Lecture 10: Computer Vision

Data C182 (Fall 2024). Week 06. Tuesday Oct 1st, 2024

Speaker: Eric Kim

Announcements

- Weekly Course Survey out (Week 06) [Gradescope link]
  - Due: Friday Oct 4th, 2024 11:59 PM PST
  - For more info, see: [link]
- HW02 ("RNNs") will be released within the next ~1 week

Announcements

- Reminder: Midterm is coming up!
  - Tuesday, October 22th 2024, 6:30 PM - 8 PM.
  - Location: Still TBD, but very likely 50% of class in 10 Evans, remainder in another building on campus.
  - If you're unable to make this time, please contact us ASAP (make a private Ed post)
  - Midterm will cover everything from:
    - Lectures, discussions, homework assignments
  - In-person, paper + pencil exam.

HW01 update

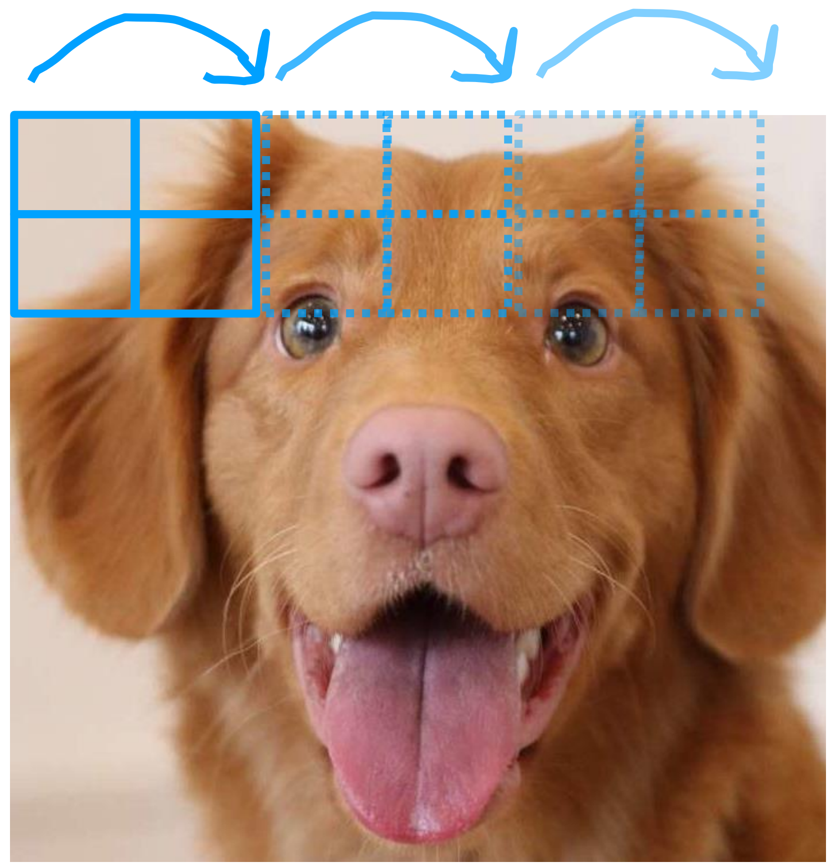
- Q4 backwards() questions are now optional (extra credit)
  - conv_backward_naive(), max_pool_backward_naive(), spatial_batchnorm_backward()
- Due date unchanged: **Tues Oct 8th, 11:59 PM PST**
  - **1 week away!**
- For more info, see Ed post: [link]

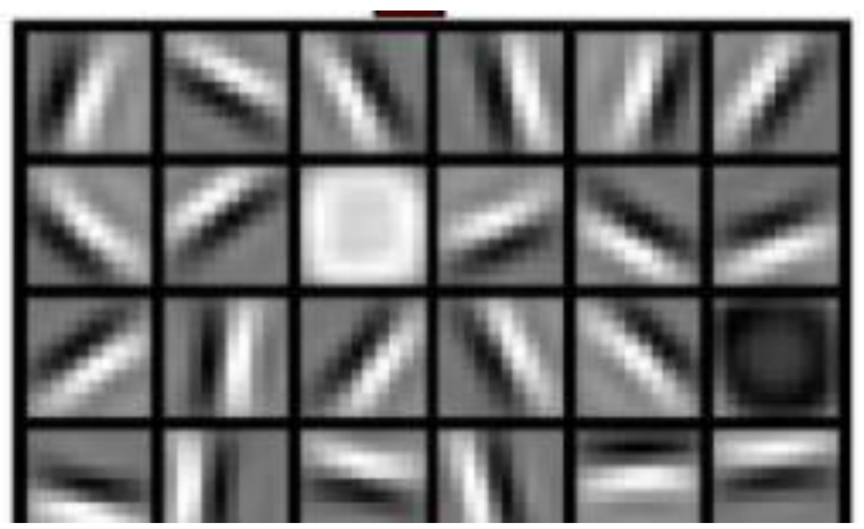
Today's lecture

- Deeper dive into Conv2d
- Computer Vision
  - Classification
  - Object detection

Convolution (2D)

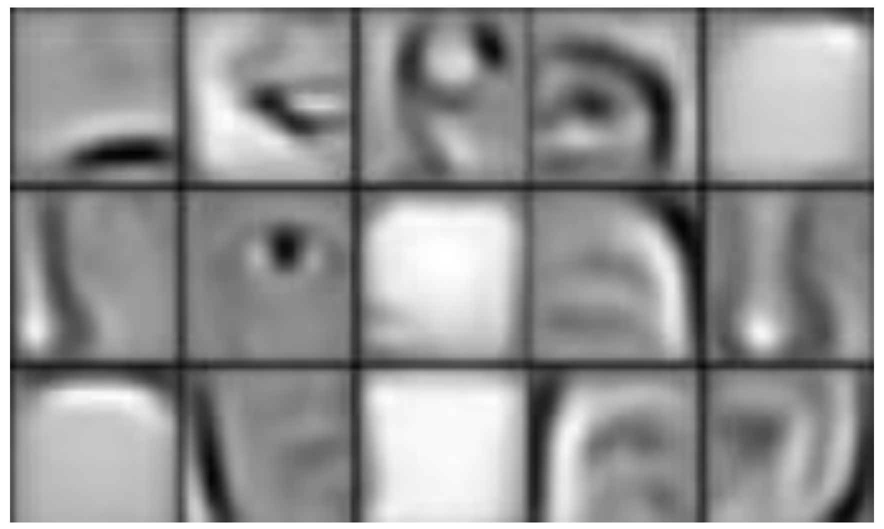
- Recall: in convolutional neural networks (CNN), a convolutional layer operates as:
- **Input**: (Spatial) feature map with shape=[num_channels, height, width]
  - Ex: input RGB color image, [3, img_height, img_width]
  - Ex: intermediate conv feature maps: [num_filters, feat_height, feat_width]
- **Output**: activation feature map, with shape [num_filters, height_out, width_out]
- **Interpretation**: output feature map is the result of sweeping a learned filter(s) over the image





Filters @ Layer 1:

edge detectors?



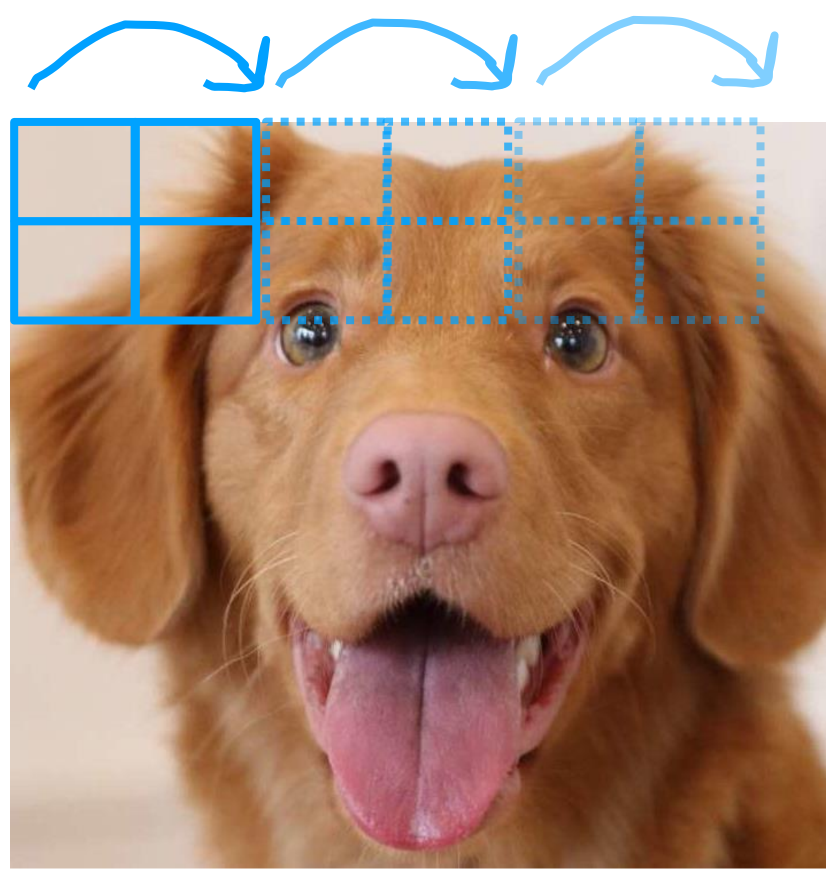
Filters @ Layer 2:

ears? noses?

Intuition: The deeper you get in the ConvNet, the more "semantic" the learned filters become

Conv2d parameters

- Parameters:
  - Filters W: shape=[num_filters, height_filter, width_filter]
  - Bias: shape=[num_filters]
  - Padding: in practice, often used to ensure that output activation feature map has same spatial shape as input
  - Stride: way to make output feature map smaller (for computation purposes)



Ex: this is a 2x2 filter, with padding=0 and stride=2.

Aside: images as tensors

In vision models (eg pytorch), RGB (three channel) images are often represented in uint8 format (ints from [0,255], 0 is black and 255 is white), with shape=[3, img_height, img_width], and in R-G-B channel order (NOT BGR):

```
from PIL import Image
import torchvision
imagepath = "UCBerkeleyCampus-scaled.jpg"
with open(imagepath, "rb") as fh:
    pil_image = Image.open(fh)
    image_tensor = torchvision.transforms.functional.pil_to_tensor(pil_image)
print(f"{type(image_tensor) = }, {image_tensor.dtype = }, {image_tensor.shape = }")
print(f"Red channel: image_tensor[0, :, :]: {image_tensor[0, :, :]}")
print(f"Green channel: image_tensor[1, :, :]: {image_tensor[1, :, :]}")
print(f"Blue channel: image_tensor[2, :, :]: {image_tensor[2, :, :]}")
```

```
type(image_tensor) = <class 'torch.Tensor'>, image_tensor.dtype = torch.uint8, image_tensor.shape = torch.Size([3, 1665, 2560])
Red channel: image_tensor[0, :, :]: tensor([[ 13,  13,  13,  ..., 141, 137, 133],
        [ 13,  13,  13,  ..., 144, 141, 139],
        [ 12,  13,  14,  ..., 144, 144, 142],
        ...,
        [ 68,  72,  87,  ...,  59,  72,  93],
        [ 78,  82,  96,  ...,  72,  73,  72],
        [ 90, 100, 114,  ...,  74,  62,  62]], dtype=torch.uint8)
Green channel: image_tensor[1, :, :]: tensor([[ 39,  39,  39,  ..., 166, 163, 159],
        [ 39,  39,  39,  ..., 169, 167, 165],
        [ 38,  39,  40,  ..., 169, 170, 168],
        ...,
...
```

Conv2d: visualized

Example: suppose we have a 1-Channel input image with img_height=img_width=3 pixels.

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Suppose we create a **Conv2D layer** with the following parameters:
filter_size: height=2, width=2, **padding**=1, **stride**=1
Num output filters: 2
Num input channels: 1 (let's only consider the Red channel for now)

Then, our layer parameters will look like:
Filters: shape=[num_filters=2, num_input_channels=1, height_filter=2, width_filter=2]
Bias: shape=[num_filters=2]

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

The **output activation map shape** will be: [num_filters=2, height_out=5, width_out=5] where:

```
height_out = 1 + (input_height + 2 * pad - filter_height) // stride
width_out = 1 + (input_width + 2 * pad - filter_width) // stride
```

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[0, 0, 0] = 1 * 0 + 2 * 0 + 0 * 0 + 0.5 * 1 = 0.5

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[0, 0, 1] = 1 * 0 + 2 * 0 + 0 * 1 + 0.5 * 128 = 64

```
output[0, :, :]:
```

$$\begin{bmatrix} 0.5 & 64 & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[0, 1, 0] = 1 * 0 + 2 * 1 + 0 * 0 + 0.5 * 4 = 4

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & 64 & & \\ 4 & & & \\ & & & \\ & & & \end{bmatrix}$$

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[0, 1, 1] = 1 * 1 + 2 * 128 + 0 * 4 + 0.5 * 100 = 307

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & 64 \\ 4 & 307 \end{bmatrix}$$

...and so on

Conv2d: computation

Let's move onto the second filter!

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[1, 0, 0] = 0 * 0 + 0.1 * 0 + 1 * 0 + 2 * 1 = 1

```
output[1, :, :]:
```

$$\begin{bmatrix} 1 & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Move onto 2nd output filter!

```
output[0, :, :]:
```

$$\begin{bmatrix} 0.5 & 64 & & \\ 4 & 307 & & \\ & & & \\ & & & \end{bmatrix}$$

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[1, 0, 1] = 0 * 0 + 0.1 * 0 + 1 * 1 + 2 * 128 = 257

```
output[1, :, :]:
```

$$\begin{bmatrix} 1 & 257 & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Move onto 2nd output filter!

```
output[0, :, :]:
```

$$\begin{bmatrix} 0.5 & 64 & & \\ 4 & 307 & & \\ & & & \\ & & & \end{bmatrix}$$

Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} & & & & & \\ & 1 & 128 & 100 & 255 & \\ & 4 & 100 & 80 & 0 & \\ & 20 & 90 & 70 & 0 & \\ & 40 & 50 & 60 & 2 & \\ & & & & & \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[1, 1, 0] = 0 * 0 + 0.1 * 1 + 1 * 0 + 2 * 4 = 8.1

```
output[1, :, :]:
```

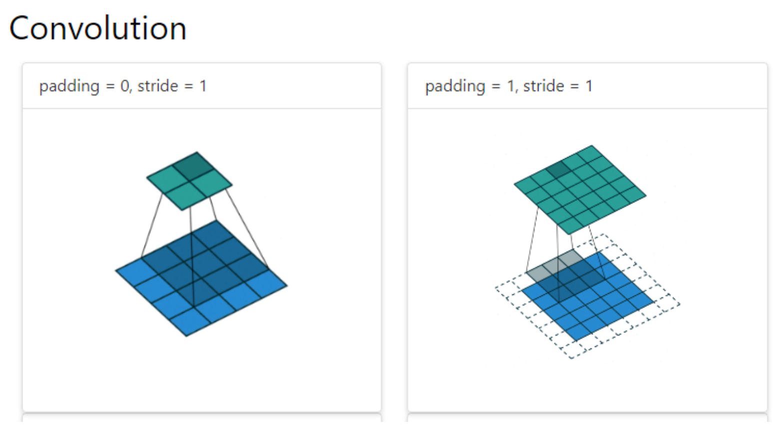
$$\begin{bmatrix} 1 & 257 & & \\ 8.1 & & & \\ & & & \\ & & & \end{bmatrix}$$

Move onto 2nd output filter!

```
output[0, :, :]:
```

$$\begin{bmatrix} 0.5 & 64 & & \\ 4 & 307 & & \\ & & & \\ & & & \end{bmatrix}$$

Tip: for a neat animation that visualizes convolution layers, see: [link]



Conv2d: computation

Convolution (aka "cross correlation"): **sliding window** computation

(Red channel only)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 100 & 80 & 0 \\ 20 & 90 & 70 & 0 \\ 40 & 50 & 60 & 2 \end{bmatrix}$$

```
>>> filters[0, 0, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

```
>>> filters[1, 0, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix}$$

output[1, 1, 1] = 0 * 1 + 0.1 * 128 + 1 * 4 + 2 * 100 = 216.8

`output[1, :, :]:`

$$\begin{bmatrix} 1 & 257 \\ 8.1 & 216.8 \end{bmatrix}$$

Move onto 2nd output filter!

...and so on

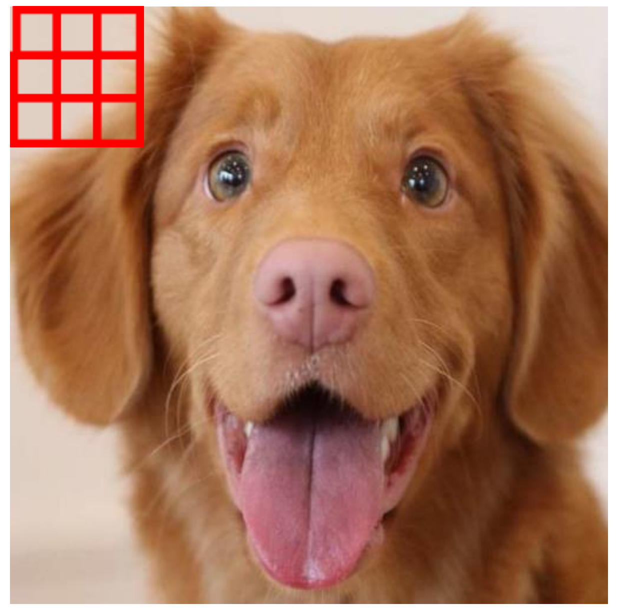
`output[0, :, :]:`

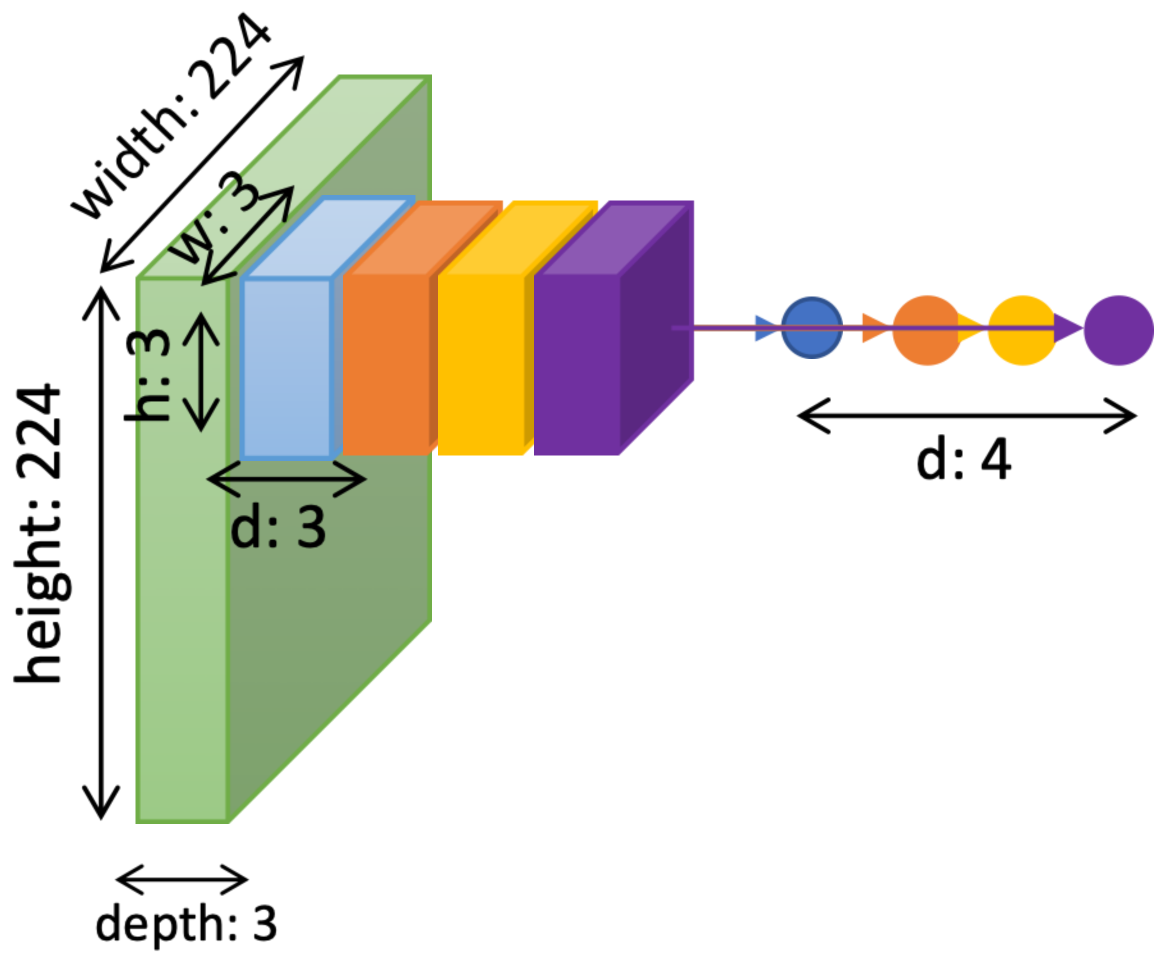
$$\begin{bmatrix} 0.5 & 64 \\ 4 & 307 \end{bmatrix}$$

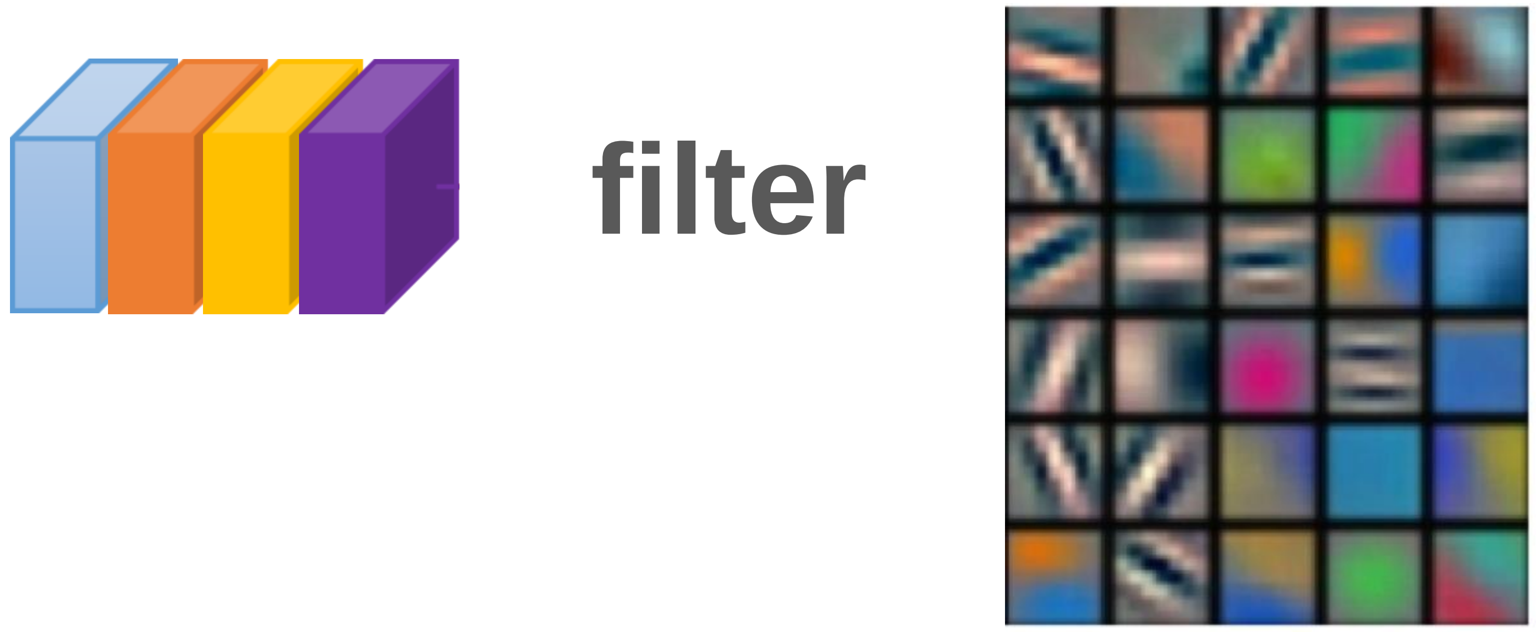
Conv2d: inputs with multiple channels

When the input to conv2d has more than 1 channel, each filter's "depth" matches the input's num_channels

Ex: when input image to conv2d has 3 channels ("RGB"), then each conv2d filter has shape=[3, filter_height, filter_width]. Computation is still element-wise multiplications between input feature map and filter.



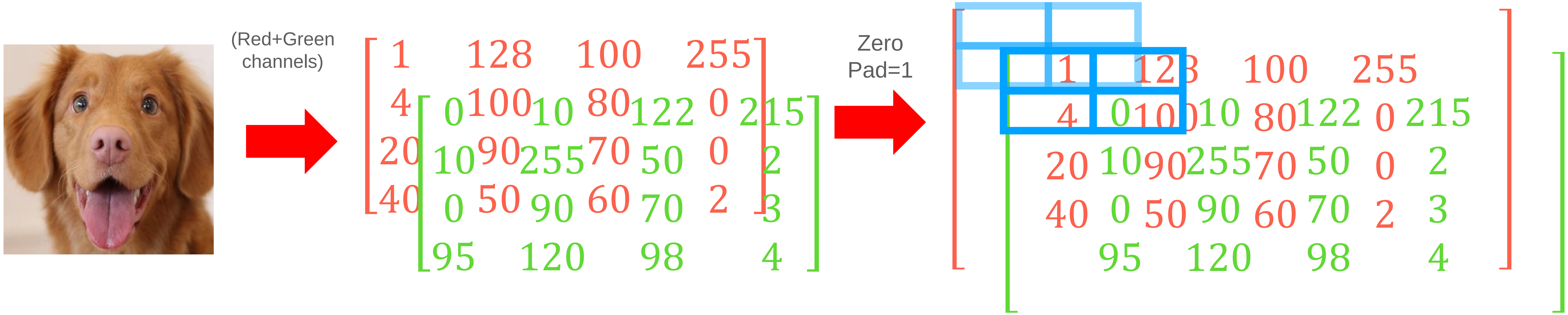




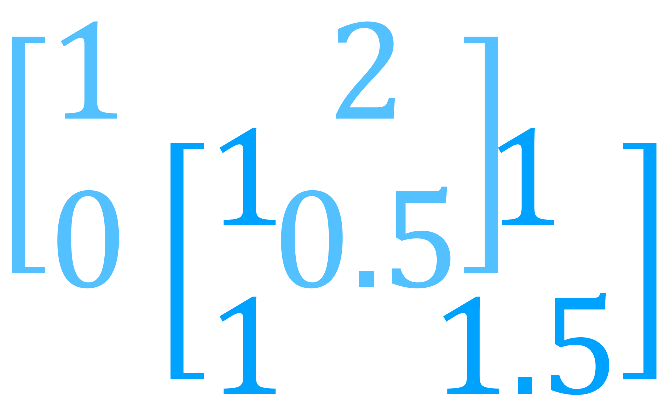
Pictured: conv2d with filter_size=3x3, num_filters=4

Conv2d: computation (mult channels)

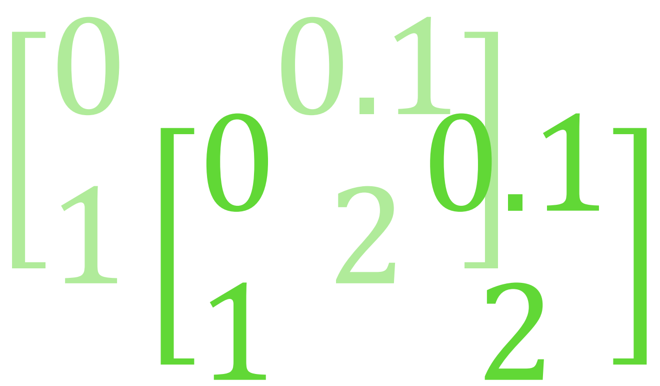
Convolution (aka "cross correlation"): **sliding window** computation




```
>>> filters[0, :, :, :]
```



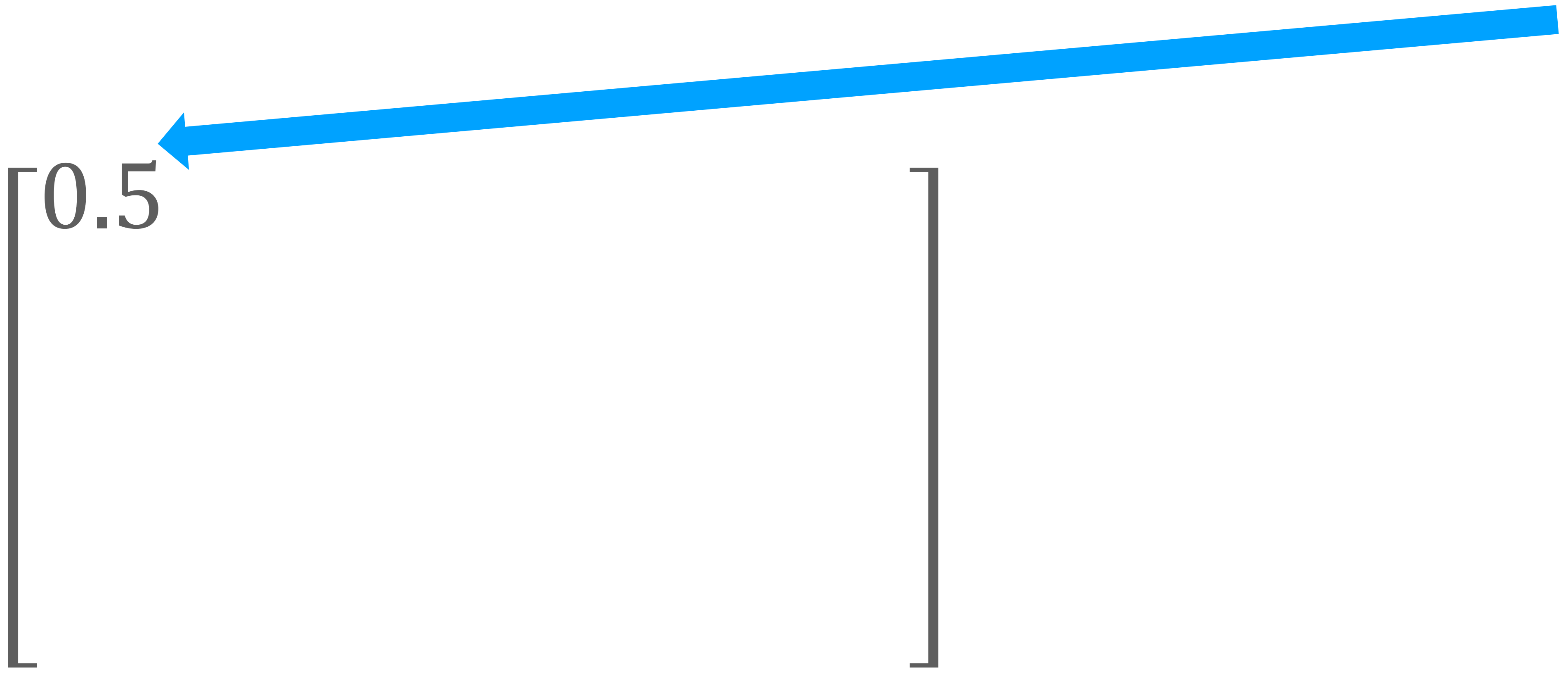
```
>>> filters[1, :, :, :]
```



\# starting from the Red channel first

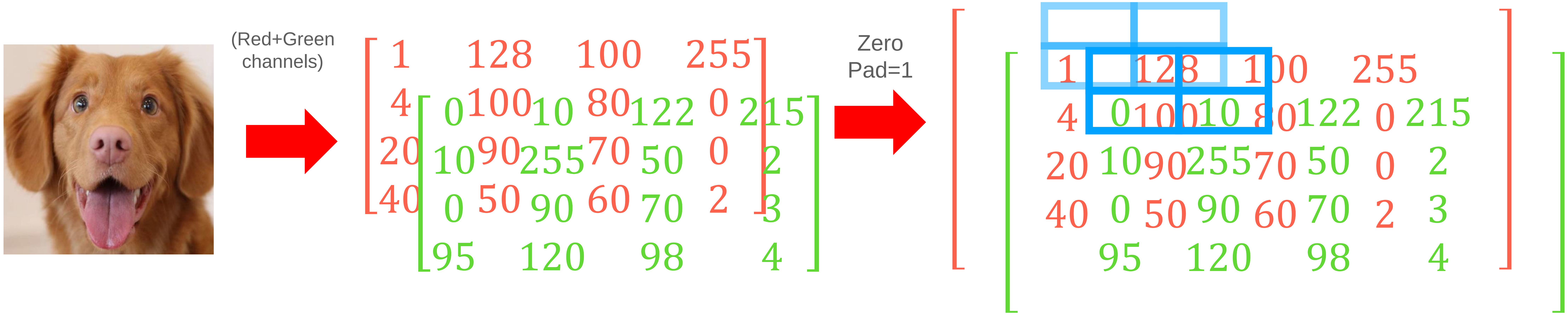
output[0, 0, 0] = 1 * 0 + 2 * 0 + 0 * 0 + 0.5 * 1 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 0 = 0.5

```
output[0, :, :]:
```

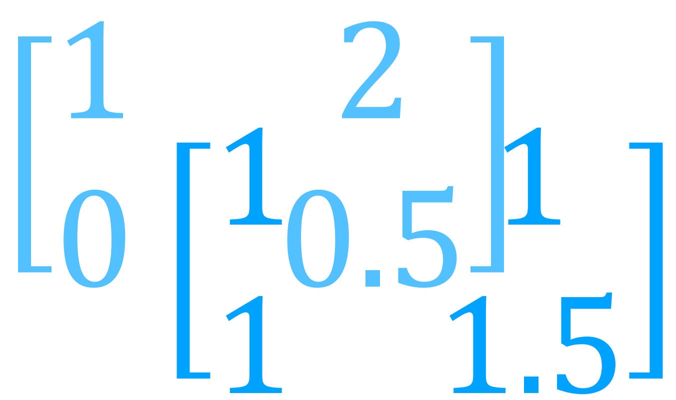


Conv2d: computation (mult channels)

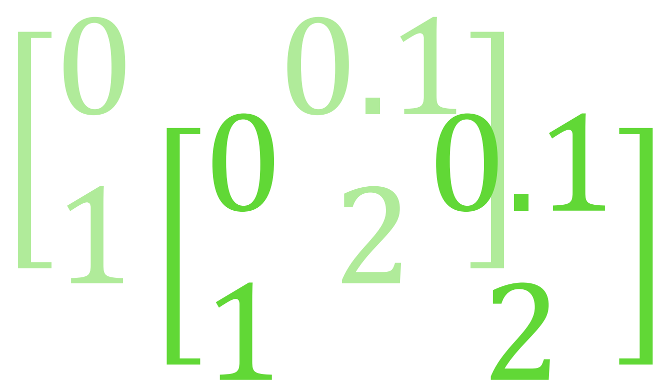
Convolution (aka "cross correlation"): **sliding window** computation




```
>>> filters[0, :, :, :]
```



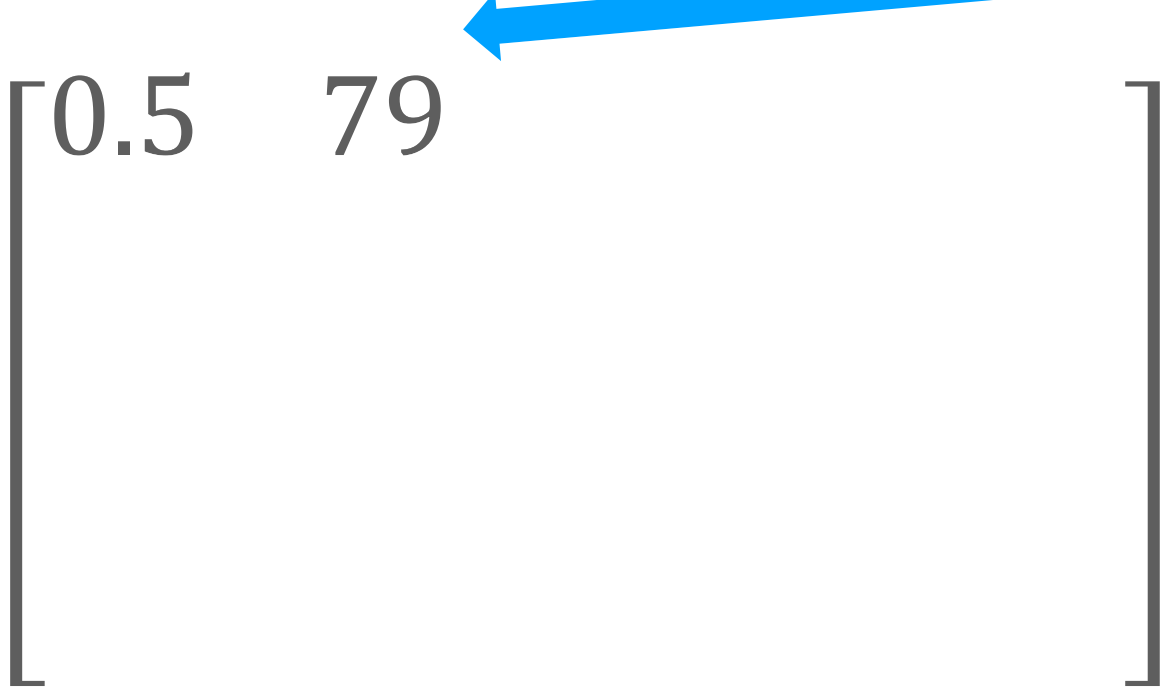
```
>>> filters[1, :, :, :]
```



starting from the Red channel first

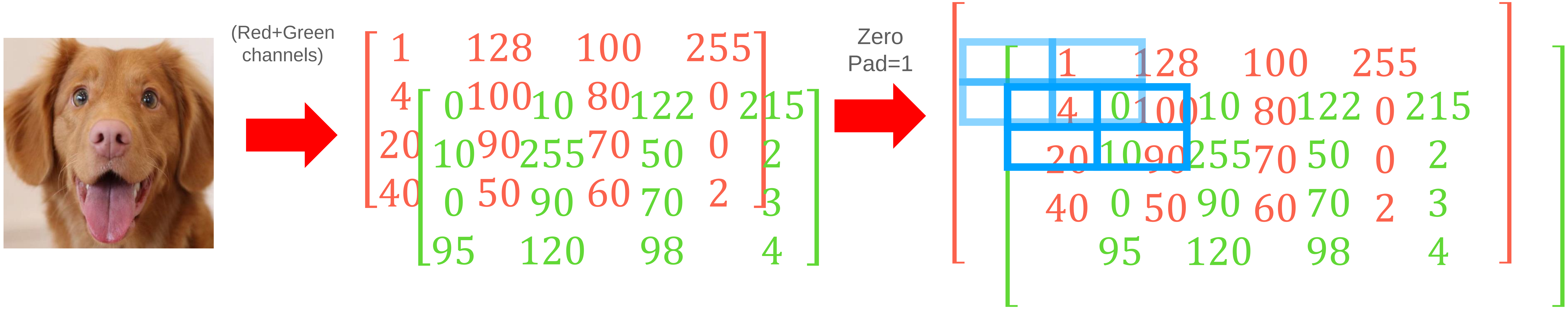
output[0, 0, 1] = 1 * 0 + 2 * 0 + 0 * 1 + 0.5 * 128 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 10 = 79

```
output[0, :, :]:
```



Conv2d: computation (mult channels)

Convolution (aka "cross correlation"): **sliding window** computation




```
>>> filters[0, :, :, :]
```

$$\begin{bmatrix}1 & 2\\ 0 & 0.5\end{bmatrix} \quad \begin{bmatrix}1 & 1\\ 1 & 1.5\end{bmatrix}$$

```
>>> filters[1, :, :, :]
```

$$\begin{bmatrix}0 & 0.1\\ 1 & 2\end{bmatrix} \quad \begin{bmatrix}0 & 0.1\\ 1 & 2\end{bmatrix}$$

starting from the Red channel first

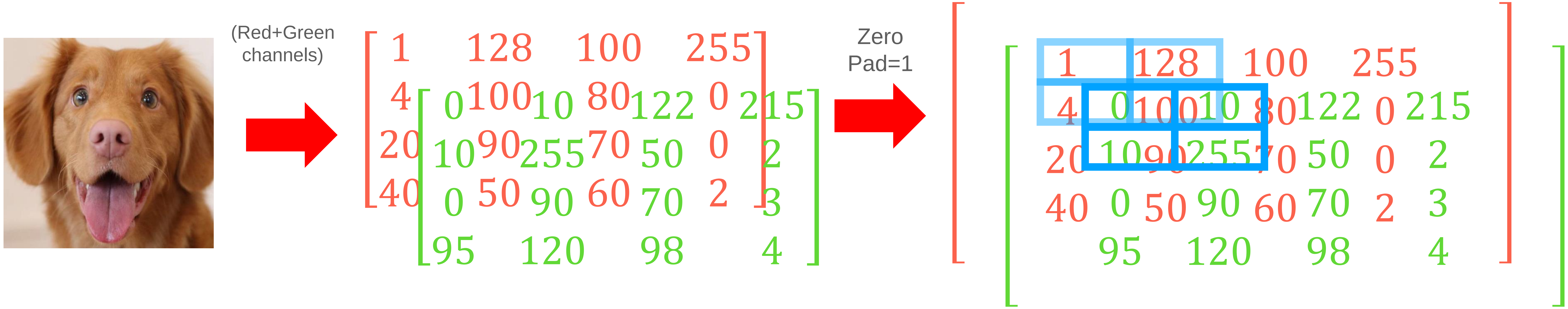
output[0, 1, 0] = 1 * 0 + 2 * 1 + 0 * 0 + 0.5 * 4 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 10 = 19

```
output[0, :, :]:
```

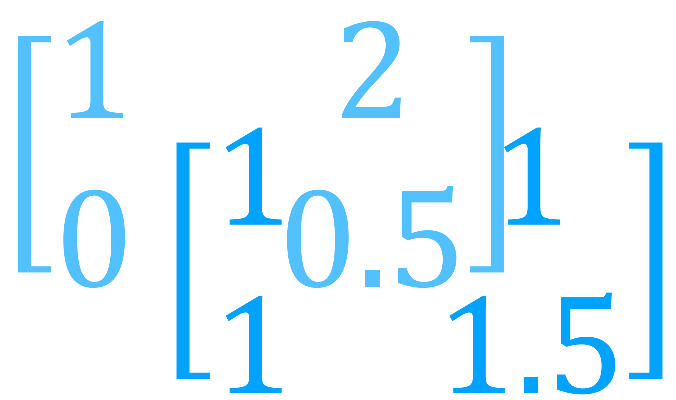
$$\begin{bmatrix}0.5 & 79\\ 19 & \\ & \end{bmatrix}$$

Conv2d: computation (mult channels)

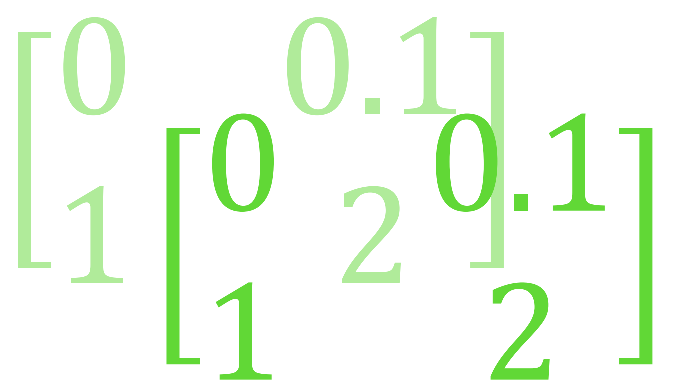
Convolution (aka "cross correlation"): **sliding window** computation




```
>>> filters[0, :, :, :]
```




```
>>> filters[1, :, :, :]
```

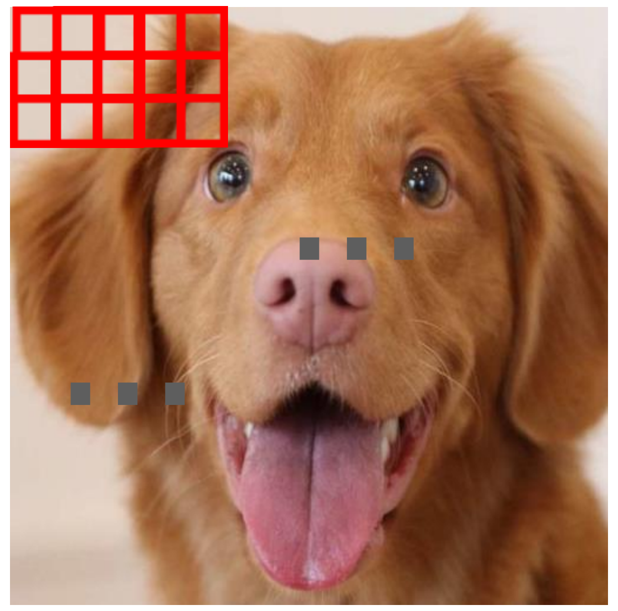


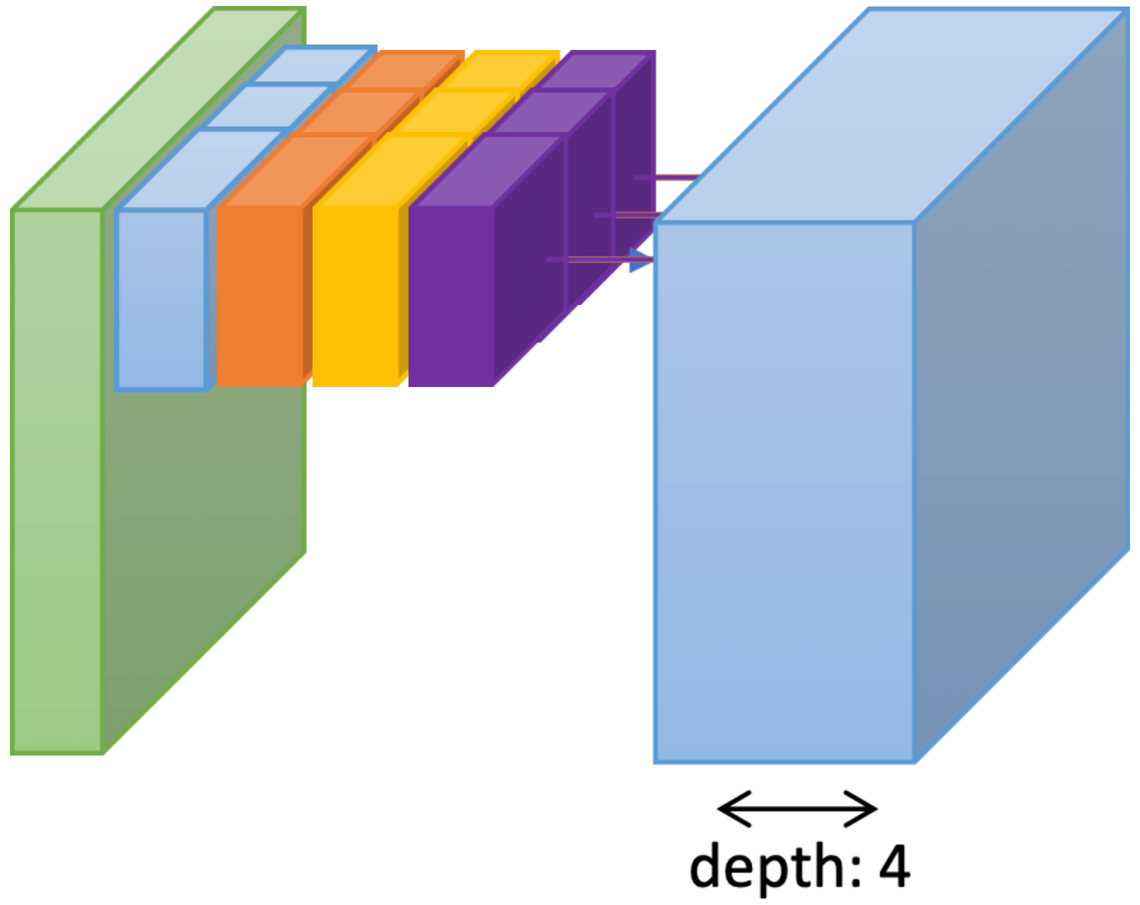
starting from the Red channel first

output[0, 1, 0] = 1 * 1 + 2 * 128 + 0 * 4 + 0.5 * 100 + 1 * 0 + 1 * 10 + 1 * 10 + 1.5 * 255 = 709.5

`output[0, :, :]:` $\begin{bmatrix} 0.5 & 79 \\ 19 & 709.5 \end{bmatrix}$

Feature map interpretation





What will our output look like?

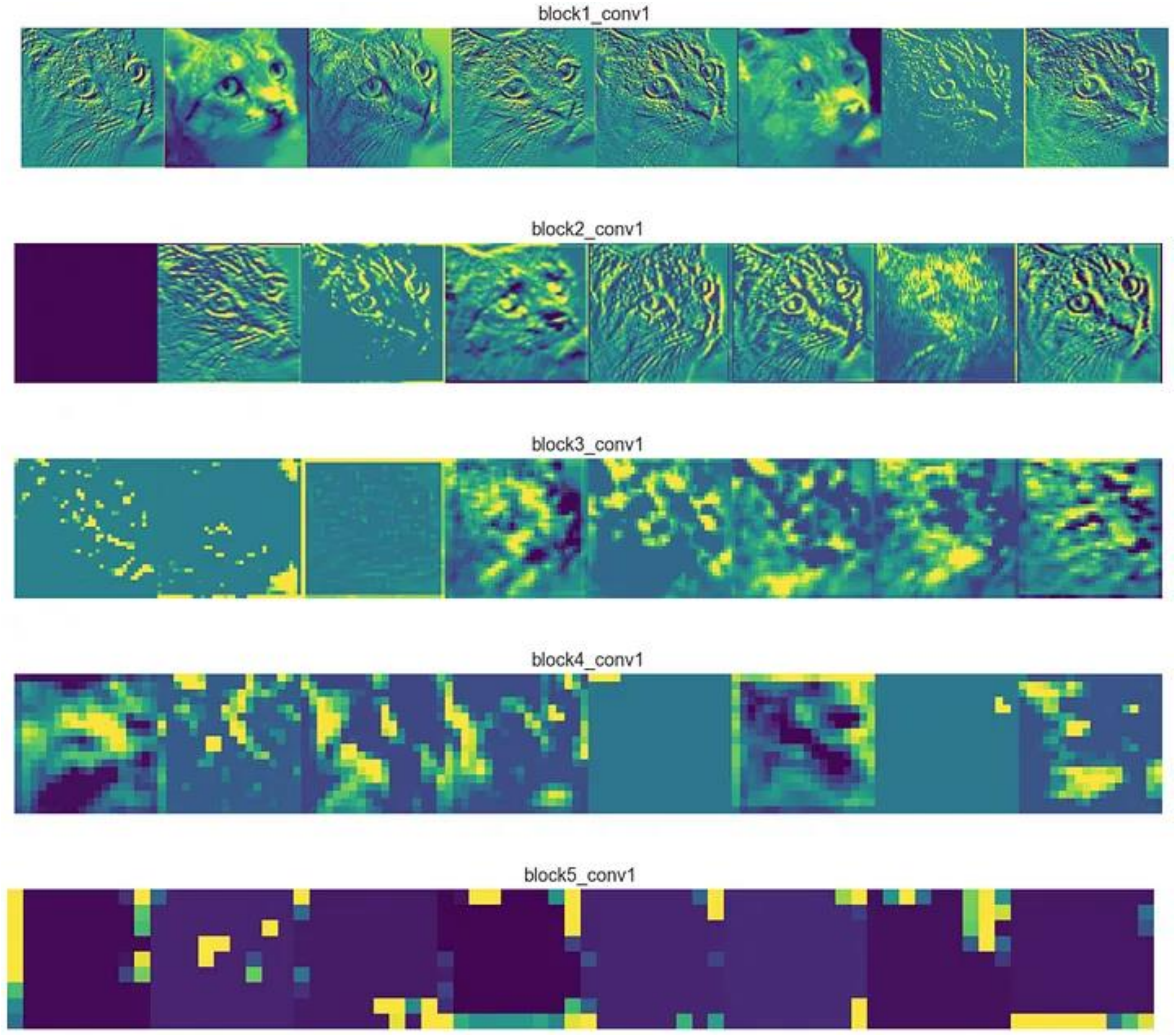
"Sliding" the filter along the image

It actually looks quite like an "image" itself… interesting…

"Spatial feature map"

Low level: edge detectors

High level: object detectors. Intuition/hope: high activations ideally mean "there is a semantic object (cat, dog) in this image!"

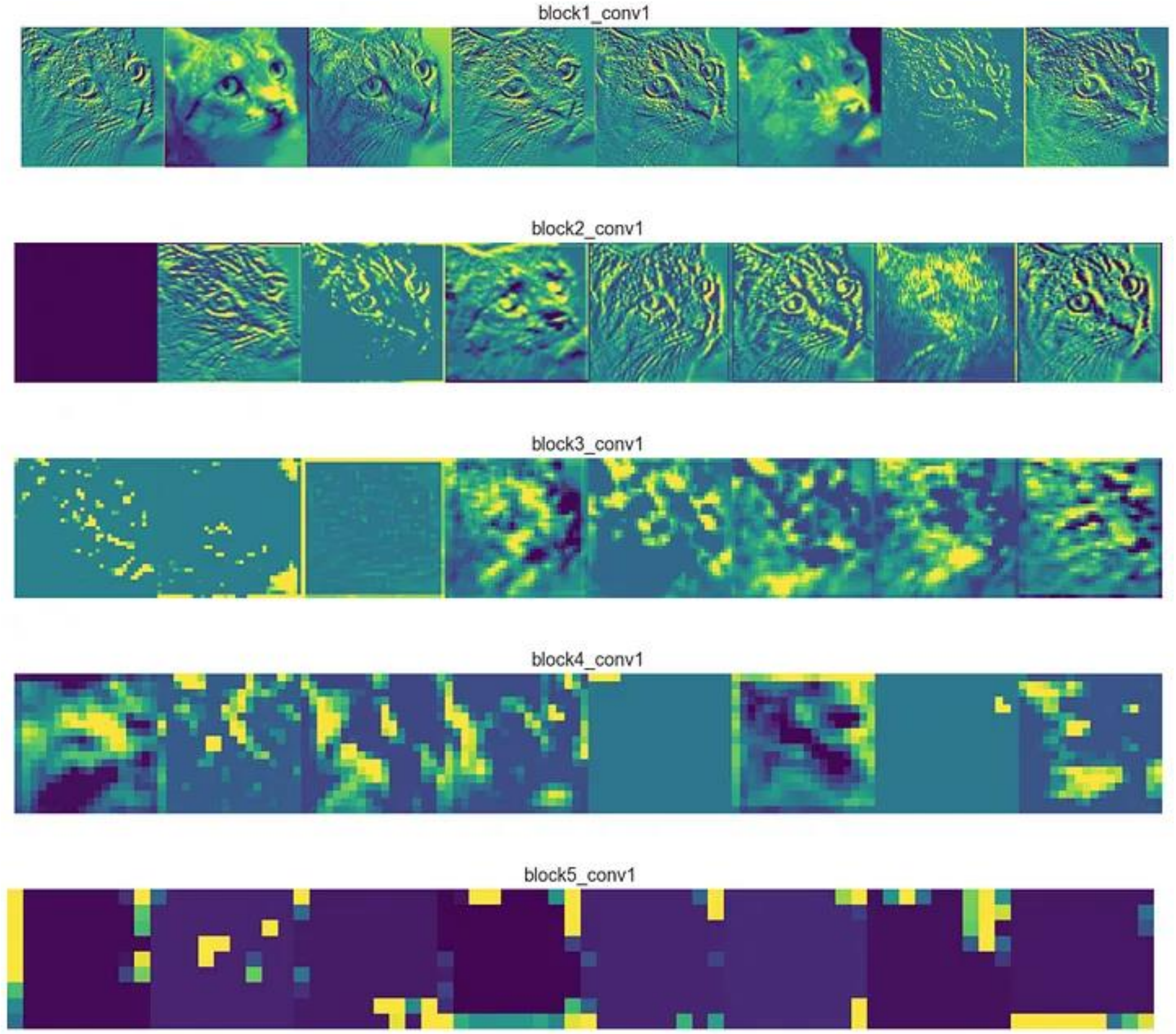


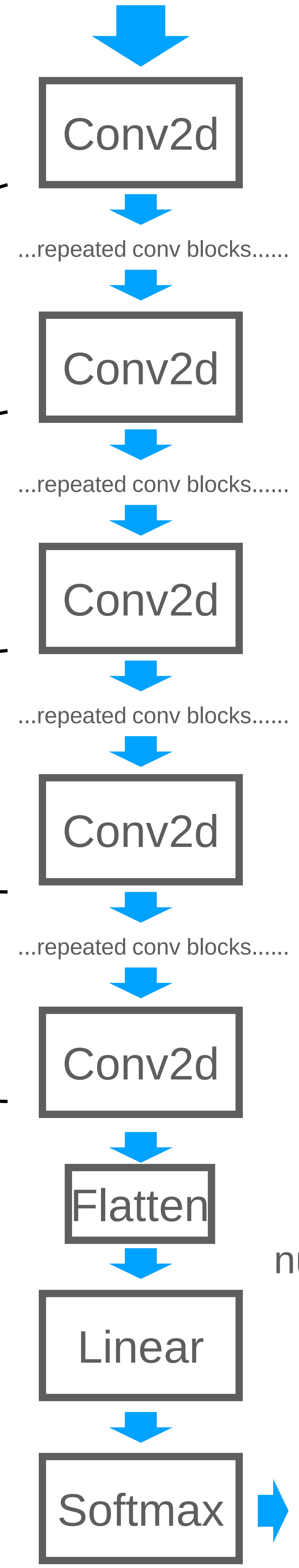
Interpretation: by stacking many conv layers, CNNs learn **hierarchical features**.
Lower level layers: Low-level image features (edges).
Middle layers: Mid-level image features (shapes)
Final layers: "semantic" features (eg part detectors, face detectors)

https://medium.com/@saba99/feature-map-35ba7e6c689e

Image classifier

Input image batch. Shape=[batchsize, 3, img_height, img_width]





Tada! Our first image classifier.

In practice, more recent CNNs will do tricks like: skip connections, multiple Linear layers, fully convolutional archs, feature pyramids...

Reshape 4D [batchsize, num_chans, height_block5, width_block5] to 2D [batchsize, num_chans*height_block5*width_block5]

Outputs logits. Shape=[batchsize, num_classes]

class_probs, shape=[batchsize, num_classes]

Final feature map shape: [batchsize, num_channels, height_block5, width_block5]

Question: how to predict class probabilities from the final feature map?

Answer: let's treat the spatial feat map as just another feature vector (flatten it), then add a Linear layer afterwards to produce the logits.

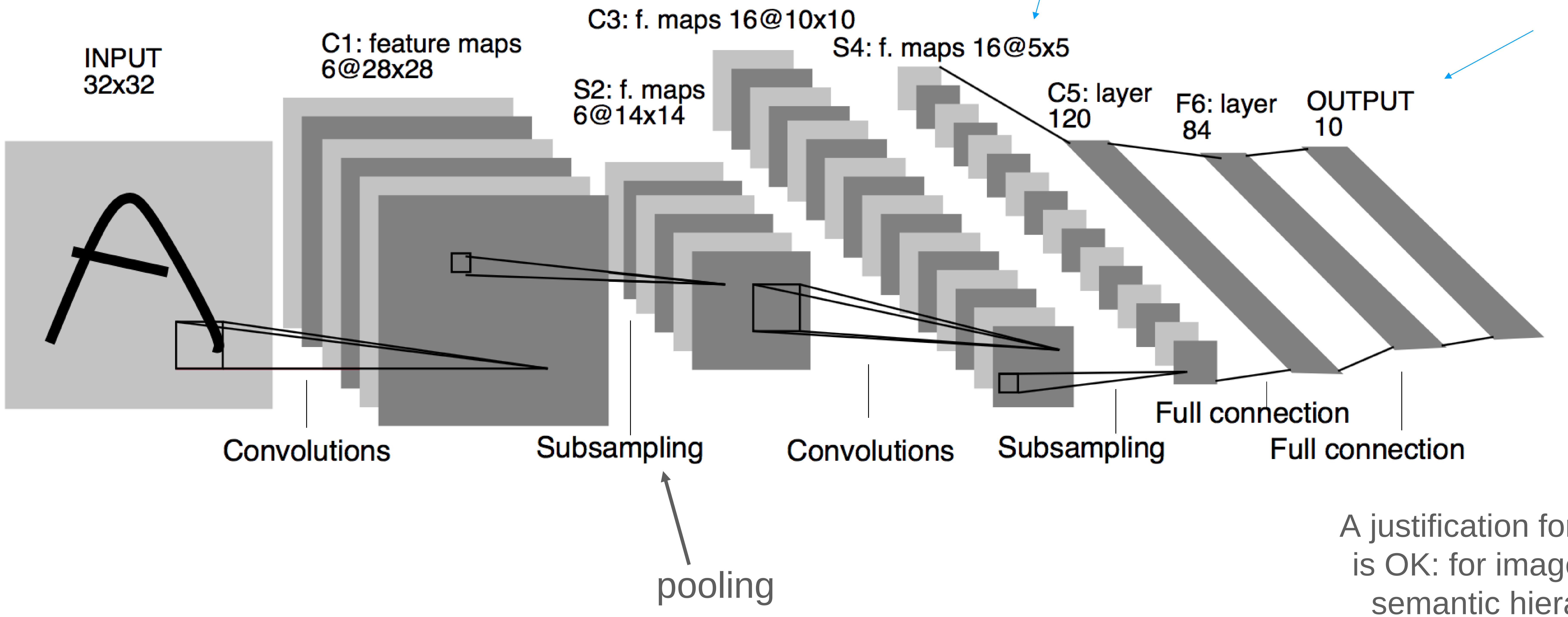
https://medium.com/@saba99/feature-map-35ba7e6c689e

What does a real conv net look like?

"LeNet" network for handwritten digit recognition

Tip: "16@5x5" means: 16 feature maps, with spatial resolution 5x5

Output is a vector with shape [10], because there are 10 target classes (all single digits)

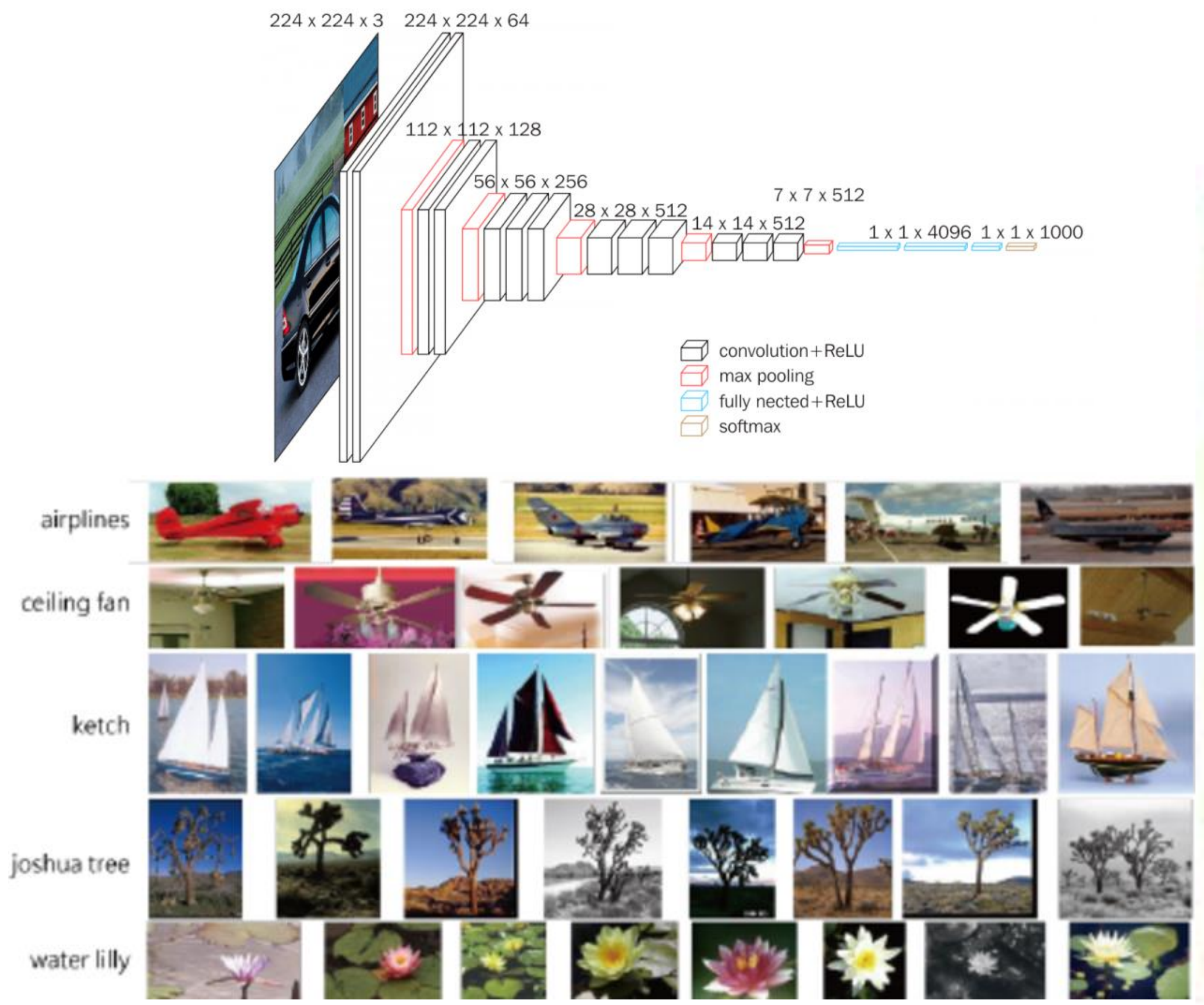


Observation: often, the spatial resolution of the feature maps gets smaller the deeper you get into the network. The main reason is due to performance: reducing spatial resolution (eg dividing by 2 every "block") dramatically improves computation speed and reduces memory requirements.

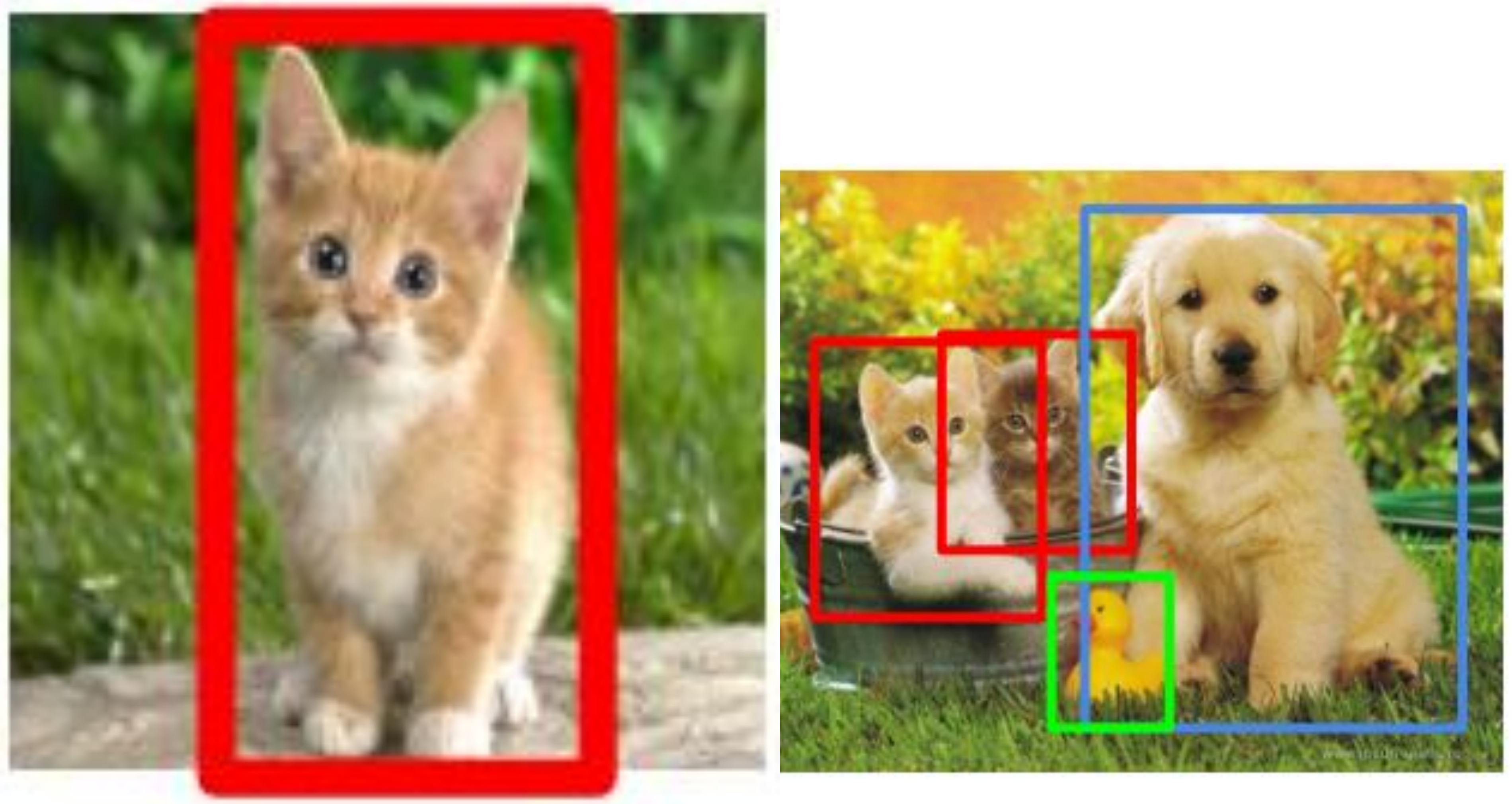
A justification for why reducing spatial resolution is OK: for image classification, the higher-level semantic hierarchical features don't need to know the "precise pixel location" of objects.

Counterpoint: for some applications (eg object detection, segmentation), "precise pixel location" is really important!

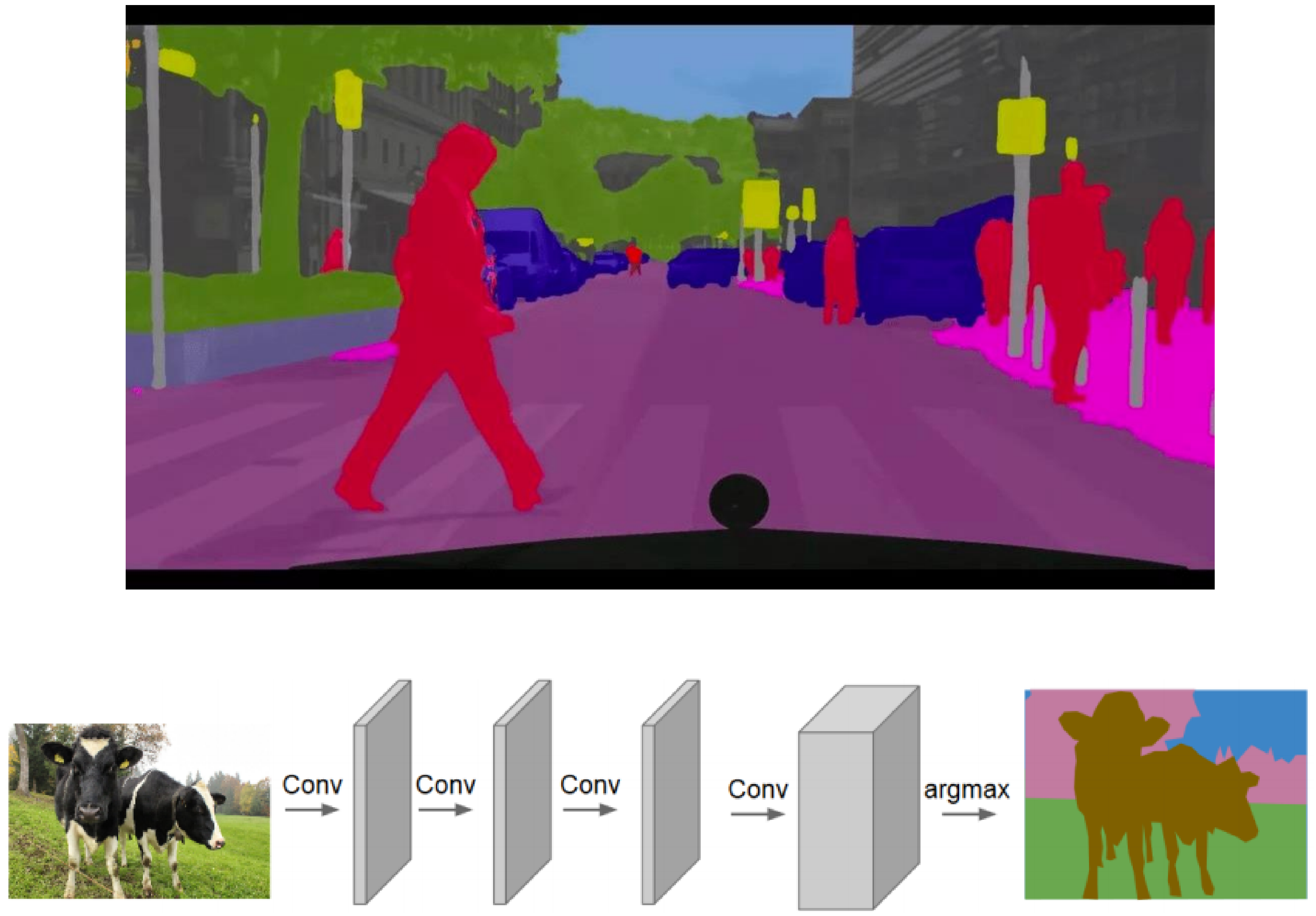
Standard computer vision problems



object classification

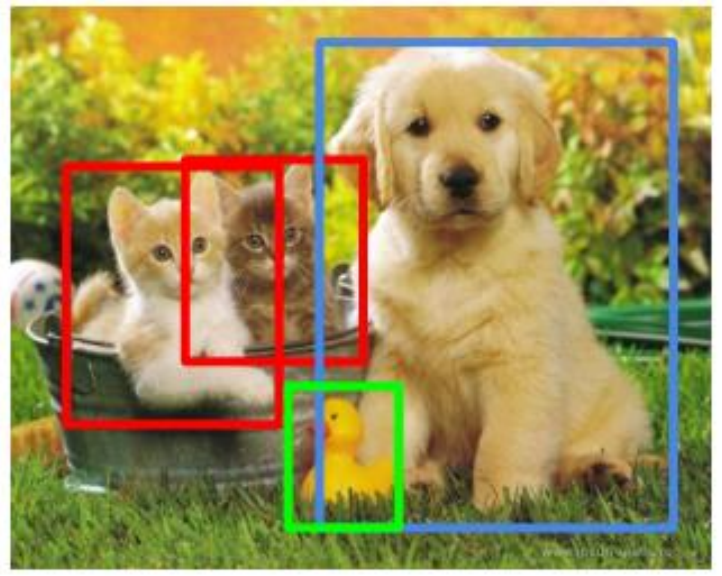


object detection



semantic segmentation
a.k.a. scene understanding

Object detection setup



Note: an image may have multiple ground truth objects!

Before: $\mathcal{D} = \{(x_i, y_i)\}$

image → x_i; class label (categorical) → y_i

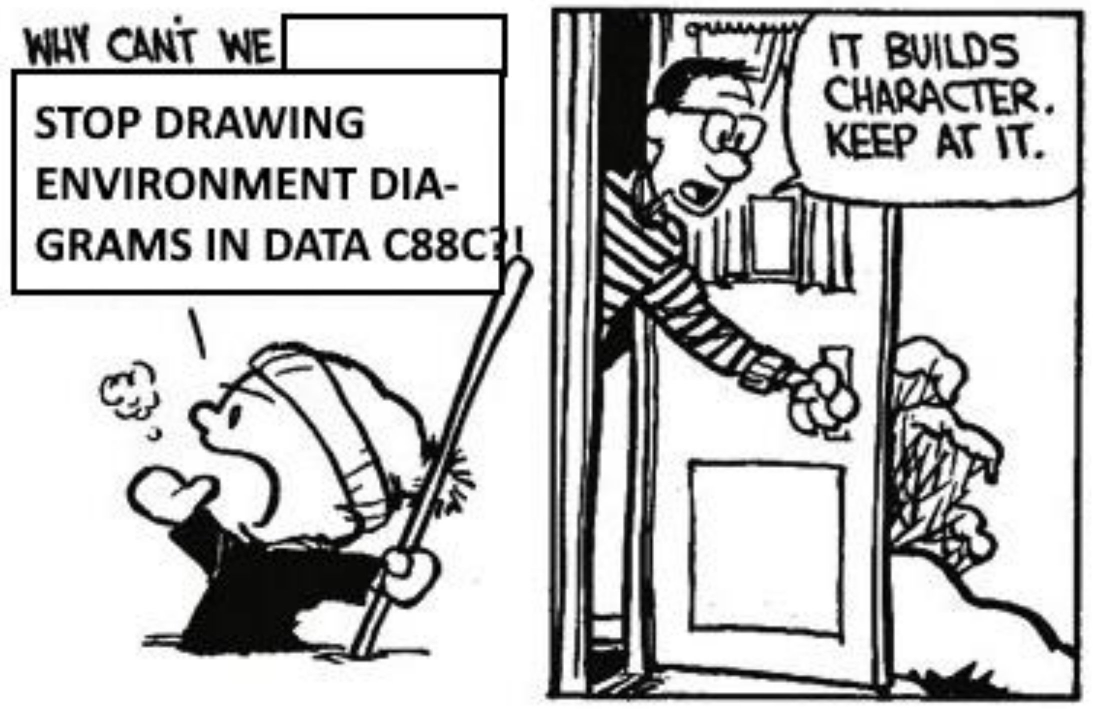
Now: $\mathcal{D} = \{(x_i, y_i)\}$

image → x_i; $y_i = (\ell_i, x_i, y_i, w_i, h_i)$

Example: ("cat", 0.2, 0, 0.6, 1.0)
Tip: rather than use pixel (absolute) coordinates, we often use preprocess the data to use "normalized" coordinates.

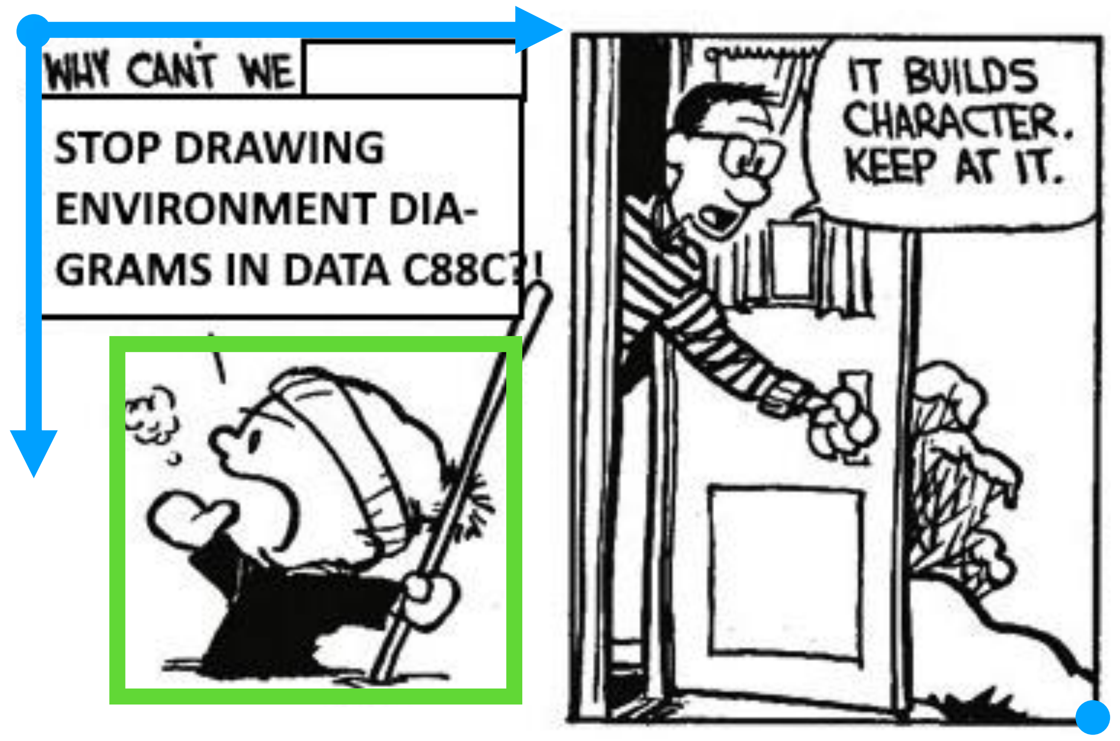
Aside: Representing bbox coordinates

- Image sizes/locs are often represented in terms of "pixels"
  - Aka "absolute" coordinates
- Alternate idea: represent bboxes in terms of "normalized" coordinates
  - x_norm = x_px / image_width, y_norm = y_px / image_height
- Question: would it be better if our bbox training set is in absolute coordinates, or normalized coordinates? Why?
- Answer: Generally normalized coordinates are preferred since it generalizes well to multiple image shapes, and makes the learning problem a little easier.
  - If you know in advance that your images are always a fixed size (eg 224x224), then absolute is maybe OK...



Example: this image has dimensions width=377px, height=253px.

Origin: (0px,0px)

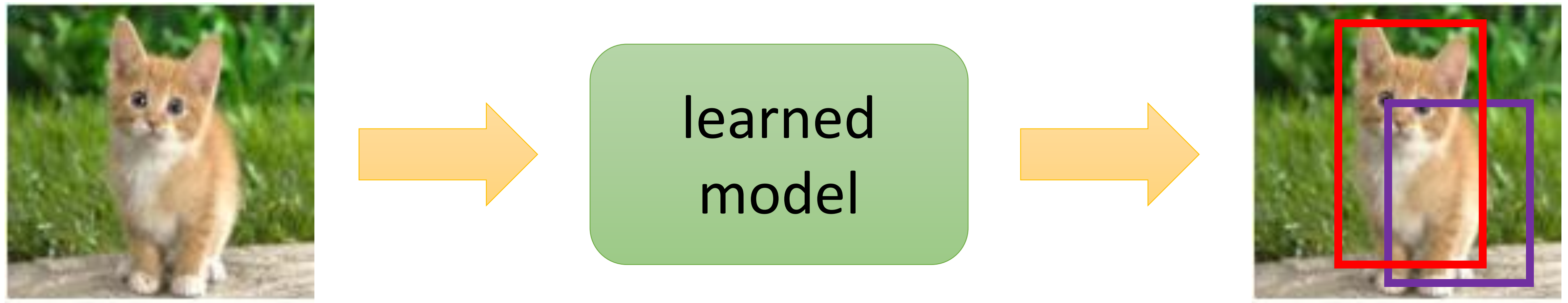


(376px, 252px)

Box_absolute: (x=30px, y=140px, w=130px, h=125px)

Box_norm: (x=0.08, y=0.55, w=0.35, h=0.49)

Measuring localization accuracy

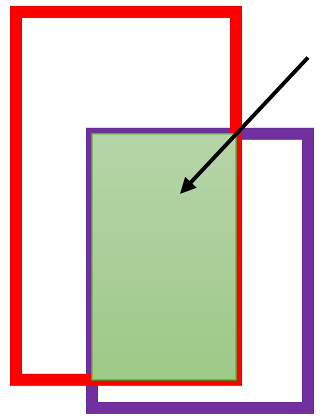


(x, y, w, h) ← predicted bounding box

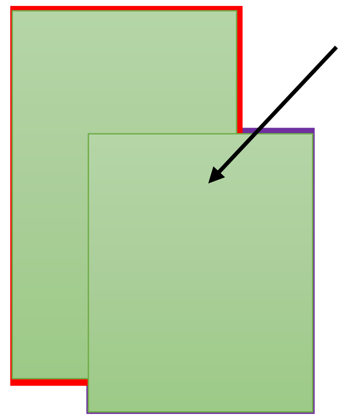
"cat": 0.64 ← prediction score (e.g., probability)

Did we get it right?

Intersection over Union (IoU)



intersection area (**I**)



union area (**U**)

IoU = I / U

Different datasets have different protocols, but one reasonable one is: **correct if IoU > 0.5**

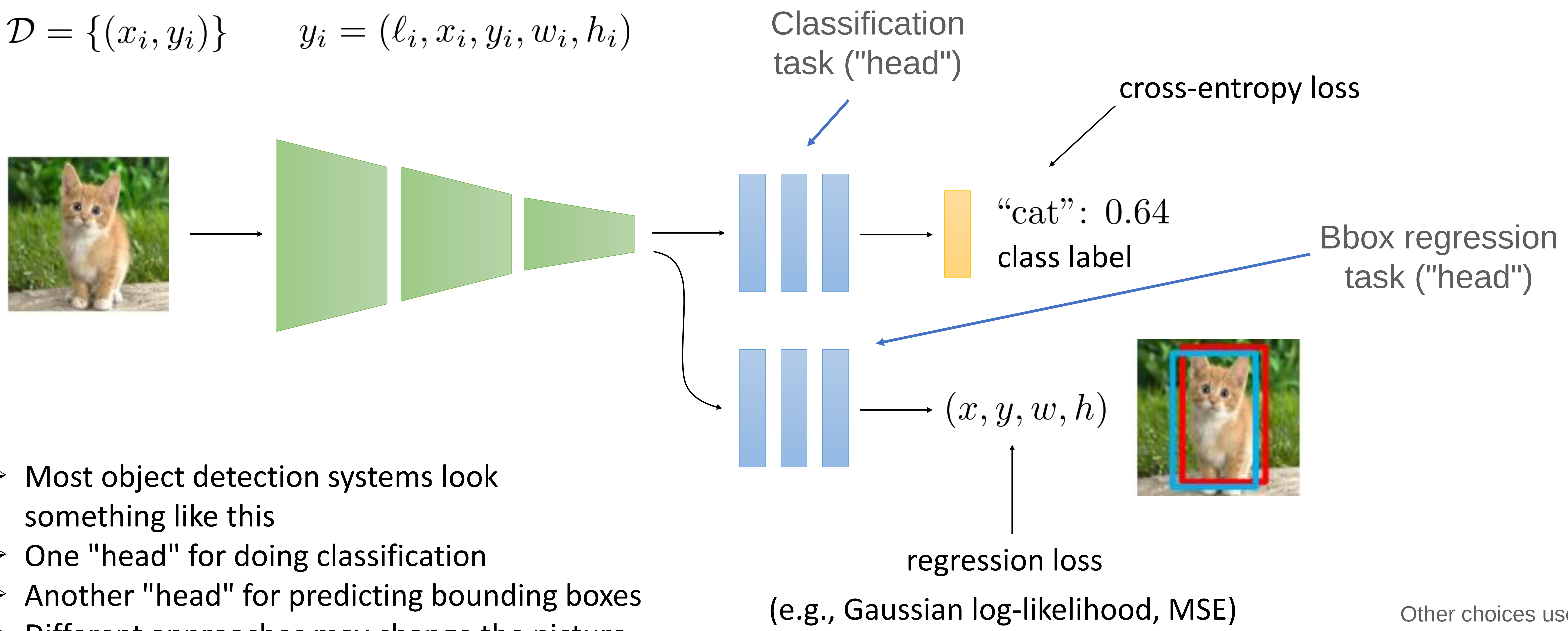
If also outputting class label (usually the case): **correct if IoU > 0.5 and class is correct**

This is **not** a loss function! Just an evaluation standard

Object localization

Object localization as regression

$\mathcal{D} = \{(x_i, y_i)\}$ $\quad y_i = (\ell_i, x_i, y_i, w_i, h_i)$



- Most object detection systems look something like this
- One "head" for doing classification
- Another "head" for predicting bounding boxes
- Different approaches may change the picture, but the spirit is still there

Reasonable bbox losses: sum squared error (L2) of each term:
Loss = (x-x')**2 + (y-y')**2 + (w-w')**2 + (h-h')**2

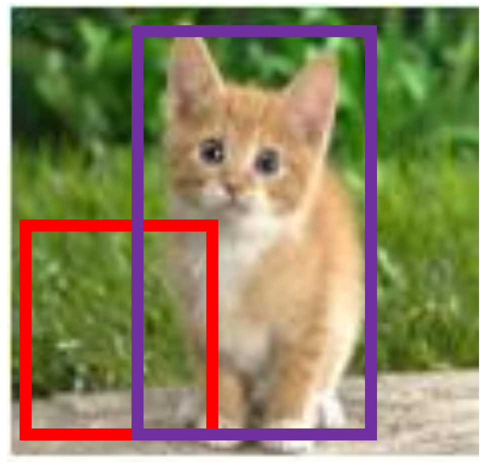
Other choices used: "Smooth" L1 error, "Focal" loss

x is predicted, x' is ground truth

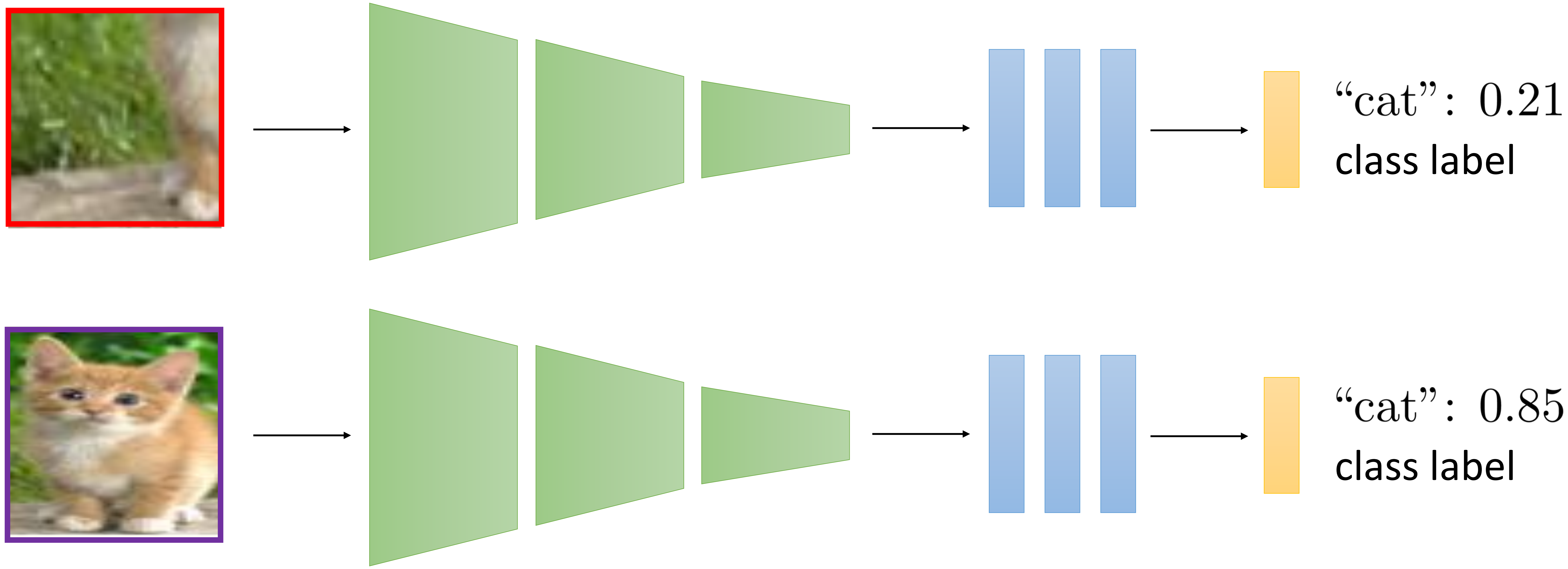
Sliding windows

$\mathcal{D} = \{(x_i, y_i)\}$ $\qquad y_i = (\ell_i, x_i, y_i, w_i, h_i)$

Problem: In theory, it could work...but it'd be so, so slow. There are so many candidate patches in an image. Not practical!

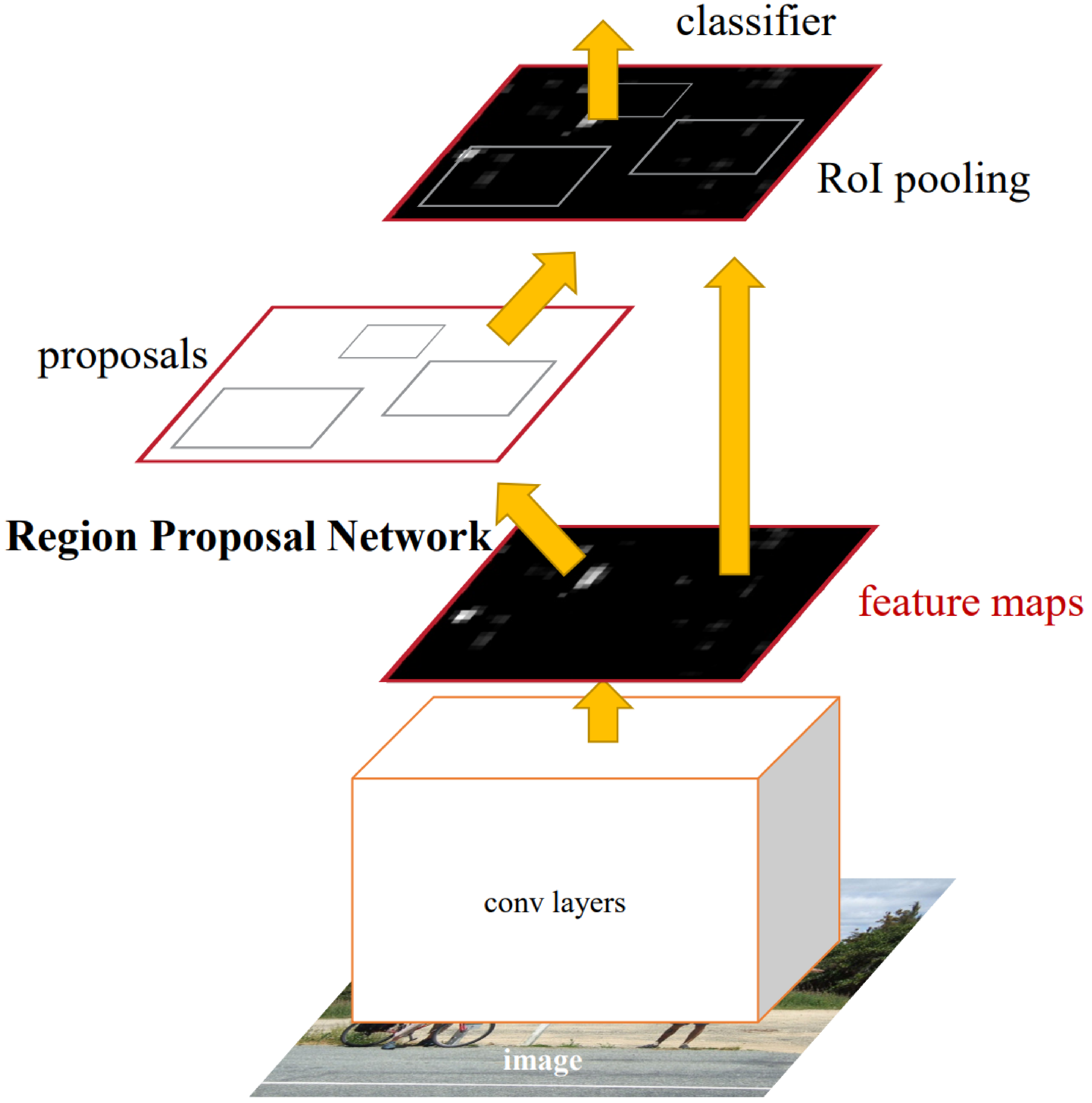


What if we classify **every** patch in the image?



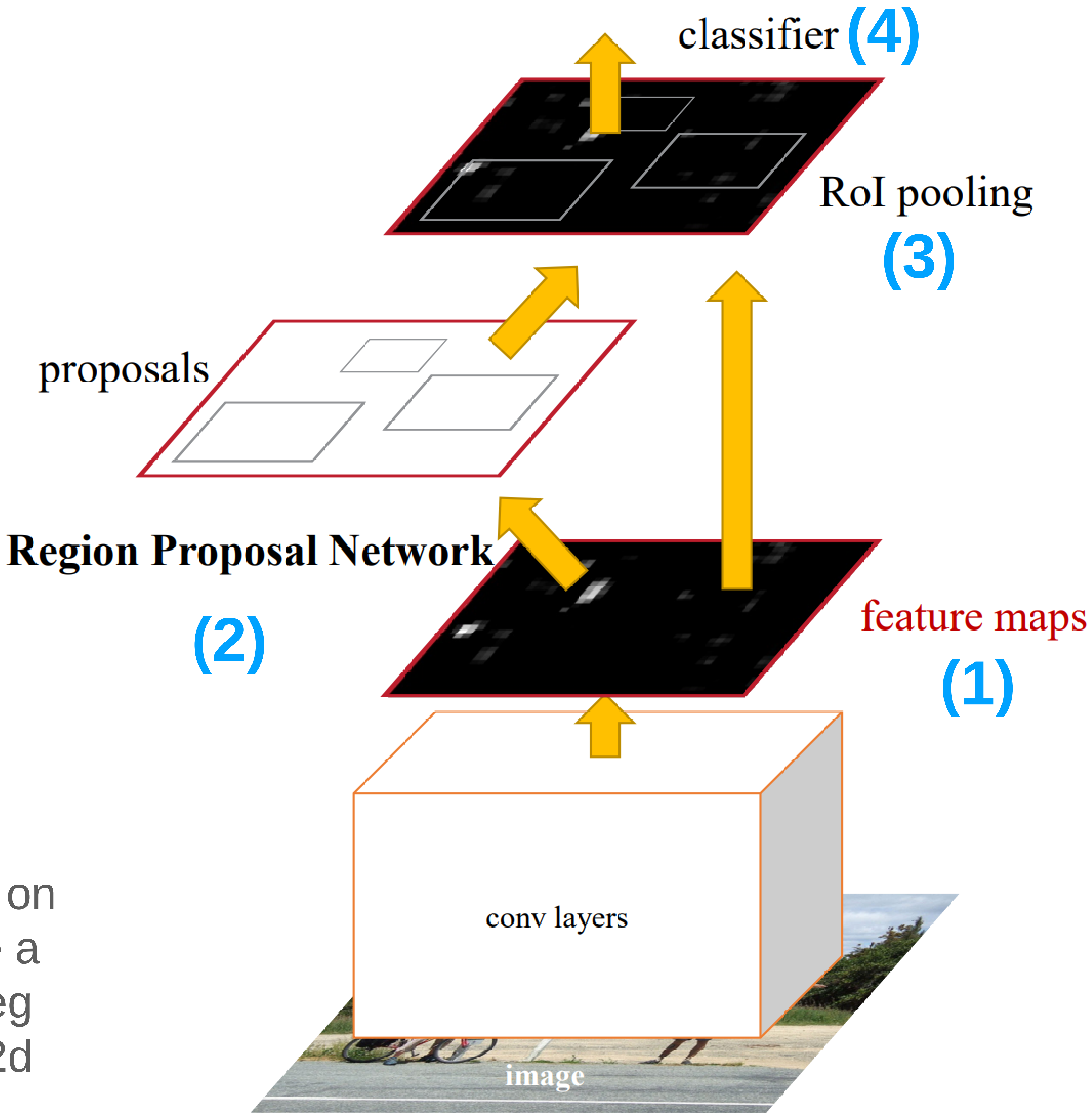
Case study: Faster-RCNN (2015)

- This is a series of work on getting object detection to work with ConvNets
  - R-CNN (2013), **Fast** R-CNN (2015), **Faster** RCNN (2015)
- Core idea with Faster-RCNN
  - **"Region proposal".** First, have the model predict high quality candidates for where an object might be
  - **"Classification + refinement"**. Then, have the model process each candidate location, and determine if there is actually an object in there.
    - Also, refine the bbox coordinates



Faster-RCNN: Forward pass

(2) We run the "Region Proposal Network" (RPN) which, given the feature map, produces a set of N "object proposals", aka bboxes that the RPN thinks has SOME object in it.



(4) For each proposal + their RoI pooling features, we run a classifier to predict the class.

(3) For each proposal, we extract the CNN features via a "Region of Interest Pooling" (RoI pooling) layer.

(1) First, we run a ConvNet on the input image to produce a final spatial feature map (eg output from the final conv2d block)

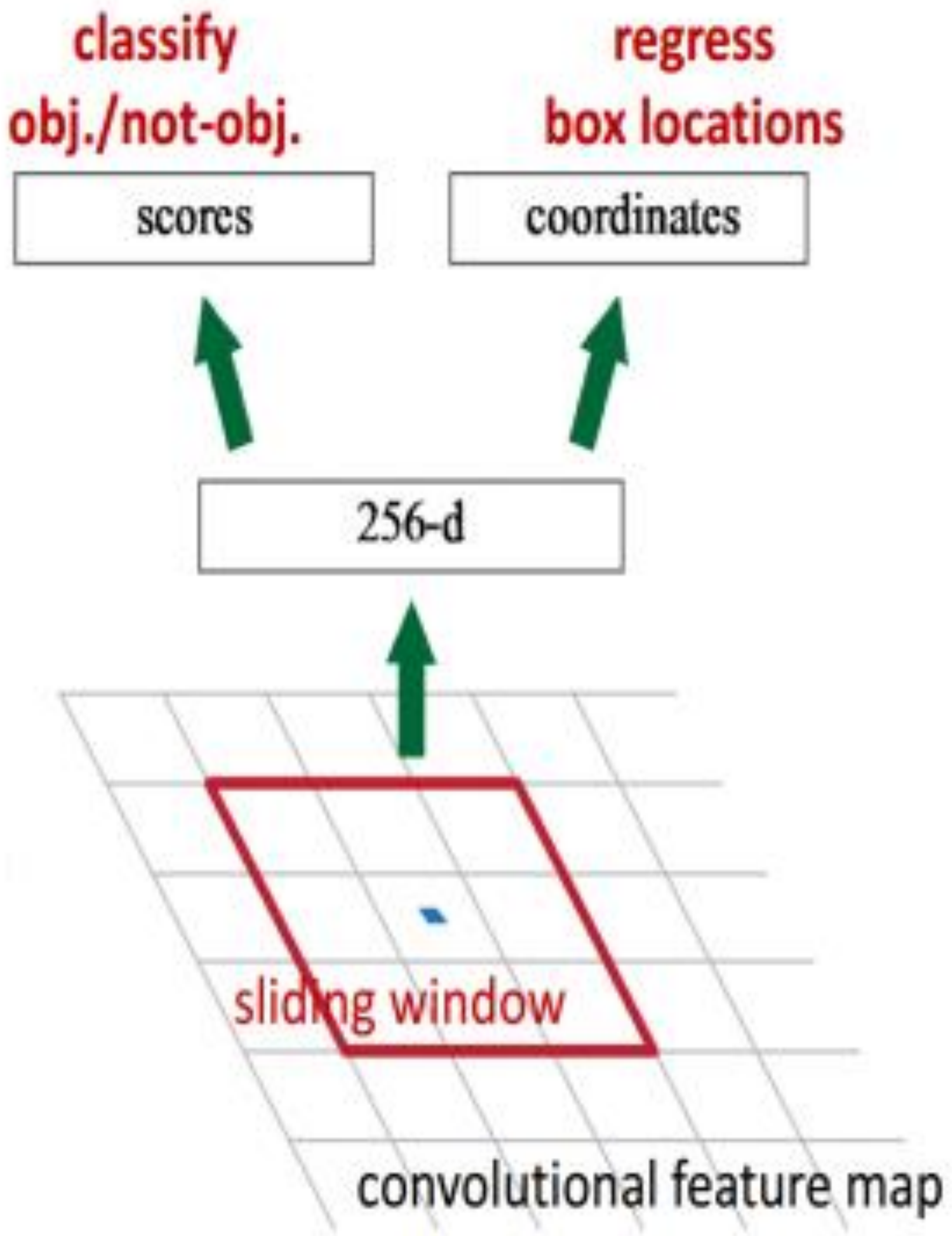
How does the Region Proposal Network work?

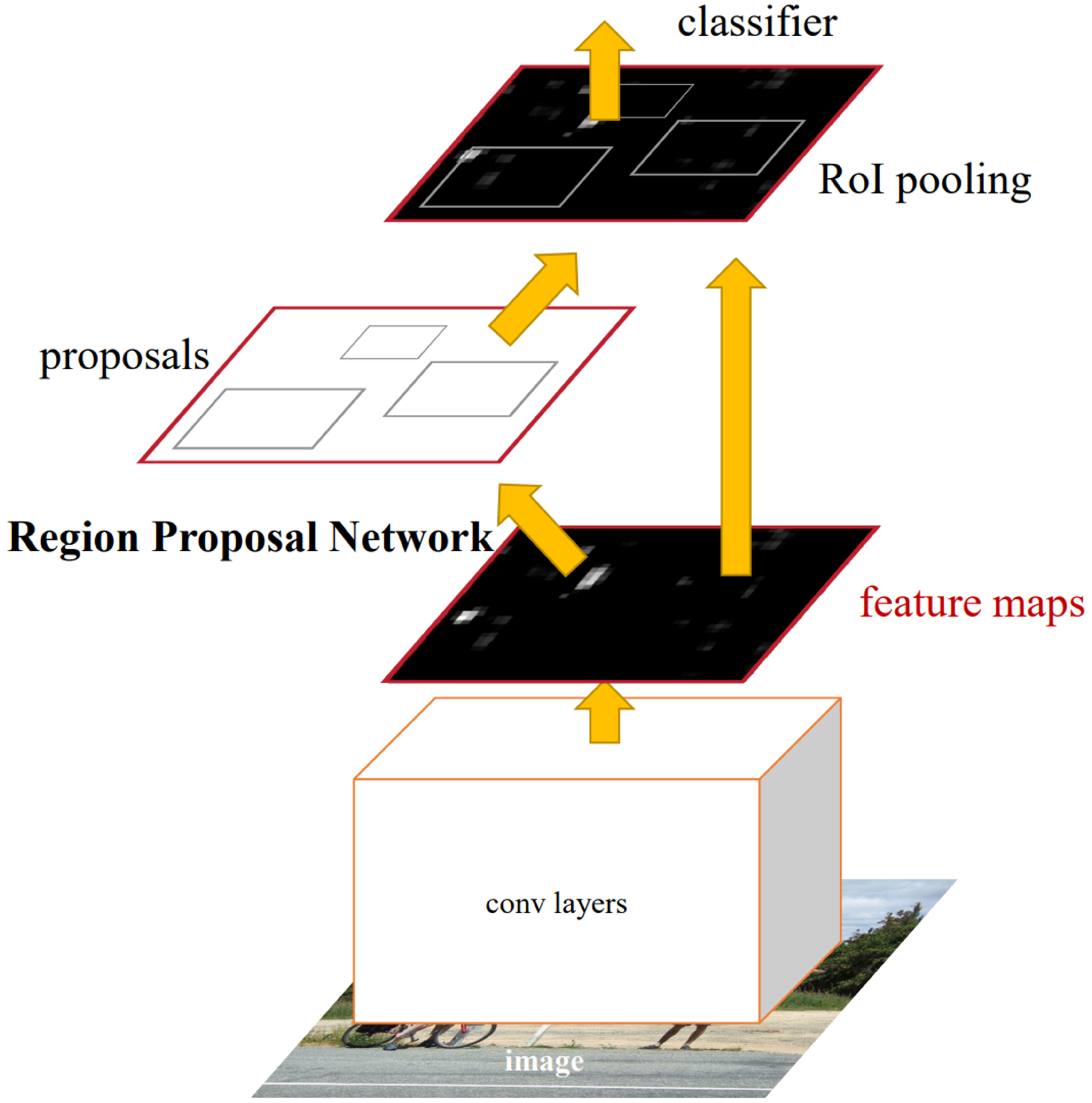
First, we do a 3x3 conv2d (num_out_filters=256 or 512), producing a [chans=256, h, w] feature map.

For each spatial location (i, j), we take the 256-dim vector and do two separate FC layers to predict

Score: is_object vs not_object
Box coords: (x, y, w, h)

Neat trick: rather than explicitly have Linear layers (FC), we can implement the score/box heads via a 1x1 conv!





Ren et al. "**Faster R-CNN.**" 2015

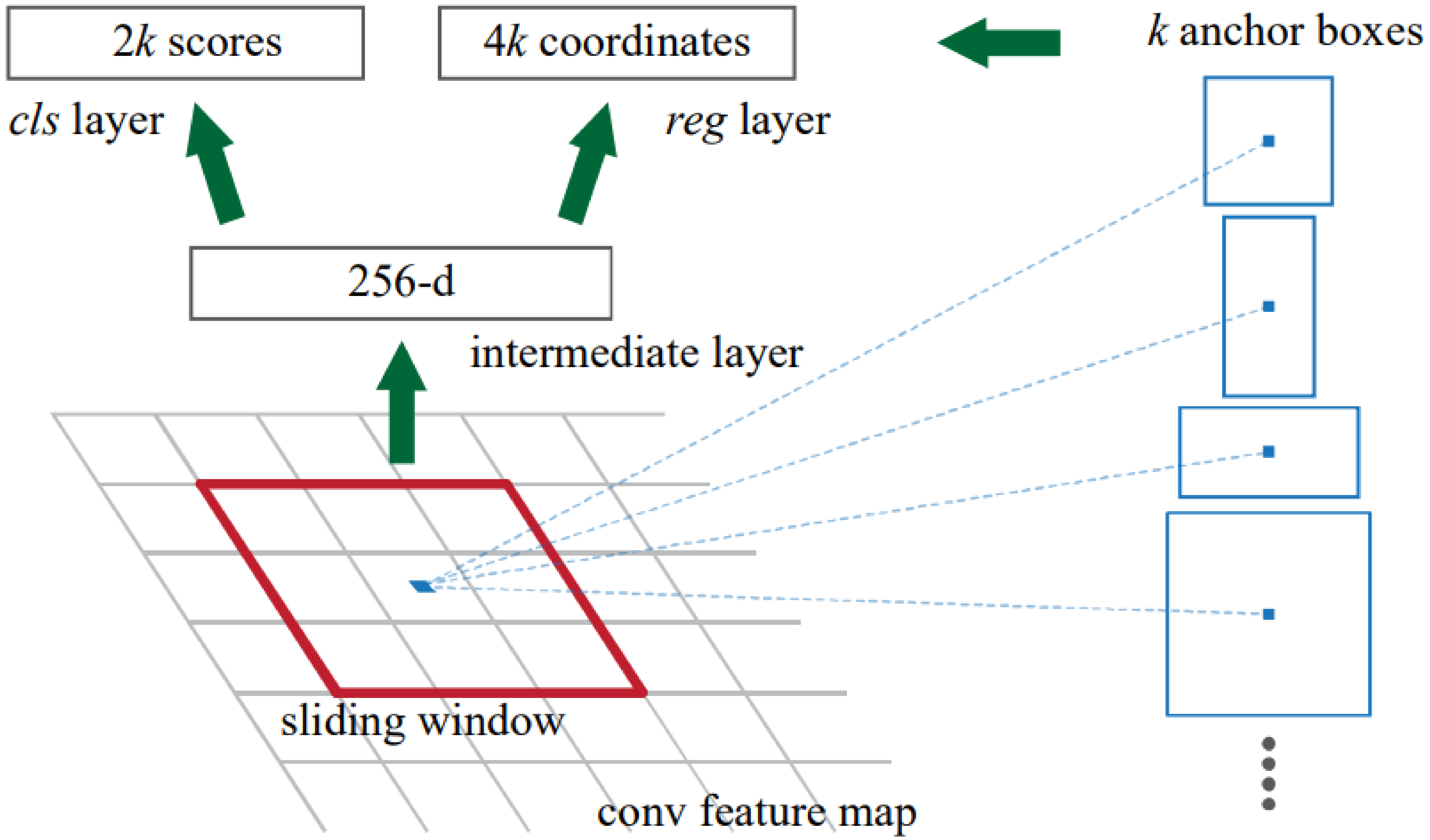
Details: Anchor Boxes

Rather than directly predict just one bbox for each spatial location (i, j), instead we predict adjustments to k=9 "anchor boxes".

Thus, the final outputs of the RPN are actually:

Scores: shape=[h*w*k, 2]
Two classes: "is_object" and "background".

Bbox Proposals: shape=[h*w*k, 4]
Each row is [dx, dy, dh, dw], aka an adjustment to the k-th anchor box.



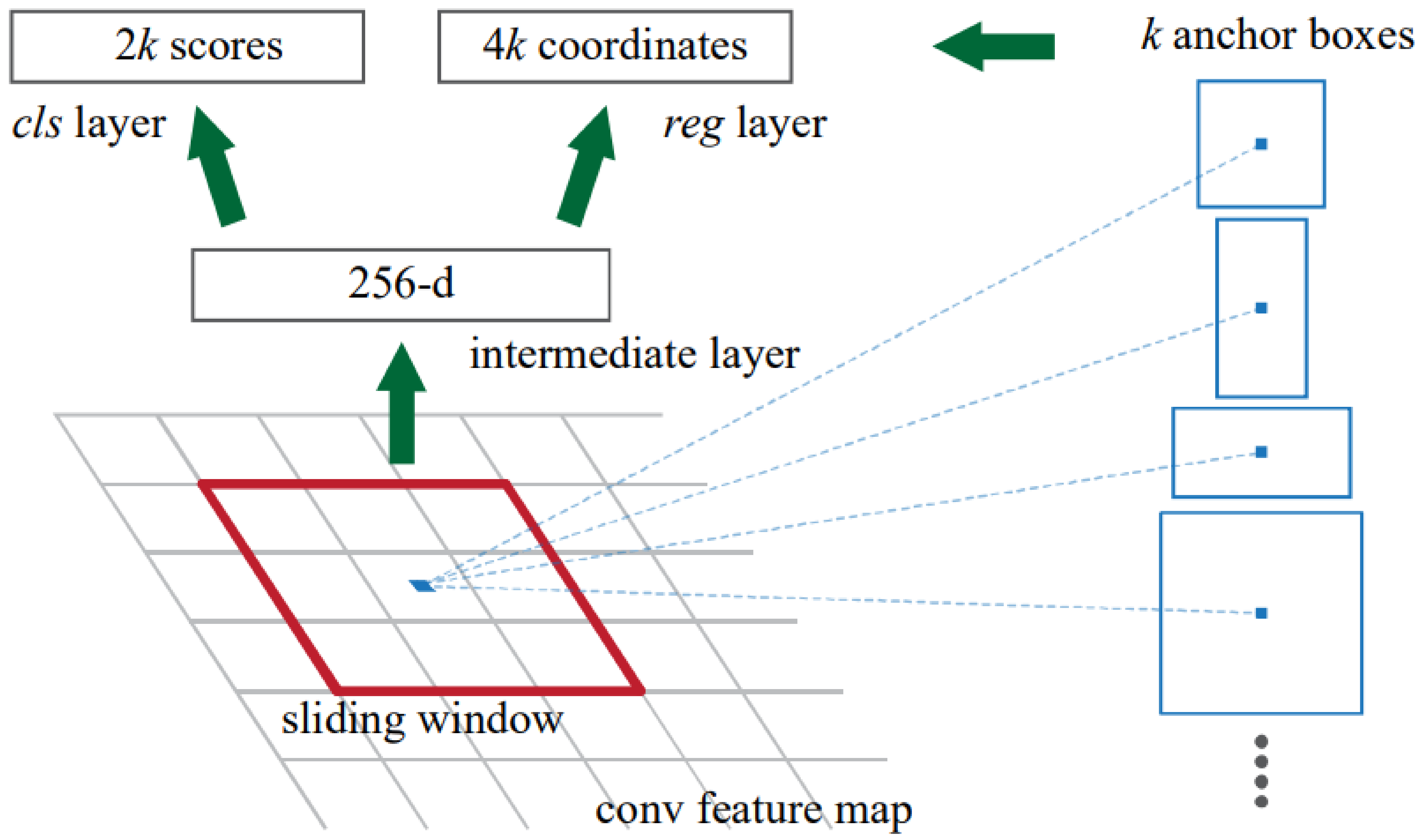
In the paper: k=9 anchor boxes. 3 different aspect ratios, then 3 different scales (small, medium, large).

Anchor box definitions are an important hyperparameter!

Ren et al. "**Faster R-CNN.**" 2015

RPN: 1x1 Conv?

Recall: the RPN's 3x3 conv2d produces a feature map with shape=[256, h, w].



Question: How do we turn this feature map into these desired outputs?

Scores: shape=[h*w*k, 2]
Two classes: "is_object" and "background". ("objectness" score)

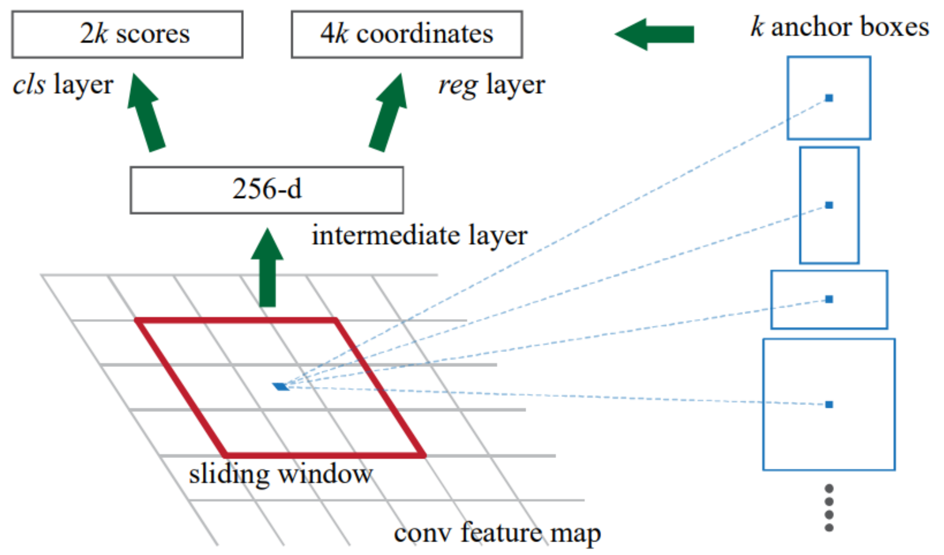
Bbox Proposals: shape=[h*w*k, 4]
Each row is [dx, dy, dh, dw], aka an adjustment to the k-th anchor box.

Their answer: 1x1 conv2ds, with a carefully chosen number of output channels!

RPN: 1x1 Conv?

```
cnn_feat_map = CNN(image)  # [batchsize, chans, h_cnn, w_cnn]
# Start: RPN
conv2d_rpn = Conv2d(filter_size=(3,3), num_chans_out=256, stride=1, pad=1)
rpn_feat_map = conv2d_rpn(cnn_feat_map)  # [batchsize, 256, h, w]

conv2d_cls = Conv2d(filter_size=(1,1), num_chans_out=2 * k, pad=0, stride=1)
conv2d_bbox = Conv2d(filter_size=(1,1), num_chans_out=4 * k, pad=0, stride=1)

preds_cls = conv2d_cls(rpn_feat_map)  # [batchsize, 2*k, h, w]
preds_bbox = conv2d_bbox(rpn_feat_map)  # [batchsize, 4*k, h, w]

# preds_cls[0, :, i, j] is the class probs [is_obj, is_bkgd] at spatial location (i,j),
#   [prob0_anchor0,prob1_anchor0, prob0_anchor1,prob1_anchor1, ...]
# pred_bbox[0, :, i, j] is the box deltas (dx,dy,dw,dh) for spatial location (i,j):
#   [dx_anchor0,dy_anchor0,dw_anchor0,dh_anchor0, dx_anchor1,dy_anchor1,dw_anchor1,dh_anchor1, ...]

# End: RPN
```

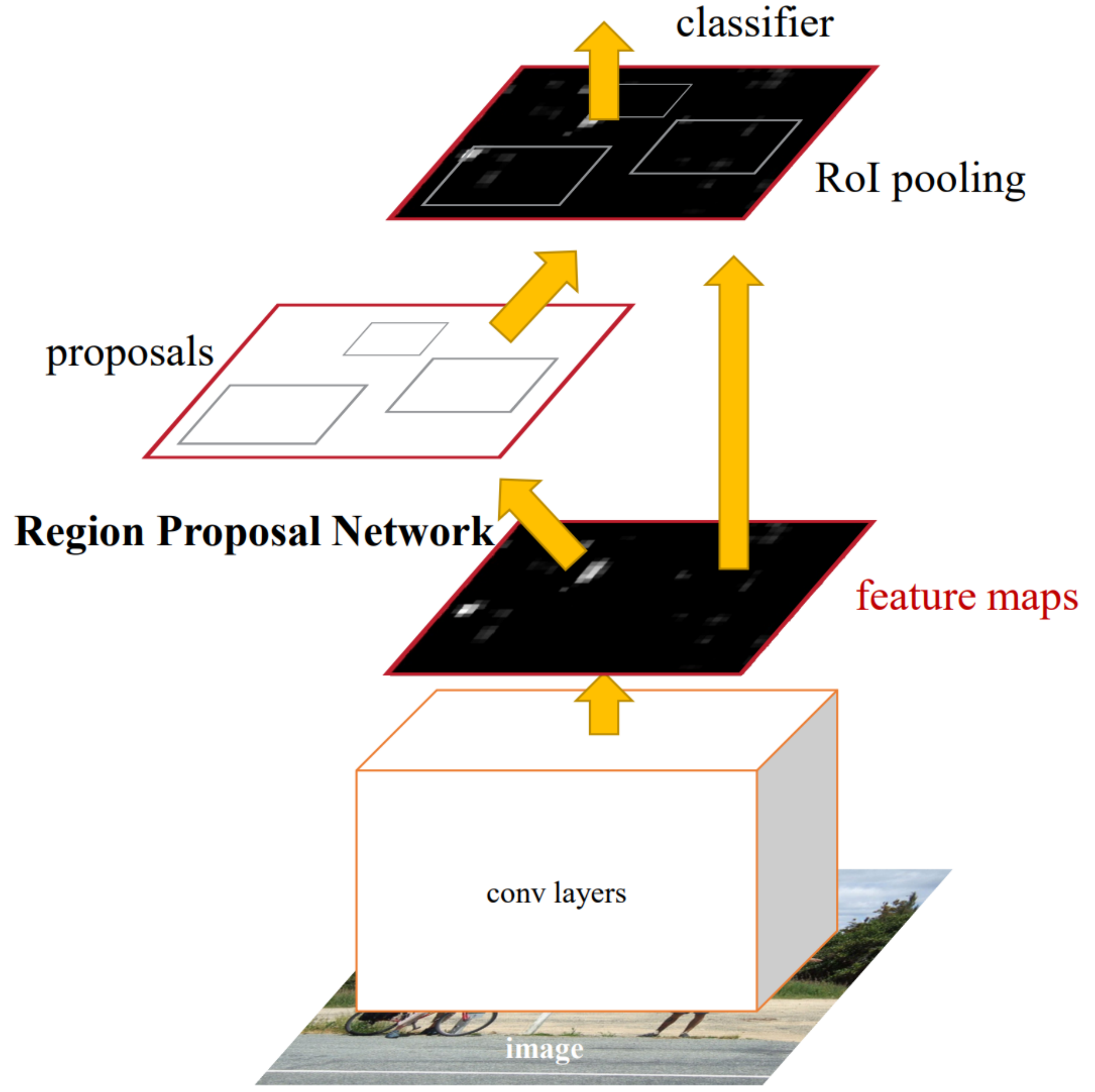
Clever trick of implementing a Linear(in=256,out={2*k,4*k}) via a 1x1 conv2d with a special choice of number of output channels. Please study this until you understand why this works!



Faster-RCNN: Forward pass

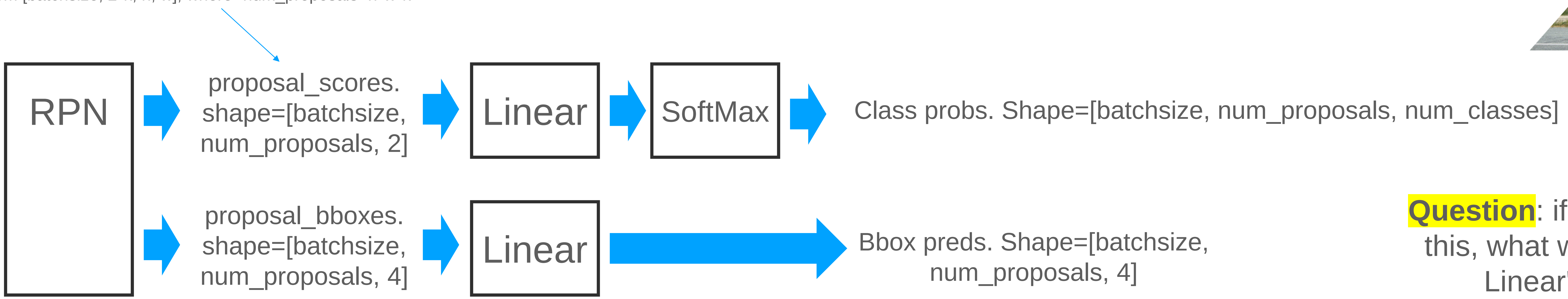
So far (post RPN): we have proposed objectness scores (shape=[batchsize, 2*k, h, w]) and bboxes (shape=[batchsize, 4*k, h, w]).

Next: what I'd like to do is classify each object proposal into the actual predictions I care about: final class predictions ("cat" vs "dog"), and box locations.



Question: suppose I trained a classifier directly on top of the RPN outputs in this way. Will this work well?

Tip: to simplify implementation, I reshaped the RPN output from [batchsize, 2*k, h, w], where `num_proposals=h*w*k`



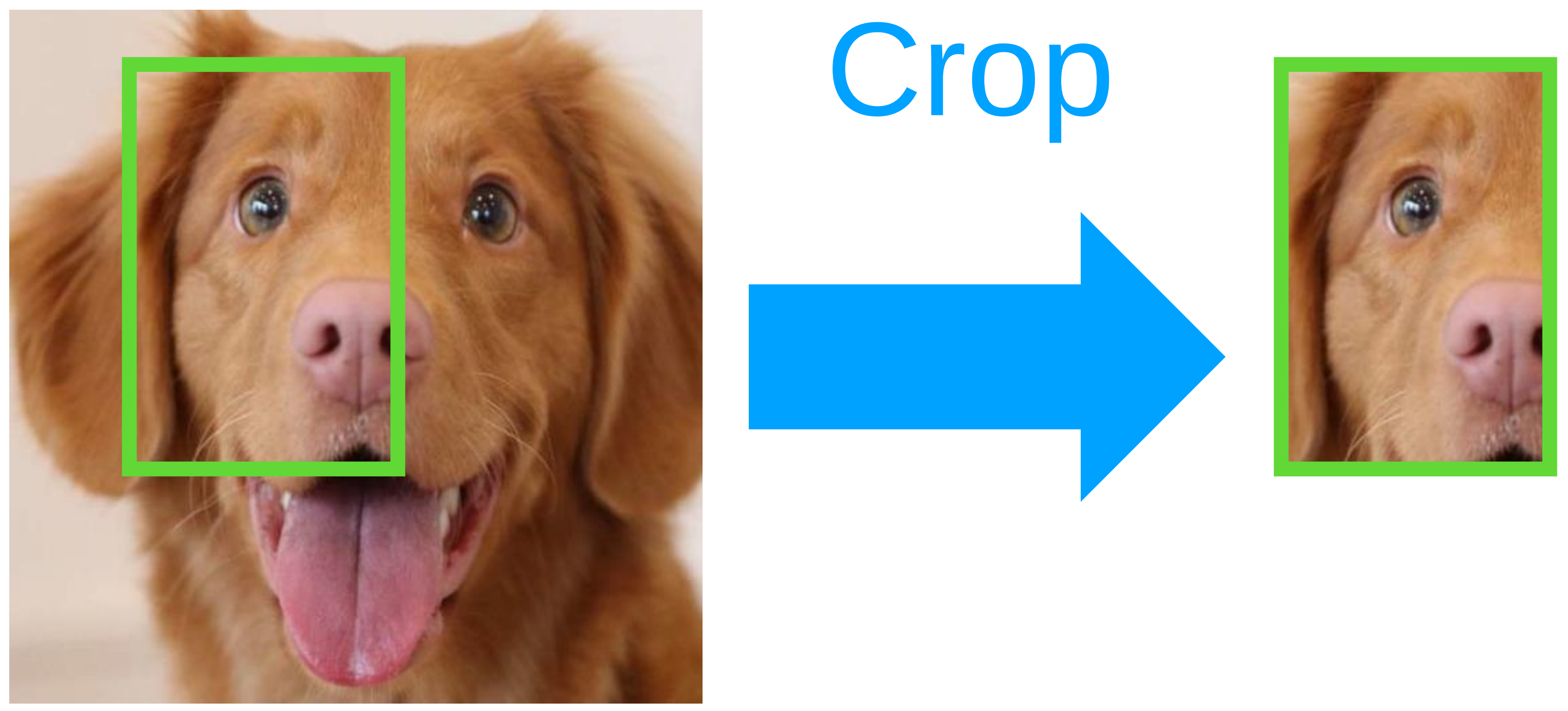
Question: if we did want to implement this, what would the shapes of each Linear's Weight matrix be?

Answer: No, this will not work well because we're not passing any **visual information** to the classification / box heads! Basically asking the detector to classify "blind".

Answer: Classification Linear.W.shape=[num_classes, 2]
Bbox Linear.W.shape=[4, 4]

Faster R-CNN

- How to get visual features for a given box proposal?
  - **Intuition**: in image pixel space, we can crop an image via a bbox
- In theory, this could work:
  - Crop the image pixels corresponding to the proposal bbox
  - Run this image patch through the CNN again to extract proposal features
  - Run the classifier/box head to produce final predictions (class, box coords)



Requires computing the proposal bbox coords from the CNN feature map coord space (eg 16x16 feat map) to the original image pixel space (eg 224x224). Which is do-able!

Feature map: [16x16]

Image: [w=260px, h=260px]

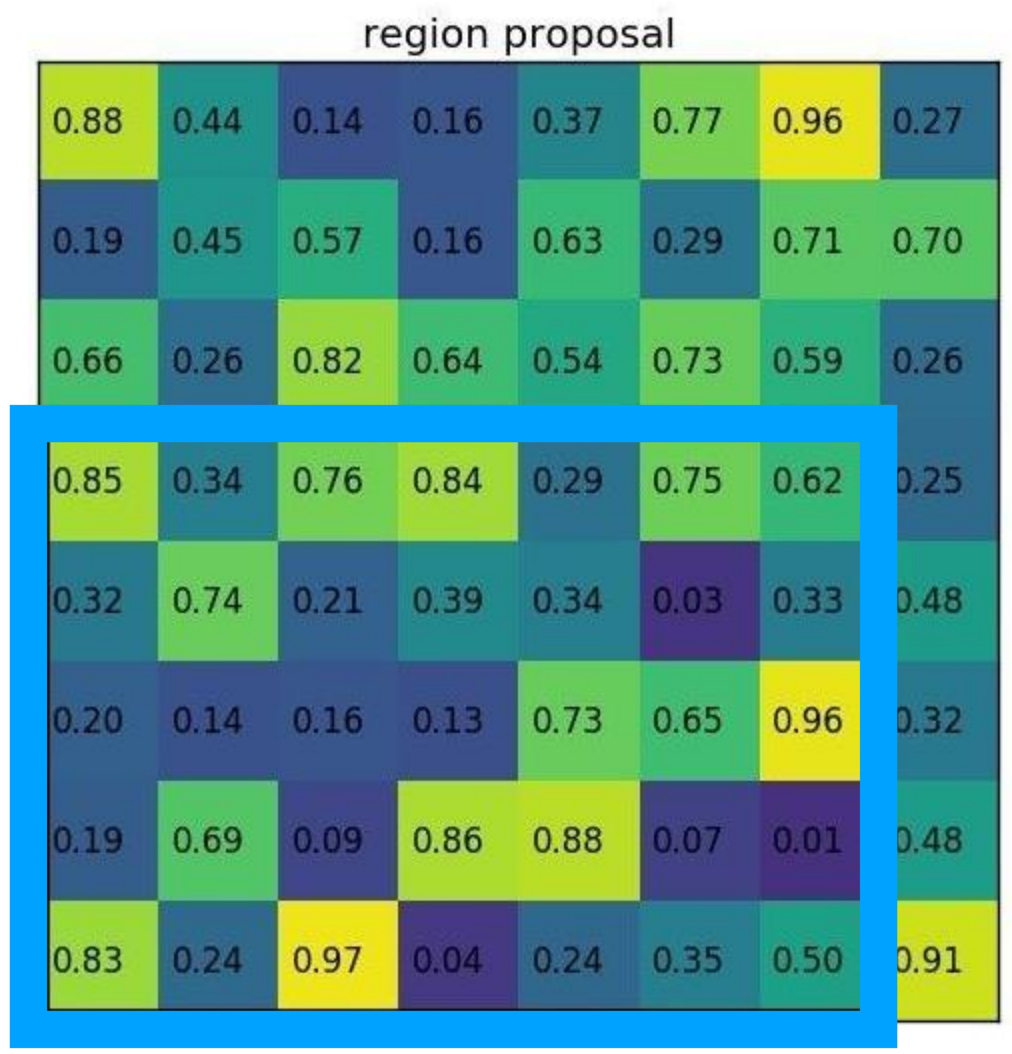
Downside: the second CNN forward pass is slow and wasteful.
Idea: let's reuse the feature map we've already computed!

RoI Pooling

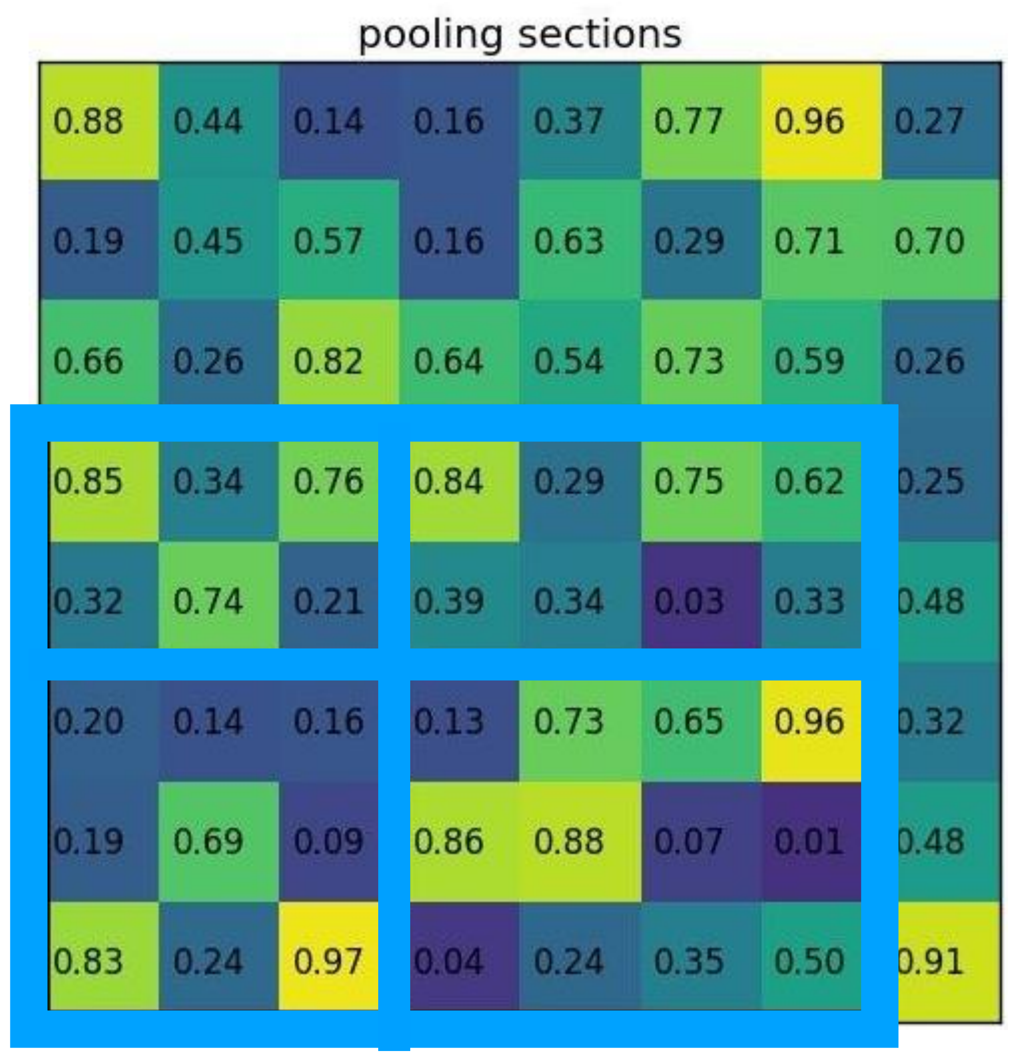
Tip: nice animation explaining RoIPooling is here: [link]

- A way of "cropping" a CNN spatial feature map via a bounding box.

Proposal box

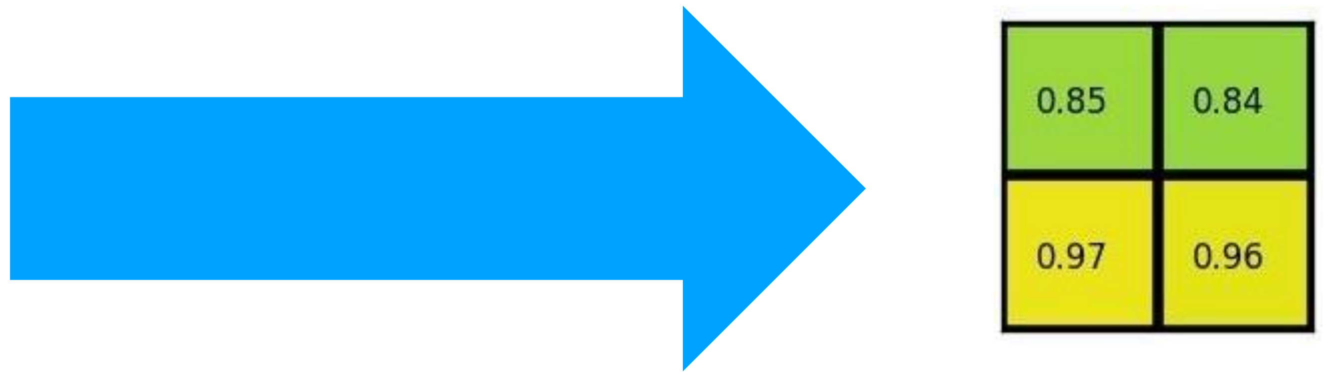


CNN spatial feature map (8x8)



(1) Grid up proposal box into a fixed number of cells (eg 4 cells)

(2) Calculate max value in each cell



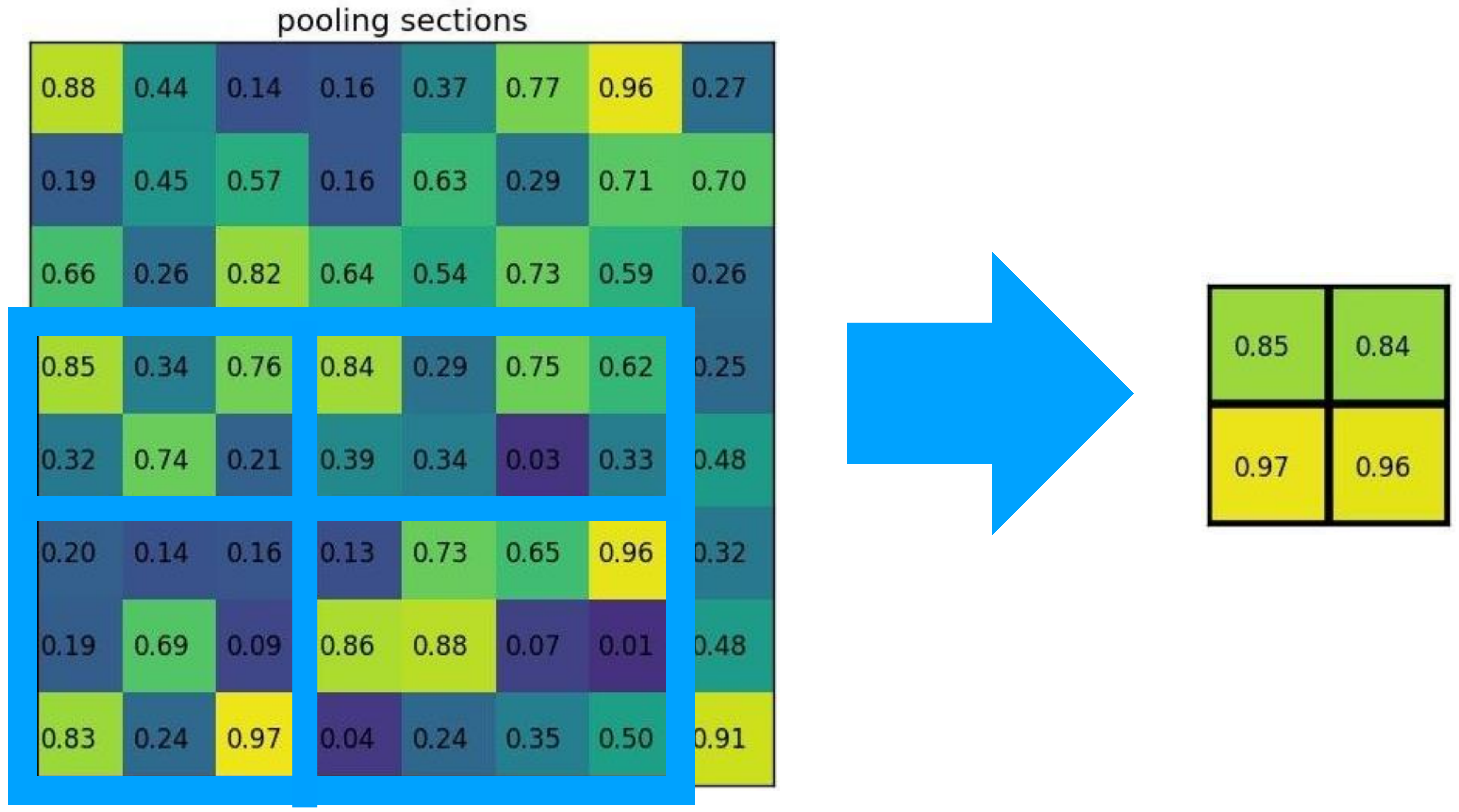
(3) Output! Feed to your downstream classifier/box head

https://deepsense.ai/region-of-interest-pooling-explained/

RoI Pooling

Tip: nice animation explaining RoIPooling is here: [link]

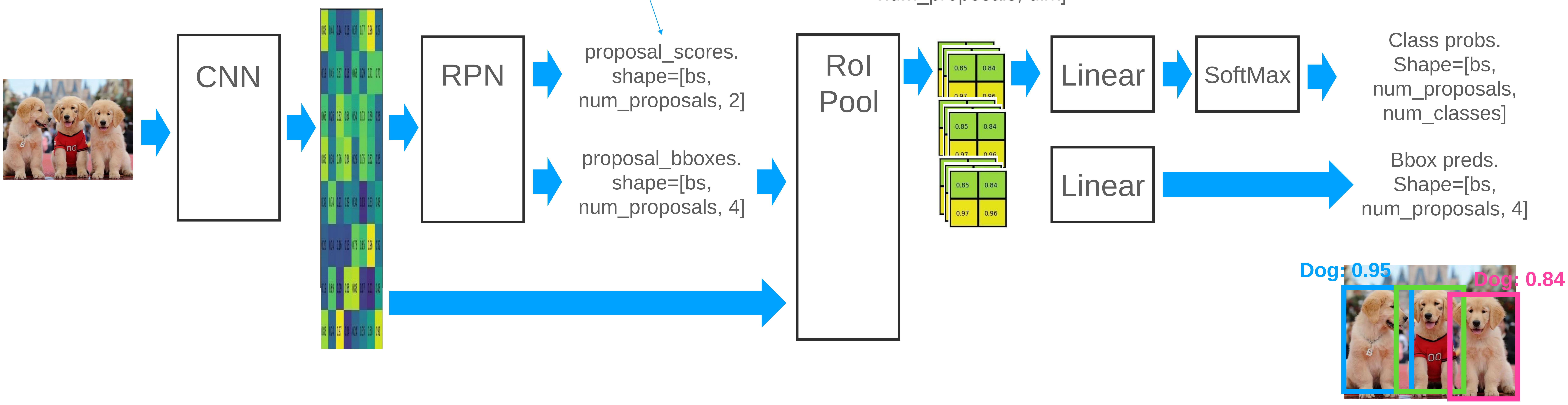
- Fortunately, we can define a backwards() for this that is reasonable.
  - Allows us to backprop through this! Amazing.
- Several variations exist, like:
  - Aggregators. Rather than max-ing each cell, take average? Median?
  - More accurate interpolating. Rather than having discrete cell edges (possibly causing mismatched cell sizes), use interpolation to calculate each cell's value
  - Generalize beyond boxes, eg: circles



https://deepsense.ai/region-of-interest-pooling-explained/

Faster R-CNN: putting it together

Takeaway: FRCNN is a neat case study to see one way to solve an important problem (object detection) via deep learning building blocks (eg Conv2d, Linear, etc)
More broadly, in this class we hope you will be able to learn the skills to come up with novel model archs to solve your own interesting problems!



Conv2d: Backwards

- Let's express conv2d forwards() as: Y = X * w + b, where:
  - Y: output, shape=[num_filters, height_out, width_out]
  - X: input, shape=[num_chans_in, height_in, width_in]
  - w: filters, shape=[num_filters, num_chans_in, height_filter, width_filter].
  - b: bias, shape=[num_filters]
  - "*" means convolution (more precisely, cross-correlation)
- To implement conv2d's backwards, we need to compute these terms:

$$\frac{dLoss}{dX}, \frac{dLoss}{dFilters}, \frac{dLoss}{dBias}$$

Recall: we're given $\frac{dLoss}{dY}$

Conv2d: Backwards: $\frac{dLoss}{dX}$

[dim(X),1] [dim(X), dim(Y)] [dim(Y), 1]

$$\frac{dLoss}{dX} = \frac{dY}{dX}\frac{dLoss}{dY}$$

We're already given this ("dout")

(1) Let's look at how the X's output each Y output (eg $\frac{dY}{dX}$):
(assume stride=1, padding=0, chans_in=1, chans_out=1)

$$w = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix}$$

$$Y = \begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} \qquad X = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}$$

$$y_{00} = x_{00}w_{00} + x_{01}w_{01} + x_{10}w_{10} + x_{11}w_{11}$$
$$y_{01} = x_{01}w_{00} + x_{02}w_{01} + x_{11}w_{10} + x_{12}w_{11}$$

$$\frac{dY_{00}}{dX_{ij}} = \begin{cases} w_{ij}, & if\ (i,j) \in filter\_inds(0,0) \\ 0, & otherwise \end{cases}$$

$$\frac{dY_{01}}{dX_{ij}} = \begin{cases} w_{ij}, & if\ (i,j) \in filter\_inds(0,1) \\ 0, & otherwise \end{cases}$$

"filter_inds(0,0)" means: the indices that the filter would have iterated over for y_00. Ex: for a 2x2 filter, filter_inds(0,0) = [[0,0], [0,1], [1,0], [1,1]]

$$\frac{dY_{00}}{dX} = \begin{bmatrix} w_{00} & w_{01} & 0 & \dots \\ w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & 0 \end{bmatrix}$$

$$\frac{dY_{01}}{dX} = \begin{bmatrix} 0 & w_{00} & w_{01} & 0 & \dots \\ 0 & w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & 0 \end{bmatrix}$$

(2) $\frac{dY}{dX}$ has a very specific structure (eg replicated filter weights w) AND is very sparse. Similar to Linear layer's $\frac{dL}{dW}$, let's take advantage of the structure to simplify calculation

Conv2d: Backwards: $\frac{dLoss}{dX}$

$$\frac{dLoss}{dX} = \frac{dY}{dX}\frac{dLoss}{dY}$$

We're already given this ("dout")

$$\frac{dLoss}{dX} = \sum_{i,j} \frac{dY_{i,j}}{dX}\frac{dL}{dY_{i,j}}$$

(by defn. of mat-mult)

$$= \begin{bmatrix} w_{00} & w_{01} & 0 & \dots \\ w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & 0 \end{bmatrix} \frac{dL}{dY_{00}} + \begin{bmatrix} 0 & w_{00} & w_{01} & 0 & \dots \\ 0 & w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & 0 \end{bmatrix} \frac{dL}{dY_{01}} + \dots$$

...hey, this is starting to look computable!

Conv2d: Backwards: $\frac{dLoss}{dX}$

$$\frac{dLoss}{dX} = \frac{dY}{dX}\underbrace{\frac{dLoss}{dY}}$$

We're already given this ("dout")

$$\frac{dLoss}{dX} = \sum_{i,j} \frac{dY_{i,j}}{dX}\frac{dL}{dY_{i,j}} = \begin{bmatrix} w_{00} & w_{01} & 0 & \dots \\ w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & 0 \end{bmatrix} \frac{dL}{dY_{00}} + \begin{bmatrix} 0 & w_{00} & w_{01} & 0 & \dots \\ 0 & w_{10} & w_{11} & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & 0 \end{bmatrix} \frac{dL}{dY_{01}} + \dots$$

```
Pseudo code:
init dloss_dx = np.zeros([batchsize, chans_in, height_in, width_in])
let filter have shape=[chans_in, filter_height, filter_width]
let dout (dLoss/dY), shape=[batchsize, chans_out, height_out, width_out]

for each location (i,j) in this conv layer's output:
    for ind_batch in range(batchsize):
        # filter_inds is list of pairs denoting the spatial locations
        # where the filter walks through the input. Ex: [[i,j], ...]
        filter_inds = compute_filter_inds(i, j, conv_param)
        for filter_i, filter_j in filter_inds:
            dloss_dx[ind_batch, :, filter_i, filter_j] += (
                filter[:, filter_i, filter_j] * dout[ind_batch, :, i, j]
            )
# Tip: make sure that compute_filter_inds() handles params like: pad, stride
```

Aside: this is a "direct" way of computing dLoss/dX. **(Optional)** Another neat way involves recognizing that dLoss/dX can be implemented as a convolution between a "rotated" w and dout! To see how this is done, see this: [link]

Conv2d: Backwards: $\frac{dLoss}{dw}$

Tip: for the following, "O" is Y ("output"), and "F" is w ("filter")

Loss gradient w.r.t the filter

We can expand the chain rule summation as:

$$\frac{dLoss}{dw} = \frac{dY}{dw}\frac{dLoss}{dY}$$

$$\frac{dLoss}{dw_{i',j'}} = \sum_{i,j} \frac{dY_{i,j}}{dX_{i',j'}}\frac{dL}{dY_{i,j}}$$

(by defn of mat-vector mult.)
Note: we're using denominator format for partial derivative here
(orig slides uses numerator format)

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{11}}$$

$$\frac{\partial L}{\partial F_{12}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{12}}$$

$$\frac{\partial L}{\partial F_{21}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{21}}$$

$$\frac{\partial L}{\partial F_{22}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{22}}$$

Assume 2x2 filter, no padding, stride=1, 1 channel

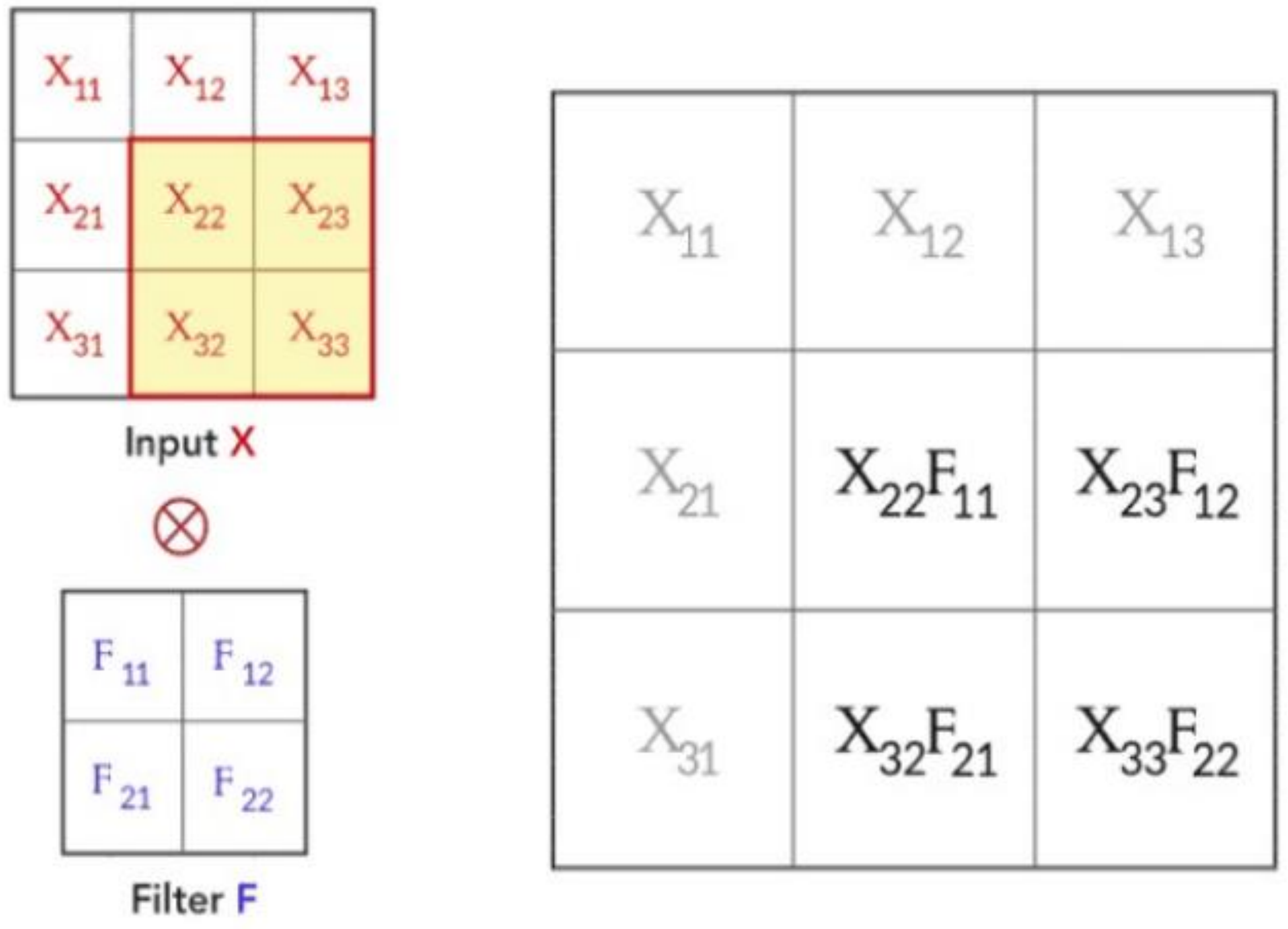
https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf

Conv2d: Backwards: $\frac{dLoss}{dw}$

Side quest: what is $\frac{dO}{dF}$ (aka $\frac{dY}{dw}$)?

Convolution Forward Pass

- Convolution between Input X and Filter F, gives us an output O. This can be represented as:



$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

$$O_{21} = X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22}$$

$$O_{22} = X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}$$

Interpretation: For a given output location O_ij, each filter value F_ij interacts (via element-wise mult) only with a single value in X

$$\frac{dO_{11}}{dF_{11}} = X_{11} \qquad \frac{dO_{11}}{dF_{12}} = 0, \dots$$

$$\frac{dO_{12}}{dF_{11}} = 0 \qquad \frac{dO_{12}}{dF_{12}} = X_{12}, \dots$$

https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf

Conv2d: Backwards: $\frac{dLoss}{dw}$

Loss gradient w.r.t the filter

- Replacing the local gradients of the filter i.e, $\frac{\partial O_i}{\partial F_i}$, we get this:

$$\begin{bmatrix} \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} \right)$$

where

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \text{Input X} \qquad \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} = \frac{\partial L}{\partial O}$$

Loss gradient from previous layer

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * X_{11} + \frac{\partial L}{\partial O_{12}} * X_{12} + \frac{\partial L}{\partial O_{21}} * X_{21} + \frac{\partial L}{\partial O_{22}} * X_{22}$$

$$\frac{\partial L}{\partial F_{12}} = \frac{\partial L}{\partial O_{11}} * X_{12} + \frac{\partial L}{\partial O_{12}} * X_{13} + \frac{\partial L}{\partial O_{21}} * X_{22} + \frac{\partial L}{\partial O_{22}} * X_{23}$$

$$\frac{\partial L}{\partial F_{21}} = \frac{\partial L}{\partial O_{11}} * X_{21} + \frac{\partial L}{\partial O_{12}} * X_{22} + \frac{\partial L}{\partial O_{21}} * X_{31} + \frac{\partial L}{\partial O_{22}} * X_{32}$$

$$\frac{\partial L}{\partial F_{22}} = \frac{\partial L}{\partial O_{11}} * X_{22} + \frac{\partial L}{\partial O_{12}} * X_{23} + \frac{\partial L}{\partial O_{21}} * X_{32} + \frac{\partial L}{\partial O_{22}} * X_{33}$$

https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf

Conv2d: Backwards: $\frac{dLoss}{dw}$

Loss gradient w.r.t the filter

- If you closely look at it, this represents an operation we are quite familiar with. We can represent it as a **convolution operation between input X** and loss gradient **∂L/∂O as shown below:**

$$\begin{bmatrix} \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} \right)$$

where

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \text{Input X} \qquad \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} = \frac{\partial L}{\partial O} \text{ Loss gradient from previous layer}$$

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * X_{11} + \frac{\partial L}{\partial O_{12}} * X_{12} + \frac{\partial L}{\partial O_{21}} * X_{21} + \frac{\partial L}{\partial O_{22}} * X_{22}$$

$$\frac{\partial L}{\partial F_{12}} = \frac{\partial L}{\partial O_{11}} * X_{12} + \frac{\partial L}{\partial O_{12}} * X_{13} + \frac{\partial L}{\partial O_{21}} * X_{22} + \frac{\partial L}{\partial O_{22}} * X_{23}$$

$$\frac{\partial L}{\partial F_{21}} = \frac{\partial L}{\partial O_{11}} * X_{21} + \frac{\partial L}{\partial O_{12}} * X_{22} + \frac{\partial L}{\partial O_{21}} * X_{31} + \frac{\partial L}{\partial O_{22}} * X_{32}$$

$$\frac{\partial L}{\partial F_{22}} = \frac{\partial L}{\partial O_{11}} * X_{22} + \frac{\partial L}{\partial O_{12}} * X_{23} + \frac{\partial L}{\partial O_{21}} * X_{32} + \frac{\partial L}{\partial O_{22}} * X_{33}$$

https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf

Conv2d: Backwards: $\frac{dLoss}{dw}$

```
x, w, b, conv_param = cache
dw = np.zeros_like(w)
stride, pad = conv_param['stride'], conv_param['pad']
N, C, H, W = x.shape
F, C, HH, WW = w.shape

# dout: (N, F, H, W)
# Tip: pad x to simplify downstream index calculations.
x = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode='constant')

for h in range(1 + (H + 2 * pad - HH) // stride):
    for w in range(1 + (W + 2 * pad - WW) // stride):
        for n in range(N):
            for f in range(F):
                for c in range(C):
                    # i1_x, j1_x denote the spatial location in (padded) x that
                    # correspond to location (h,w) in the output Y
                    # Tip: since x is already padded, we don't have to consider
                    #   padding in this index calculation
                    i1_x = h * stride
                    j1_x = w * stride
                    # compute element-wise mult (aka cross-corr) between X and dout
                    x_region = x[n, c, i1_x:h*stride + HH, j1_x:w*stride+WW]
                    dw[f, c, :, :] += (x_region * dout[n, f, h, w])
```

Tip: this might be a little slow, so vectorizing it will help a lot with performance.
Note: writing performant vectorized numpy code is NOT the focus of this course