Machine Learning

Convolutional Neural Networks

Gabriele Russo Russo Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Motivation: Image Classification

- Suppose we are asked to classify images of cats and dogs
- Consider a dataset of 1-megapixel labeled photographs
  - Each input instance has 1 million pixels
- Even an aggressive reduction to 1,000 hidden units would require a fully connected layer with $10^6 \times 10^3 = 10^9$ parameters
  - Training would be extremely difficult or even unfeasible

Motivation (2)

- Recall the “Fashion MINST” example, where we classified images using a NN
- Image = 2D grid of pixels
- We ignored this structure and *flattened* images into vectors
  - We could even shuffle the vector elements, as the NN was invariant to feature order
- How to leverage prior knowledge that nearby pixels are typically related to each other?

Convolutional Neural Networks (CNN)

LeCun et al., "Backpropagation applied to handwritten zip code recognition". Neural Computation (1989)

- Convolutional Neural Networks (CNN): specialized NNs for processing data that has a known grid-like topology, e.g.:
  - time-series data (1D topology)
  - images (2D grids of pixels)
- "Convolutional" because of the convolution operation used in the NN
- A CNN is a neural network that uses convolution in at least one layer

Key Ideas

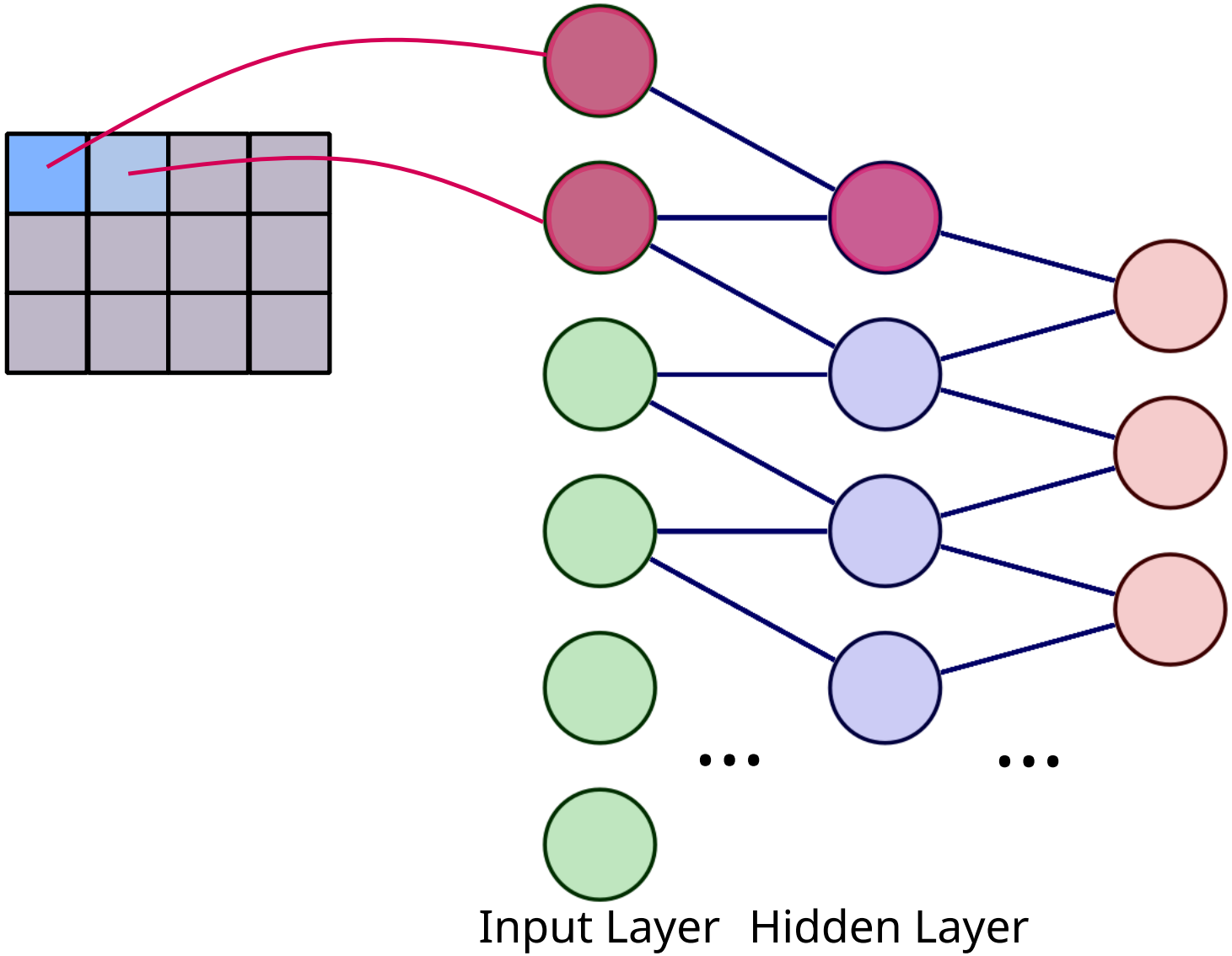
CNNs leverages 3 important ideas to overcome the limitations mentioned above:

- sparse interactions
- parameter sharing
- equivariant representations

Sparse Interactions

- So far, we have considered *fully connected* layers
  - every unit interacts with every input unit
- With a large number of units (as in the motivating example at the beginning), this approach leads to a huge number of parameters!
- Idea: let each unit interact with a reduced portion of the input

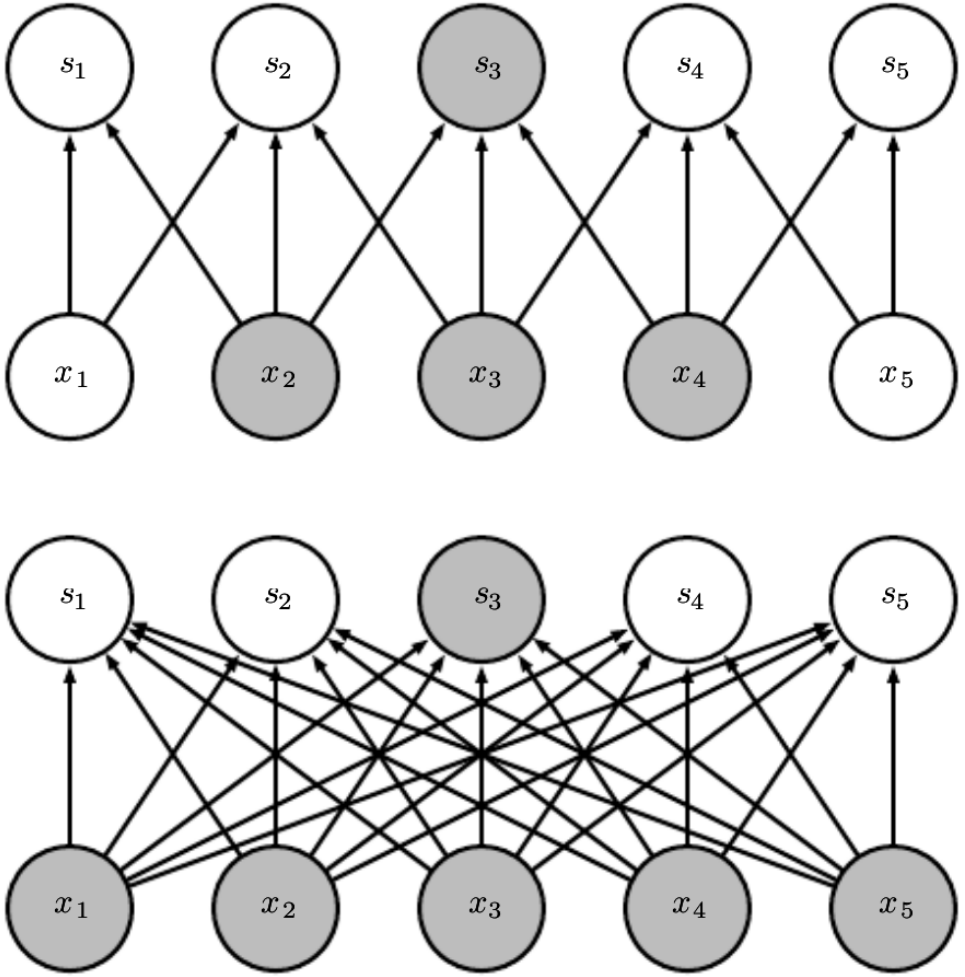
Example: Sparse Connections



Connections & Receptive Field

- With sparse connections, it seems that we loose the ability of detecting relevant interactions between distant portions of the input
- Let's consider the receptive field of each unit: the set of inputs that impact the output of a given unit

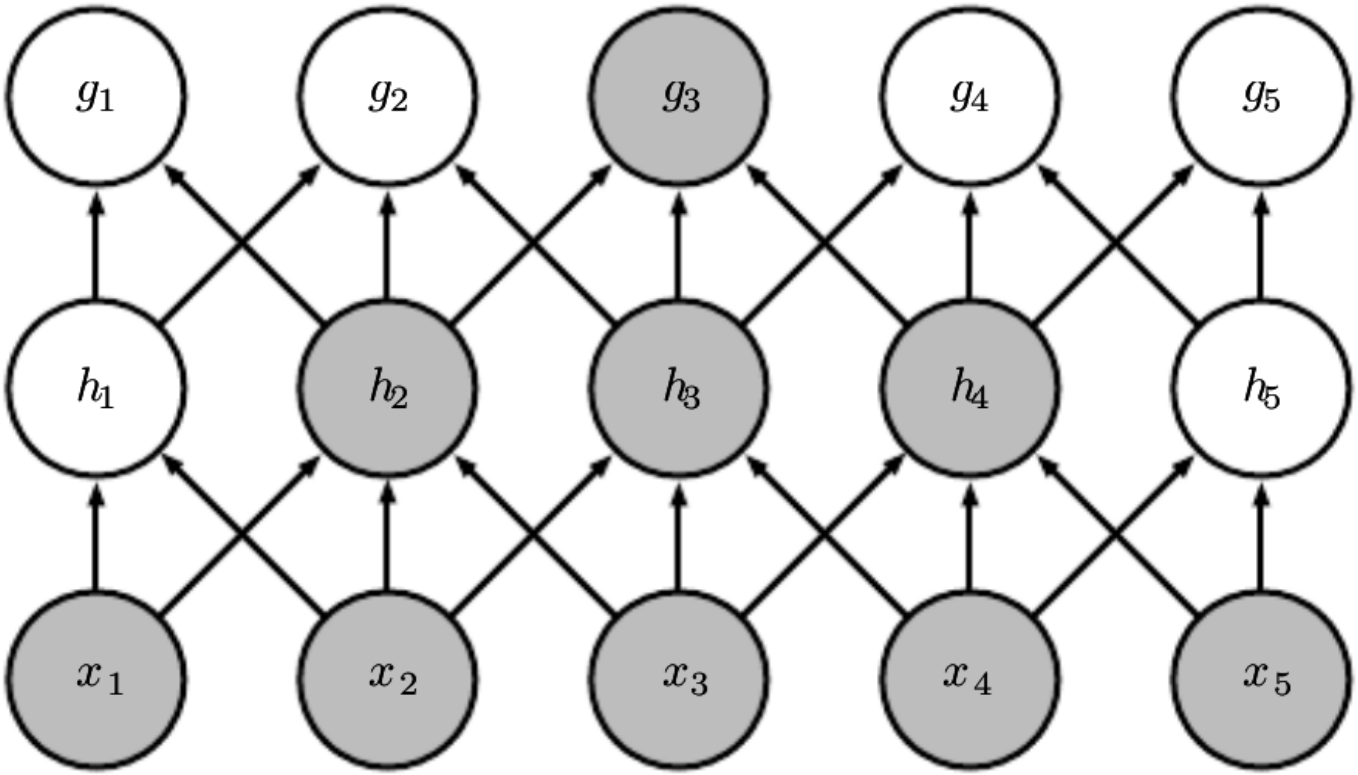
Example: Receptive Field



Sparse Connections

Fully Connected

Example: Receptive Field with More Layers

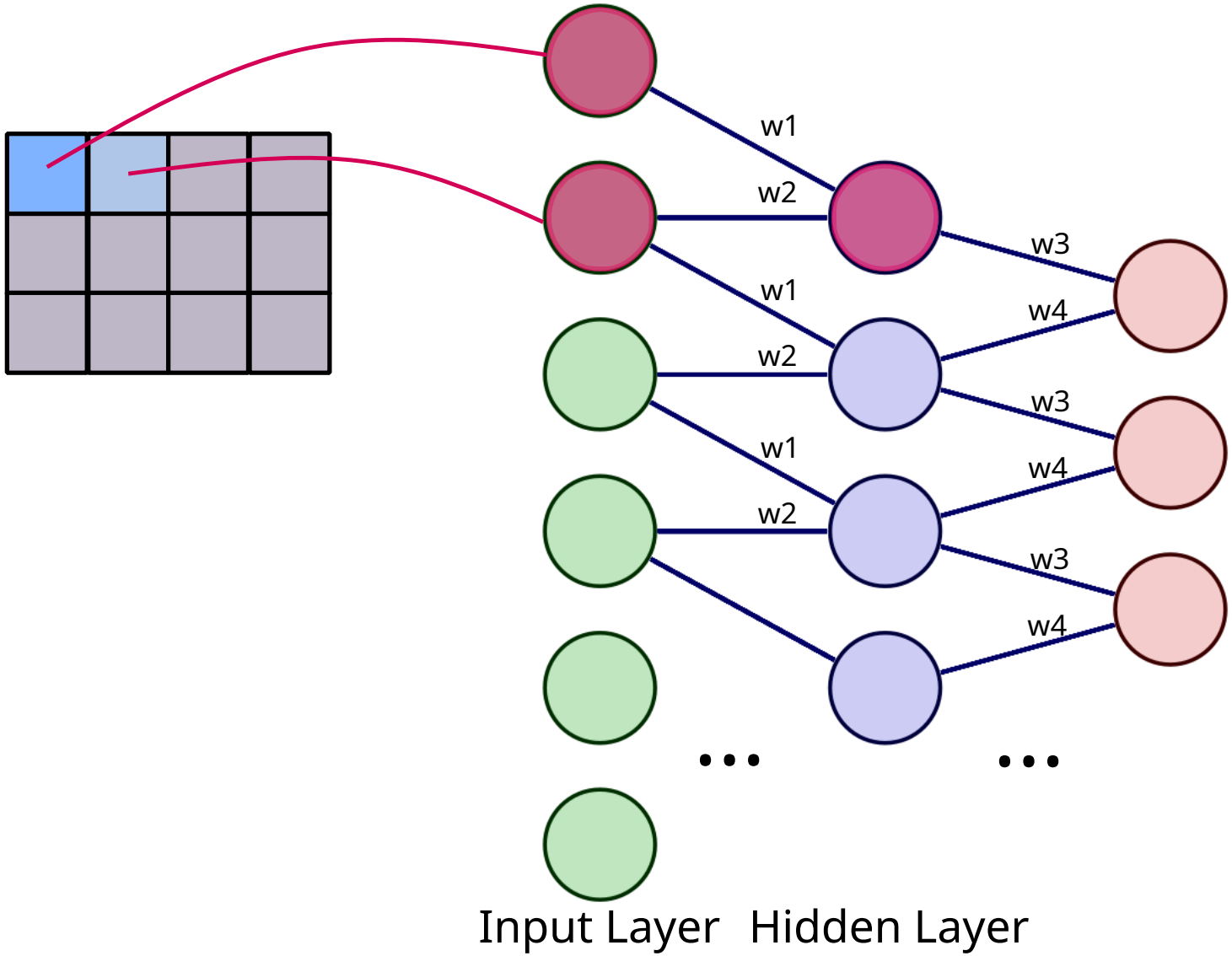


Despite sparsity of *direct* connections, units in higher layers *indirectly* interact with all or most of the input

Parameter Sharing

- In a traditional NN, each pair of connected units has its own weight w_{ij}
- Convolutional layers re-use the same set of parameters to compute the output of each unit
  - the same parameters are shared across all the units
- While the complexity of backpropagation does not change, we have important benefits
- Reduced memory requirement!
- Reduced number of parameters to learn!

Example

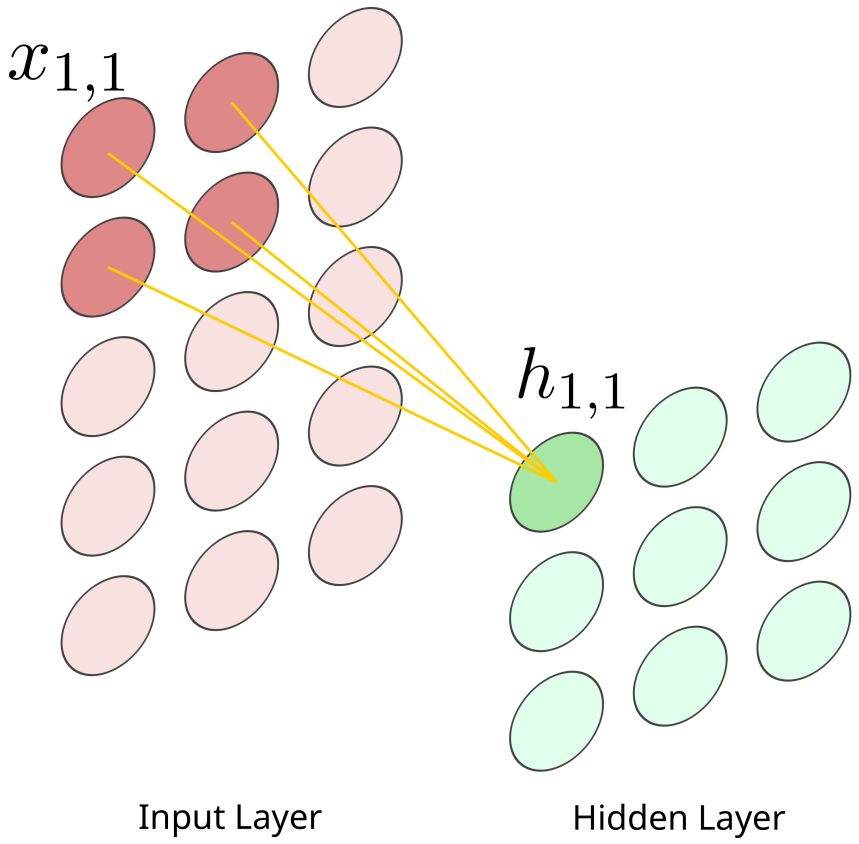


Equivariant Representation

- Parameter sharing (as defined above) causes the layer to have a property called equivariance to translation
- Because the same parameters are used across different parts of the input, if we move an object in the input, its representation will move the same amount in the output
- A traditional NN can completely fail if we change the order in which features appear!
- Clearly, parameter sharing does not make CNNs equivariant to some other transformations, such as changes in the scale or rotation of an image

Towards Convolutional Layers

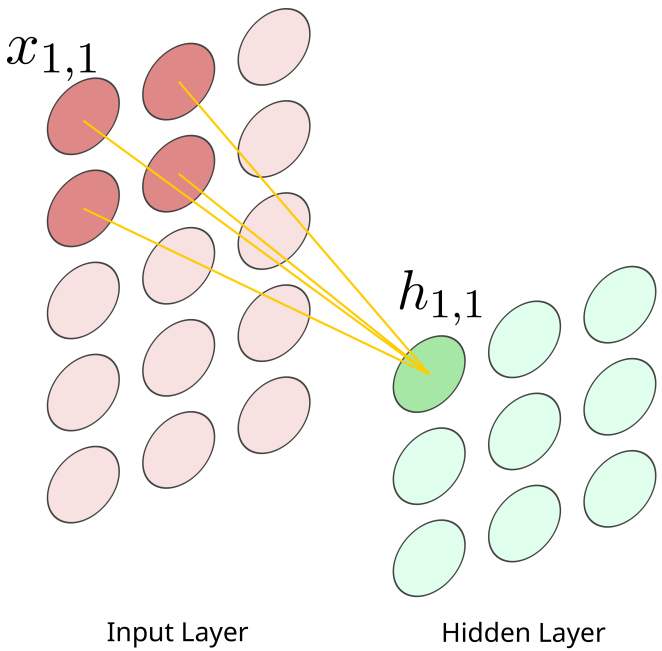
- When working with grid data (e.g., images), it is useful to think of hidden units being arranged in a grid as well
- Each unit still connected to a small portion of the input



Towards Convolutional Layers (2)

$$h_{1,1} = k_{1,1}x_{1,1} + k_{1,2}x_{1,2} + k_{2,1}x_{2,1} + k_{2,2}x_{2,2}$$

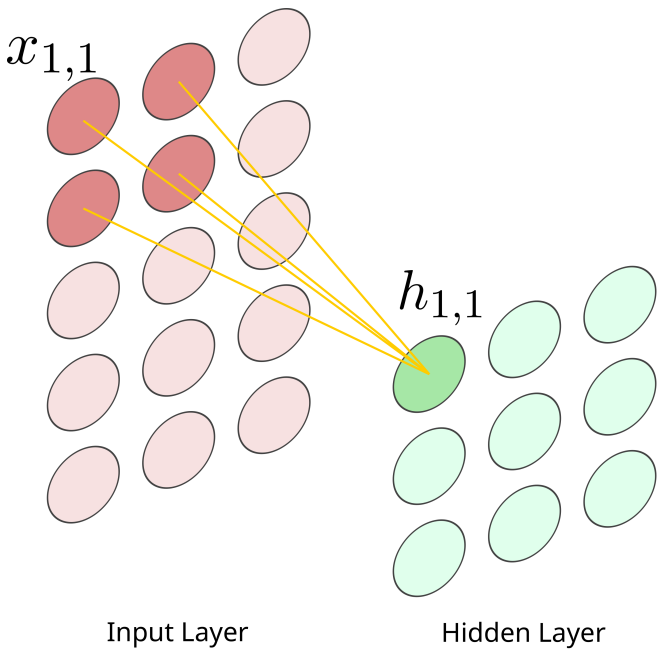
$$h_{i,j} = k_{1,1}x_{i,j} + k_{1,2}x_{i,j+1} + k_{2,1}x_{i+1,j} + k_{2,2}x_{j+1,i+1}$$



- Params $k_{i,j}$ form matrix K (similar to the W used in fully connected layers)
- Called kernel or filter

Example

$$K = \begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$



- What are we computing?
- We are just averaging the value of 4 neighboring pixels

Convolution

- The operation of convolutional layers can be mathematically described as a convolution applied to the input tensor and the kernel
- E.g., for a 2D image $\boldsymbol{X}$ as input, we use a 2D kernel $\boldsymbol{K}$

$$(X * K)(i, j) = \sum_a \sum_b K(a, b) X(i + a, j + b)$$

Convolution vs. Cross-Correlation

- Actually, convolution on discrete data is defined as:

$$S(i,j) = \sum_a \sum_b K(a,b) X(i-a, j-b)$$

- Most the ML libraries use the term "convolution" but actually implement cross-correlation

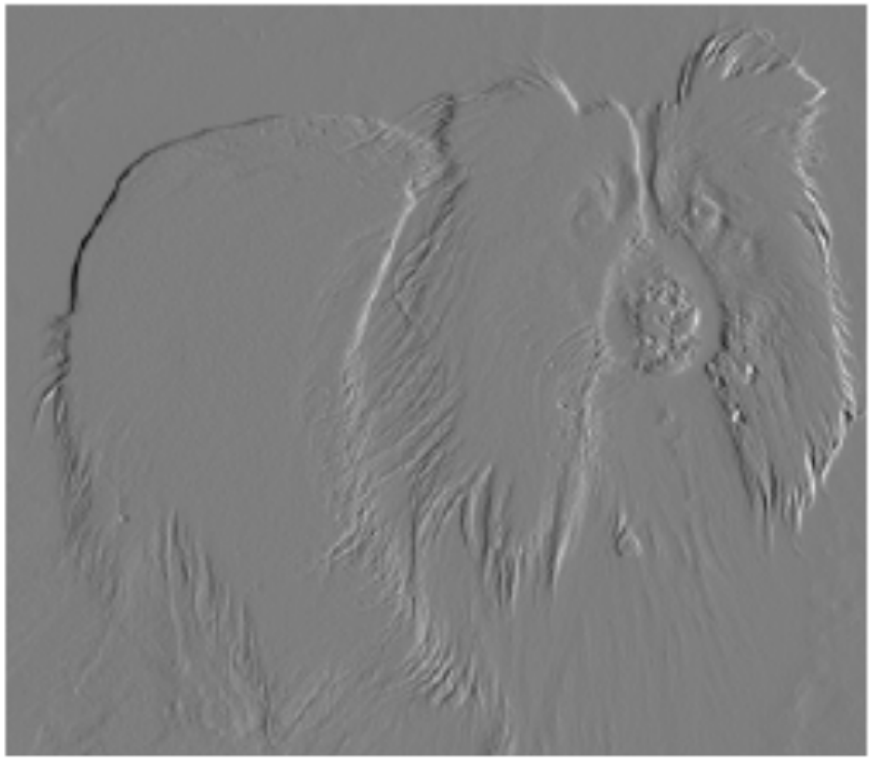
$$S(i,j) = \sum_a \sum_b K(a,b) X(i+a, j+b)$$

- Besides losing commutative property with cross-correlation, there is no impact in practice because we are going to *learn* the kernel (flipped or not)

Basic Convolutional Layer

- Let's use convolution to construct a hidden layer
- A convolutional layer takes a tensor $\boldsymbol{X}$ in input and produces an output tensor $\boldsymbol{H}$
- $\boldsymbol{H}$ computed by calculating convolution $\forall(i, j)$
  - We are skipping many details at this point, including the use of a nonlinear activation function
- The kernel $\boldsymbol{K}$ comprises the parameters of the layer (to be trained)

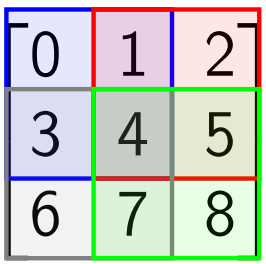
Example: Detecting Vertical Edges



A 2-pixel kernel is enough: $K = [1, -1]$

Example

Input X



Kernel K

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

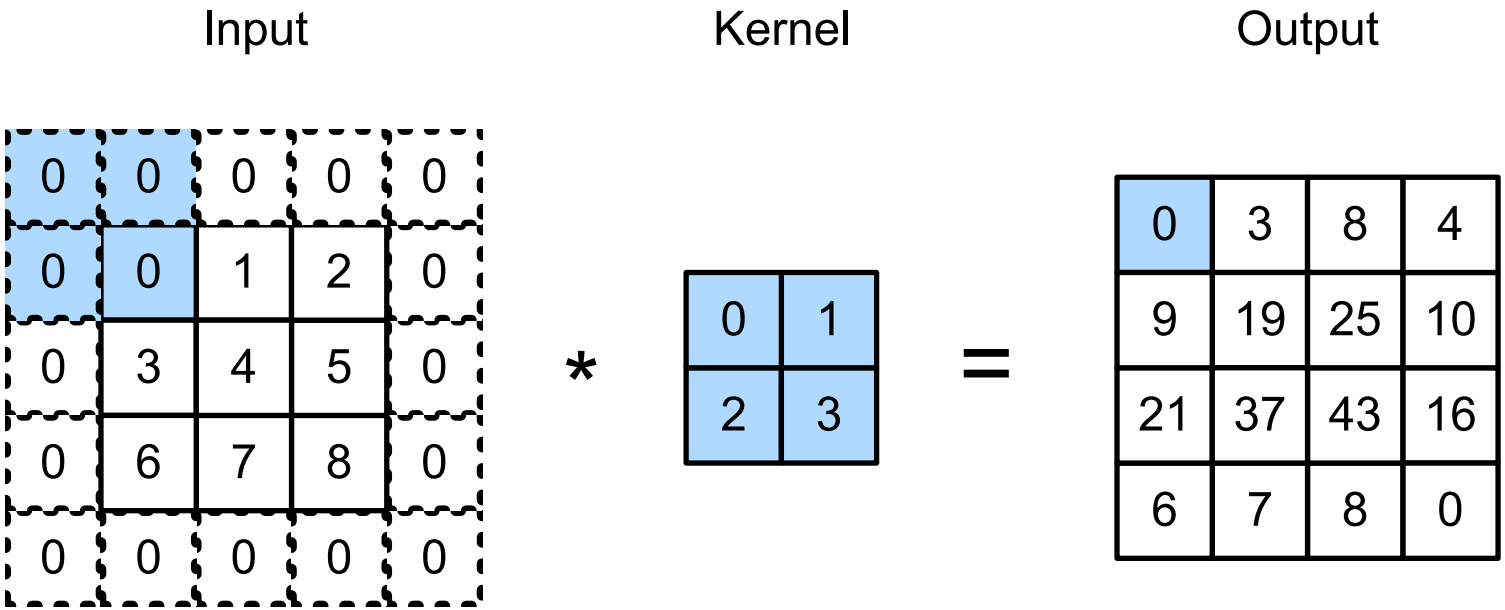
H

$$\begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$$

- We started with a 3x3 input and got a 2x2 output
- What if we want to stack N convolutional layers??

Zero Padding

- To solve this problem, we can add extra pixels of filler around the boundary of our input (padding)
- Typically, we set the values of the extra pixels to zero



Zero Padding (2)

- With input of size $n_h \times n_w$ and kernel of size $k_h \times k_w$, output has shape $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- Adding p_h rows and p_w columns of padding, we get: $(n_h + p_h - k_h + 1) \times (n_w + p_w - k_w + 1)$
- To give input and output the same size, usually we set $p_h = k_h - 1$ and $p_w = k_w - 1$
- CNN kernels usually have odd width and height
  - If k_h is odd, we exactly add $\frac{p_h}{2}$ rows both on top and bottom (similarly for columns)
  - As an additional benefit, the output element $\boldsymbol{H}_{ij}$ is calculated with the convolution window centered on $\boldsymbol{X}_{ij}$

Example (with padding)

Input (3x4)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 0 & 0 \\ 7 & 8 & 9 & 1 \end{bmatrix}$$

With Padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 2 & 3 & 4 & 0 \\ 0 & 2 & 5 & 0 & 0 & 0 \\ 0 & 7 & 8 & 9 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel K

$$\begin{bmatrix} a & b & c \\ c & a & b \\ e & c & f \end{bmatrix}$$

$$(K * I)(1,1) = \ldots$$

Convolution as Matrix Multiplication

- Convolution (and cross-correlation) can be regarded as traditional matrix multiplication
- The matrices to multiply must be constructed from the original input and kernel
- For instance, consider the following input A and kernel K (without padding):

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \qquad K = \begin{bmatrix} k_{1,1} & k_{1,2} \\ k_{2,1} & k_{2,2} \end{bmatrix}$$

We construct a matrix M from K and a vector v from A

$$M = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

$$v = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{2,1} & a_{2,2} & a_{2,3} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$Mv^T = \begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{1,3} \cdot 0 + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} + a_{2,3} \cdot 0 + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,1} \cdot 0 + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{2,3} \cdot 0 + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} + a_{2,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1} \cdot 0 + a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,1} \cdot 0 + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Reshaping into a 2x2 matrix we get the result:

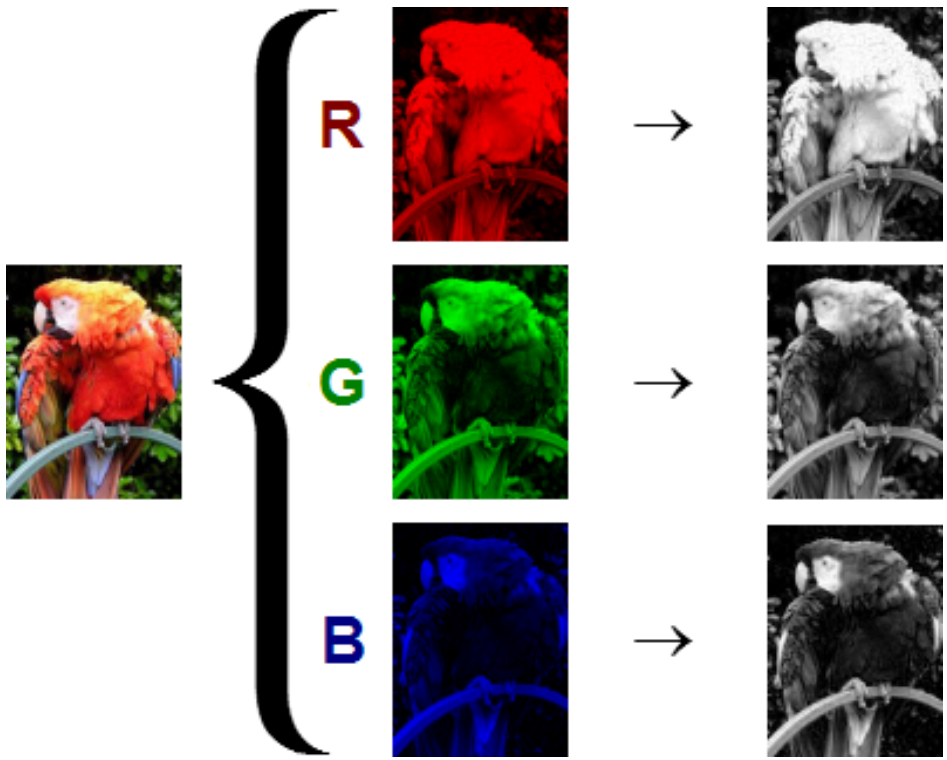
$$\begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} & a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} \\ a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} & a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Convolution as Matrix Multiplication (2)

- Performing convolution as matrix multiplication does not look much more efficient (larger matrices are involved!)
- However, this interpretation shows that conceptually convolutional layers are not different than fully connected layers, in terms of mathematical operations involved
- Therefore, we are guaranteed that backpropagation can still be used to compute gradients!

Channels

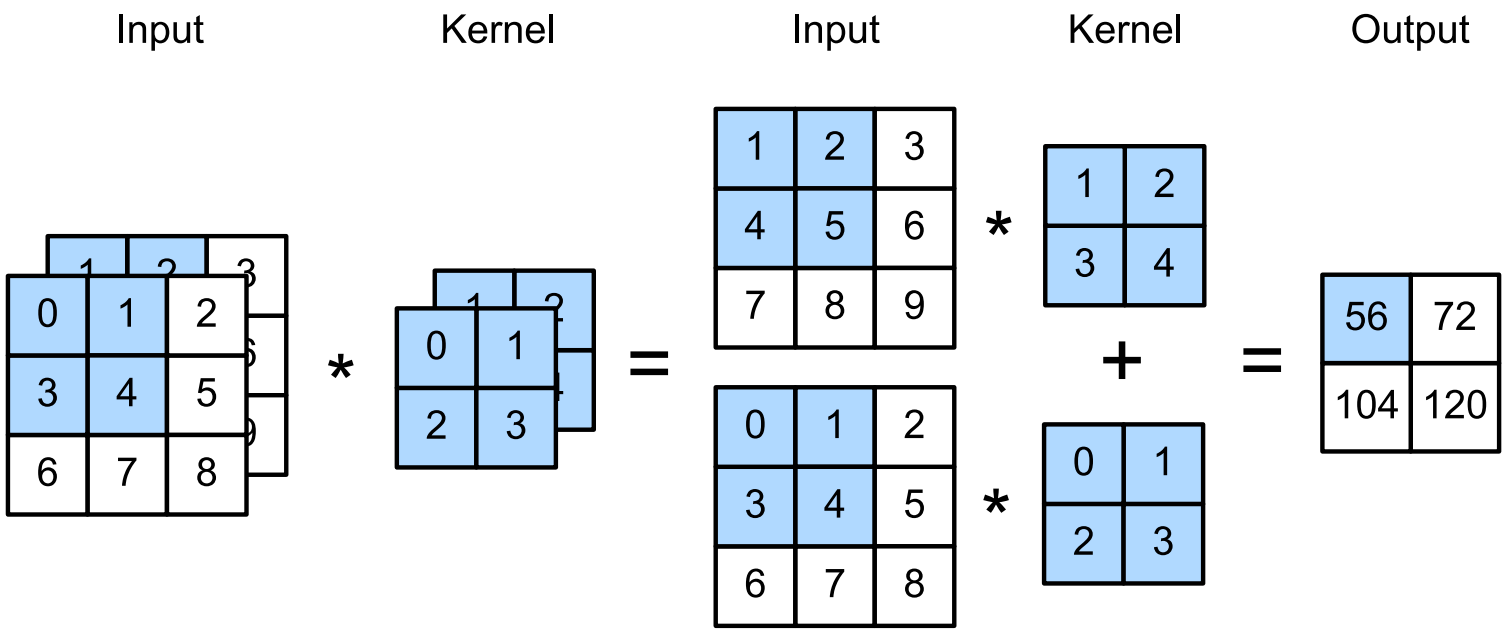
- Images usually consist of multiple channels, usually three: red, green and blue (RGB)
- Represented as 3D tensors with shape: $3 \times h \times w$



Multiple Input Channels

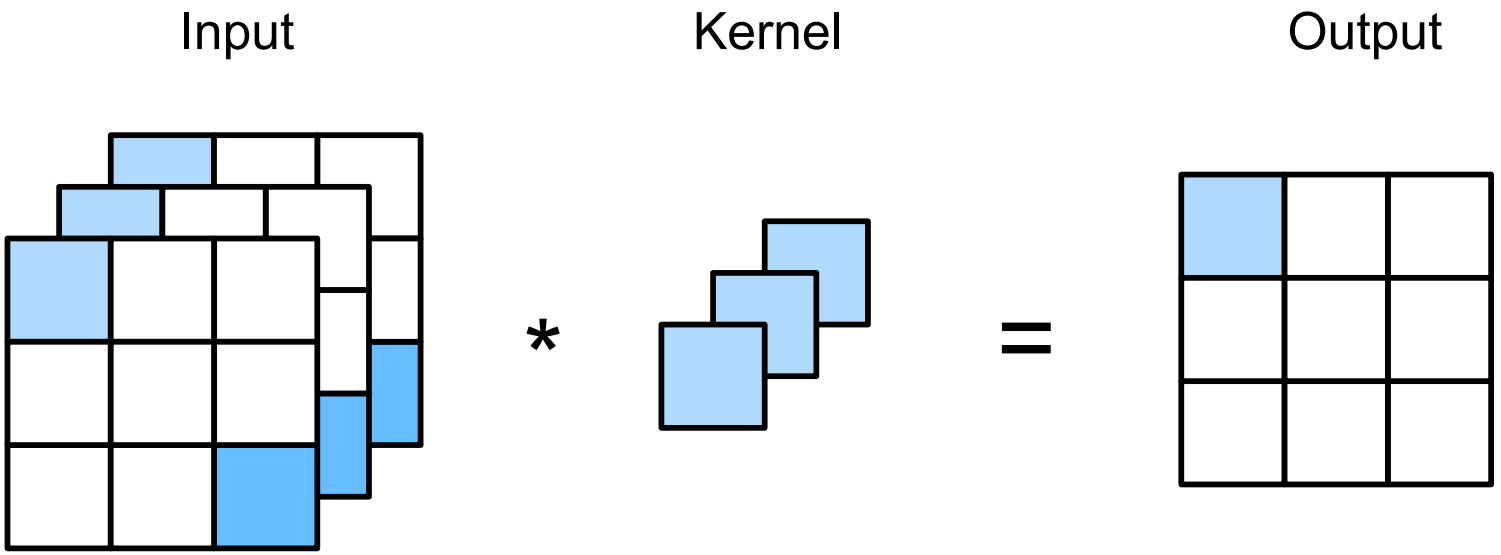
With $c_i > 1$ input channels, we need a 3D kernel as well (but we still get a 2D output tensor)

$$H_{i,j} = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$$



Example: 1x1 kernel

- Apparently, a 1×1 kernel does not make much sense, as it cannot correlate any pixels!
- With multi-channel input, a $1 \times 1 \times c_i$ kernel can be actually useful
  - e.g., to average the value of a pixel across the channels



Multiple Output Channels

- Regardless of the number of input channels, so far we always ended up with 1 output channel
- In practice, a convolutional layer applies multiple kernels at the same time, each leading to an output channel
  - Different kernels can discover different things in the input
- Having c_o kernels with shape $k_h \times k_w \times c_i$, equivalent to a single kernel with shape $k_h \times k_w \times c_i \times c_o$

$$H_{i,j,k} = \sum_a \sum_b \sum_c K_{a,b,c,k} X_{i+a,j+b,c}$$

Convolutional Layer: Revisited

- Tensor $\boldsymbol{X}$ in input, with shape $n_h \times n_w \times c_i$
- Kernel $\boldsymbol{K}$ with shape $k_h \times k_w \times c_i \times c_o$ comprises the parameters of the layer
  - Plus a bias term for every kernel
- Output tensor $\boldsymbol{H}$ with shape $n_h \times n_w \times c_o$
  - Assuming zero padding is used as described above
- $\boldsymbol{H}$ computed by calculating convolution with $\boldsymbol{K}$ at every input position, and possibly applying a nonlinear activation function (e.g., ReLU)

$$H_{i,j,k} = \phi\left(\sum_{u,v,c} K_{u,v,c,k} X_{i+u,j+v,c} + b_k\right)$$

where b_k is the bias term for the k-th output channel.

Convolution with Stride

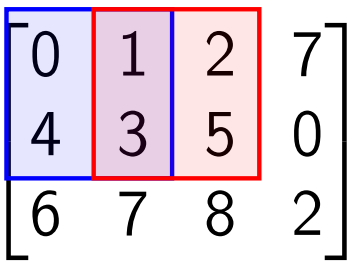
- When computing cross-correlation, the convolution window starts from the upper-left corner of the input, and then slides over all locations down and to the right
  - So far, we defaulted to sliding one element at a time
- Sometimes we prefer to move the window more than one element at a time, skipping intermediate locations
  - Computational efficiency
  - Downsampling the input image
- Stride = number of rows and columns traversed per slide
  - e.g., with stride 2, we halve both width and height of the image

Example

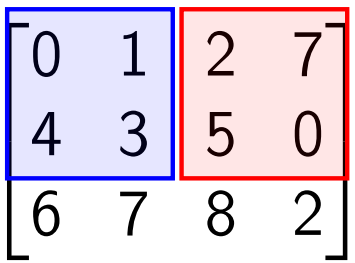
Kernel K

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

Stride 1



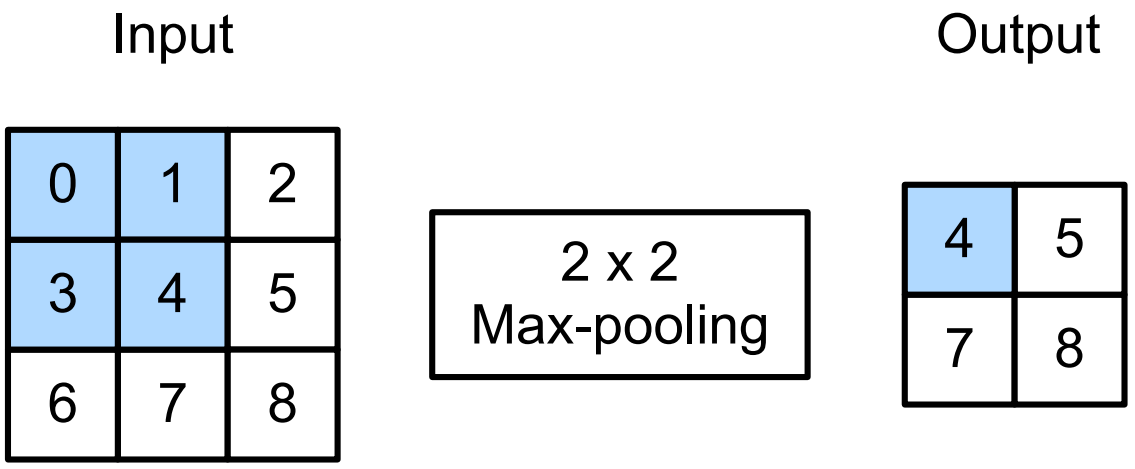
Stride 2



Pooling

- Convolutional layers are usually followed by pooling layers
  - Some books just consider convolution and pooling distinct steps of the same “complex” layer
- Pooling operator: fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed
- Similar to convolution, but with no parameters!
- Pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window
  - “max pooling” and “average pooling” (for short)

Example



Pooling (2)

- In presence of multiple channels, pooling is applied to each channel separately
  - Output tensor has the same number of channels as the input
- Since pooling aggregates information from an area, it is frequent to match pooling window size and stride
  - with a 3×3 window, stride is set to 3

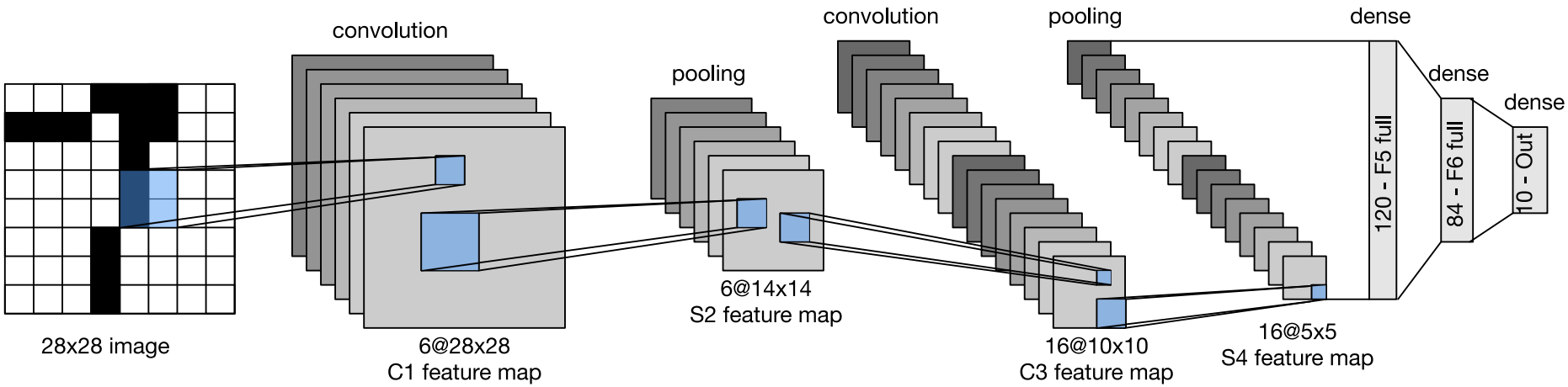
LeNet-5 (1998)

- We are now ready to assemble a fully-functional CNN
- LeNet-5 (or simply, LeNet) is one of the first published CNNs
  - Introduced by (and named for) Yann LeCun, after a decade of research on CNNs
  - Received wide attention for its performance
- Initially aimed at recognizing handwritten digits
  - Matched the performance of support vector machines, then a dominant approach in supervised learning, achieving an error rate of less than 1% per digit
  - Eventually adapted to recognize digits for deposits in ATM machines (still used in some ATMs today!)

LeNet: Architecture

At a high level, LeNet consists of two parts:

- a convolutional encoder consisting of 2 convolutional layers
- a dense block consisting of 3 fully connected layers



tf_lenet.ipynb

Example: LeNet and Fashion MNIST

- We consider again the Fashion MNIST dataset we have used in the previous lectures
- Instead of using a MLP, this time we use LeNet

tf_lenet_fashion.ipynb

Beyond LeNet

- Most of the modern successful CNN models deeply inspired by LeNet-5
- No major advancements until 2010s
  - Waiting for more data and hardware!
- ImageNet Large Scale Visual Recognition Challenge[1]: barometer of progress on supervised learning in computer vision since 2010
  - From 27% to 2% error in 5-6 years
  - We review some CNN architectures that were winners or runners-up

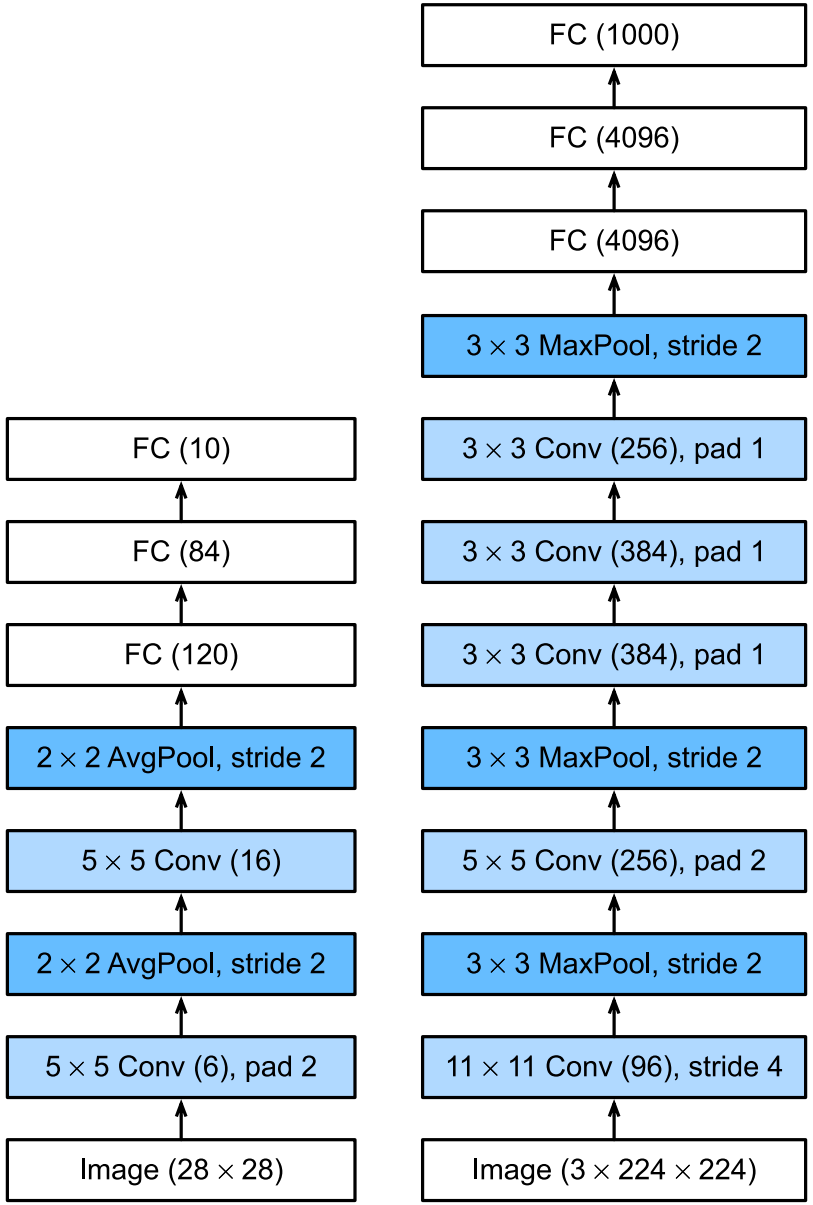
[1] `https://www.image-net.org/challenges/LSVRC/index.php`

AlexNet (2012)

- Won the ImageNet competition in 2012 by a large margin
- Architecture similar to LeNet, with important differences:
  - Deeper (8 layers)
  - Larger layers (e.g., dense layers require 160 MB of weights)
  - Larger and multi-channel input images
  - ReLU replaces sigmoid
  - Dropout
- Originally trained for 6 days on two NVIDIA GeForce GTX 580 (2010 GPU)


tf_alexnet_fashion.ipynb (GPU recommended!)

AlexNet vs. LeNet

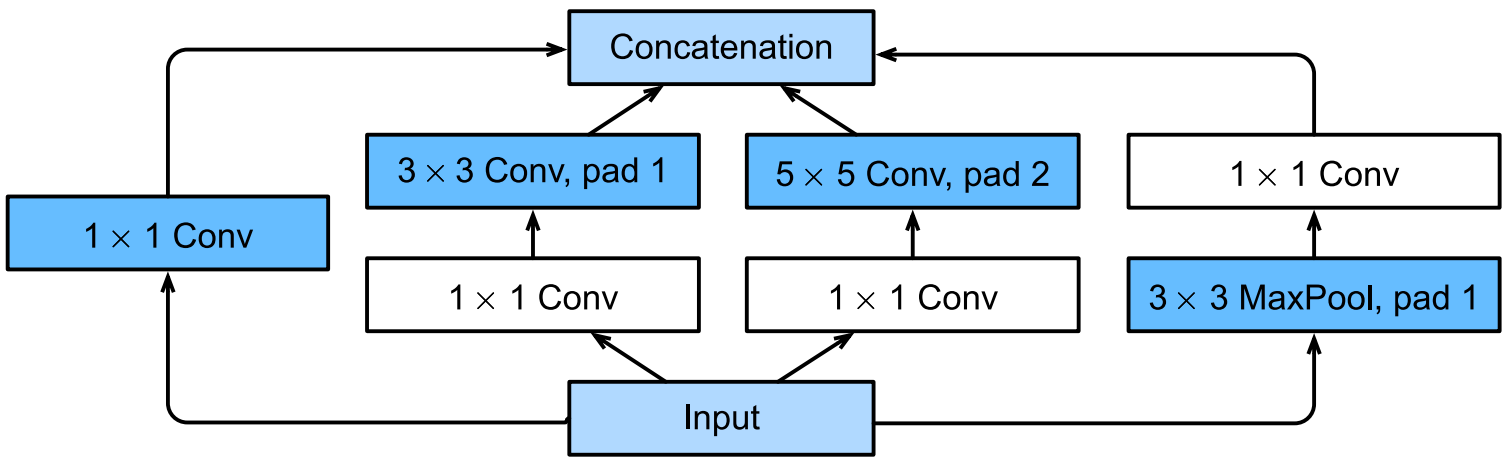


GoogLeNet (2014)

- Winner of the ImageNet competition in 2014
- Clear distinction of:
  - Stem (data ingestion): first 2-3 conv. layers
  - Body (data processing): other convolutional blocks
  - Head (prediction)
- We see a slightly simplified version of GoogLeNet: the original design included a number of tricks for stabilizing training, no longer necessary due to the availability of improved training algorithms

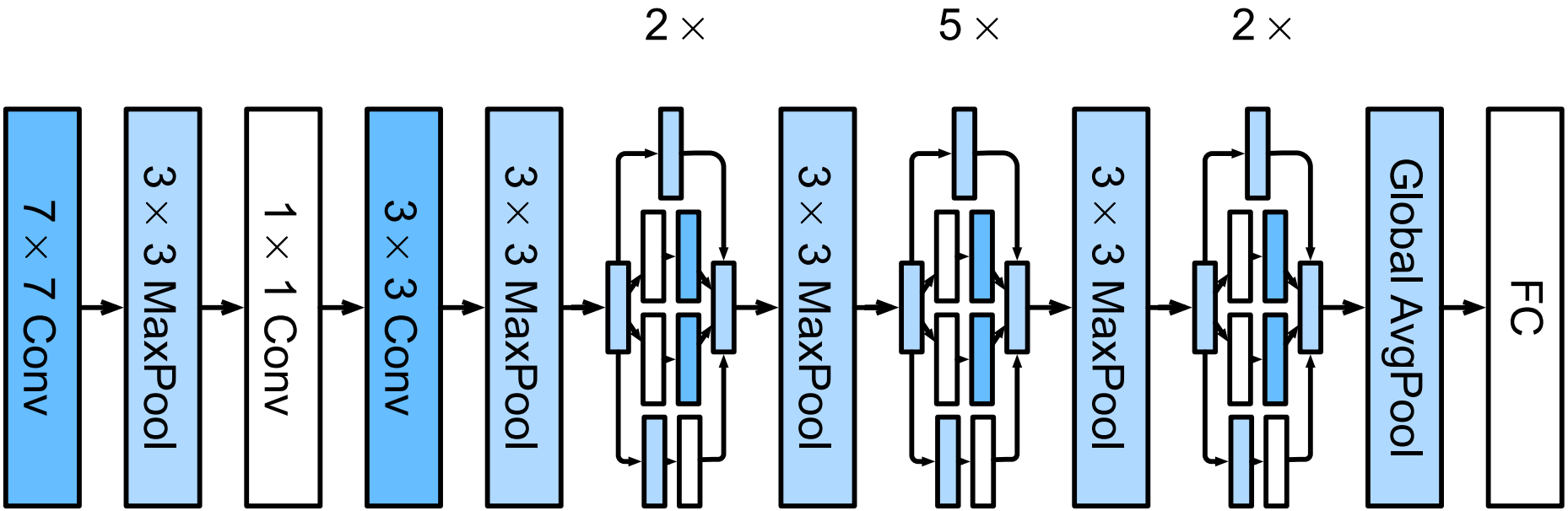
GoogLeNet: Inception Block

- Key contribution of GoogLeNet
- Other works tried to identify the best convolution kernel (from 1×1 to 11×11)
- GoogLeNet simply concatenates multi-branch convolutions (Inception block)



GoogLeNet: Architecture

- A total of 9 Inception blocks
- *Global pooling* at the end to reduce each channel to a single value
- Fewer parameters than AlexNet

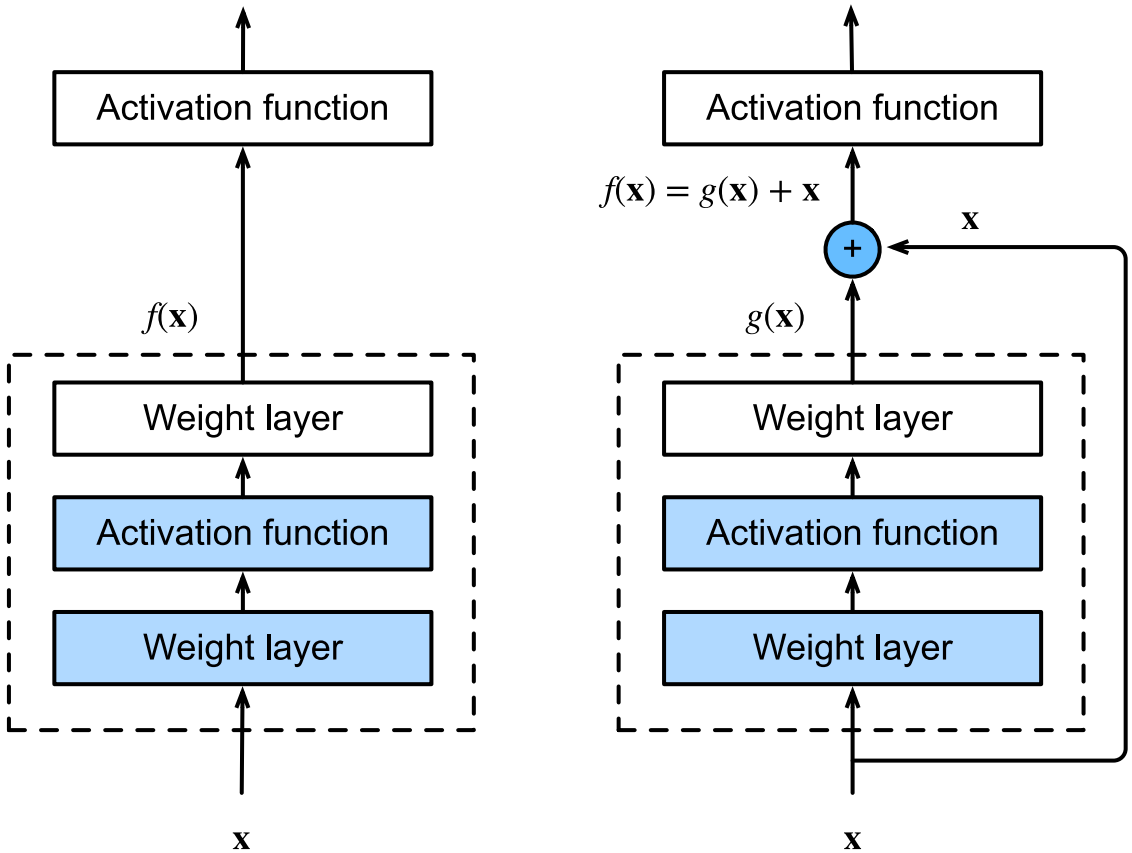


ResNet (2015)

- Winner of the ImageNet competition in 2015
- Deeper model (100+), with fewer parameters
  - Trend already started in previous years
- In 2015, ResNet-50 trained in about 10 days (<20 minutes today on several GPUs)
- Key idea: skip connections (or, shortcut connections)
  - Output of a layer re-used at a higher point in the stack
- Used to create residual blocks

Residual Block

- Let $f(x)$ be the function we aim to model
- With a residual block, we learn $g(x)$ and then compute $f(x) = g(x) + x$

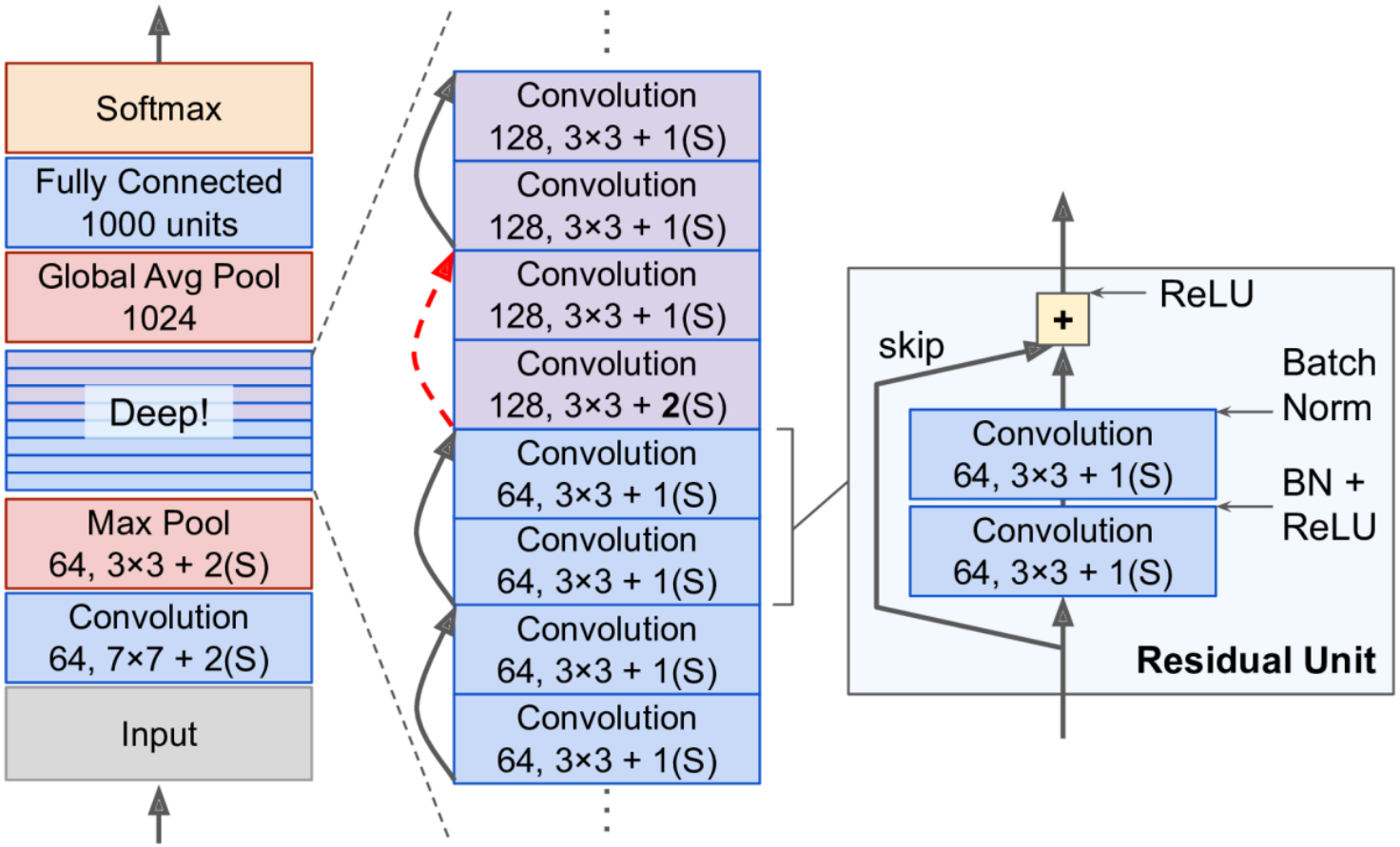


Residual Block: Why?

- The motivation behind residual learning is beyond the scope of the lecture
- Just an intuition: if weights are zero-initialized, your model will predict something close to 0
- With a residual block, you can easily initialize your model with the identity function (i.e., it just outputs the input)
- Being able to easily learn the identity function is beneficial for learning
- Furthermore, skip connections allow the NN to make progress even when many layers have not started learning yet

ResNet: Architecture

- Start and end similar to GoogLeNet
- Each res. block has 2 convolutional layers (and no pooling!)
- Batch Normalization and ReLU



Beyond ResNet

- ResNet is quite easy to modify
  - e.g., by tweaking number of channels and residual blocks, we can create different ResNet models, such as the deeper 152-layer ResNet-152
- Several other popular and successful CNN architectures proposed following ResNet
  - e.g., Google's Inception-v4 merged the ideas of GoogLeNet and ResNet
- Recently, proposals for network design spaces (e.g., RegNetX)
  - Given a few parameters, you automatically obtain a new NN model that enjoys some properties

Using Pre-Trained Models

- To use a well-known CNN model, you usually don't need to implement it
- Keras provides many pre-trained models with a single line of code

```
m = keras.applications.resnet50\
      .ResNet50(weights="imagenet")
```

tf_pretrained_cnn.ipynb

Transfer Learning

- What if you want to predict classes that were not considered in the existing model?
- Transfer learning: taking features learned on one problem, and leveraging them on a new, similar problem
- Especially useful for tasks where your dataset has too little data to train a full-scale model from scratch

Transfer Learning for DNNs

In the context of DL:

1. Take layers from a previously trained model
2. Freeze the lowest layers, so as to avoid destroying any of the information they contain during future training rounds
3. Add some new, trainable layers on top of the frozen layers
4. Train the new layers on your dataset
5. *Optional - Fine-tuning*: unfreeze the entire model and re-train it on the new data with a very low learning rate

Note: the term "fine-tuning" is often used to indicate the whole workflow, not only the last step

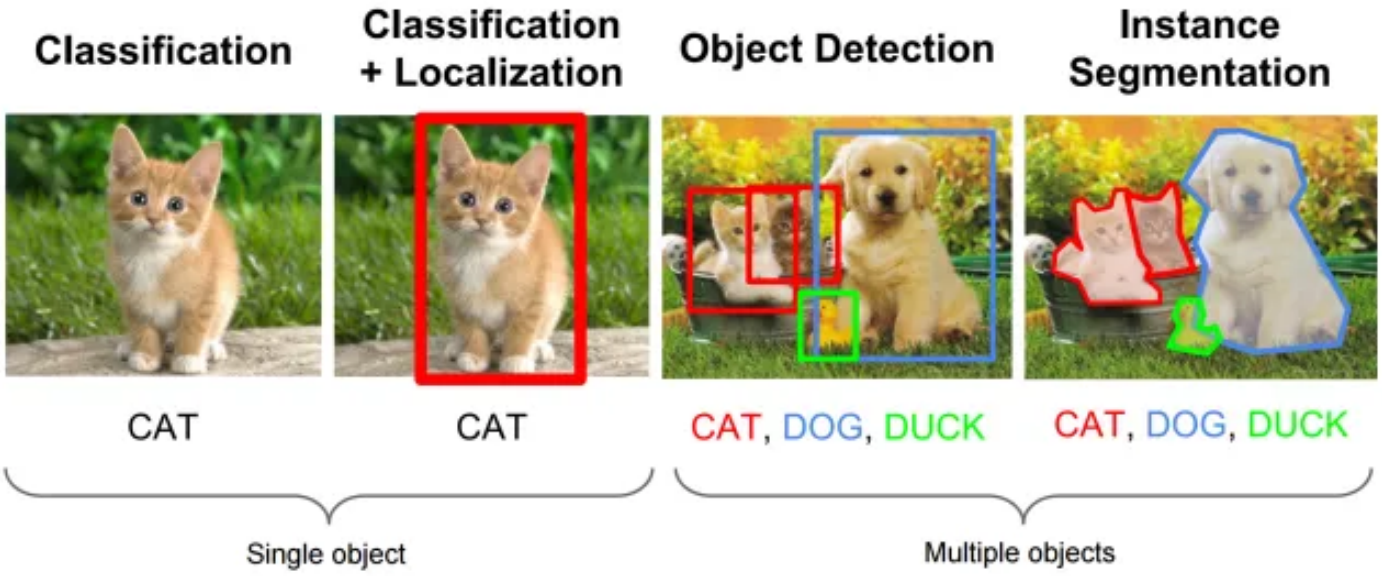
Example

- We load the pre-trained Xception CNN, a variant of GoogLeNet with improved performance
- The model has been trained to classify the ImageNet images (1,000 classes)
- We aim to reuse it for a binary classification problem (i.e., cat vs dog)
  - a “specialized” classification task

tf_transfer.ipynb

Beyond Classification

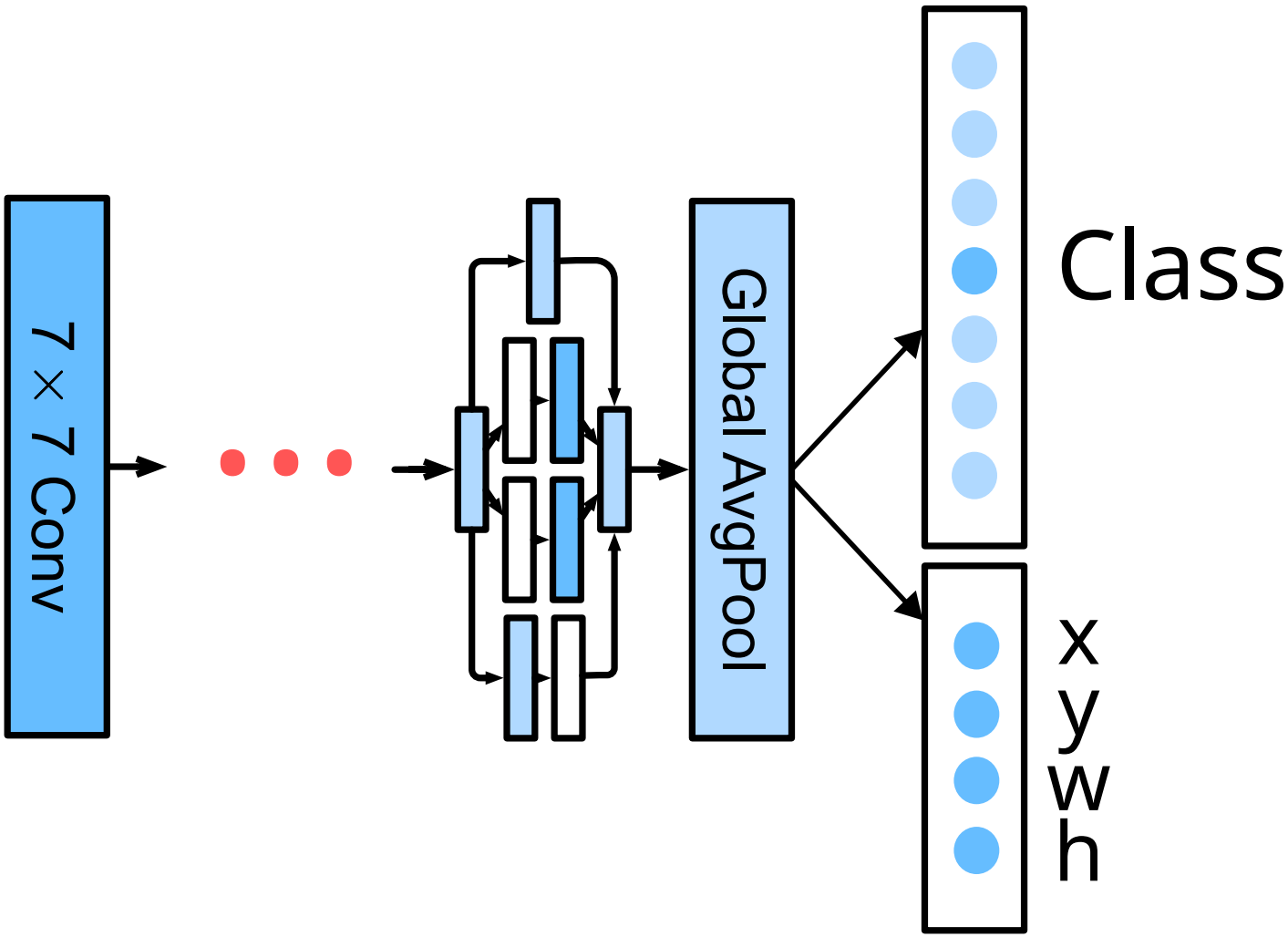
- So far we focused on image classification
  - associating each image with a label
- We may want to perform additional or more complex tasks:
  - localization+classification
  - object detection
  - semantic segmentation



Localization

- Our goal is predicting the bounding box of the object along with its class
- A bounding box may be defined in terms of 4 values:
  - (x, y) coordinates of the object center
  - object width and height
- Predicting a bounding box is a regression task!
- We can use the same CNN to solve both the classification and the regression tasks, by means of separate dense layers

Example



Example

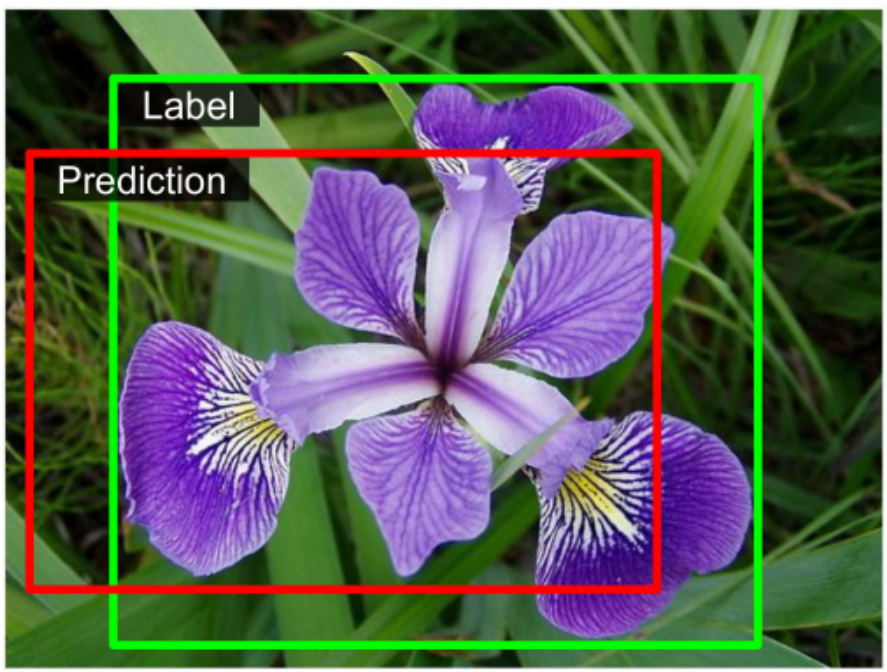
```
base = applications.xception.Xception(
    weights="imagenet", include_top=False)

avg = GlobalAveragePooling2D()(base.output)
cls_out = Dense(N, activation="softmax")(avg)
loc_out = Dense(4)(avg)
model = Model(inputs=base.input,
      outputs=[cls_out, loc_out])

model.compile(loss=
    ["sparse_categorical_crossentropy", "mse"],
    loss_weights=[0.8, 0.2],
    optimizer=optimizer, metrics=["accuracy"])
```

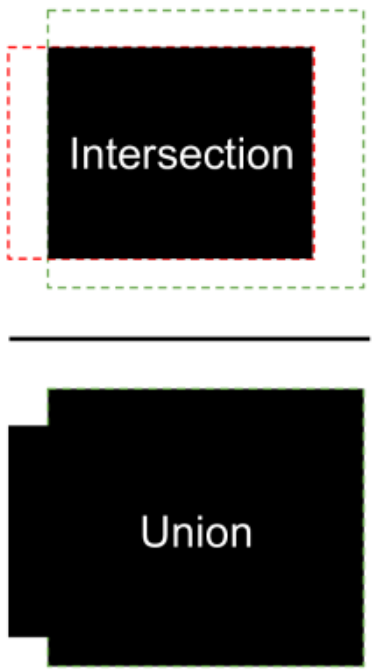
Labels for Localization

- To train such a NN, we need richer labels in the training set
- Class + bounding box for each instance
- Obtaining these labels can be a very long and difficult work
  - usually done by hand
  - crowdsourcing platforms (e.g., Amazon Mechanical Turk)

Metrics

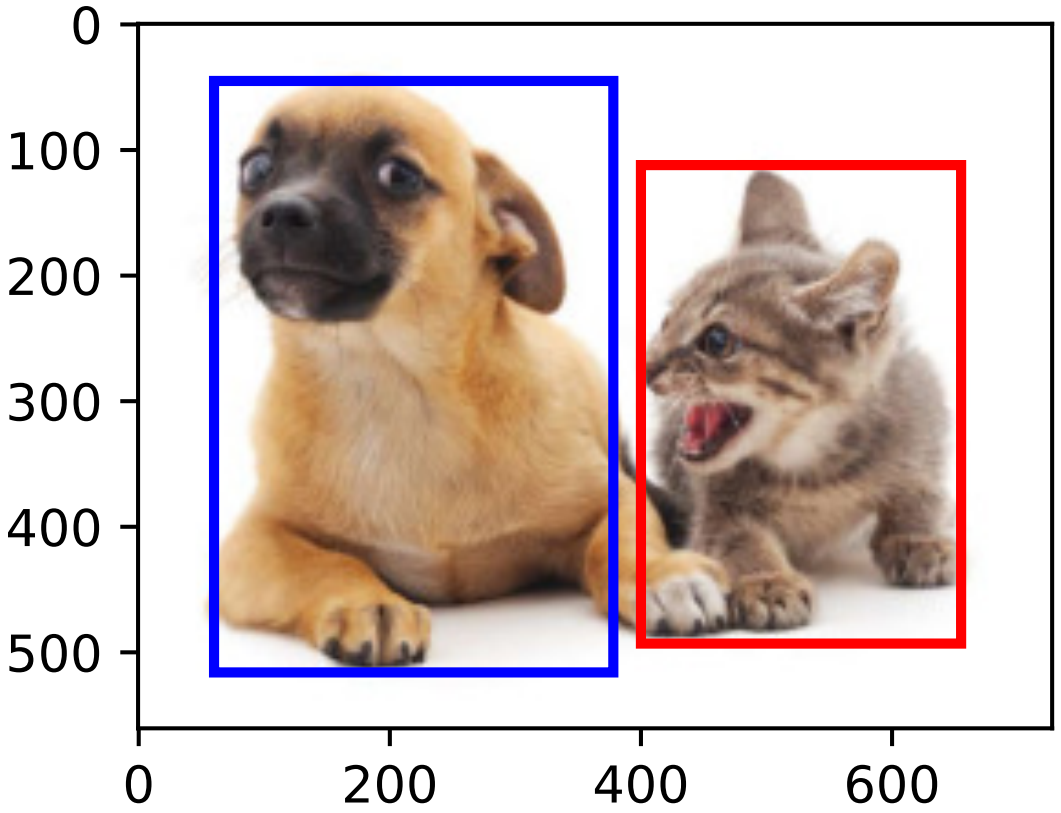
- The MSE is a good and popular choice as a loss function for for bounding box training
- To evaluate the quality of the predicted bounding boxes, a common metric is Intersection over Union (IoU)





Object Detection

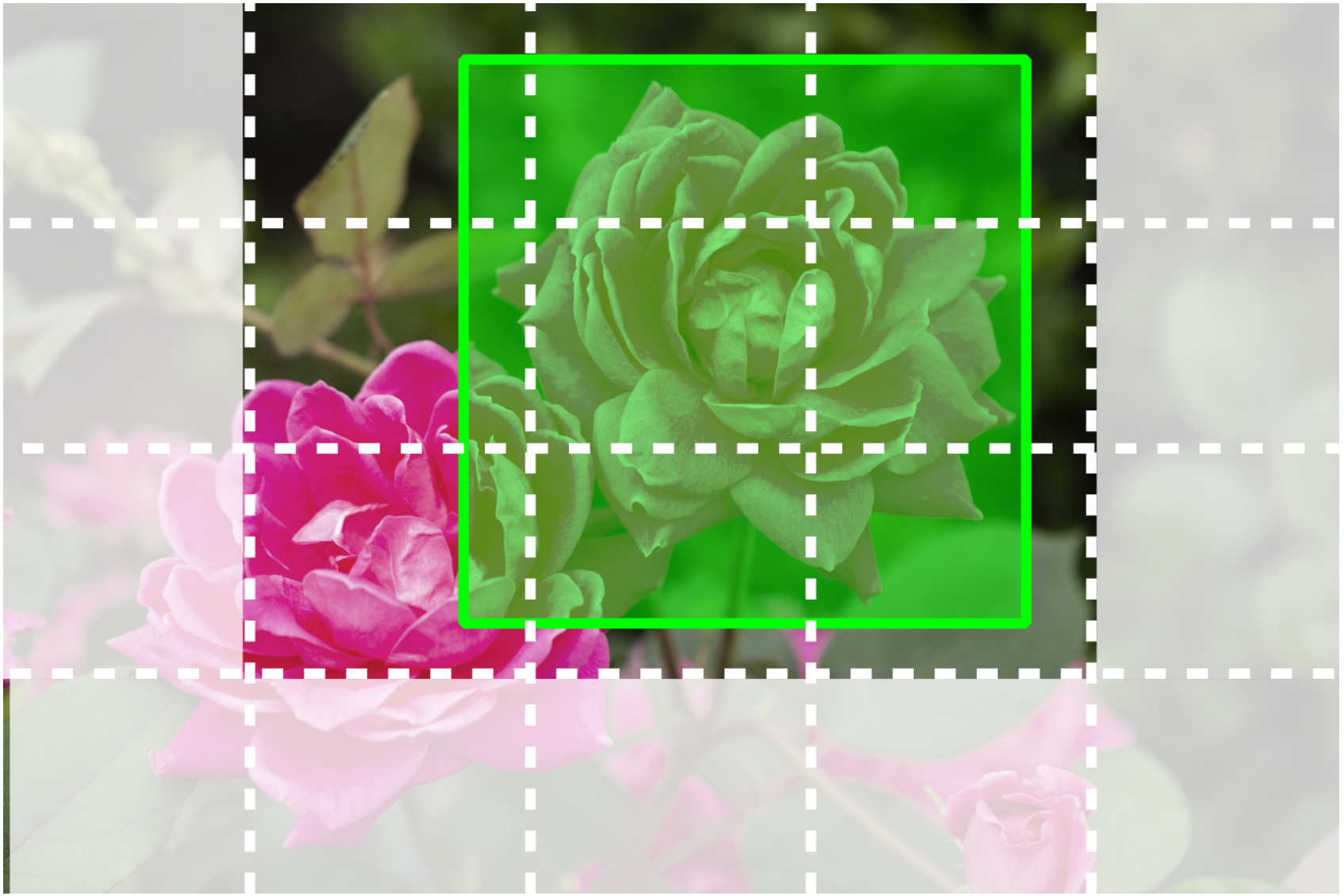
- Input images might depict more than a single object
- Object detection = classifying and localizing multiple objects in an image



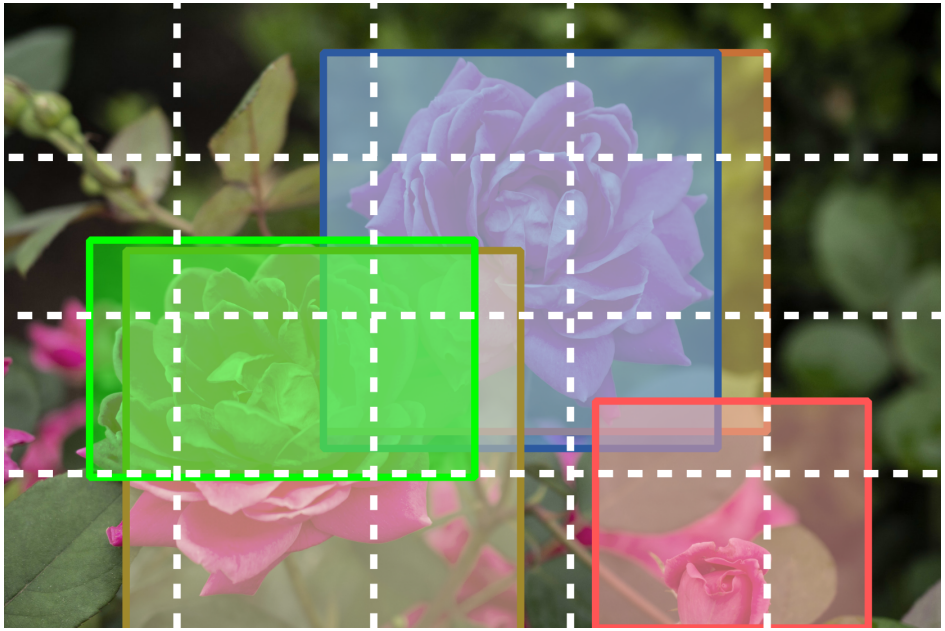
Naive Sliding Window Approach

- Train a CNN for single-object localization, with objects roughly in the middle of the input images; slide this CNN across the actual image and make predictions at each step
- The CNN may output an additional confidence score
  - estimated probability that the image does indeed contain the classified object
  - can be computed by a final output unit with sigmoid activation, based on predicted class probabilities
- You may also want to use multiple CNNs, to slide over larger/smaller regions
- Clearly, this approach does not scale

Example: Sliding Window



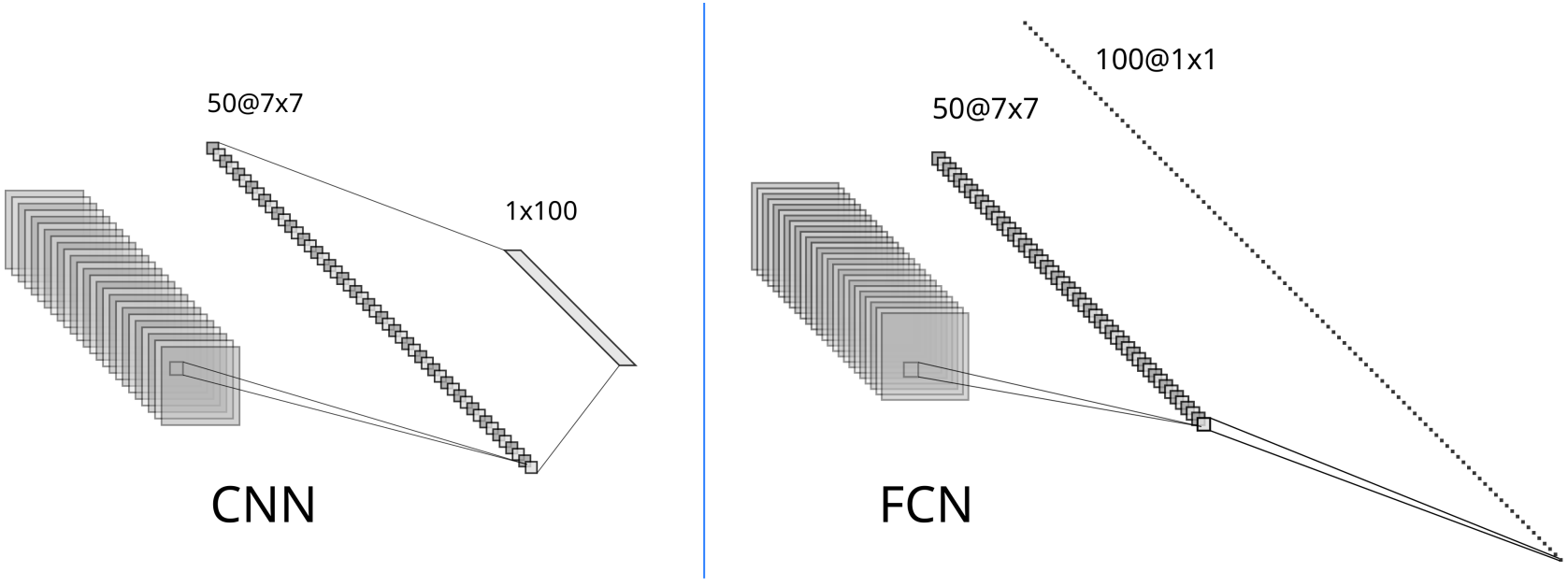
Example: Sliding Window (2)



- Some postprocessing required (non-max suppression)
- Bounding boxes with confidence lower than a threshold (e.g., 0.5) should be discarded
- For the remaining boxes, IoU should be used to discard highly overlapping ones

Fully Convolutional Networks (FCN)

- Consider a CNN with a final 100-unit dense layer
- Each output unit computes a weighted sum of all $50 \times 7 \times 7$ activations produced by the pooling layer
- Let's replace the dense layer with another conv. layer with 100 filters 7×7, without padding



What we get is a Fully Convolutional Network

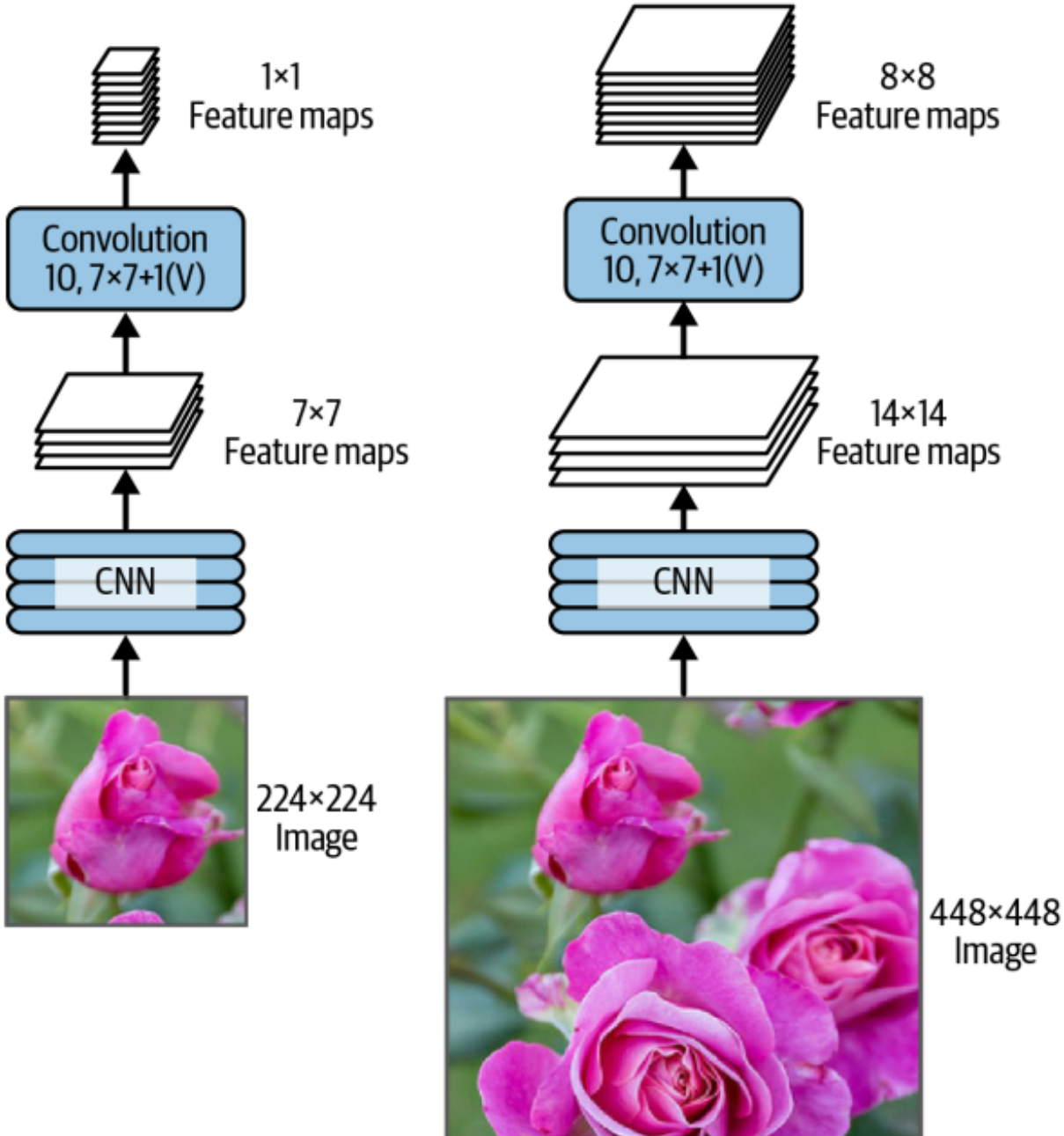
Fully Convolutional Networks (2)

- The two final layers both output 100 numbers
  - They only differ in the output "shape": 100-element vector vs. 1x1x100 tensor
- Suppose that we feed a larger input image to the two networks. What happens?
- The CNN does not work any more, because the dense layer will receive a longer input vector, which cannot be handled
- The FCN will still output 100 feature maps, larger than 1x1
- A FCN can handle input images of different sizes, with no re-training!

Fully Convolutional Networks (3)

- We trained a CNN for flower classification and localization, on 224x224 images. It outputs 10 numbers:
  - 5 class probabilities; 4 for bounding box; 1 confidence score
- Suppose the last conv. layer outputs 7x7 feature maps
- If we feed the FCN a 448x448 image, the conv. layer will output 14x14 feature maps
- The output will be composed of 10 features maps, each of size 8×8 ($14 - 7 + 1 = 8$)
- The FCN will output a 8x8 grid where each cell contains 10 numbers
- Like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column, but with a single pass!

Example


1×1
Feature maps
Convolution
10, 7×7+1(V)
7×7
Feature maps
CNN
224×224
Image
8×8
Feature maps
Convolution
10, 7×7+1(V)
14×14
Feature maps
CNN
448×448
Image

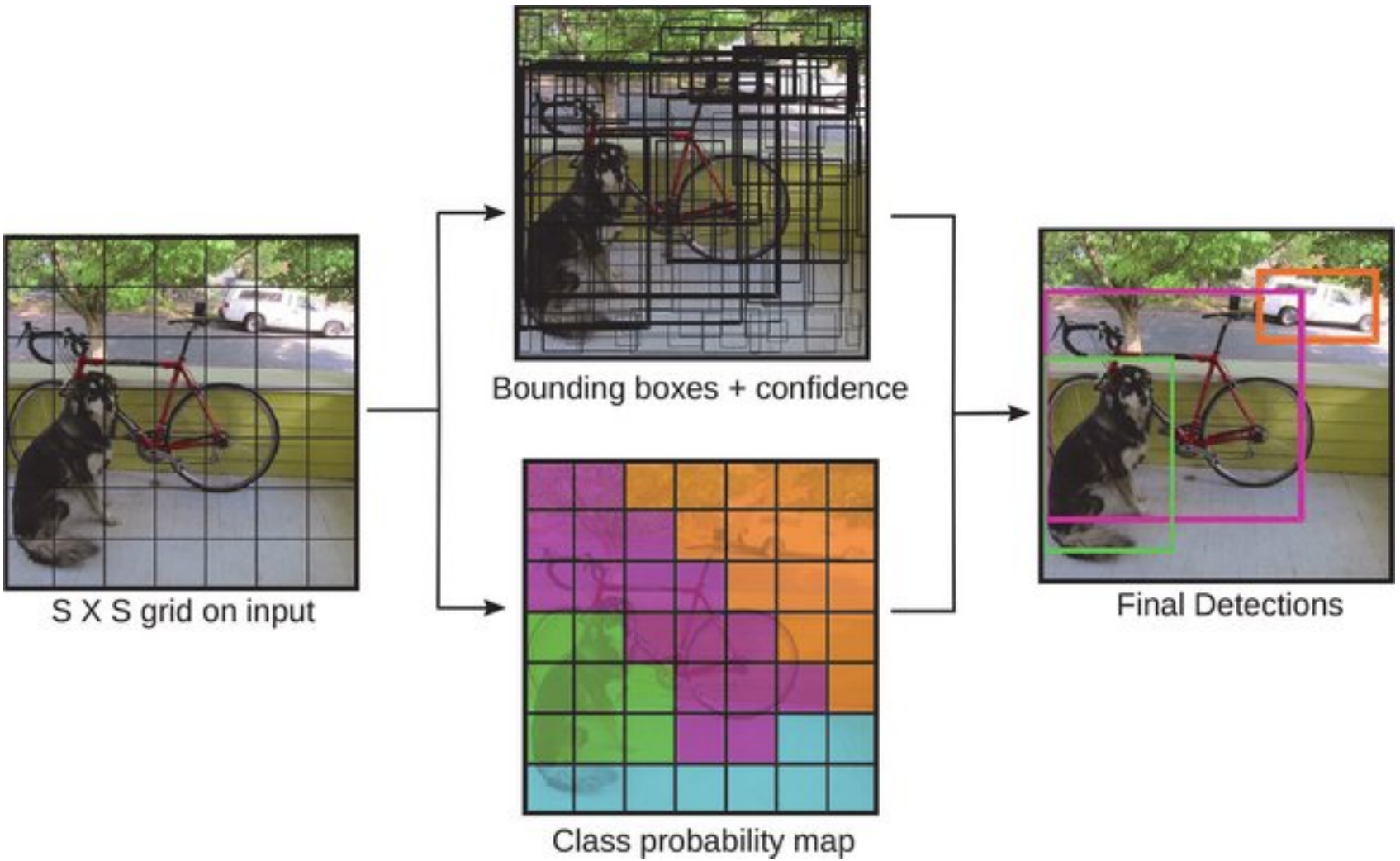
YOLO

- You Only Look Once (YOLO): fast and accurate object detection architecture
- Demo: `https://www.youtube.com/watch?v=MPU2HistivI`
- First proposed in 2015: very fast, not very accurate
  - up to 145 fps
- Accuracy greatly improved in newer versions
  - YOLOv2 (2016)
  - YOLOv3 (2018)
  - …
  - YOLOv7 (2022)
- Open-source implementations available

YOLO (2)

- Compared to sliding window approaches, YOLO sees the input image once (faster!) and sees the entire image (can use contextual information!)
- Input image divided in a $S \times S$ grid
- For each cell, YOLO predicts:
  - B bounding boxes (2 in YOLOv1)
  - C class probabilities
- So, a single dominant class predicted per cell (not per box!)
- A bounding box consists of:
  - x, y, w, h: normalized offset and size w.r.t. cell
  - confidence score
- A bounding box may extend beyond a single cell to detect larger objects!

Example


S X S grid on input
Bounding boxes + confidence
Class probability map
Final Detections

YOLO (3)

- After generating $S \times S \times B$ bounding boxes, IoU is used to eliminate redundant detections of the same object
  - If two boxes have high IoU, they likely detected the same object
  - Pick the one with highest confidence
- At training time, IoU is also used to compute confidence scores:

$$P(Object) * IoU_{truth}^{pred}$$

where $P(Object) \in \{0, 1\}$, depending on whether an object centered in the cell exists

YOLO (4)

- YOLOv1 used 24 convolutional layers + 2 fully connected layers
- A tiny/fast variant exists with less layers and smaller filters
- For training, the network is first trained on ImageNet as a classifier; then, the highest layers are replaced and training is completed for the detection task

- Starting with YOLOv2, they moved to a fully convolutional architecture

YOLO: Anchor Boxes

- Introduced in YOLOv3
- The goal is “suggesting” bounding box shapes to the network
- K-Means executed on training instances to identify B most frequent bounding box shapes (e.g., pedestrians will show similar shapes...): anchor boxes
- Predicted bounding box size expressed as relative to the associated anchor box (e.g., 1x1 = the anchor box itself)

YOLO: Some Implementations

- YOLOv3/YOLOv4: `https://pjreddie.com/darknet/yolo/`
  - Implemented in C, must be compiled
- YOLOv5: `https://github.com/ultralytics/yolov5`
  - Implemented using PyTorch; easy to install using a virtual environment

Object Tracking

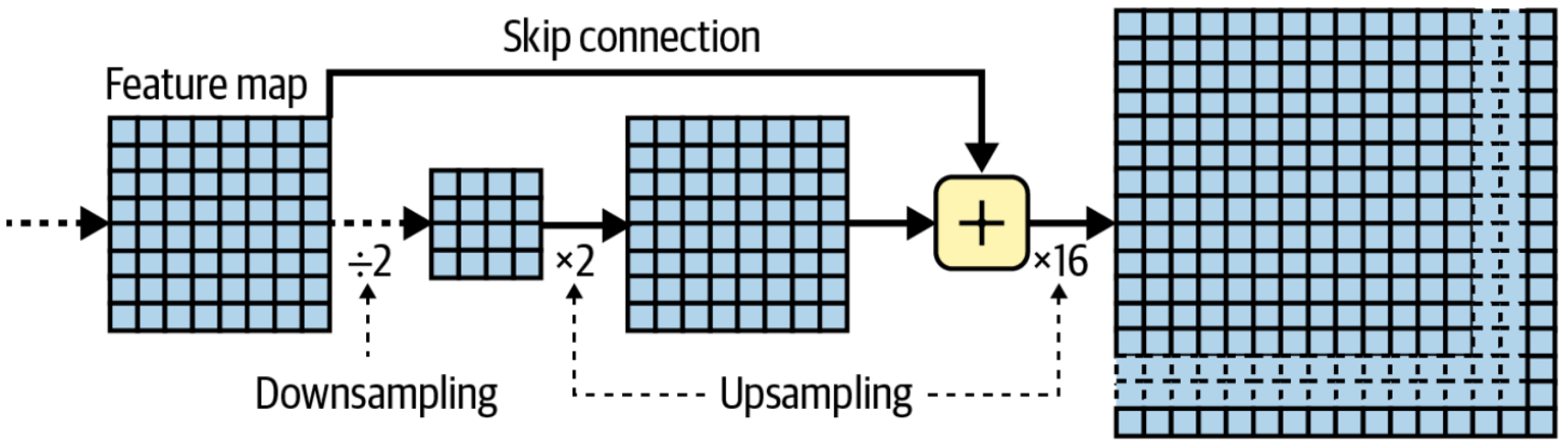
- Goal: tracking the position of a detected object in a video
- Naive idea: running detection for every frame
  - cannot distinguish newly appeared objects and moved objects
- DeepSORT
  - Kalman filter to predict motion
  - DL model to measure the resemblance between new detections and existing tracked objects
- DeepSORT + YOLOv4:
  `https://github.com/theAIGuysCode/yolov4-deepsort`

Semantic Segmentation

- Each pixel is classified according to the class of the object it belongs to
- More accurate localization compared to a bounding box

Semantic Segmentation (2)

- Active research topic, various models recently proposed
- First FCN proposed in 2015 to address this issue
- Key ideas: upscaling + skip connections
- Upscaling via transposed conv. layer
  - Imagine a convolution with fractional stride (e.g., 0.5)



References

- Goodfellow et al.: 9.1–9.3, 9.5
- D2L: 7, 8.1, 8.4, 8.6, 14.1–14.3, 14.11
- Hands-on ML: Chapter 14