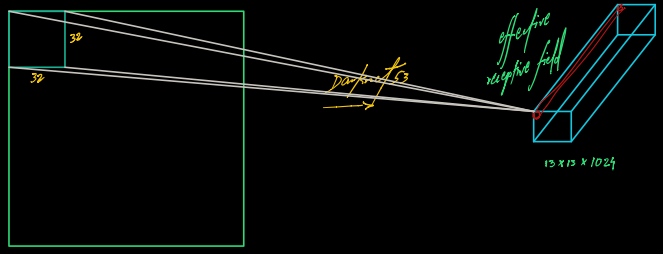
→ interesting result of YOLO v3

If there's only one object, that "Pc" ↓ probability of having an object of interest will be zero or close to zero

YOLO is designed to work on square images but we can always resize the input to be a square



what if the object spans multiple receptive fields?

↓

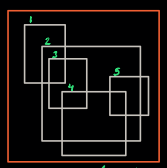
the center of BB has to inside receptive field, the height/width can be anything

↓

anchor boxes solve some of the issue

first they tried to predict (bx, by, bw, bh) for a bounding box → Turned out to be too hard to train.

solution → have anchor boxes.



5 anchor boxes

let's represent bb as a modification of anchor boxes

↳ (cx, cy, pw, ph)

The increased performance by 5% or 4% by using anchor boxes.

range → 0 to ∞ → it'll be utmost one priority away. They want bb to be very close to anchor boxes

sigmoid function

σ(tx) + cx

σ(ty) + cy

(bx, by, bw, bh)

$p_w \cdot e^{t_w}$

$p_h \cdot e^{t_h}$

→ kmeans clustering on train data's bounding boxes

k = 5 anchor boxes
↓
5 boxes in which there's a high likelihood of finding an object.
↓
this can act as "prior" info to find objects more successfully.

Darknet weights are used for init only. loss are propagated to all the layers.

let's work with objectness score (P_0)
prob. of finding an object in this box.

beauty of deep learning & → loss fn design
↗ model learns all this implicitly since we penalize the model accordingly

t_x	t_y	t_w	t_h	P_0	P_1	P_2	. . .	P_c

↳ if class = woman
↳ if class = person
} → if we do find a woman, do we put her in person or woman class. → softmax will make the nw only one class ↳ softmax doesn't work here.

so we use squared loss

for every class we multiply P_0 with P_i →
↓
general prob. to finding

loss function.
if we keep $\beta = 3$

13 × 13 × 255

∴ 13 × 13 × 3 possible boxes

they found: ↙ it's a hyp. parameter.
$\lambda_{coord} = 5$
$\lambda_{noobj} = 0.5$ works well

that's why the λ values for weight

∴ Σ [① λ_{coord} * squared loss on t_w, t_h, t_x & t_y (some constant)
② + log. loss for P_0 (since its binary)
③ + log. loss for each P_i (↳ since its logistic regression.)]

→ range will be more. # imbalance in loss

↓ for each bounding box containing an object in ground truth.

+ [Σ (no object boxes in ground truth) $\lambda_{no.obj}$ · log loss for P_0] → 0 to 1

→ over here we don't care about anything else. So we just want P_0 to be close to zero or zero.

We use pretrained Darknet but we're not freezing the weights.

logloss as it's a probability
squared loss because real value.

Multi-Scale Prediction

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Darknet-53 model

over here numbers are different because input dimensions are different.

the photo is for layer architecture reference

more resolution, better smaller objects.

52 × 52 × 256 → NN1 → 52 × 52 × 3 bb per cell

26 × 26 × 512 → NN2 → 26 × 26 × 3 bb per cell

13 × 13 × 1024 → NN3 → 13 × 13 × 3 bb per cell

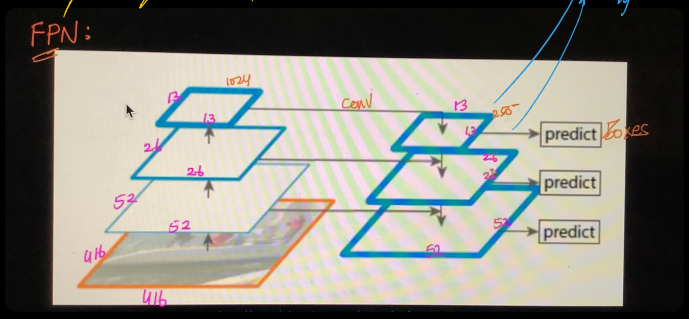
each NN works with diff layer outputs.

we want to use info from the last. We want NN1 to use info from NN3

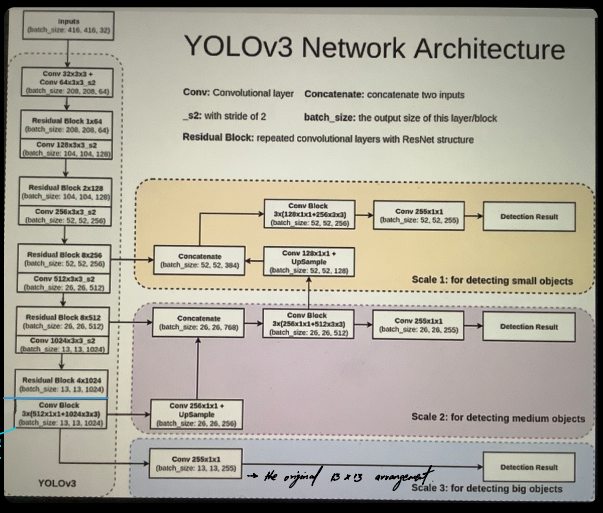
10,647 boxes

Feature Pyramid Network:

FPN:

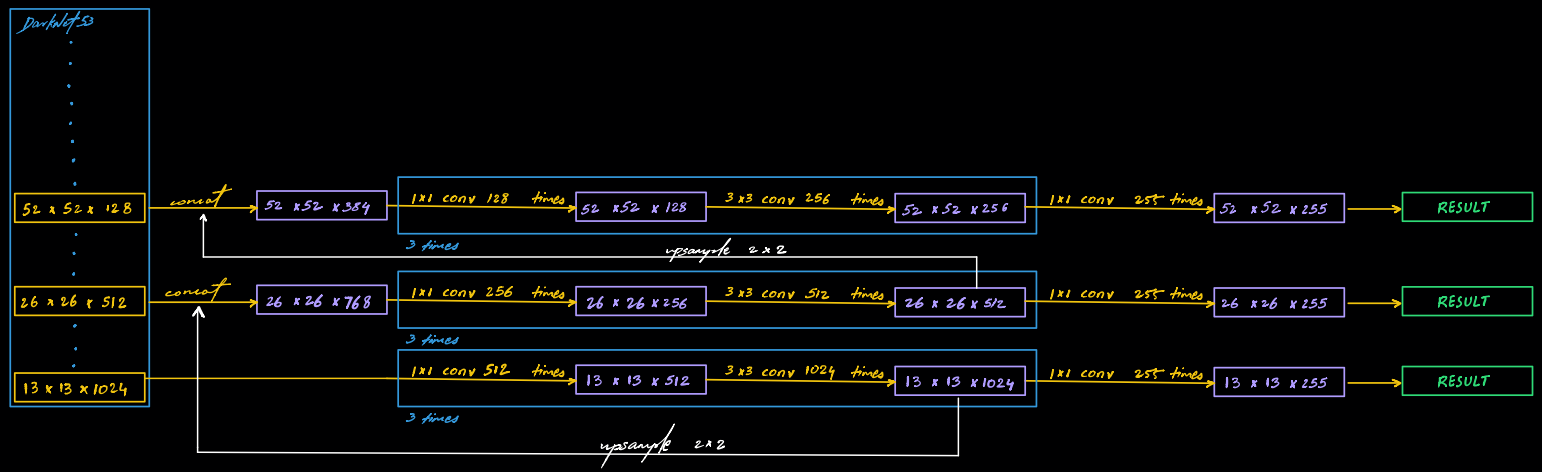


each one uses info from before → more efficient.



output of Darknet also has 3x3 to incorporate info from neighbouring cell

→ the original 13 x 13 arrangement



People know the Darknet part well, then they fail on the remaining part.

Code taken from Jason Brownlee's blog

code is pretty obvious, it's Keras. so I'm not noting much

Just for reference

	Type	Filters	Size
	Convolutional	32	3 × 3
	Convolutional	64	3 × 3 / 2
1×	Convolutional	32	1 × 1
	Convolutional	64	3 × 3
	Residual		
	Convolutional	128	3 × 3 / 2
2×	Convolutional	64	1 × 1
	Convolutional	128	3 × 3
	Residual		
	Convolutional	256	3 × 3 / 2
8×	Convolutional	128	1 × 1
	Convolutional	256	3 × 3
	Residual		
	Convolutional	512	3 × 3 / 2
8×	Convolutional	256	1 × 1
	Convolutional	512	3 × 3
	Residual		
	Convolutional	1024	3 × 3 / 2
4×	Convolutional	512	1 × 1
	Convolutional	1024	3 × 3
	Residual		
	Avgpool		Global
	Connected		1000
	Softmax		

Darknet-53 model

```python
def make_yolov3_model():
    input_image = Input(shape=(None, None, 3))
    # Layer  0 => 4
    x = _conv_block(input_image, [{'filter': 32, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 0},
                                  {'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 1},
                                  {'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 2},
                                  {'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 3}])
    # Layer  5 => 8
    x = _conv_block(x, [{'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 5},
                        {'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 6},
                        {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 7}])
    # Layer  9 => 11
    x = _conv_block(x, [{'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 9},
                        {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 10}])
    # Layer 12 => 15
    x = _conv_block(x, [{'filter': 256, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 12},
                        {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 13},
                        {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 14}])
    # Layer 16 => 36
    for i in range(7):
        x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 16+i*3},
                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 17+i*3}])
    skip_36 = x
    # Layer 37 => 40
    x = _conv_block(x, [{'filter': 512, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 37},
                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 38},
                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 39}])
    # Layer 41 => 61
    for i in range(7):
        x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 41+i*3},
                            {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 42+i*3}])
    skip_61 = x
    # Layer 62 => 65
    x = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 62},
                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 63},
                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 64}])
    # Layer 66 => 74
    for i in range(3):
        x = _conv_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 66+i*3},
                            {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 67+i*3}])
    # Layer 75 => 79
    x = _conv_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 75},
                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 76},
                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 77},
                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 78},
                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 79}], skip=False)
    # Layer 80 => 82
    yolo_82 = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 80},
                              {'filter':  255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer_idx': 81}], skip=False)
    # Layer 83 => 86
    x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 84}], skip=False)
    x = UpSampling2D(2)(x)
    x = concatenate([x, skip_61])
    # Layer 87 => 91
    x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 87},
                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 88},
                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 89},
                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 90},
                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 91}], skip=False)
    # Layer 92 => 94
    yolo_94 = _conv_block(x, [{'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 92},
                              {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer_idx': 93}], skip=False)
    # Layer 95 => 98
    x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,   'layer_idx': 96}], skip=False)
    x = UpSampling2D(2)(x)
    x = concatenate([x, skip_36])
    # Layer 99 => 106
    yolo_106 = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 99},
                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 100},
                               {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 101},
                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 102},
                               {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 103},
                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 104},
                               {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer_idx': 105}], skip=False)
    model = Model(input_image, [yolo_82, yolo_94, yolo_106])
    return model
```

Excellent example of how useful Keras functional API is

layer_idx 4 is a skip connection. we add it later.

not 8 because already written once above

saving for FPN concat

13×13×1024 at this point

13×13 result done

26×26 result

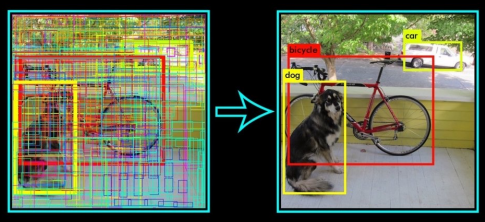
52×52 result

Excellent example of how useful Keras functional API is

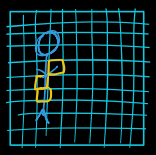
we still need to find more hacks to make this work.

out of the 1064 boxes → only take $\text{score}_i = (p_o * p_i) \geq 0.5$

Non max suppression ⇒ after filtering for $p_c \geq 0.5$, we still have a fuck ton of boxes left.

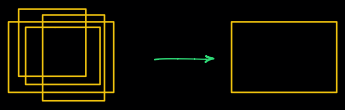


first delete every box with $p_c \leq 0.6$. Just to filter..



each yellow box thinks it got the midpoint of this person, but the person has only one midpoint.

Andrew Ng's video on NMS is better for this one



① Pick the box with highest to highest confidence. Lets call it B

② For every other box, compute its IOU with B

③ Delete the ones that have a high overlap with B ($IOU \geq 0.5$)

in this way we're deleting the non-maximal boxes. Thus the name NMS

depends on how densely packed the objects are in the image.

YOLO v3 uses intelligence & engineering, Google just throws big compute on problems. This is why YOLO v3 is so interesting.

PJ Reddie: Blog has all documentation for the C version of YOLO v3, the faster version.

the documentation is really good, I'm not going to note down any of it. → and pretty obvious

Interesting → They have Tiny Yolo v3 for low powered devices.

conv2DTranspose has more params vs upsampling. ∴ We choose upsampling