Announcements

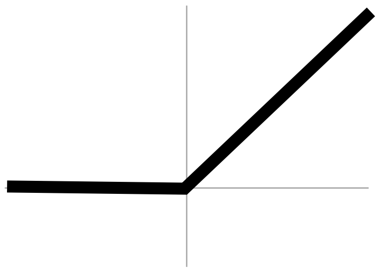
- HW 3 due **Wednesday, October 19 at 8pm**
- Quiz 6 due **Thursday, October 20 at 8pm**

Agenda

- **Neural networks**
  - Hyperparameter tuning
  - Implementation
- **Computer vision**
  - Prior to deep learning
  - Convolutional layers
  - Convolutional neural networks
  - Feature visualization

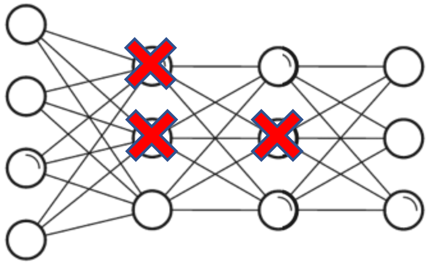
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



Dropout

Managing Training

Neural Network Tips & Tricks

- **Neural networks**
  - Design the model family
  - Backpropagation to compute gradient

- **Optimization**
  - Gradient descent
  - Momentum
  - Adaptive step sizes
  - Learning rate schedules
  - Initialize weights properly

Neural Network Tips & Tricks

- **Layers**
  - Use ReLU activations to avoid vanishing gradients
  - Use batch normalization at all layers to avoid "covariate shift"
  - Use dropout at last few layers for regularization

- **Regularization**
  - Use early stopping (or choose best model on validation set)
  - Use data augmentation if possible

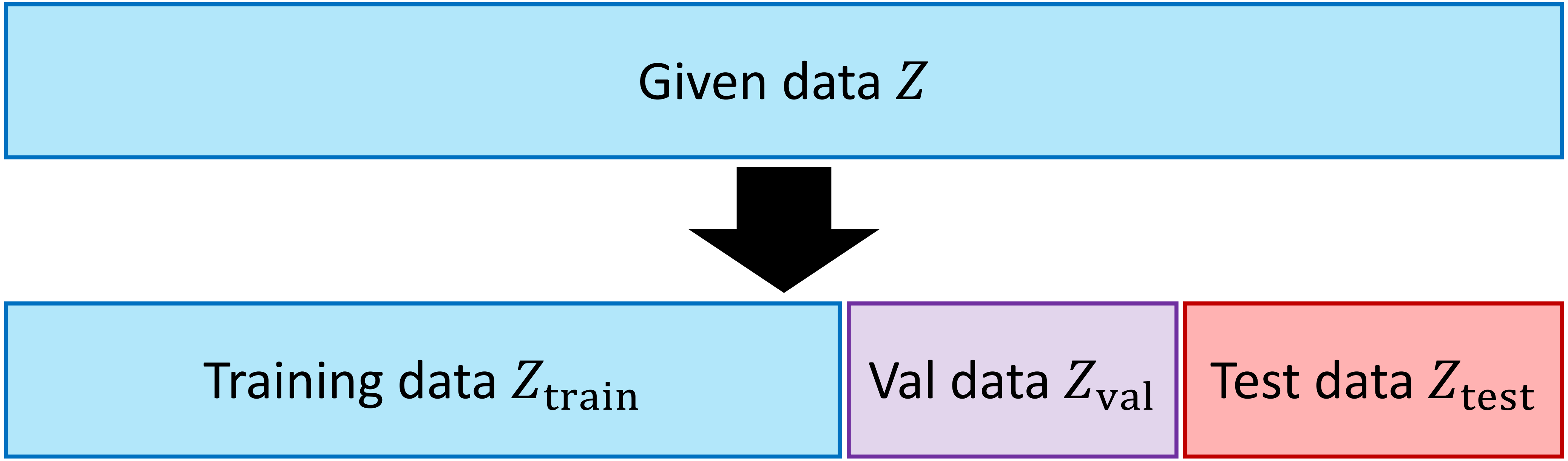
- Lots of hyperparameters! How to tune?

Hyperparameteter Choices

- **Architecture:** Stick close to tried-and-tested architectures (esp. for images)
- **SGD variant:** Adam, second choice SGD + 0.9 momentum
- **Learning rate:** 3e-4 (Adam), 1e-4 (for SGD + momentum)
- **Learning rate schedule**: Divide by 10 every time training loss stagnates
- **Weight initialization**: “Kaiming” initialization (scaled Gaussian)
- **Activation functions**: ReLU
- **Regularization**: BatchNorm (& cousins), L2 regularization + Dropout on some or all fully connected layers
- **Hyperparameter Optimization**: Random sampling (often uniform on log scale), coarse to fine

Hyperparameter Optimization

- **Recall:** Use cross-validation to tune hyperparameters!
  - Typically use one held-out validation set for computational tractability
  - E.g., 60/20/20 split
  - Can use smaller validation/test sets if you have a very large dataset



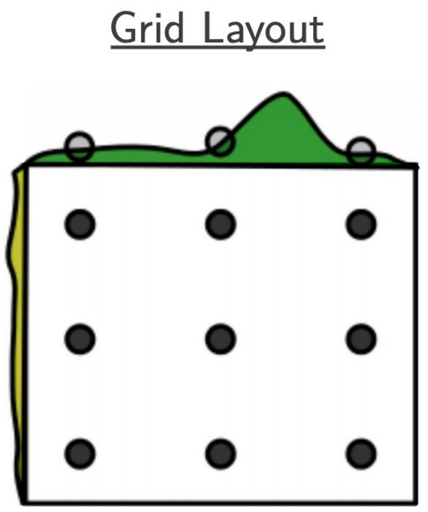
Hyperparameter Optimization Tips

- Keep the number of hyperparameters as small as possible
    - **Most important:** Learning rate

- **Strategy:** Automatically search over grid of hyperparameters and choose the best one on the validation set
    - Easy to parallelize across many machines
    - Record hyperparameters of all runs carefully!
    - Use the same random seeds for all runs

Hyperparameter Optimization Tips

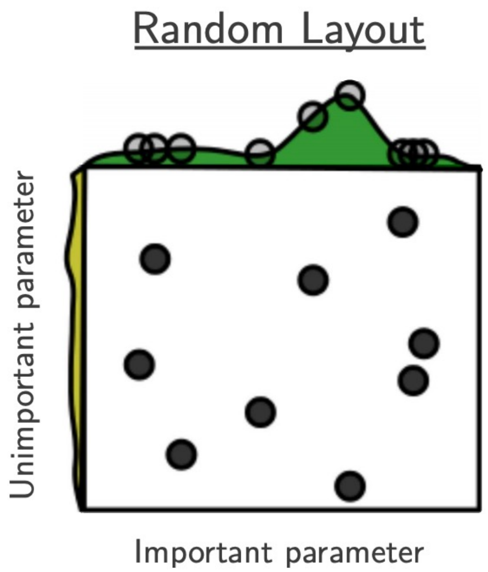
- **What about multiple hyperparameters?**
  - For 2 or 3 hyperparameters, do a systematic "grid search"



[Bergstra & Bengio, JMLR 2012]

Hyperparameter Optimization Tips

- **What about multiple hyperparameters?**
  - For >3 hyperparameters, do random search

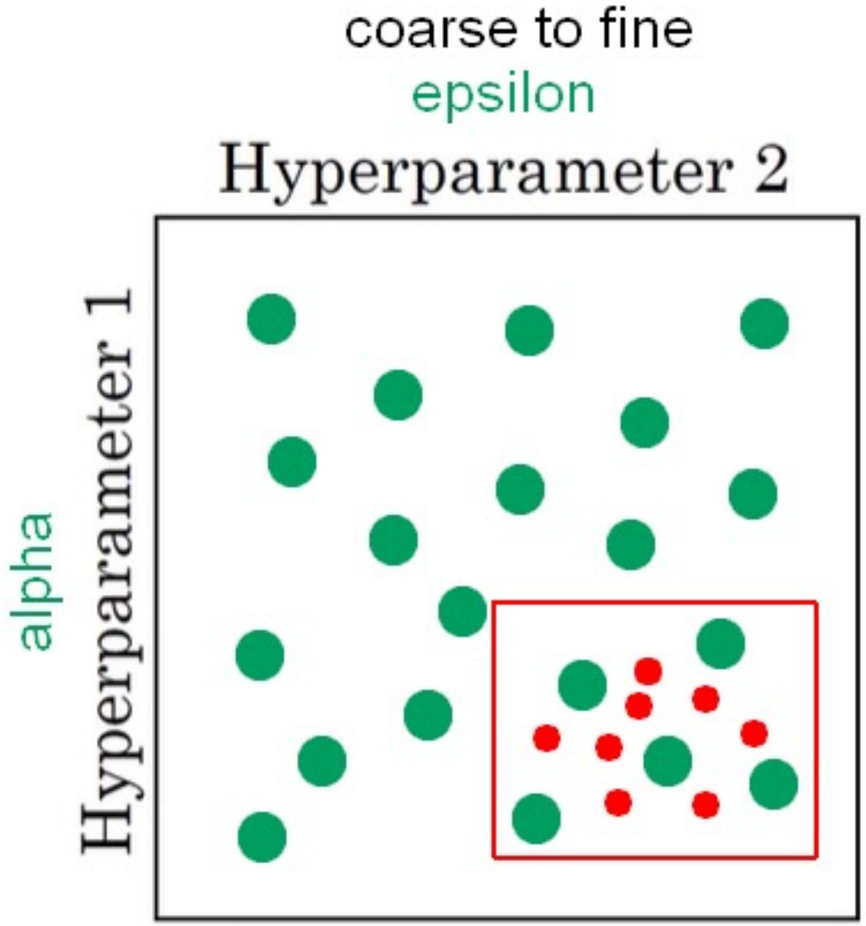


[Bergstra & Bengio, JMLR 2012]

Hyperparameter Optimization Tips

- **Coarse-to-find search**
  - Iteratively search over a window of hyperparameters
  - If the best results are near the boundary, center it on best hyperparameters
  - Otherwise, set a smaller window centered on the best hyperparameters

- **Bayesian optimization:** ML-guided search across hyperparameter trials to find good choices



https://www.andreaperlato.com/aipost/hyperparameters-tuning-in-ai/

More Practical Tips

- **Andrej Karpathy's blog post:**
  - http://karpathy.github.io/2019/04/25/recipe
  - Fix random seed during debugging
  - Overfit a tiny dataset first
  - With everything (architecture, learning algorithm, data etc.), start simple and build complexity slowly over iterations
  - Plot weight and gradient magnitudes to detect vanishing/exploding gradients

- **Additional reading:**
  - Chapter 11 of the Deep Learning textbook: "Practical Methodology"
  - https://www.deeplearningbook.org/contents/guidelines.html

Agenda

- **Neural networks**
  - Hyperparameter tuning
  - Implementation

- **Computer vision**
  - Prior to deep learning
  - Convolutional layers
  - Convolutional neural networks
  - Feature visualization

Pytorch

- Open source packages have helped democratize deep learning

Pytorch

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

Common parent class: nn.Module

Constructor: Defining layers of the network

```python

class Net(nn.Module):
    def __init__(self, in_features=10, num_classes=2, hidden_features=20):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.fc2 = nn.Linear(hidden_features, num_classes)

    def forward(self, x):
        x1 = self.fc1(x)
        x2 = F.relu(x1)
        x3 = self.fc2(x2)
        log_prob = F.log_softmax(x3, dim=1)

        return log_prob
```

Forward propagation

What about backward propagation?

Pytorch

- Open source packages have helped democratize deep learning

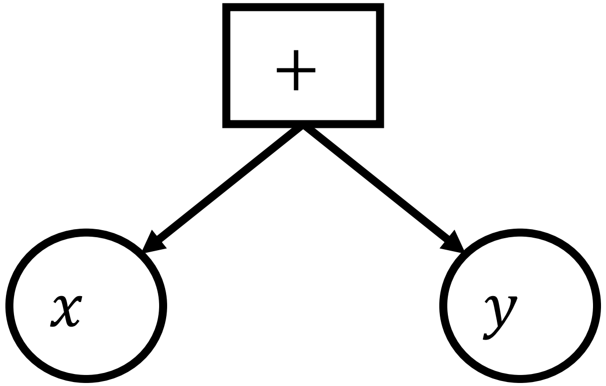
- Backpropagation implemented for all neural network architectures
  - Most modern libraries, including Tensorflow, Mxnet, Caffe, Pytorch, and Jax
  - Only need gradients of new layers

- **Basic Idea:** Provide model family as sequence of functions $[f_1, \ldots, f_m]$
  - What about more general compositions?
  - **Solution:** Composition of functions can be represented as graphs!

Computation Graphs

- The **tensor** datatype represents a **computation graph**
  - **Not just a numpy array!**
  - Instead, performing the computation produces a numpy array

- **Example:**
  - Suppose x is tensor that evaluates to $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
  - Suppose y is a tensor evaluates to $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
  - Then, $x + y$ is a tensor that evaluates to $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$

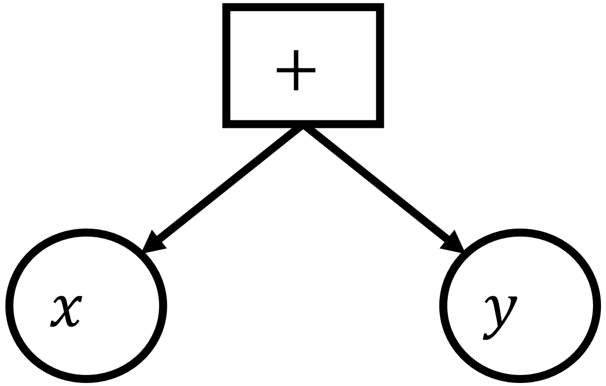


Toy Implementation of Computation Graphs

```
class Constant(tensor):
      def __init__(self, val):
            self.val = val
      def backpropagate(self):
            ...


class Add(tensor):
      def __init__(self, t1, t2):
            self.t1 = t1
            self.t2 = t2
            self.val = self.t1.val + self.t2.val
      def backpropagate(self):
            ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + y  # z has type Add
```

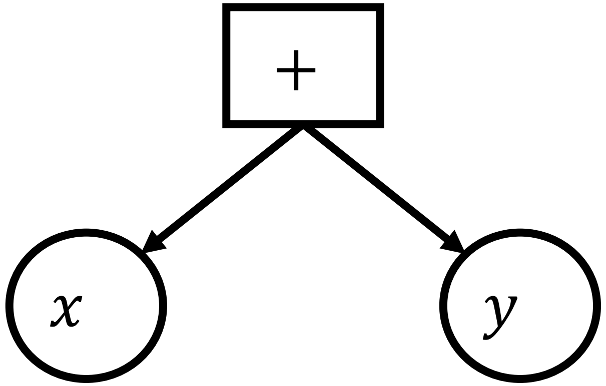


Toy Implementation of Computation Graphs

```
class Constant(tensor):
      def __init__(self, val):
            self.val = val
      def backpropagate(self):
            ...

class Add(tensor):
      def __init__(self, t1, t2):
            self.t1 = t1
            self.t2 = t2
            self.val = self.t1.val + self.t2.val
      def backpropagate(self):
            ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + x + y   # Z has type Add
```



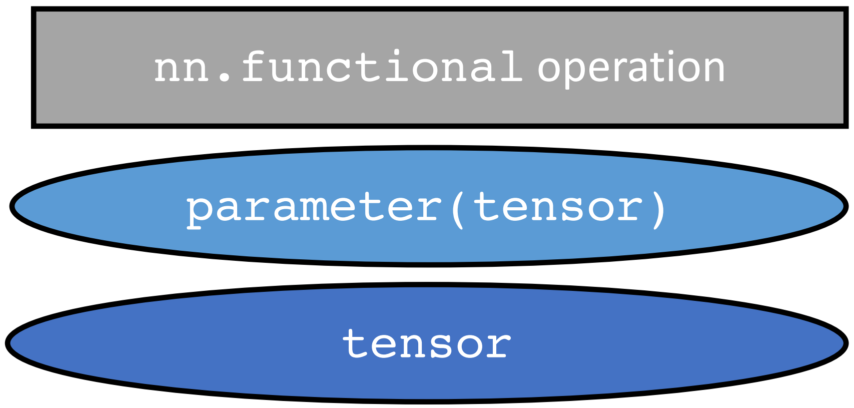
Computation Graphs

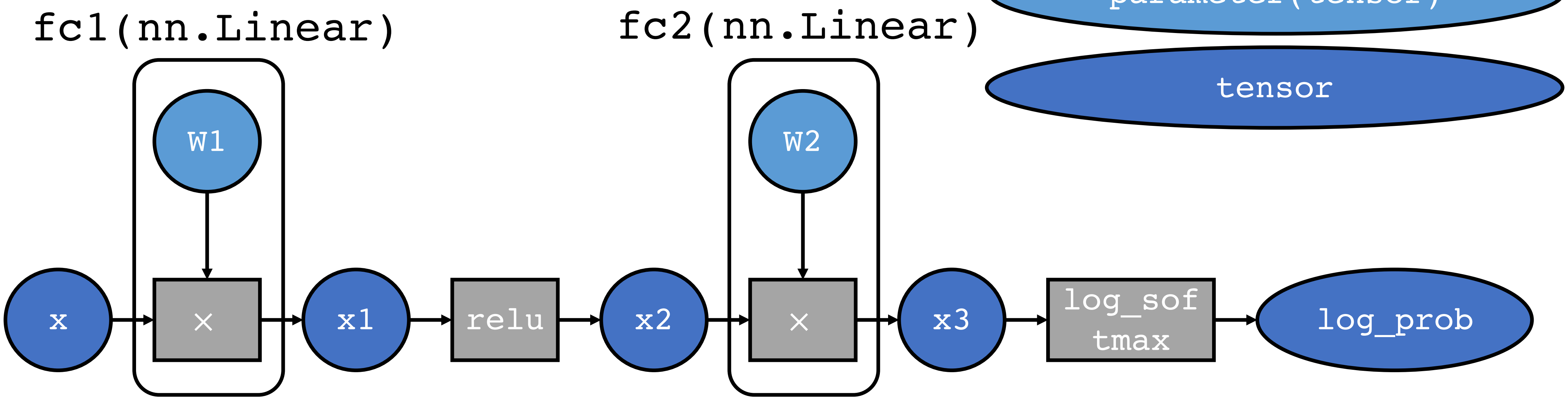
- Layers are implemented as tensors
  - **Examples:** addition, multiplication, ReLU, sigmoid, softmax, matrix multiplication/linear layers, MSE, logistic NLL, concatenation, etc.
  - You can also implement your own by providing forward pass and derivatives
- Tensors can be composed together to form neural networks

Computation Graphs

- **Forward propagation:** Values are evaluated as they are constructed
- **Backpropagation:** Automatically compute derivative of scalar with respect to all parameters based on derivatives of layers
  - `x.backwards()`
  - Does not perform any gradient updates!

Computation Graphs






```python
    def forward(self, x):
        x1 = self.fc1(x)
        x2 = F.relu(x1)
        x3 = self.fc2(x2)
        log_prob = F.log_softmax(x3, dim=1)

        return log_prob
```

Pytorch Training Loop

```python
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

Looping over mini-batches

Zero out all old gradients

Runs forward pass model.forward(data)

Loss computation

Backpropagation

Gradient step

Pytorch Training Loop

```
def main():
    torch.manual_seed(1)
    device = torch.device("cuda")
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=64, shuffle=True)

    model = Net().to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    sc                                              e=1, gamma=0.9)
    for epoch in range(1, 15):
        train(model, device, train_loader, optimizer, epoch)
        scheduler.step()
```

Load dataset

Loop over epochs (full passes over data)

Minibatch SGD for one epoch

Update base learning rate

Pytorch Model

- To use your model (once it has been trained):

```
label = model(input)
```

Agenda

- **Neural networks**
  - Hyperparameter tuning
  - Implementation

- **Computer vision**
  - Prior to deep learning
  - Convolutional layers
  - Convolutional neural networks
  - Feature visualization

Lecture 13: Computer Vision (Part 1)

CIS 4190/5190

Fall 2022

Images as 2D Arrays

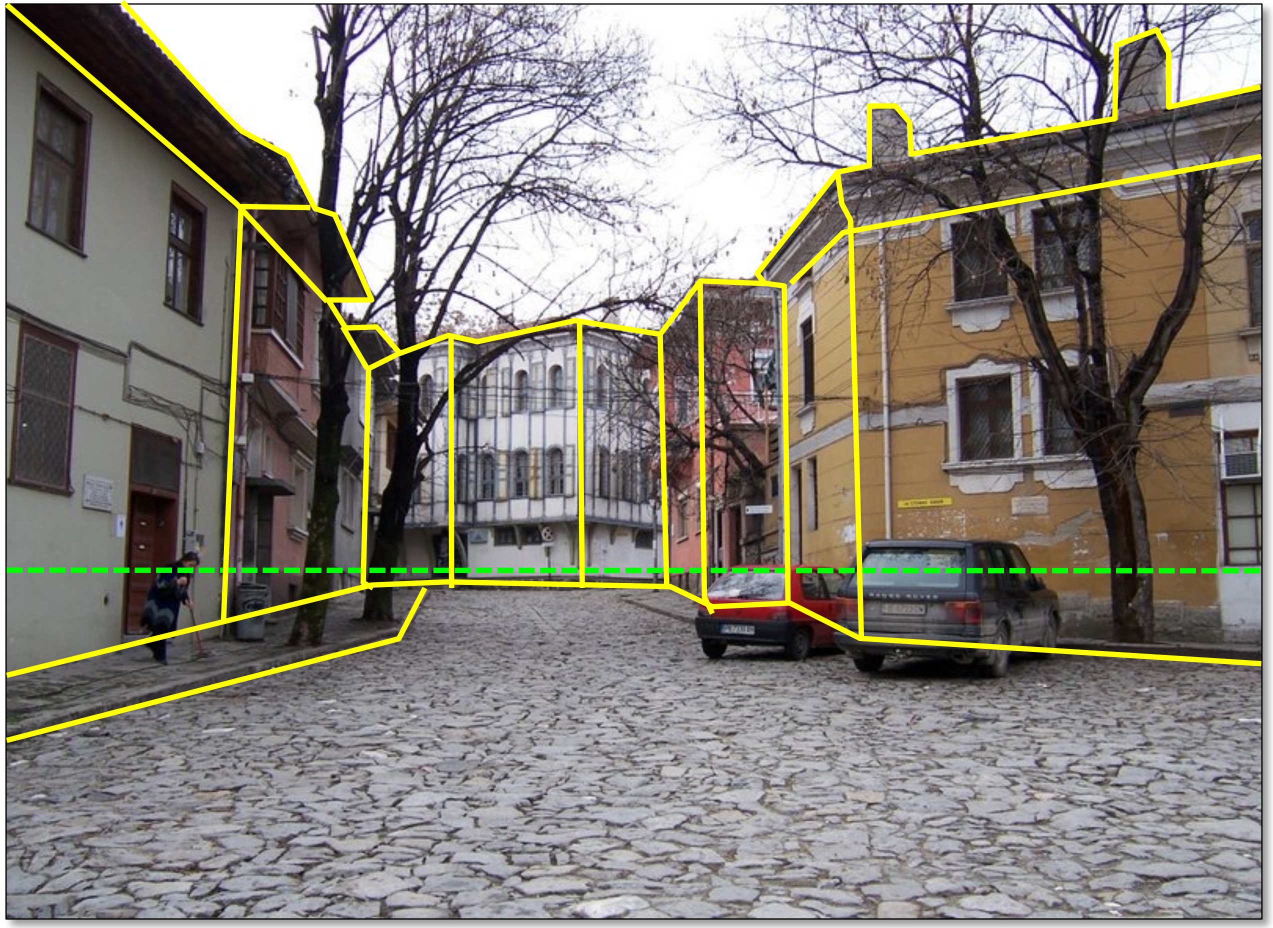
- Grayscale image is a 2D array of pixel values
- Color images are 3D array
  - 3^{rd} dimension is color (e.g., RGB)
  - Called "channels"

0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

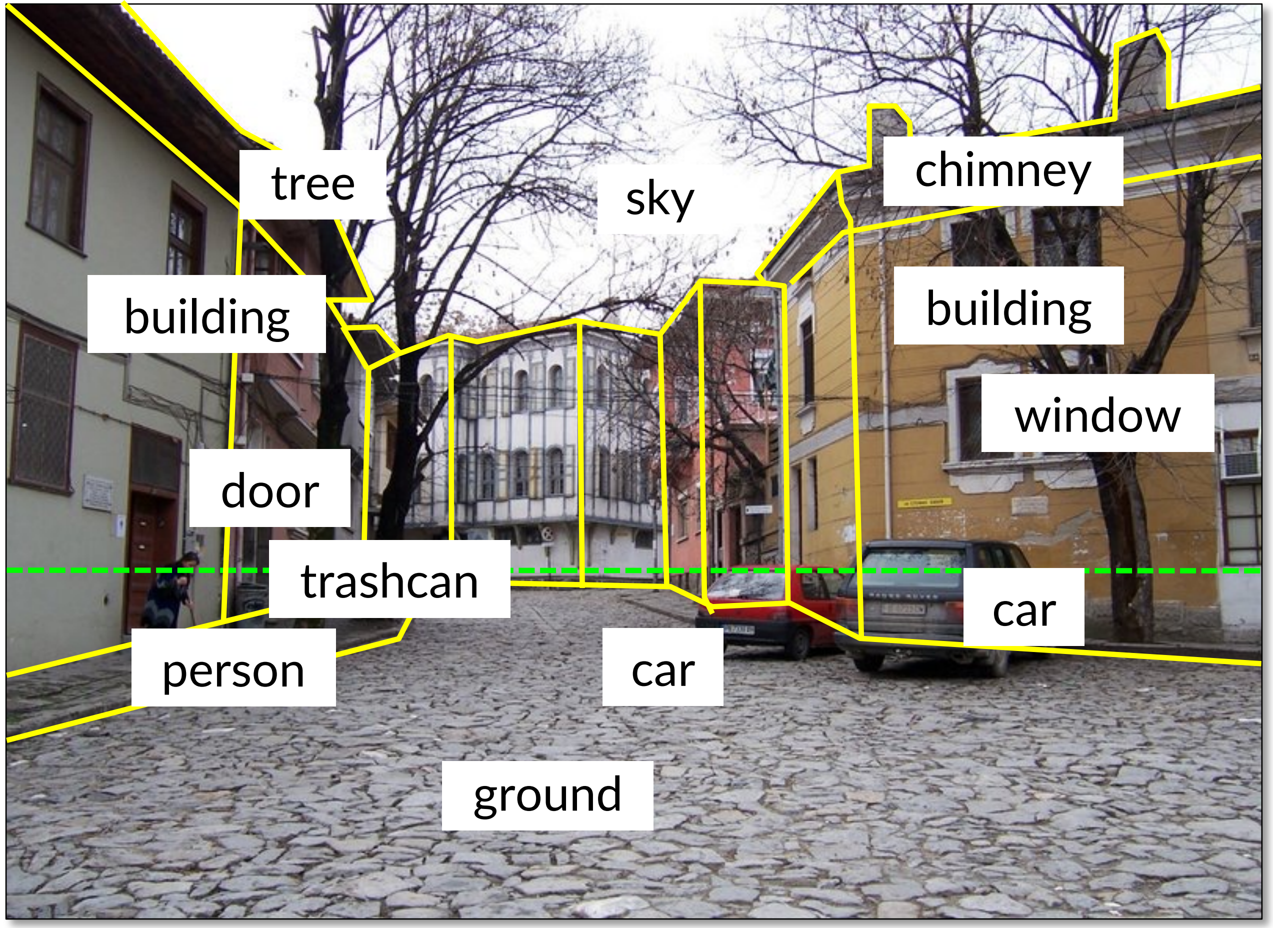
Source: S. Narasimhan, S. Lazebnik

Structure in Images

Structure in Images



Structure in Images



Outdoor scene

City

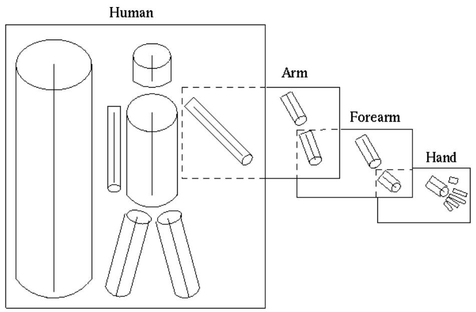
European

History of Computer Vision

- **Deceptively challenging task**
  - In the 1960s, Marvin Minsky assigned some undergrads to program a computer to use a camera to identify objects in a scene
  - Half a century later, we are still working on it

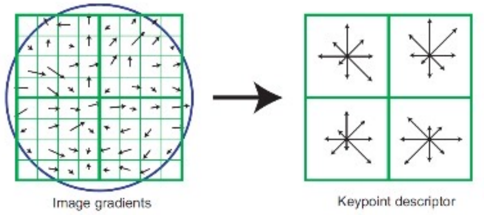
- **Moravec's paradox**
  - Motor and perception skills require enormous computational resources
  - Largely unconscious, biasing our intuition
  - Likely innate to some degree

History of Computer Vision



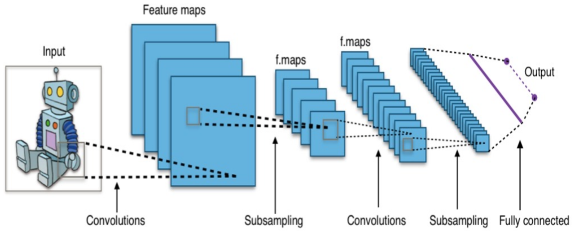
Very **old: 60's – Mid 90's**

Image → hand-def. features → hand-def. classifier



Old: Mid 90's – 2012

Image → hand-def. features → learned classifier



Current: 2012 – Present

Image → jointly learned features + classifier

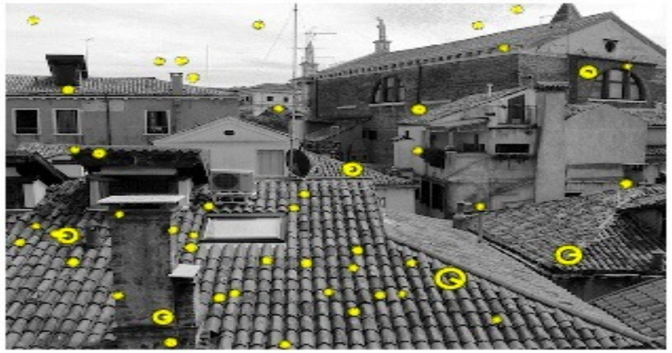
Prior to Deep Learning

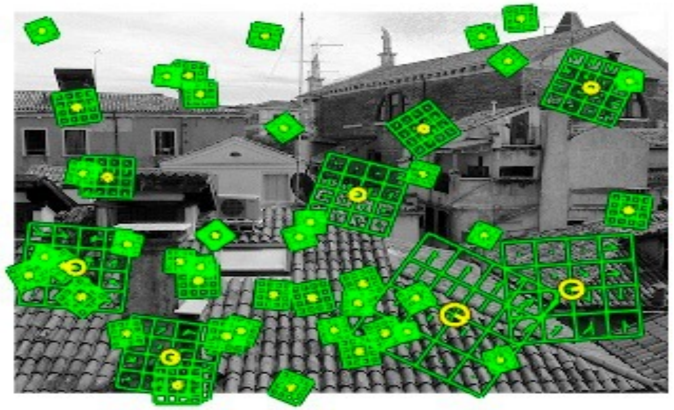
- **Step 1:** Find “pixels of interest”
  - E.g., corner points or “difference of gaussians”

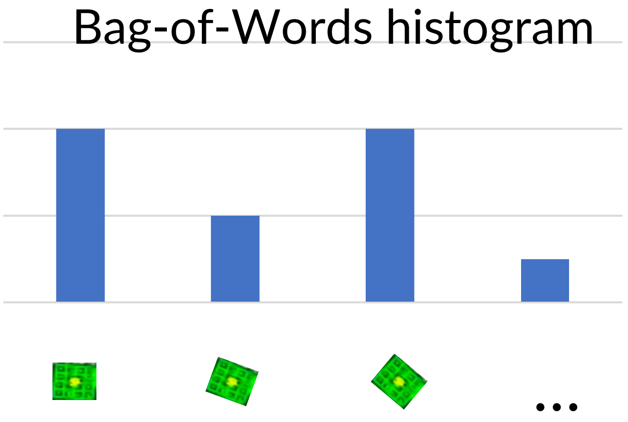
- **Step 2:** Compute features at these points
  - E.g., “SIFT”, “HOG”, “SURF”, etc.

- **Step 3:** Convert to feature vector via statistics of features such as histograms
  - E.g., “Bag of Words”, “Spatial Pyramids”, etc.

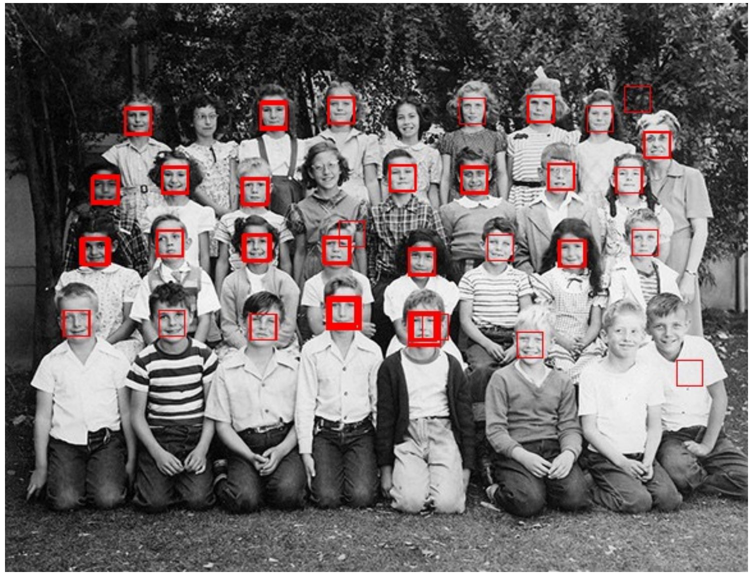
- **Step 4:** Use standard ML algorithm





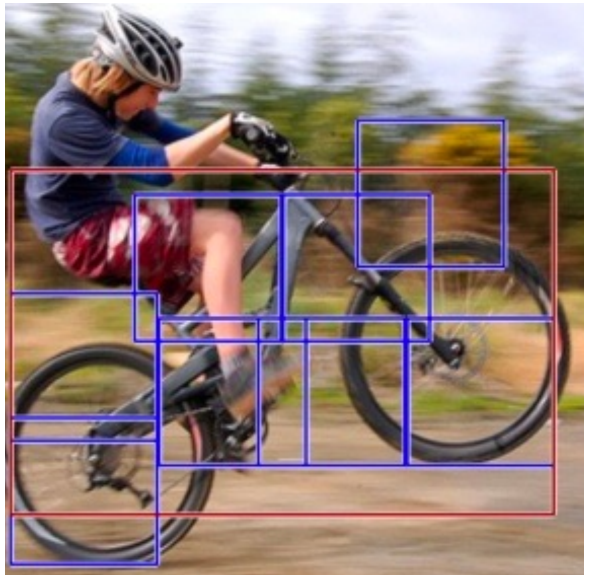


Prior to Deep Learning



https://github.com/alexdemartos/ViolaAndJones

Viola-Jones face detector
(with AdaBoost!)
~2000



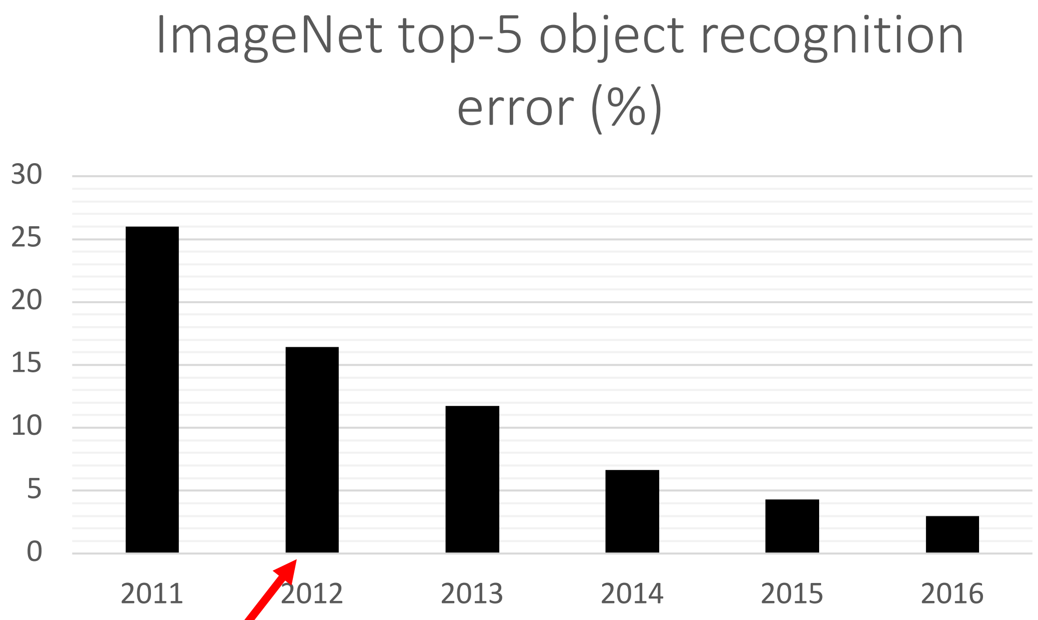
Deformable Parts Model
object detection
(with linear classifiers!)
~2010

GIST
Scene retrieval
(with nearest neighbors!)
~2006

See libraries such as VLFeat and OpenCV

Impact of Deep Learning

ImageNet 1000-object category recognition challenge

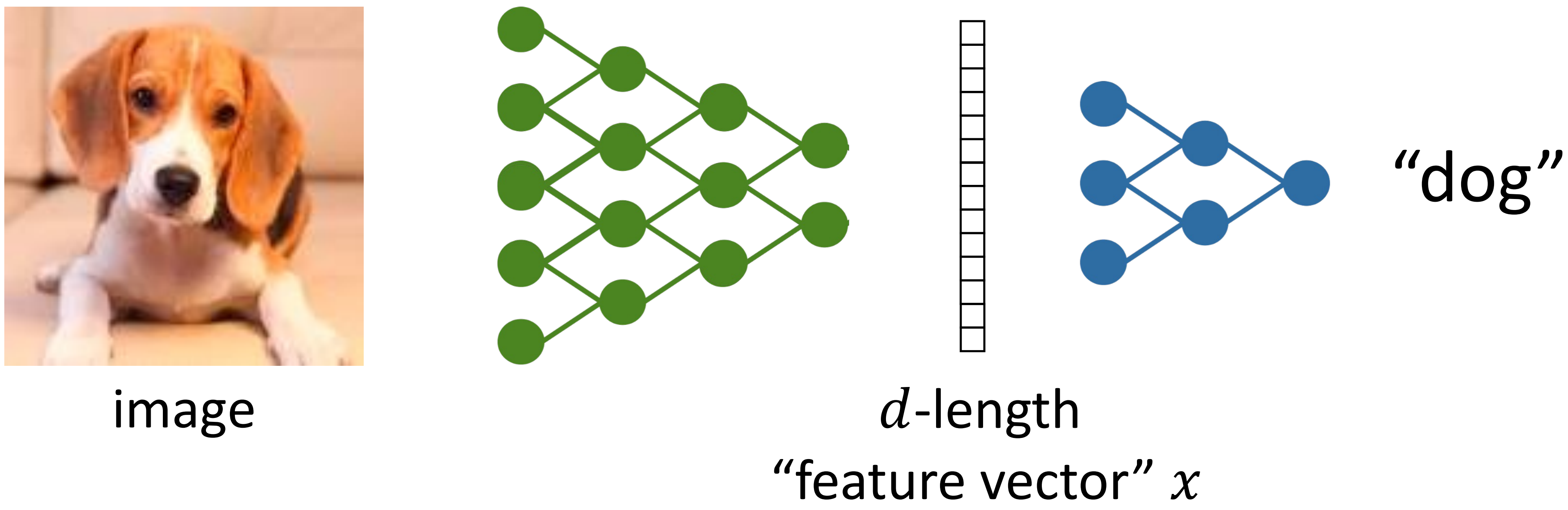


Deep learning breakthrough

Agenda

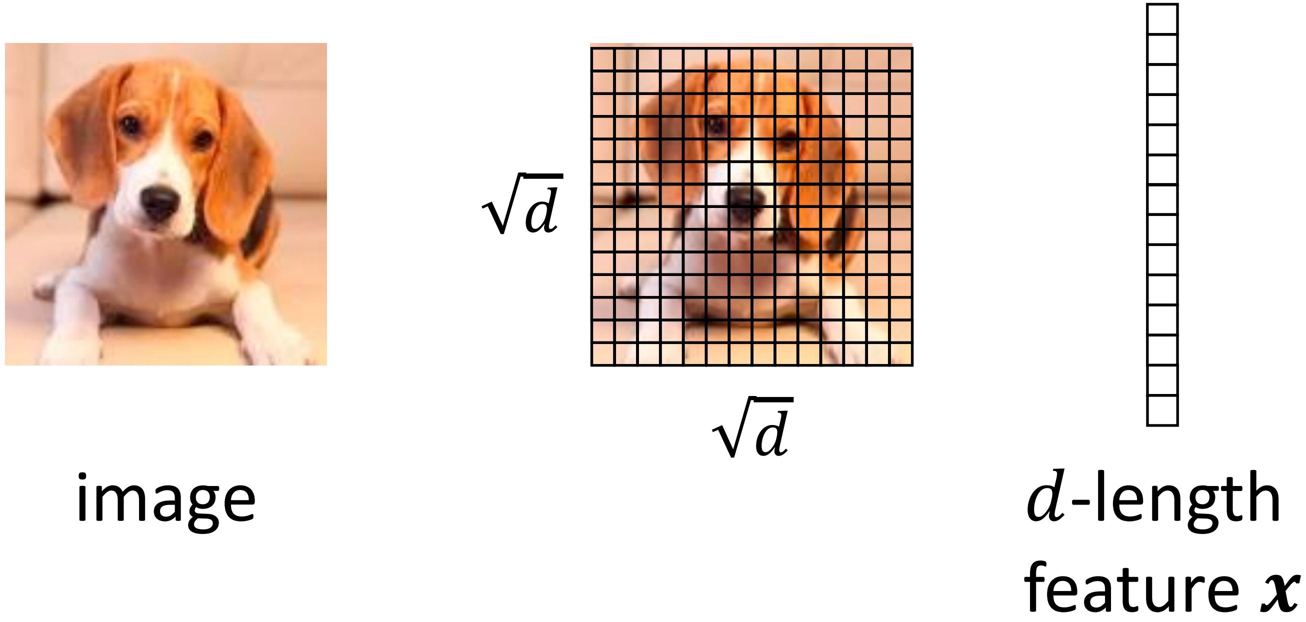
- **Neural networks**
  - Hyperparameter tuning
  - Implementation
- **Computer vision**
  - Prior to deep learning
  - Convolutional & pooling layers
  - Convolutional neural networks

Representation Learning



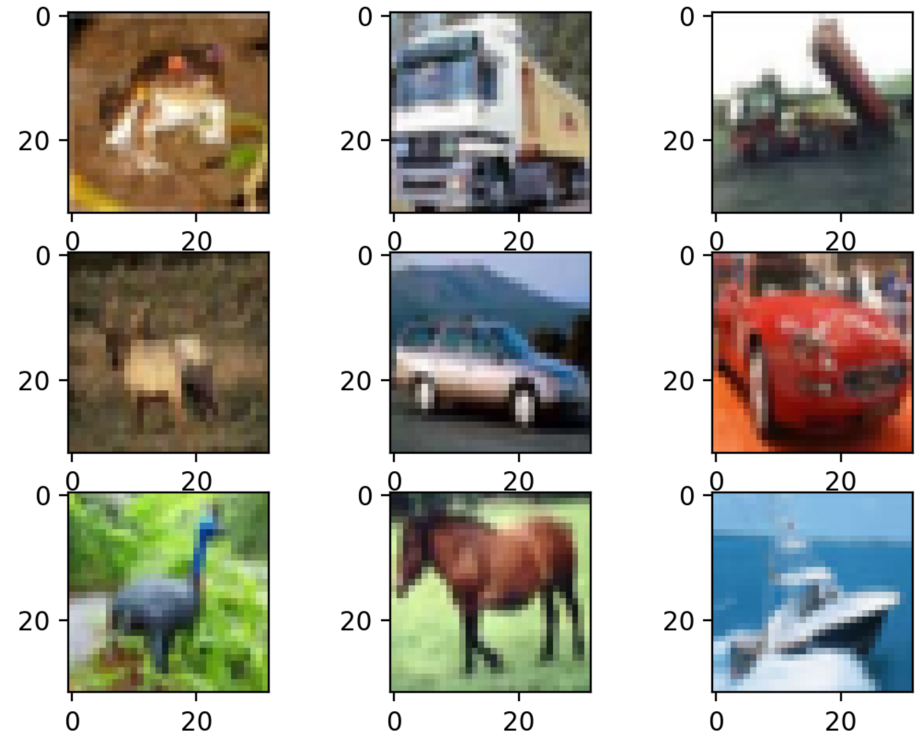
Representing Images as Inputs

- **Naïve strategy**
  - Feed image to neural network as a vector of pixels



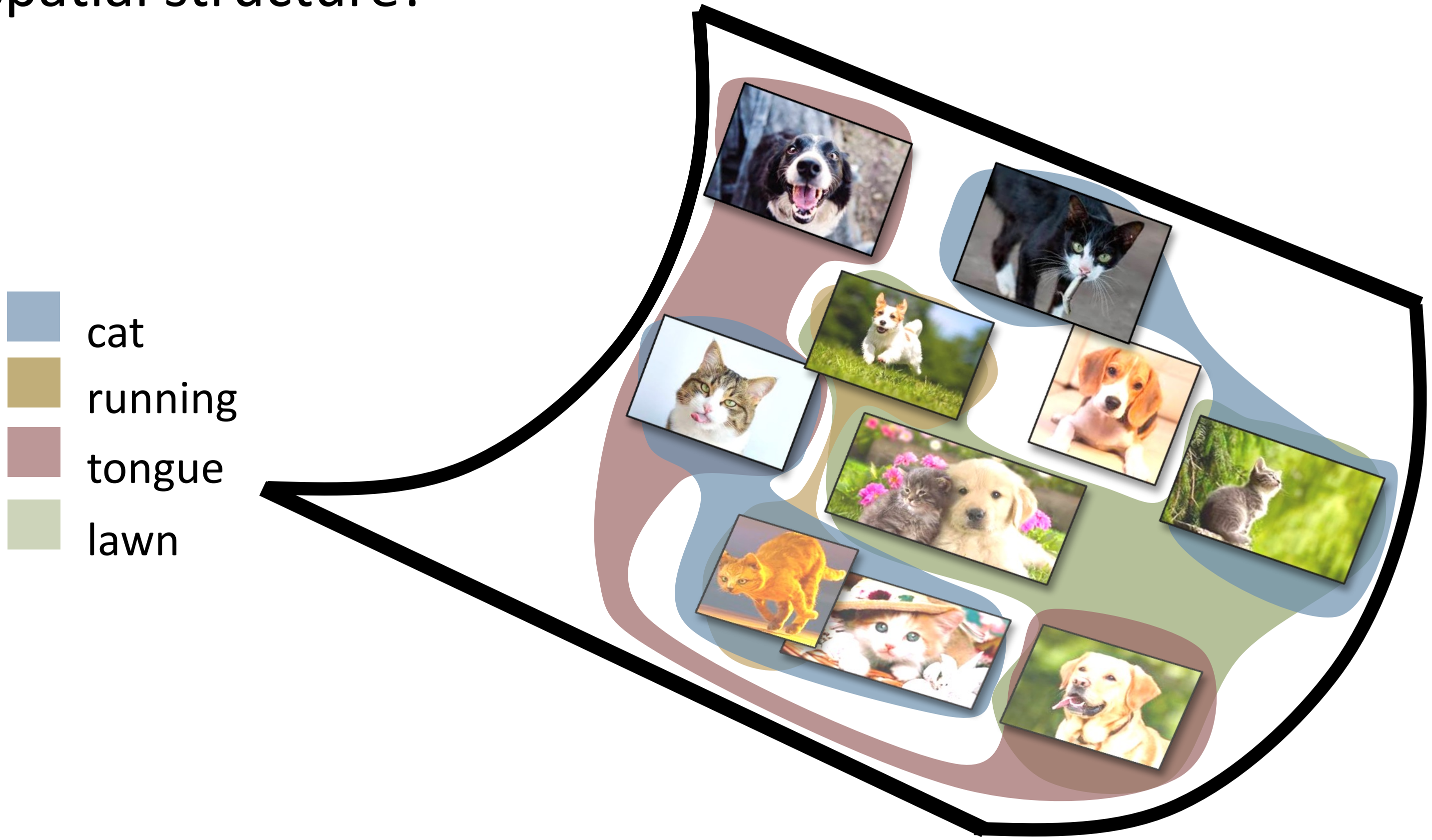
Representing Images as Inputs

- **Shortcomings**
  - Very high dimensional! $32 \times 32 \times 3 = 3072$ dimensions



Representing Images as Inputs

- **Shortcomings**
  - Ignores spatial structure!

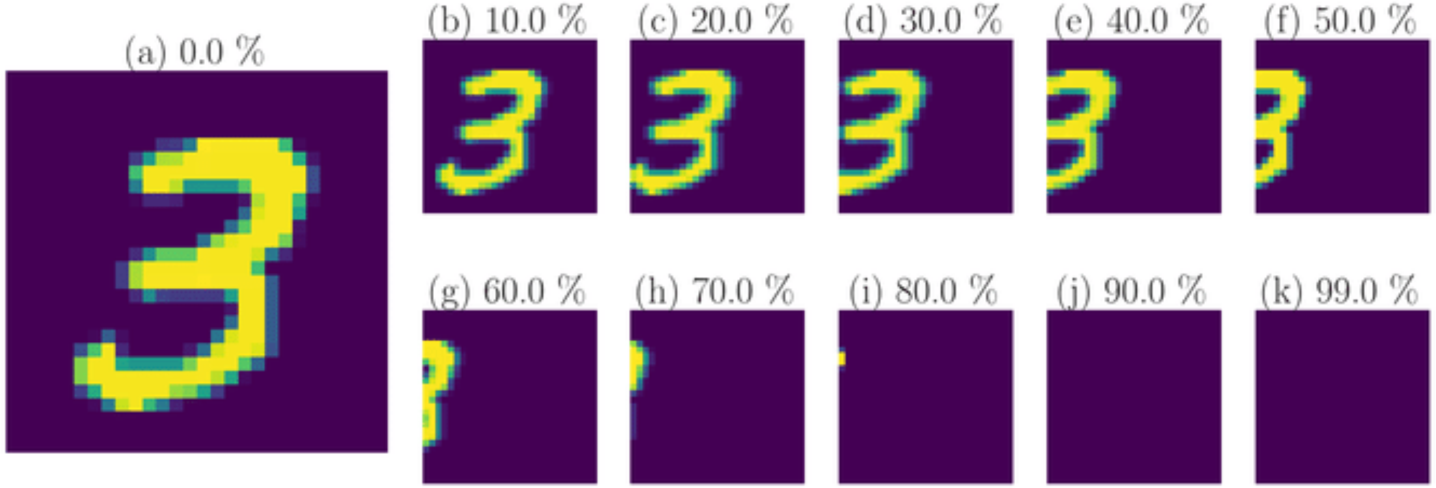


Structure in Images

- **2D image structure**
  - Location associations and spatial neighborhoods are meaningful
  - So far, we can shuffle the features without changing the problem (e.g., $\beta^\top x$)
  - Not true for images!

Structure in Images

- **Translation invariance**
  - Consider image classification (e.g., labels are cat, dog, etc.)
  - **Invariance:** If we translate an image, it does not change the category label



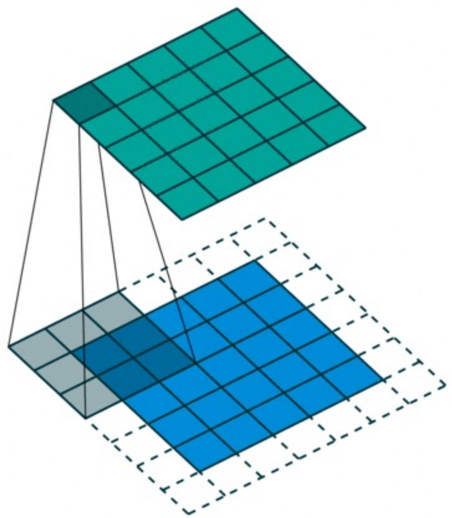
Source: Ott et al., Learning in the machine: To share or not to share?

Structure in Images

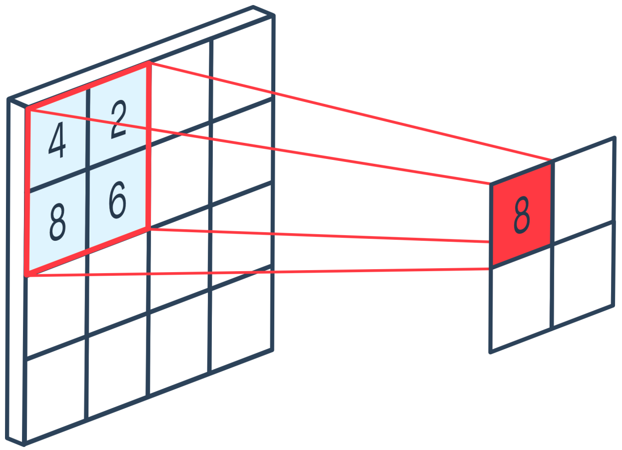
- **Translation equivariance**
  - Consider object detection (e.g., find the position of the cat in an image)
  - **Equivariance:** If we translate an image, the the object is translated similarly

Structure in Images

- Use layers that capture structure



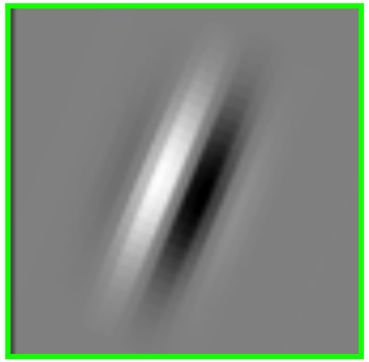
Convolution layers
(Capture equivariance)



Pooling layers
(Capture invariance)

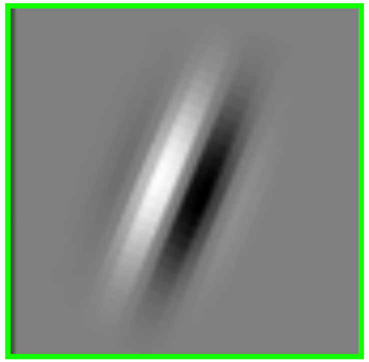
https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d
https://peltarion.com/static/2d_max_pooling_pa1.png

Convolution Filters



graphic credit: S. Lazebnik

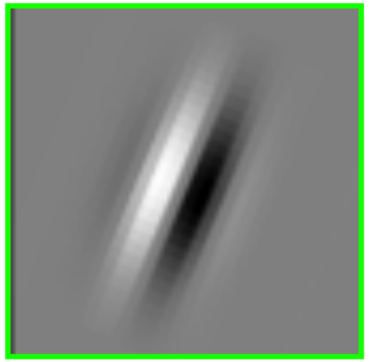
Convolution Filters



$$\text{output}[0,0] = \sum_{\tau=0}^{k-1}\sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[0+\tau, 0+\gamma]$$

graphic credit: S. Lazebnik

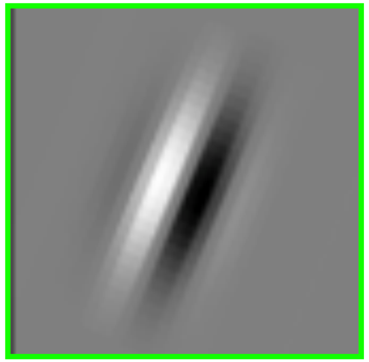
Convolution Filters



$$\text{output}[0,1] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau, \gamma] \cdot \text{image}[0+\tau, 1+\gamma]$$

graphic credit: S. Lazebnik

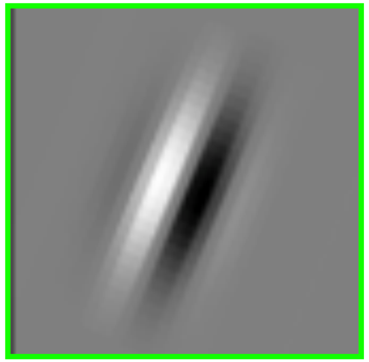
Convolution Filters



$$\text{output}[0,2] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[0+\tau, 2+\gamma]$$

graphic credit: S. Lazebnik

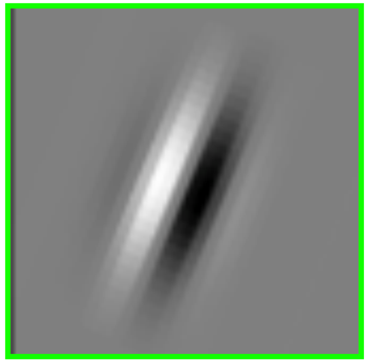
Convolution Filters

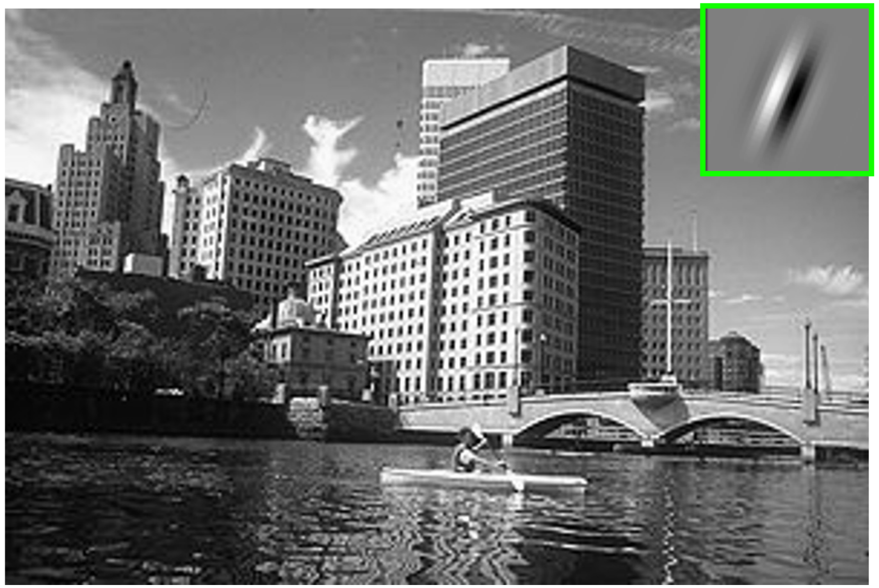


$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters

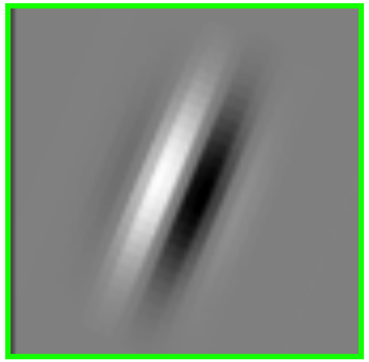


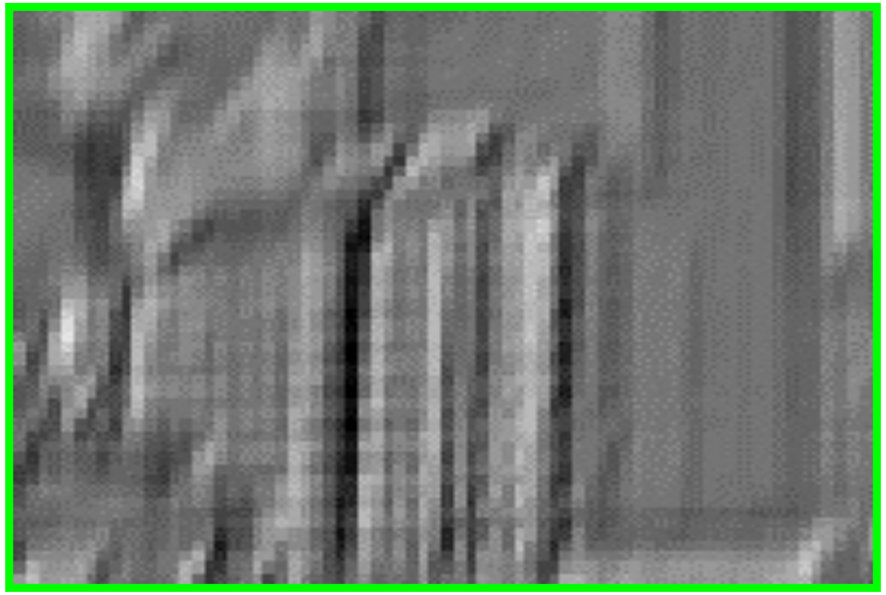


$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters





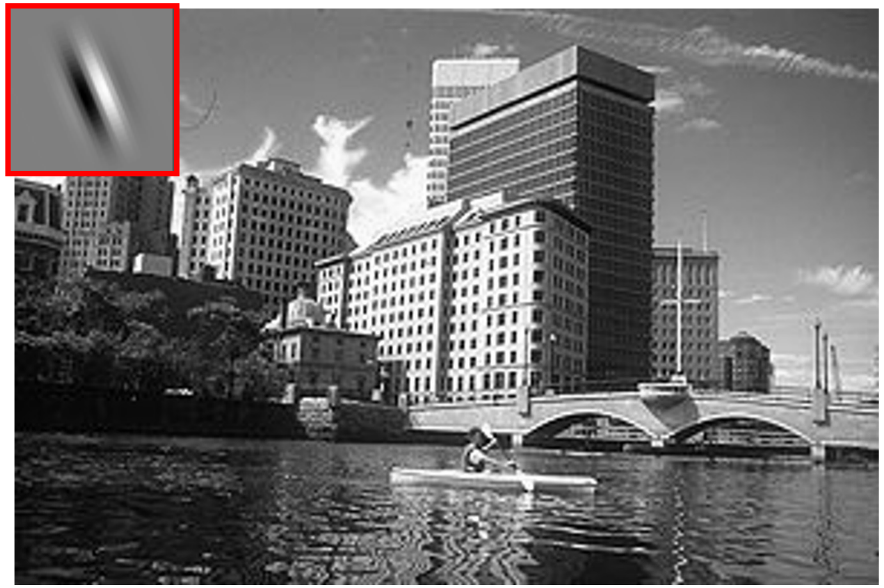
$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters

graphic credit: S. Lazebnik

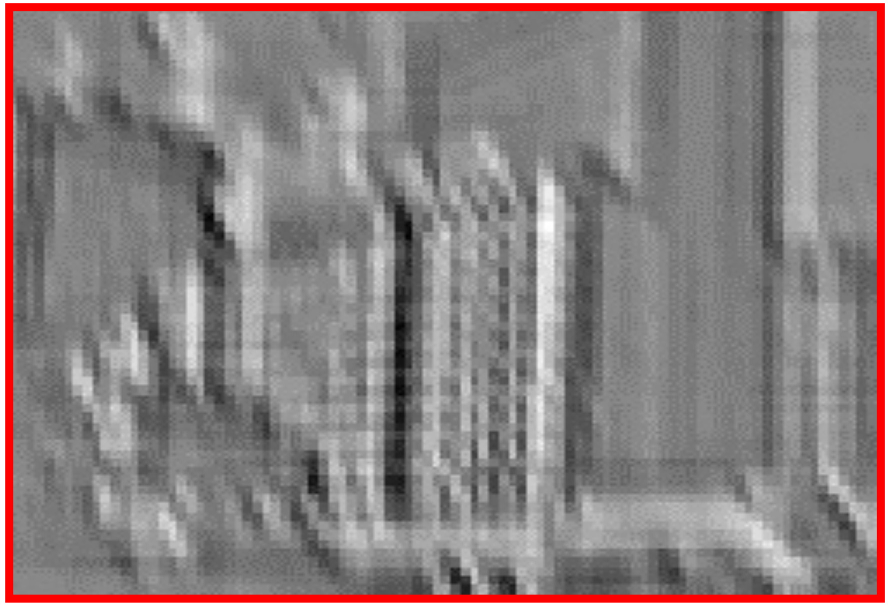
Convolution Filters



$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters



$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

1D Convolution Filters

- **Given:**
  - 1D sequence $x$is 1D
  - 1D **kernel** k

- Convolution is the following:

$$y[t] = \sum_{\tau=0}^{|k|-1} k[\tau] \cdot x[t+\tau]$$

- Technically **cross-correlation**

1D Convolution Filters

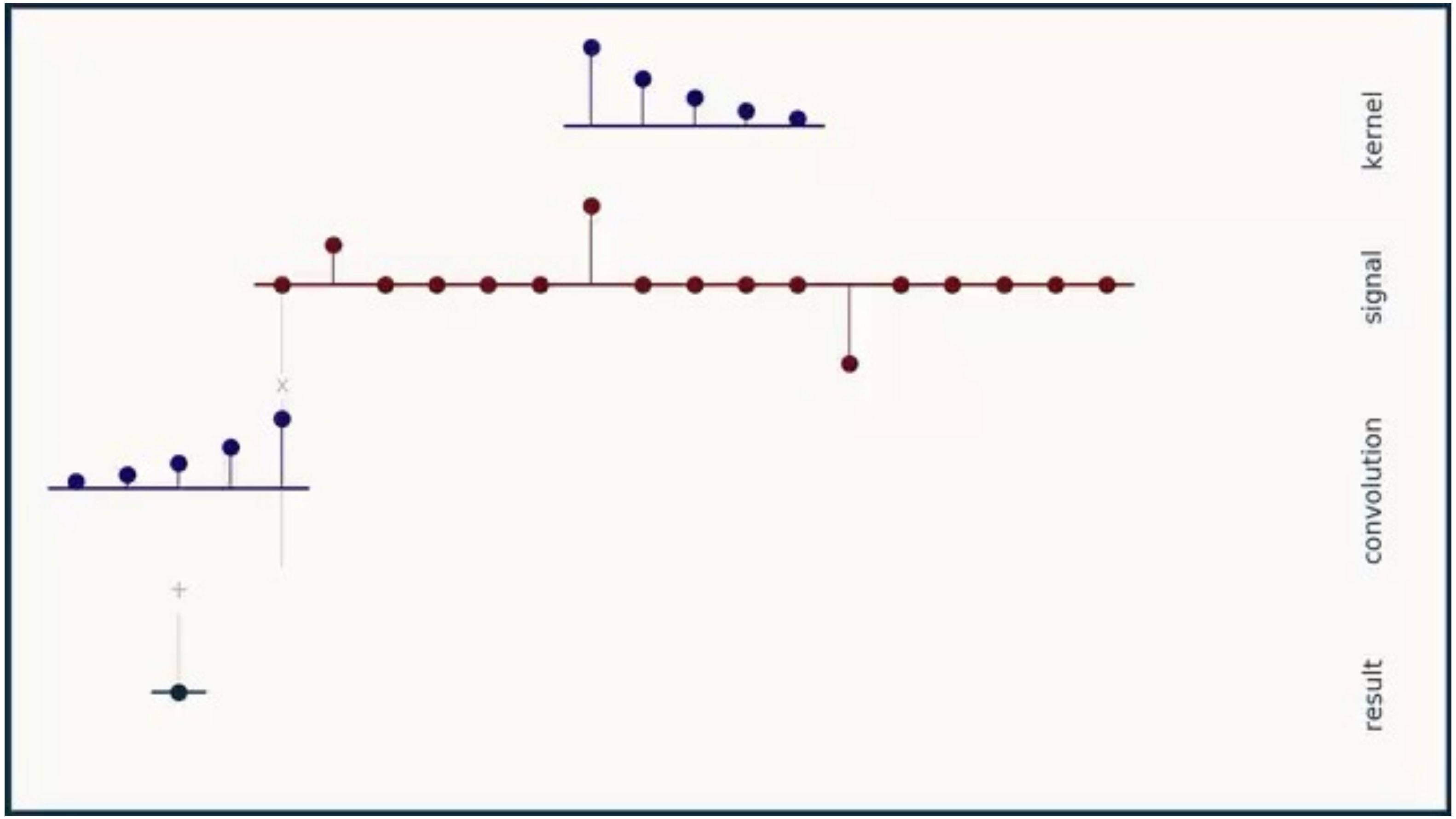
- **Example:**
  - $x = [25000, 28000, 30000, 21000, 18000, \ldots]$
  - $k = [-1, 1, -1]$

- **Convolution:**

$$y[t] = \sum_{\tau=0}^{|k|-1} k[\tau] \cdot x[t+\tau]$$

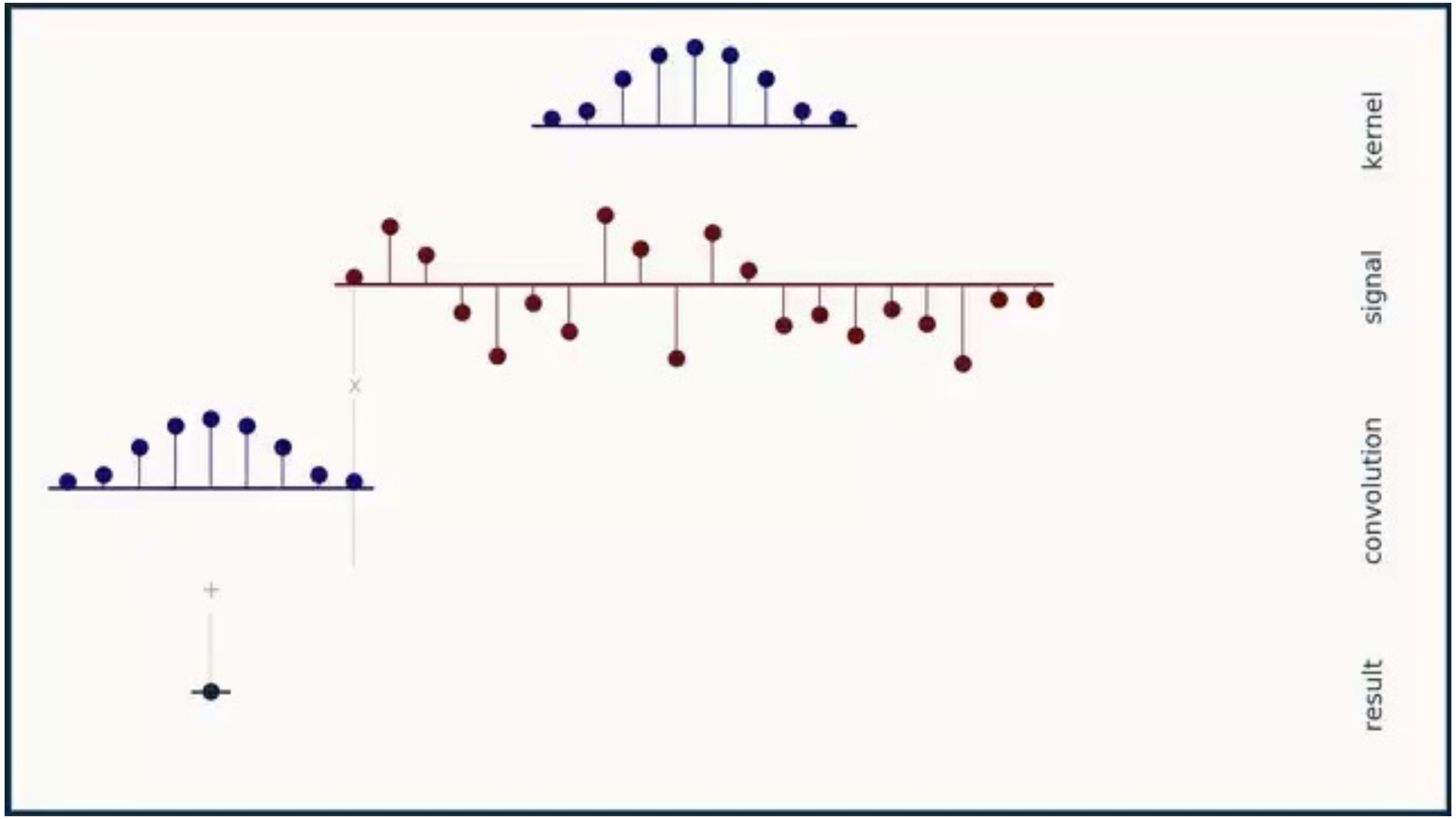
$$y[0] = k[0]x[0] + k[1]x[1] + k[2]x[2] = -25000 + 28000 - 30000$$
$$y[1] = k[0]x[1] + k[1]x[2] + k[2]x[3] = -28000 + 30000 - 21000$$
$$y[2] = k[0]x[2] + k[1]x[3] + k[2]x[4] = -30000 + 21000 - 18000$$

1D Convolution Filters



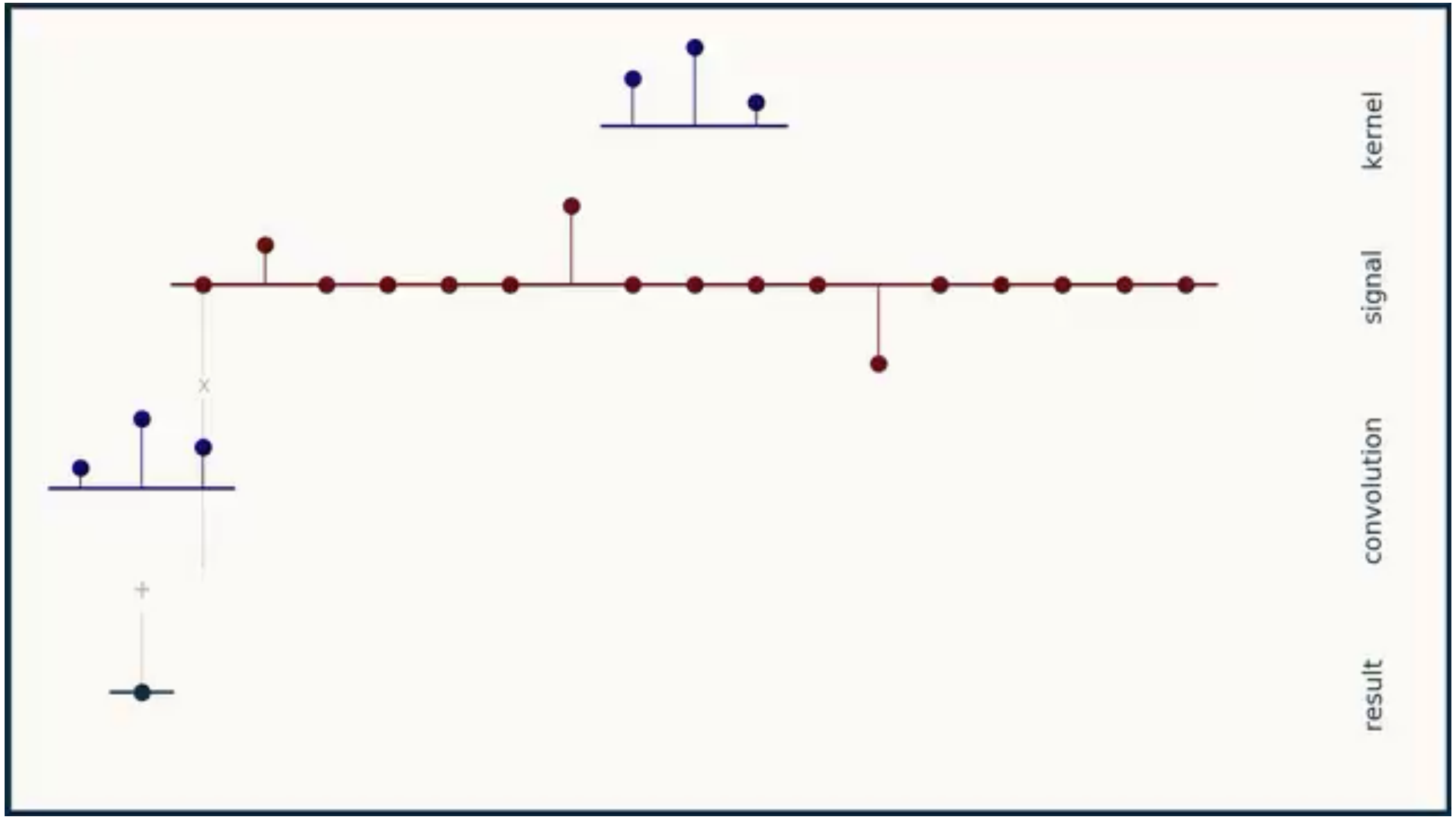
https://gitlab.com/brohrer/

1D Convolution Filters



https://gitlab.com/brohrer/

1D Convolution Filters



https://gitlab.com/brohrer/

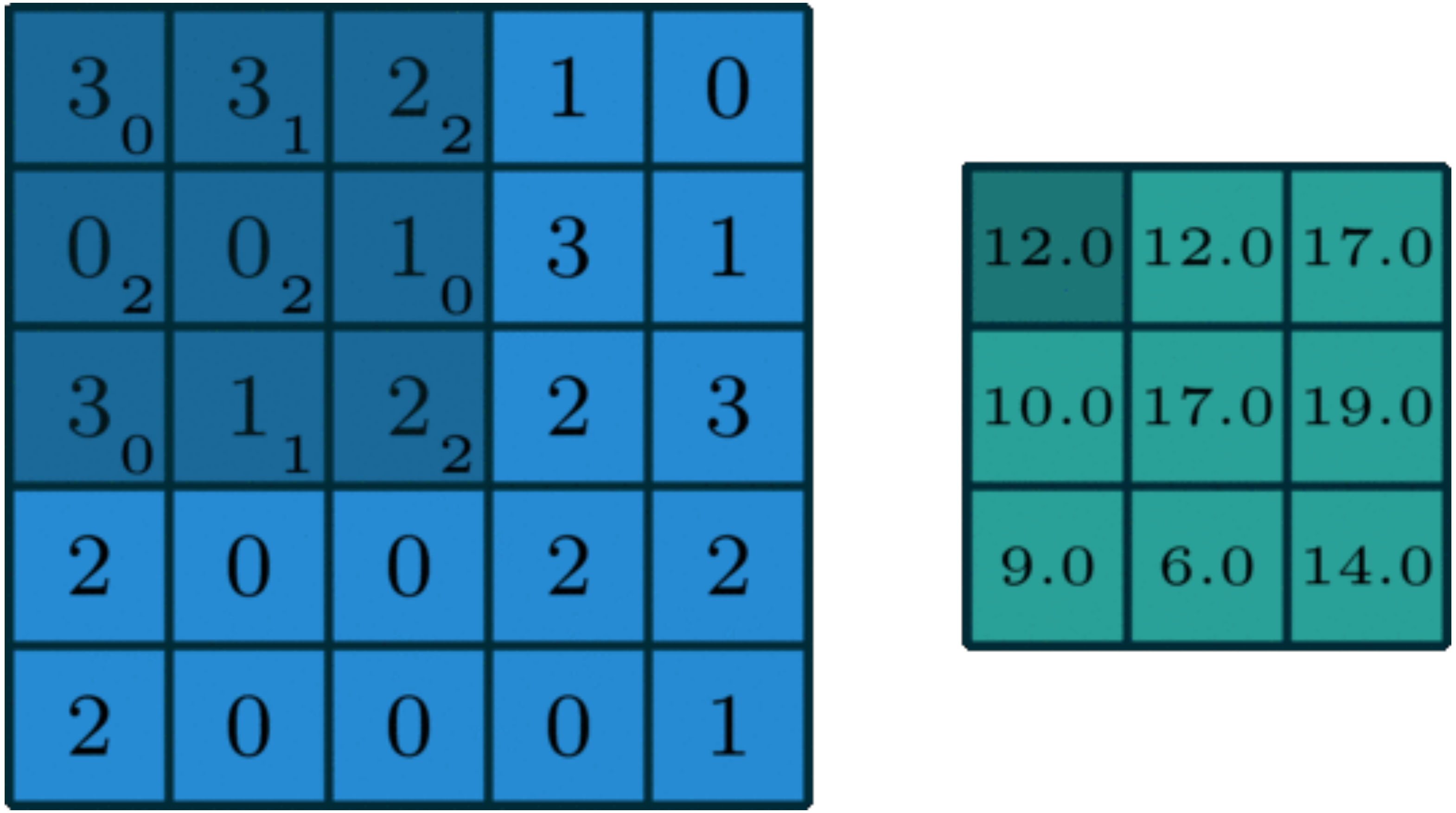
2D Convolution Filters

- **Given:**
  - A 2D input x
  - A 2D $h \times w$ kernel k

- The 2D convolution is:

$$y[s,t] = \sum_{\tau=0}^{h-1} \sum_{\gamma=0}^{w-1} k[\tau,\gamma] \cdot x[s+\tau, t+\gamma]$$

2D Convolution Filters



2D Convolution Filters

- Historically (until late 1980s), kernel parameters were handcrafted
  - E.g., “edge detectors”

2D Convolution Filters

-1	-1	-1
2	2	2
-1	-1	-1

Horizontal lines

-1	2	-1
-1	2	-1
-1	2	-1

Vertical lines

-1	-1	2
-1	2	-1
2	-1	-1

45 degree lines

2	-1	-1
-1	2	-1
-1	-1	2

135 degree lines

Example Edge Detection Kernels

Result of Convolution with Horizontal Kernel

https://aishack.in/tutorials/image-convolution-examples/

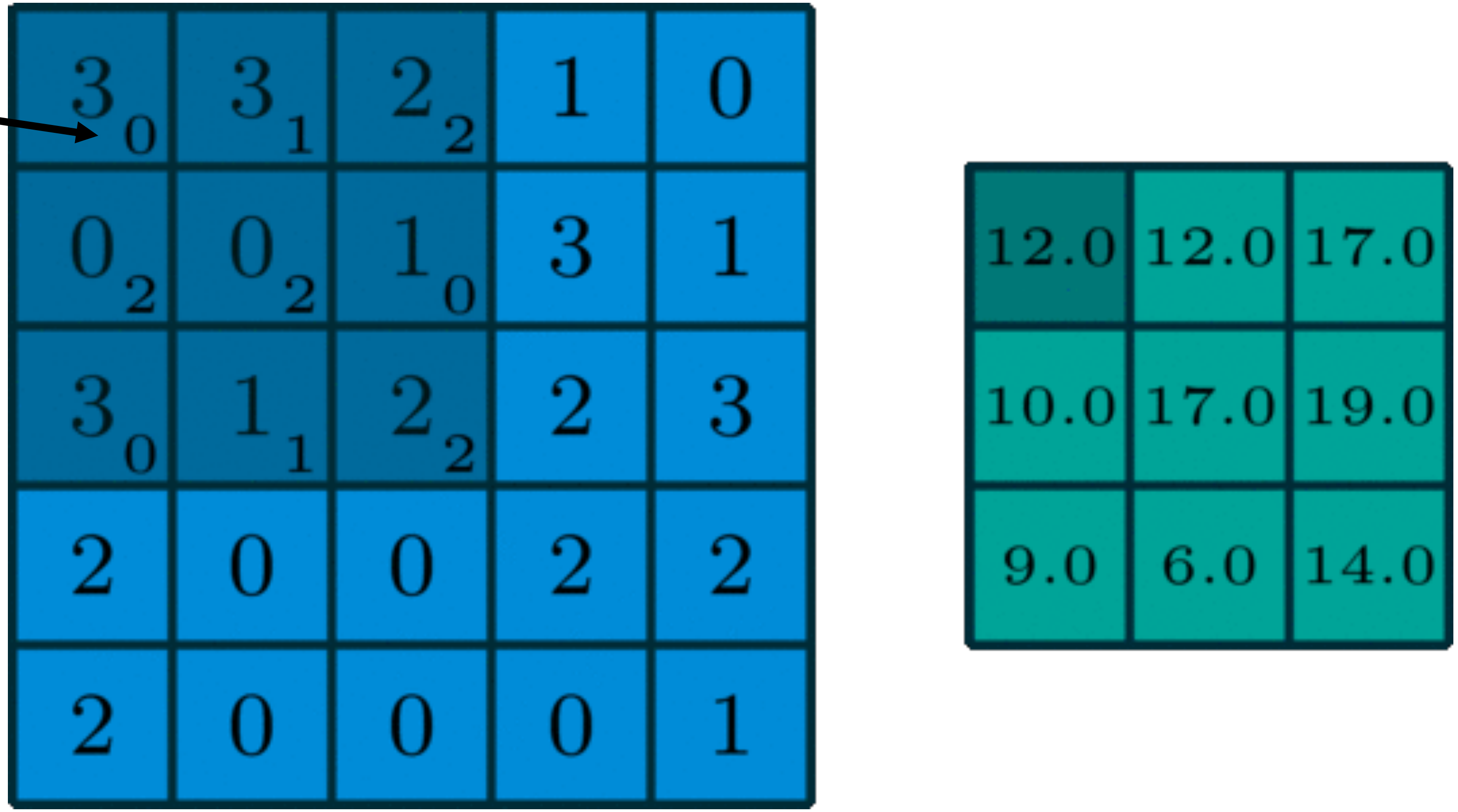
2D Convolution Filters

- Historically (until late 1980s), kernel parameters were handcrafted
  - E.g., “edge detectors”

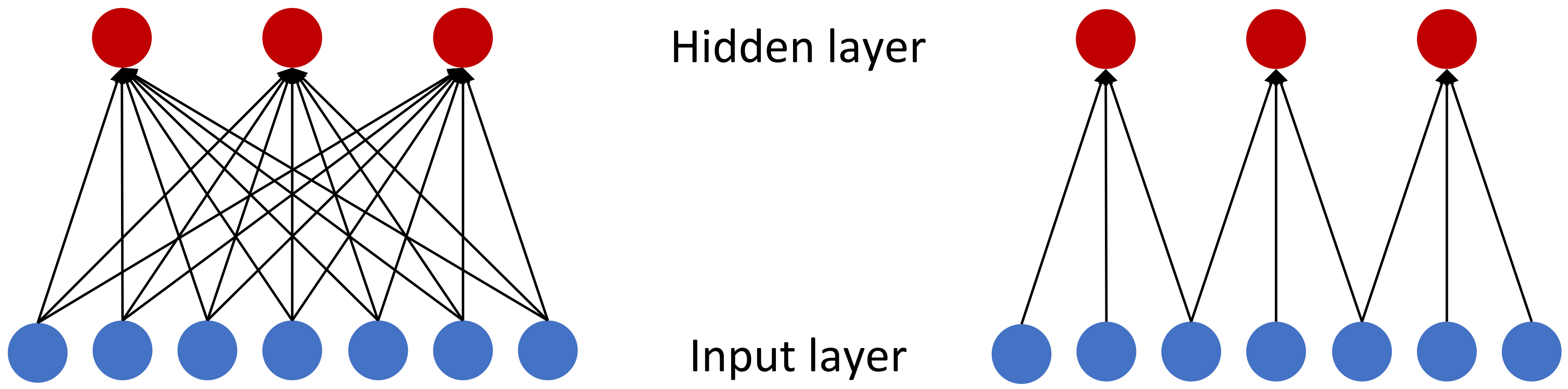
- In convolutional neural networks, they are learned
  - Essentially a linear layer with fewer “connections”
  - Backpropagate as usual!

Convolution Layers

Learnable parameters



Convolution Layers

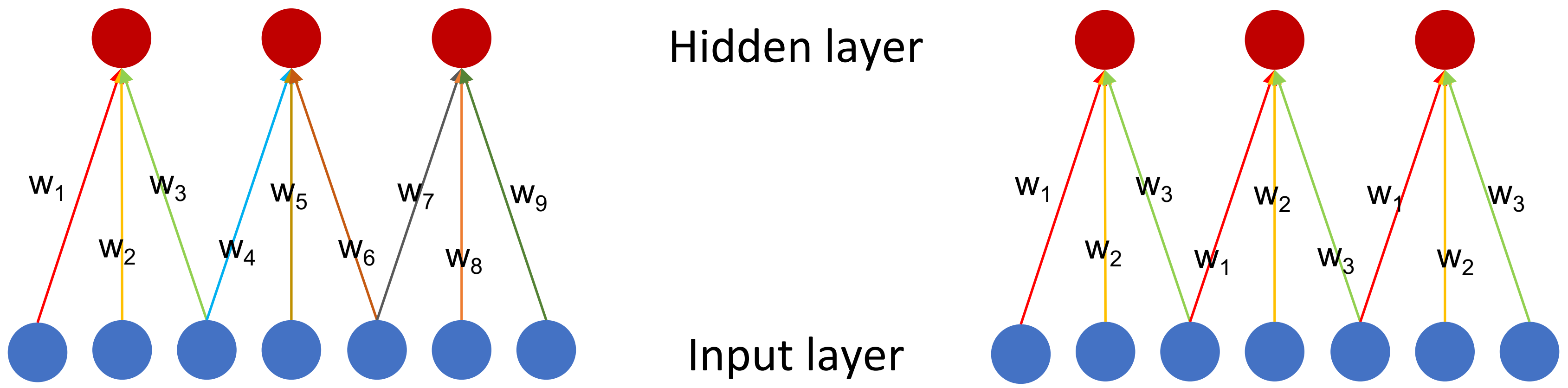


Fully connected
(3 input × 7 output = 21 parameters)

Locally connected
(3 input × 3 output = 9 parameters)

Slide credit: Jia-Bin Huang

Convolution Layers

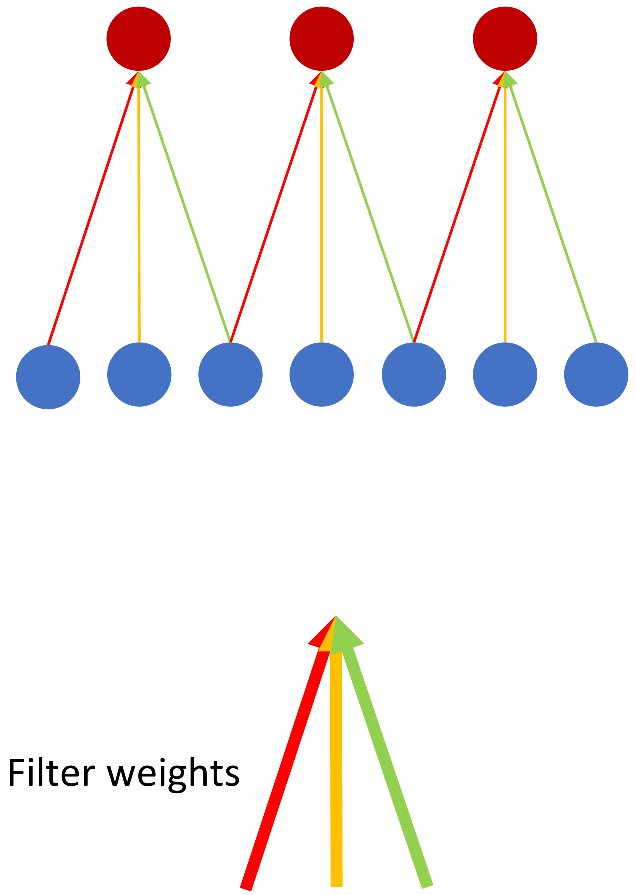


Without weight sharing
(3 input × 3 output = 9 parameters)

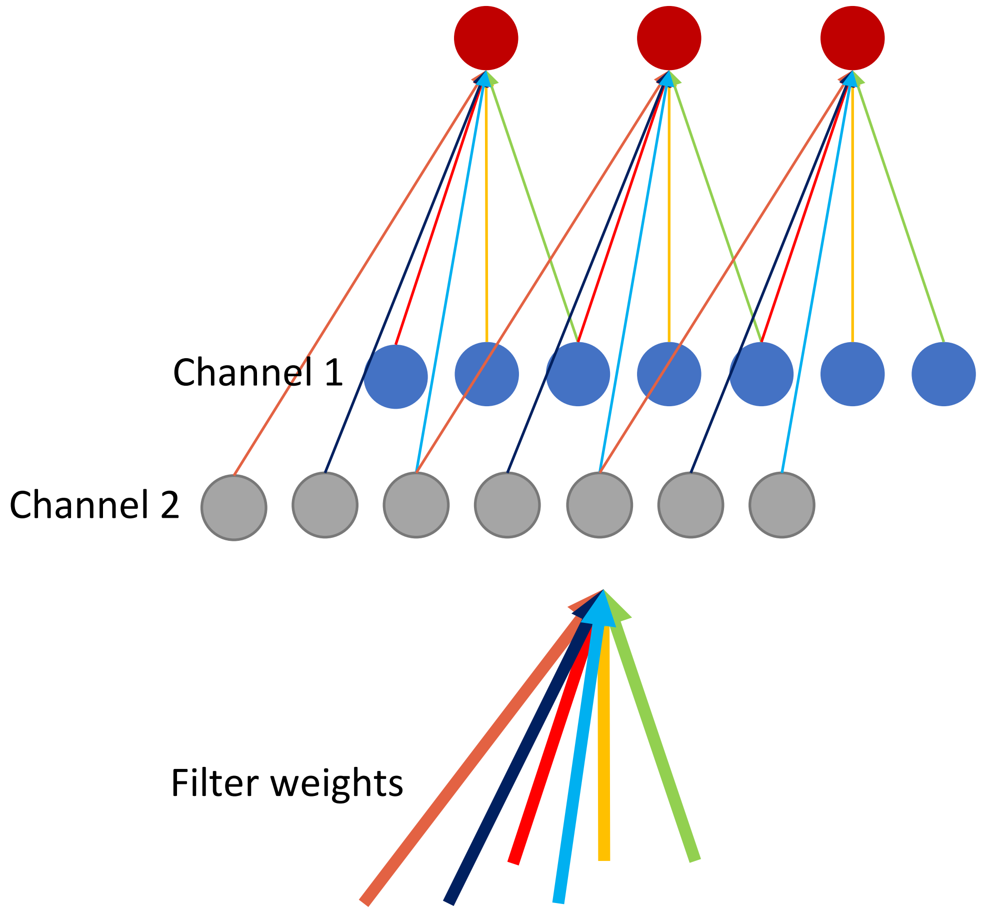
With weight sharing
(3 parameters)

Slide credit: Jia-Bin Huang

Convolution Layers



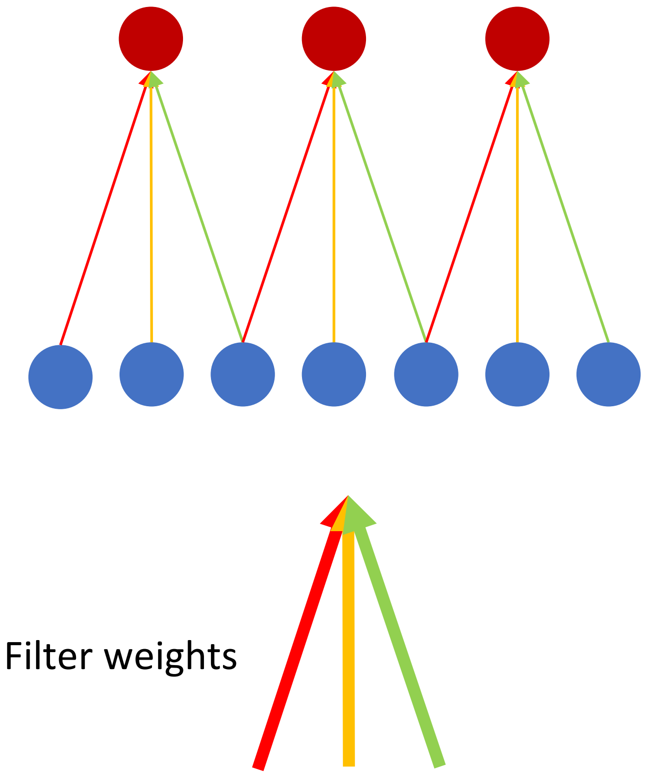
Single input channel



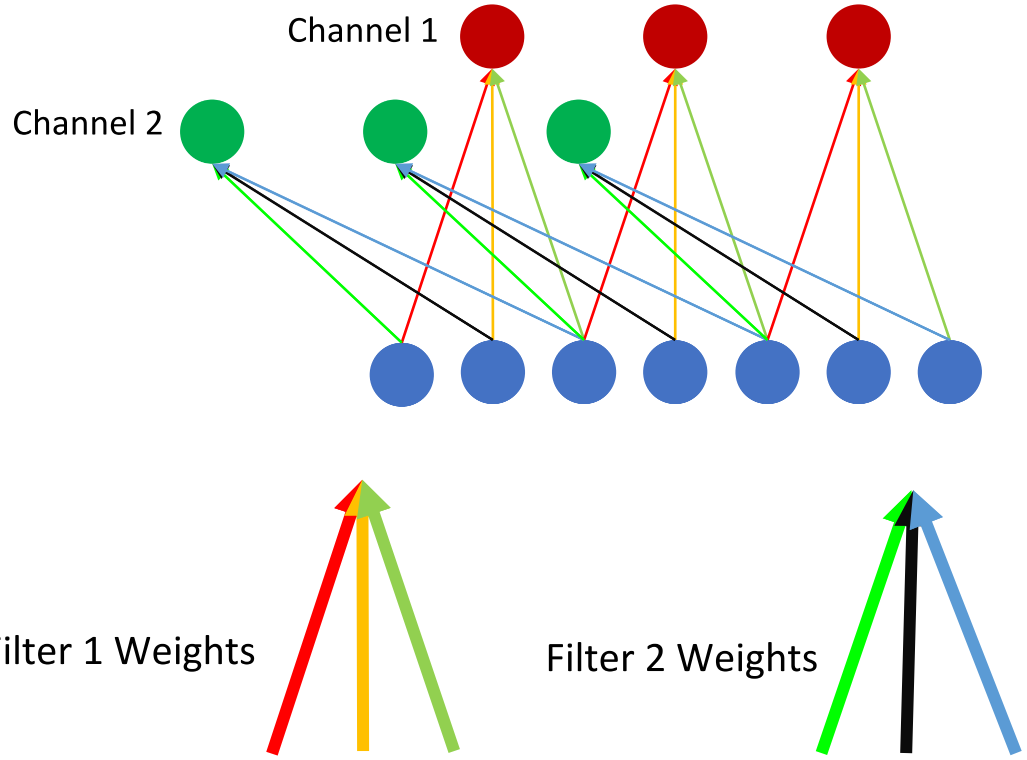
Multiple input channels

Slide credit: Jia-Bin Huang

Convolution Layers



Single output map



Multiple output maps

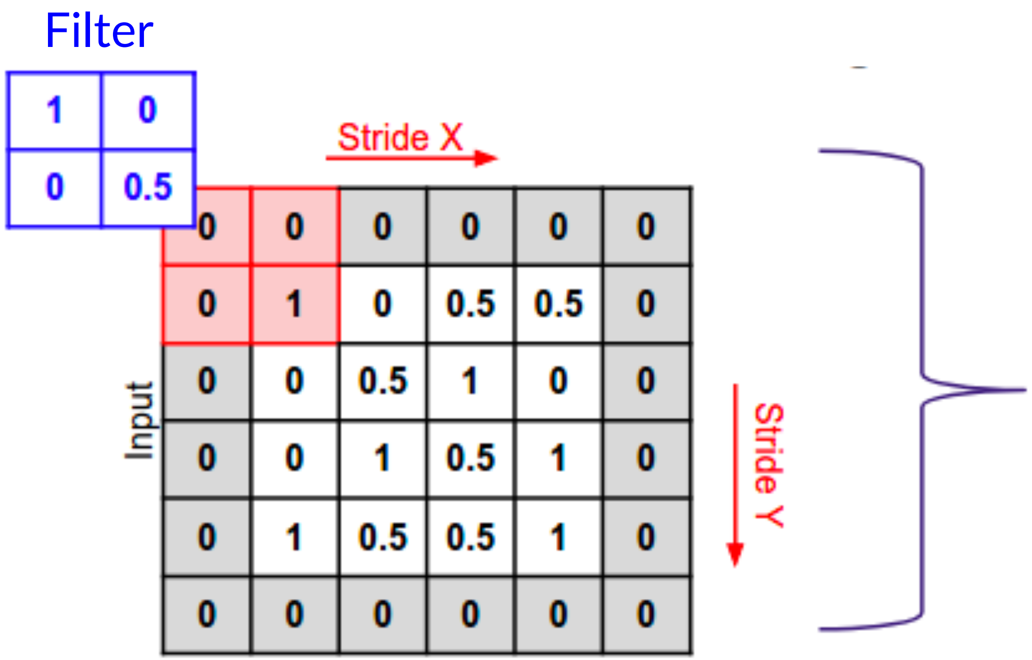
Slide credit: Jia-Bin Huang

Convolution Layers

- **Summary**
  - Local connectivity
  - Weight sharing
  - Handling multiple input/output channels
  - Retains location associations

Convolution Layer Parameters

- **Stride:** How many pixels to skip (if any)
  - **Default:** Stride of 1 (no skipping)

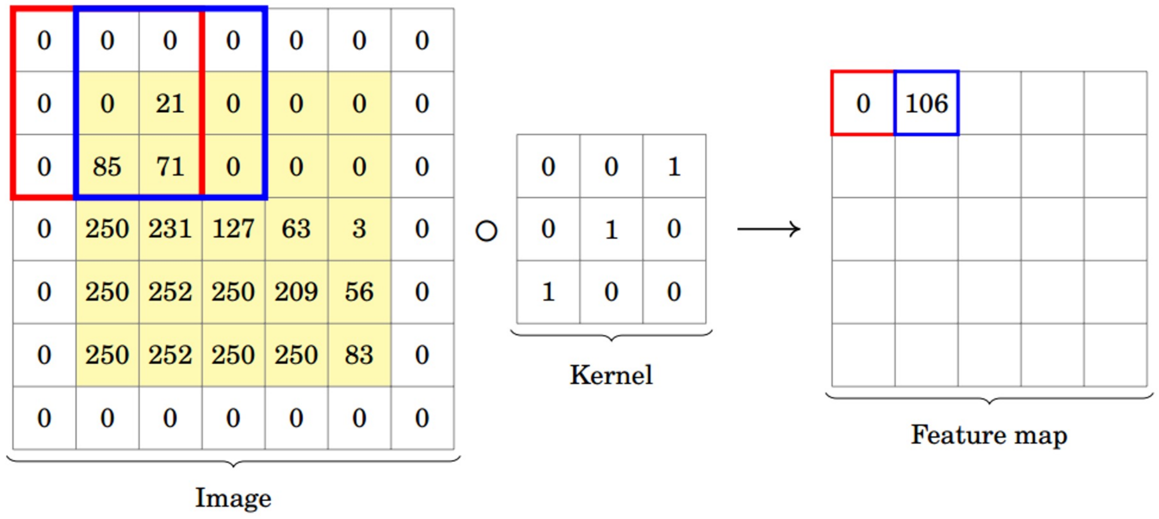


$$\text{OutDimension} = \frac{\text{InputDimension}}{\text{StrideDimension}}$$

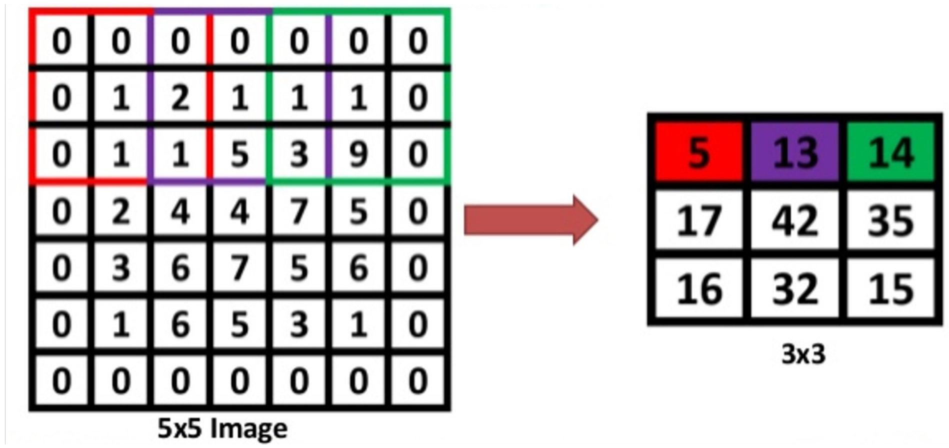
https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Padding:** Add zeros to edges of image to capture ends
  - **Default:** No padding



stride = 1, zero-padding = 1



stride = 2, zero-padding = 1

https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Summary:** Hyperparameters
  - Kernel size
  - Stride
  - Amount of zero-padding
  - Output channels

- Together, these determine the relationship between the input tensor shape and the output tensor shape

- Typically, also use a single bias term for each convolution filter

Convolution Layers

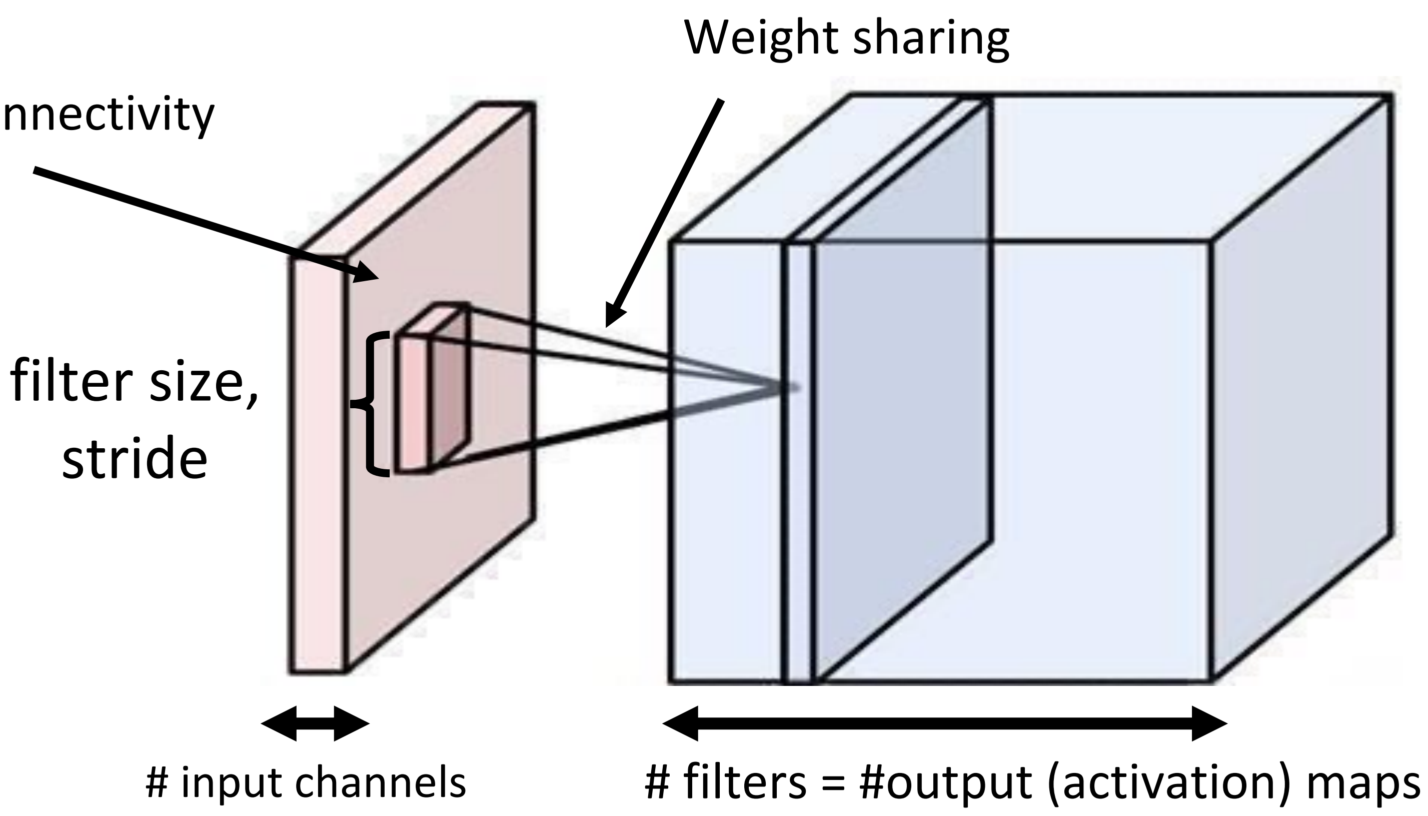
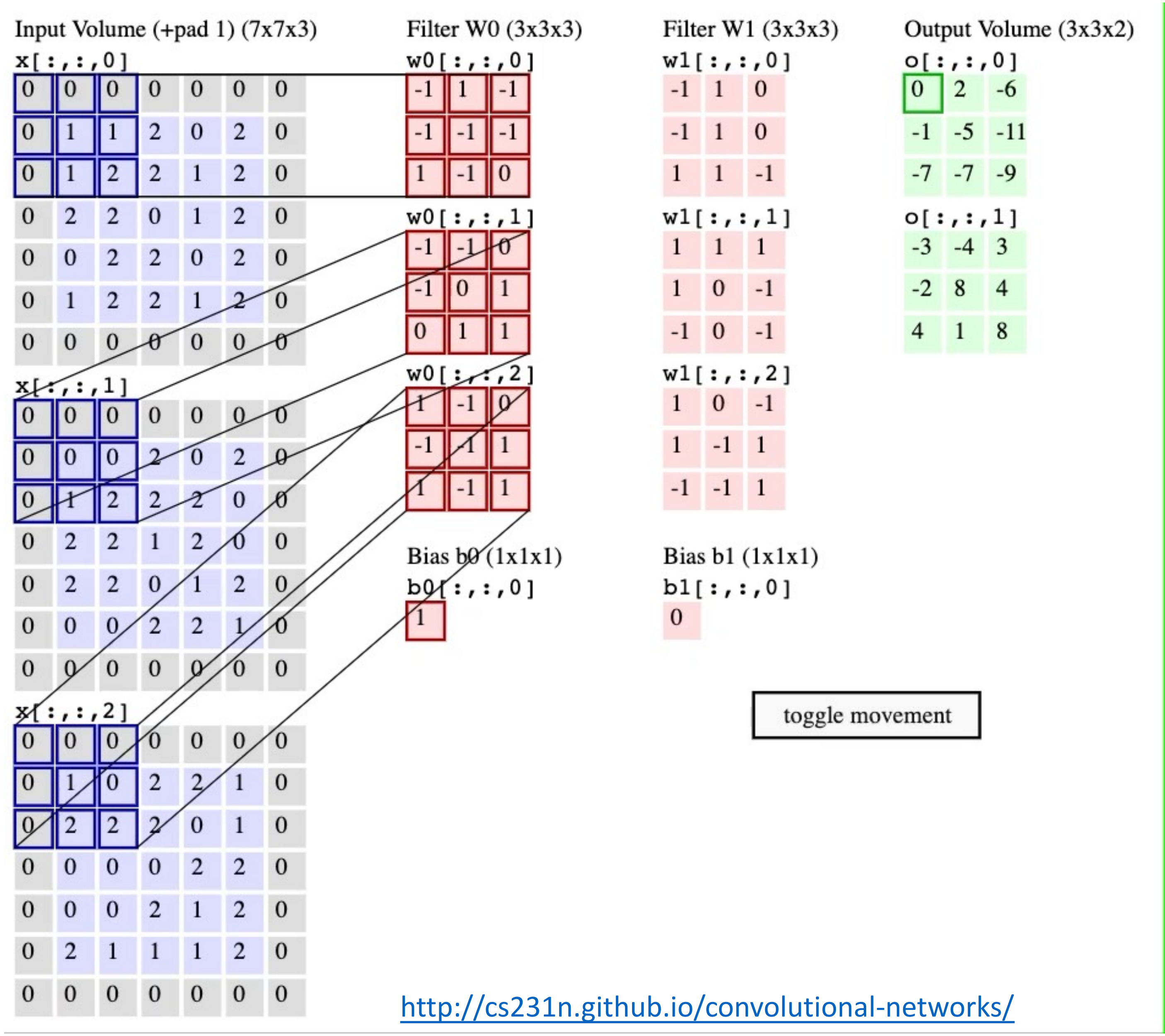


Image credit: A. Karpathy Slide credit: Jia-Bin Huang

Example

- Kernel size 3, stride 2, padding 1
- 3 input channels
  - Hence kernel size 3×3×3
- 2 output channels
  - Hence 2 kernels
- Total # of parameters:
  - $(3\times3\times3 + 1)\times2 \; = \; 56$



http://cs231n.github.io/convolutional-networks/