.1804.02767)
```

```
    Ayoosh Kathuria, Whats new in YOLO v3?, April, 23, 2018. (https://towardsdatascience.com/yolo-v3-object-
detection-53fb7d3bfe6b)
    Sanna Persson, YOLOv3 from Scratch, Mar 21, 2021. (https://sannaperzon.medium.com/yolov3-implementation-with-
training-setup-from-scratch-30ecb9751cb0)
Implementation of YOLOv3 architecture
"""
import torch
import torch.nn as nn
"""
Information about architecture config:
    Tuple is structured by (filters, kernel_size, stride)
    Every conv is a same convolution.
    List is structured by "B" indicating a residual block followed by the number of repeats
    "S" is for scale prediction block and computing the yolo loss
    "U" is for upsampling the feature map and concatenating with a previous layer
"""
config = [
    (32, 3, 1),   # (32, 3, 1) is the CBL, CBL = Conv + BN + LeakyReLU
    (64, 3, 2),
    ["B", 1],     # (64, 3, 2) + ["B", 1] is the Res1, Res1 = ZeroPadding + CBL + (CBL + CBL + Add)*1
    (128, 3, 2),
    ["B", 2],     # (128, 3, 2) + ["B", 2] is th Res2, Res2 = ZeroPadding + CBL + (CBL + CBL + Add)*2
    (256, 3, 2),
    ["B", 8],     # (256, 3, 2) + ["B", 8] is th Res8, Res8 = ZeroPadding + CBL + (CBL + CBL + Add)*8
    (512, 3, 2),
    ["B", 8],     # (512, 3, 2) + ["B", 8] is th Res8, Res8 = ZeroPadding + CBL + (CBL + CBL + Add)*8
    (1024, 3, 2),
    ["B", 4],     # (1024, 3, 2) + ["B", 4] is th Res4, Res4 = ZeroPadding + CBL + (CBL + CBL + Add)*4
    # to this point is Darknet-53 which has 52 layers
    # 52 = 1 + (1 + 1*2) + (1 + 2*2) + (1 + 8*2) + (1 + 8*2) + (1 + 4*2) ?
    (512, 1, 1),  #
    (1024, 3, 1), #
    "S",
    (256, 1, 1),
    "U",
    (256, 1, 1),
    (512, 3, 1),
    "S",
    (128, 1, 1),
    "U",
    (128, 1, 1),
    (256, 3, 1),
```

```
    "S",
    # 252 = 1 + 3 + (4+7) + (4+7*2) + (4+7*8) + (4+7*8) + (4+7*4) + 19 + 5 + 19 + 5 + 19 ?
]

config = [
    (32 // 2, 3, 1),
    (64 // 2, 3, 2),
    ["B", 1],     # (64, 3, 2) + ["B", 1] is the Res1
    (128, 3, 2),
    ["B", 2],     # (128, 3, 2) + ["B", 2] is th Res2
    # (256, 3, 2),
    # ["B", 8],     # (256, 3, 2) + ["B", 8] is th Res8
    (512, 3, 2),
    ["B", 4],     # (512, 3, 2) + ["B", 8] is th Res8
    (1024 // 2, 3, 2),
    ["B", 1],     # ["B", 4], to this point is Darknet-53, which has 53 layers?
    # 52 = 1 + (1 + 1*2) + (1 + 2*2) + (1 + 8*2) + (1 + 8*2) + (1 + 4*2) ?
    (512 // 2, 1, 1),
    (1024, 3, 1),
    "S",
    (256, 1, 1),
    "U",
    (256 // 2, 1, 1),
    (512 // 2, 3, 1),
    "S",
    (128 // 2, 1, 1), #
    "U",
    (128 // 2, 1, 1),
    (256 // 2, 3, 1),
    "S",
    # 252 = 1 + 3 + (4+7) + (4+7*2) + (4+7*8) + (4+7*8) + (4+7*4) + 19 + 5 + 19 + 5 + 19 ?
]


class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, bn_act=True, **kwargs):
        super(CNNBlock, self).__init__()
        # if we do use bn activation function in the block, then we do not want to use bias, its unnecessary
        # **kwargs will be the kernal size, the stride and padding as well
        self.conv = nn.Conv2d(in_channels, out_channels, bias=not bn_act, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels)
        self.leaky = nn.LeakyReLU(negative_slope=0.1) # default negative_slope=0.01
```

```python
        self.use_bn_act = bn_act # indicating if the block is going to use a batch norm NN activation function

    def forward(self, x):
        # using if-else statement in the forward pass might lose on some performance, negligible?
        # we use bn activation by default, except for scale prediction
        if self.use_bn_act:
            return self.leaky(self.bn(self.conv(x))) # bn_act()
        # for scale prediction we don't want to use batch norm LeakyReLU on our output, just normal Conv
        else:
            return self.conv(x)


class ResidualBlock(nn.Module):
    def __init__(self, channels, use_residual=True, num_repeats=1):
        super(ResidualBlock, self).__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_repeats): # repeat for num_repeats
            self.layers += [
                nn.Sequential(
                    CNNBlock(channels, channels // 2, kernel_size=1, padding=0), # down samples or reduces the number
of filters
                    # CNNBlock(channels // 2, channels, kernel_size=3, padding=1), # then brings it back again
                    CNNBlock(channels // 2, channels, kernel_size=3, padding=1),
                )
            ]
        # 1. why specify use_residual in a ResidualBlock? is because in some cases we are going to use skip
        # connections, in some cases we just going through the config file and build the ordinary ResidualBlock
        # 2. why we need to store these?
        self.use_residual = use_residual # indicating using residual
        self.num_repeats = num_repeats   # number of repeats set to 1 by default

    def forward(self, x):
        for layer in self.layers:
            x = layer(x) + x if self.use_residual else layer(x)
            # if self.use_residual:
            #     # x = x + layer(x)
            #     x = layer(x) + x
            # else:
            #     x = layer(x)
        return x
```

```python
class ScalePrediction(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(ScalePrediction, self).__init__()
        # for every single cell grid we have 3 anchor boxes, for every anchor box we have 1 node for each of the
classes
        # for each bounding box we have [P(Object), x, y, w, h] and that's 5 values
        self.pred = nn.Sequential(
            # CNNBlock(in_channels, 2 * in_channels, kernel_size=3, padding=1),
            CNNBlock(in_channels, 2 * in_channels, kernel_size=3, padding=1),
            CNNBlock(2 * in_channels, 3 * (num_classes + 5), bn_act=False, kernel_size=1),
        )
        self.num_classes = num_classes

    def forward(self, x):
        # we want to return the prediction of x, then we want to reshape it to the number of examples in our batch
        # split out_channel "3 * (num_classes + 5)" into two different dimensions "3, (num_classes + 5)", instead of
        # having a long vector of bounging boxes, and change the order of the dimensions
        return (
            self.pred(x)
            .reshape(x.shape[0], 3, self.num_classes + 5, x.shape[2], x.shape[3])
            .permute(0, 1, 3, 4, 2)
        )
        # [x.shape[0], 3, x.shape[2], x.shape[3], self.num_classes + 5], e.g. [N, 3, 13, 13, 5+num_classes]
        # for scale one, we have N examples in our batch, each example has 3 anchors, each anchor has 13-by-13 grid
        # and every cell has (5+num_classes) output, output dimension = N x 3 x 13 x 13 x (5+num_classes)


class YOLOv3(nn.Module):
    def __init__(self, in_channels=3, num_classes=1):
        super(YOLOv3, self).__init__()
        self.num_classes = num_classes
        self.in_channels = in_channels
        # we want to create the layers using the config file, and store them in a nn.ModuleList()
        self.layers = self._create_conv_layers() # we immediately call _create_conv_layers() to initialize the layers

    def forward(self, x):
        # need to keep track of outputs and route connections
        outputs = []           # we have one output for each scale prediction, should be 3 in total
        route_connections = [] # e.g. after upsampling, we concatenate the channels of skip connections

        for i, layer in enumerate(self.layers):
            if isinstance(layer, ScalePrediction): # if it's ScalePrediction
```

```
                outputs.append(layer(x)) # we're going to add that layer
                continue # and then continue from where we were previously, not after ScalePrediction

            # calling layer(x) is equivalent to calling layers.__call__(x), and __call__() is actually calling 
layer.forward(x)
            # which is defined in class layer(nn.Module), but in practice we should use layer(x) rather than 
layer.forward(x)
            x = layer(x) #
            print(f"layer {i}: ", x.shape)

            # skip layers are connected to ["B", 8] based on the paper, original config file
            if isinstance(layer, ResidualBlock) and layer.num_repeats != 1: #
            # if isinstance(layer, ResidualBlock) and layer.num_repeats == 8:
                route_connections.append(x)

            elif isinstance(layer, nn.Upsample): # if we use the Upsample
                # we want to concatenates with the last route connection, with the last one we added
                x = torch.cat([x, route_connections[-1]], dim=1) # why concatenate along dimension 1 for the channels
                route_connections.pop() # after concatenation, we remove the last one

        # print(f"outputs: {outputs}")
        return outputs

    # create the layers using the config files
    def _create_conv_layers(self):
        layers = nn.ModuleList()       # keep track of all the layers in a ModuleList, which supports tools like 
model.eval()
        in_channels = self.in_channels # only need to specifies the first in_channels, I suppose

        # go through and parse the config file and construct the model line by line
        for module in config:
            # if it's a tuple (filters, kernel_size, stride), e.g. (32, 3, 1), then it's just a CNNBlock
            if isinstance(module, tuple):
                out_channels, kernel_size, stride = module # we want to take out the (filters, kernel_size, stride)
                layers.append(
                    CNNBlock(
                        in_channels,
                        out_channels,
                        kernel_size=kernel_size,
                        stride=stride,
                        # padding=1 if kernel_size == 3 else 0, # if kernel_size == 1 then padding = 0
                        padding=1 if kernel_size == 3 else 0,
```

```
                )
            )
            # the in_channels for the next block is going to be the out_channels of this block
            in_channels = out_channels # update the in_channels of the next layer

        # if it's a List, e.g. ["B", 1], then it's a ResidualBlock
        elif isinstance(module, list):
            num_repeats = module[1] # we want to take out the number of repeats, which is going to be module[1]
            # and module[0] should be "B", which indicates that this is a ResidualBlock
            layers.append(ResidualBlock(in_channels, num_repeats=num_repeats,))

        # if it's a String, e.g. "S" or "U", then it might be ScalePrediction or Upsampling
        elif isinstance(module, str):
            # "S" for ScalePrediction
            if module == "S":
                layers += [
                    ResidualBlock(in_channels, use_residual=False, num_repeats=1),
                    CNNBlock(in_channels, in_channels // 2, kernel_size=1),
                    ScalePrediction(in_channels // 2, num_classes=self.num_classes),
                ]
                # after ScalePrediction, we want to continue from CNNBlock, since we have scale_factor=2
                in_channels = in_channels // 2 # we then wnat to divide in_channels by 2
            # "U" for Upsampling
            elif module == "U":
                layers.append(nn.Upsample(scale_factor=2),)
                in_channels = in_channels * 3 # 3 == 2 + 1, concatenated the channels from previously

    return layers


if __name__ == "__main__":
    # actual parameters
    num_classes = 1 # 20
    # YOLOv1: 448, YOLOv2/YOLOv3: 416 (with multi-scale training)
    IMAGE_SIZE = 16 # multiples of 32 are workable with stride [32, 16, 8]
    # stride = [8, 4, 2]
    stride = [16, 8, 4] # 16
    # stride = [32, 16, 8] # 32

    # simple test settings
    num_examples = 2
    num_channels = 3 # num_anchors
```

```
    model = YOLOv3(num_classes=num_classes) # initialize a YOLOv3 model as model
    # simple test with random inputs of 2 examples, 3 channels, and IMAGE_SIZE-by-IMAGE_SIZE input
    x = torch.randn((num_examples, num_channels, IMAGE_SIZE, IMAGE_SIZE))
    out = model(x)

    print("Output Shape: ")
    print("[num_examples, num_channels, feature_map, feature_map, num_classes + 5]")
    for i in range(num_channels):
        print(out[i].shape)

    assert out[0].shape == (2, 3, IMAGE_SIZE//stride[0], IMAGE_SIZE//stride[0], num_classes + 5) # [2, 3, 13, 13,
num_classes + 5]
    assert out[1].shape == (2, 3, IMAGE_SIZE//stride[1], IMAGE_SIZE//stride[1], num_classes + 5) # [2, 3, 26, 26,
num_classes + 5]
    assert out[2].shape == (2, 3, IMAGE_SIZE//stride[2], IMAGE_SIZE//stride[2], num_classes + 5) # [2, 3, 52, 52,
num_classes + 5]
    print("Success!")

# layer 0:  torch.Size([2, 16, 16, 16])
# layer 1:  torch.Size([2, 32, 8, 8])
# layer 2:  torch.Size([2, 32, 8, 8])
# layer 3:  torch.Size([2, 128, 4, 4])
# layer 4:  torch.Size([2, 128, 4, 4])
# layer 5:  torch.Size([2, 512, 2, 2])
# layer 6:  torch.Size([2, 512, 2, 2])
# layer 7:  torch.Size([2, 512, 1, 1])
# layer 8:  torch.Size([2, 512, 1, 1])
# layer 9:  torch.Size([2, 256, 1, 1])
# layer 10:  torch.Size([2, 1024, 1, 1])
# layer 11:  torch.Size([2, 1024, 1, 1])
# layer 12:  torch.Size([2, 512, 1, 1])
# layer 14:  torch.Size([2, 256, 1, 1])
# layer 15:  torch.Size([2, 256, 2, 2])
# layer 16:  torch.Size([2, 128, 2, 2])
# layer 17:  torch.Size([2, 256, 2, 2])
# layer 18:  torch.Size([2, 256, 2, 2])
# layer 19:  torch.Size([2, 128, 2, 2])
# layer 21:  torch.Size([2, 64, 2, 2])
# layer 22:  torch.Size([2, 64, 4, 4])
# layer 23:  torch.Size([2, 64, 4, 4])
# layer 24:  torch.Size([2, 128, 4, 4])
```

```
# layer 25:  torch.Size([2, 128, 4, 4])
# layer 26:  torch.Size([2, 64, 4, 4])
# Output Shape:
# [num_examples, num_channels, feature_map, feature_map, num_classes + 5]
# torch.Size([2, 3, 1, 1, 6])
# torch.Size([2, 3, 2, 2, 6])
# torch.Size([2, 3, 4, 4, 6])
# Success!
# layer 0:  torch.Size([2, 32, 416, 416])
# layer 1:  torch.Size([2, 64, 208, 208])
# layer 2:  torch.Size([2, 64, 208, 208])
# layer 3:  torch.Size([2, 128, 104, 104])
# layer 4:  torch.Size([2, 128, 104, 104])
# layer 5:  torch.Size([2, 256, 52, 52])
# layer 6:  torch.Size([2, 256, 52, 52])
# layer 7:  torch.Size([2, 512, 26, 26])
# layer 8:  torch.Size([2, 512, 26, 26])
# layer 9:  torch.Size([2, 1024, 13, 13])
# layer 10:  torch.Size([2, 1024, 13, 13])
# layer 11:  torch.Size([2, 512, 13, 13])
# layer 12:  torch.Size([2, 1024, 13, 13])
# layer 13:  torch.Size([2, 1024, 13, 13])
# layer 14:  torch.Size([2, 512, 13, 13])
# layer 16:  torch.Size([2, 256, 13, 13])
# layer 17:  torch.Size([2, 256, 26, 26])
# layer 18:  torch.Size([2, 256, 26, 26])
# layer 19:  torch.Size([2, 512, 26, 26])
# layer 20:  torch.Size([2, 512, 26, 26])
# layer 21:  torch.Size([2, 256, 26, 26])
# layer 23:  torch.Size([2, 128, 26, 26])
# layer 24:  torch.Size([2, 128, 52, 52])
# layer 25:  torch.Size([2, 128, 52, 52])
# layer 26:  torch.Size([2, 256, 52, 52])
# layer 27:  torch.Size([2, 256, 52, 52])
# layer 28:  torch.Size([2, 128, 52, 52])
# Output Shape:
# [num_examples, num_channels, feature_map, feature_map, num_classes + 5]
# torch.Size([2, 3, 13, 13, 6])
# torch.Size([2, 3, 26, 26, 6])
# torch.Size([2, 3, 52, 52, 6])
# Success!
```