Maths of AI -

Deepness & Convolution

Instructor - Simon Lucey

THE UNIVERSITY of ADELAIDE

AUSTRALIAN INSTITUTE FOR MACHINE LEARNING

Let’s have a play!!!

https://colab.research.google.com/github/Tobias-Fischer/RVSS2022/blob/main/Visual_Learning/Session1/Classification_MLP_2021.ipynb

Some things to try!!!

- What happens to performance if you use a linear function?

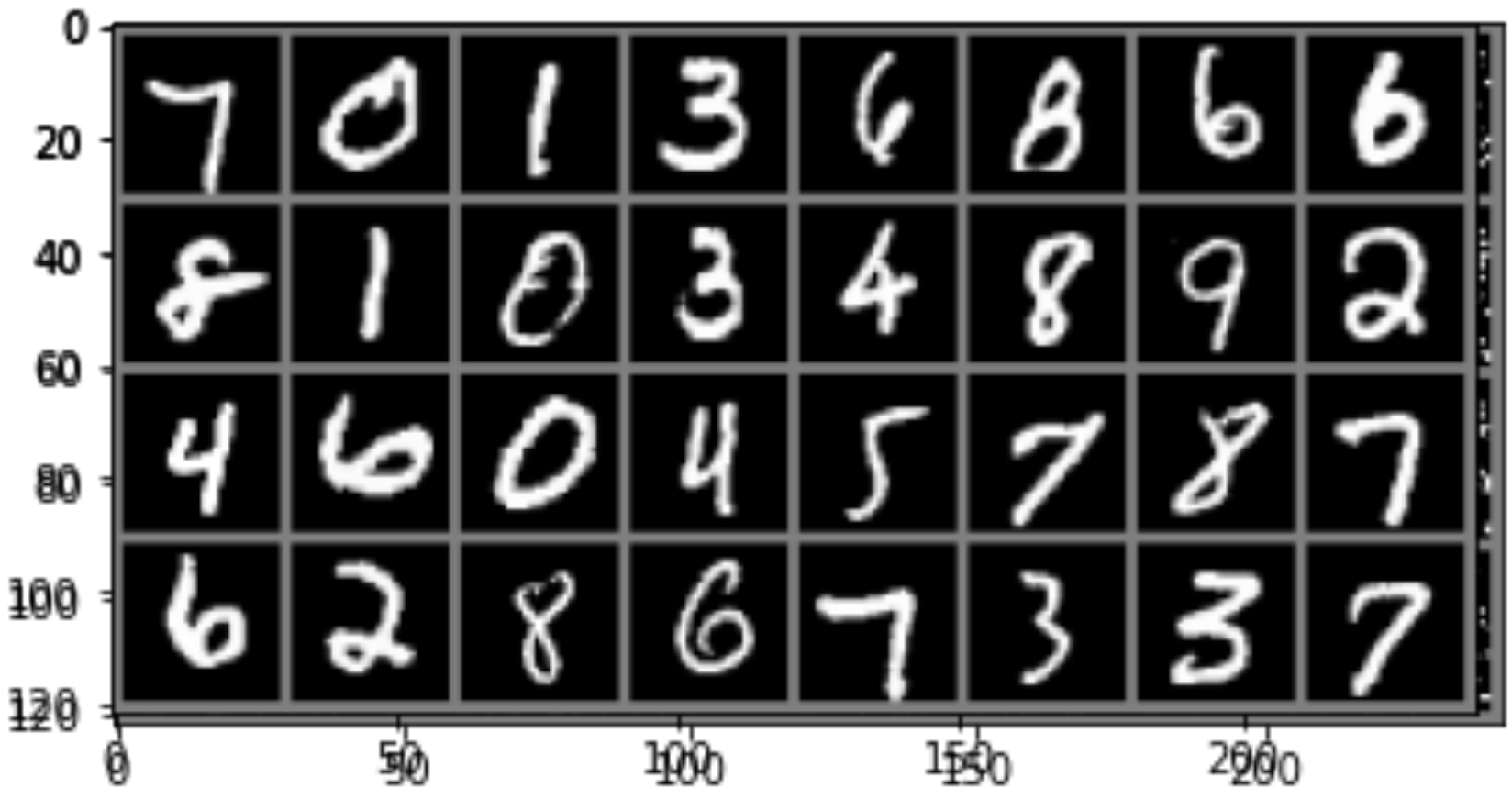
```
class MLPNet(nn.Module):
    def __init__(self):
        super(MLPNet, self).__init__()

        # First fully connected layers input image is 28x28 = 784 dim.
        self.fc0 = nn.Linear(784, 10) # nparam = 784*256 = 38400
        # Two more fully connected layers
        #self.fc1 = nn.Linear(256, 84)
        #self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        # Flattens the image like structure into vectors
        x = torch.flatten(x, start_dim=1)

        # fully connected layers with activations
        x = self.fc0(x)
        #x = F.relu(x)
        #x = self.fc1(x)
        #x = F.relu(x)
        #x = self.fc2(x)
        # Outputs are log(p) so softmax followed by log.
        #return(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Why don’t you need to regularize?

Some things to try!!!

- What happens to performance if you use a linear function?
- What happens when you permute the pixels?

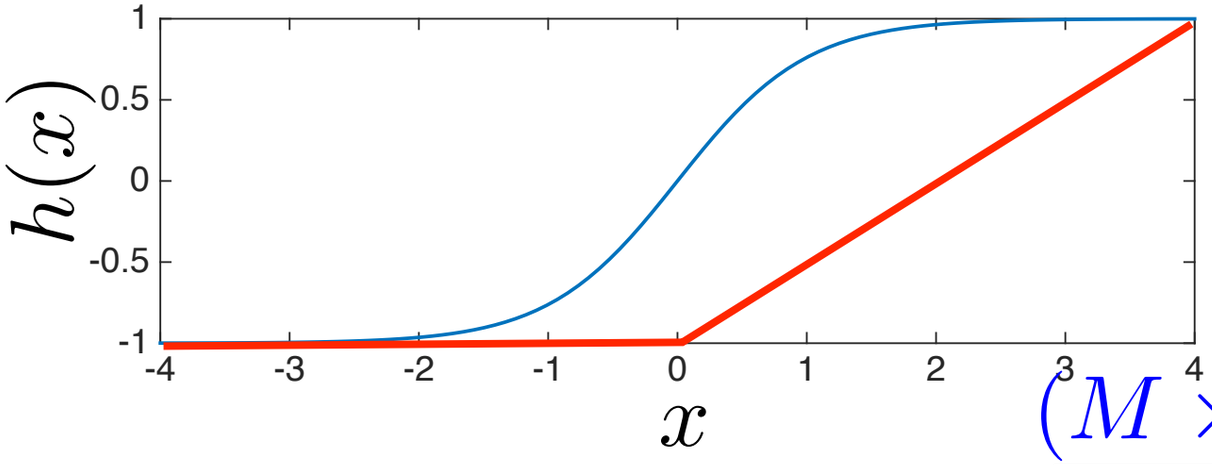
```
from numpy.random import permutation
idx_permute = torch.from_numpy(permutation(784))
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Lambda(lambda x: x.view(-1)[idx_permute].view(1, 28, 28) ),
                               transforms.Normalize((0.5,), (0.5,)),
                               ])
```

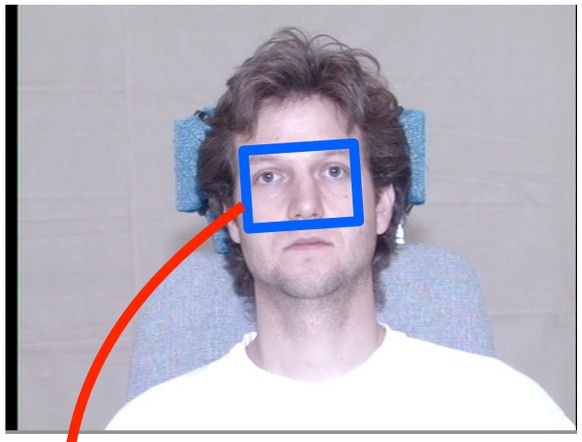


Today

- Shallow Visual Learning
- Deep Visual Learning
- Convolutional Neural Network

Two-Layer Perceptron





$(M \times D)$

$(1 \times M)$

$\mathbf{x} \in \mathbb{C}_1$

$\geq$ 0

$<$

$\mathbf{x} \in \mathbb{C}_2$

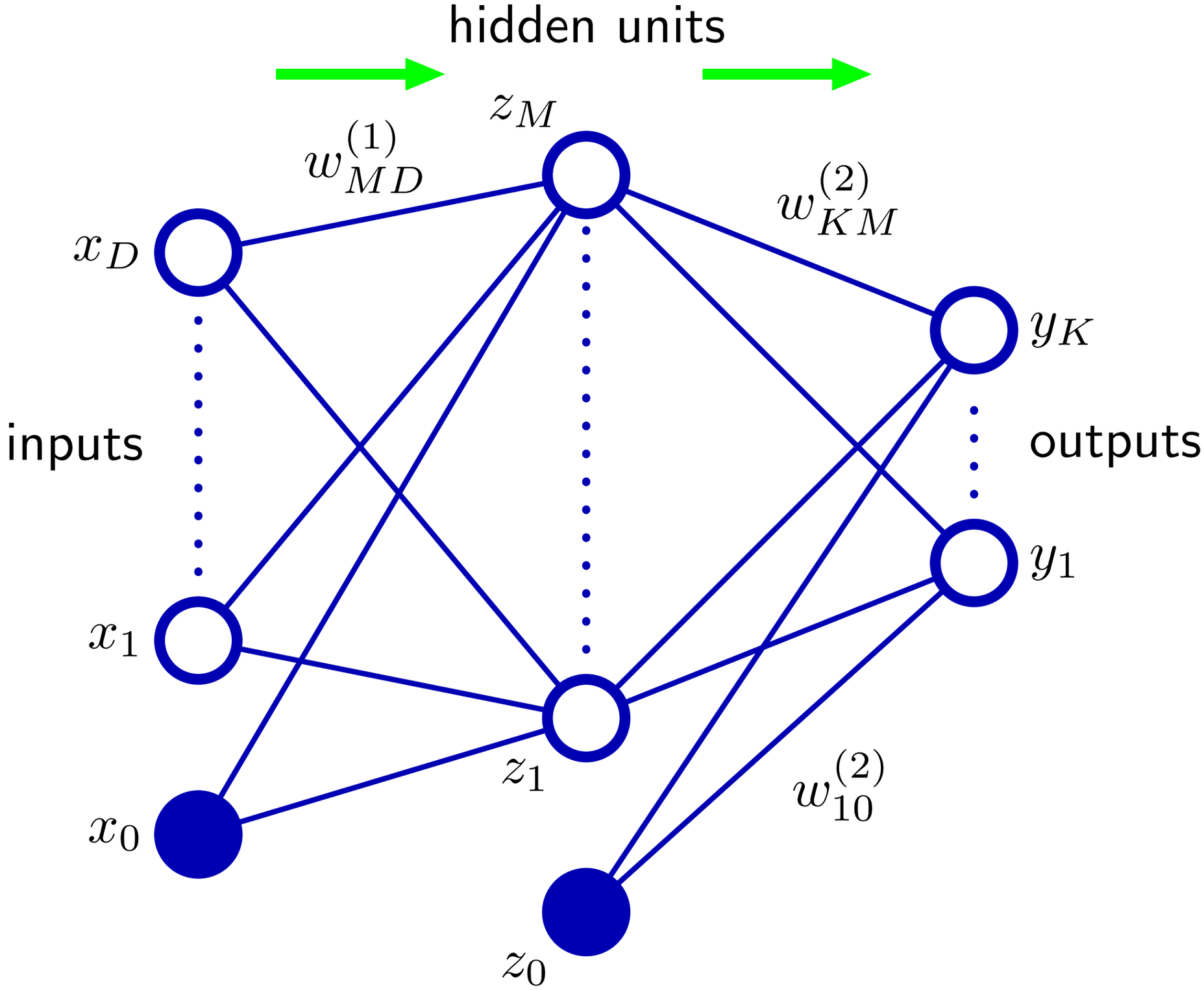
$[\mathbf{w}^{(2)}]^T$

$h(\mathbf{W}^{(1)}\mathbf{x})$

$\mathbf{W}^{(1)}$

$\mathbf{x}$

Multi-Layer Perceptron



Layer 1 - MLP

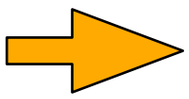
$$\mathbf{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_M \end{bmatrix} \leftarrow \begin{bmatrix} h[\mathbf{x}^T \mathbf{w}_1^{(1)}] \\ \vdots \\ h[\mathbf{x}^T \mathbf{w}_M^{(1)}] \end{bmatrix}$$

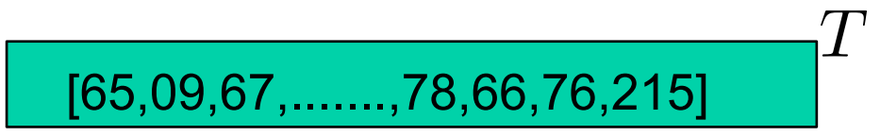
$h() =$ non-linear function

$[\mathbf{w}_1^{(1)}, \ldots, \mathbf{w}_M^{(1)}] =$ 1st layer's $D \times M$ weights

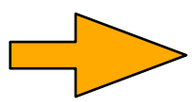
$\mathbf{x} = D \times 1$ raw input

Layer 2 - MLP





$$\mathbf{x} \in \mathbb{R}^D$$



$$\mathbf{z} \in \mathbb{R}^M$$

$$\mathbf{z} \in \mathbb{C}_1$$

$$\mathbf{z}^T \mathbf{w}^{(2)} \quad \begin{matrix} \geq \\ < \end{matrix} \quad 0$$

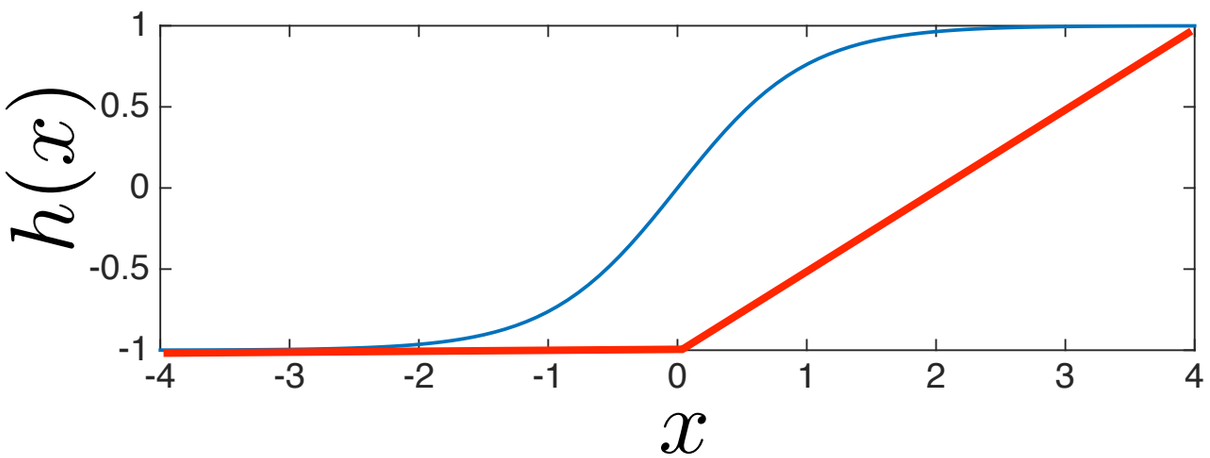
$$\mathbf{z} \in \mathbb{C}_2$$

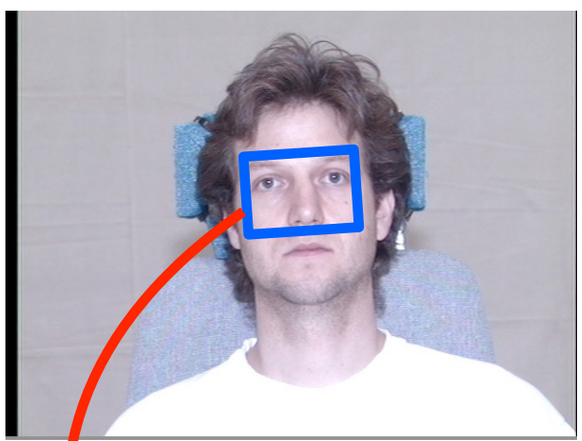
$\mathbf{z} = M \times 1$ output of layer 1
$\mathbf{w}^{(2)} =$ 2nd layer's $M \times 1$ weight vector

Example in Code

```
class MLPNet(nn.Module):
    def __init__(self):
        super(MLPNet, self).__init__()

        # First fully connected layers input image is 28x28 = 784 dim.
        self.fc0 = nn.Linear(784, 256) # nparam = 784*256 = 38400
        # Two more fully connected layers
        self.fc1 = nn.Linear(256, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        # Flattens the image like structure into vectors
        x = torch.flatten(x, start_dim=1)

        # fully connected layers with activations
        x = self.fc0(x)
        x = F.relu(x)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
```

Why the Non-Linearity?





$\mathbf{x} \in \mathbb{C}_1$

≥ 0
$<$

$$[\mathbf{w}^{(2)}]^T h(\mathbf{W}^{(1)}\mathbf{x})$$

$\mathbf{x} \in \mathbb{C}_2$

Shallow vs. Deep Learning

- A shallow learner has **only one** hidden unit,

$$\mathbf{W}^{(2)} h(\mathbf{W}^{(1)} \mathbf{x})$$

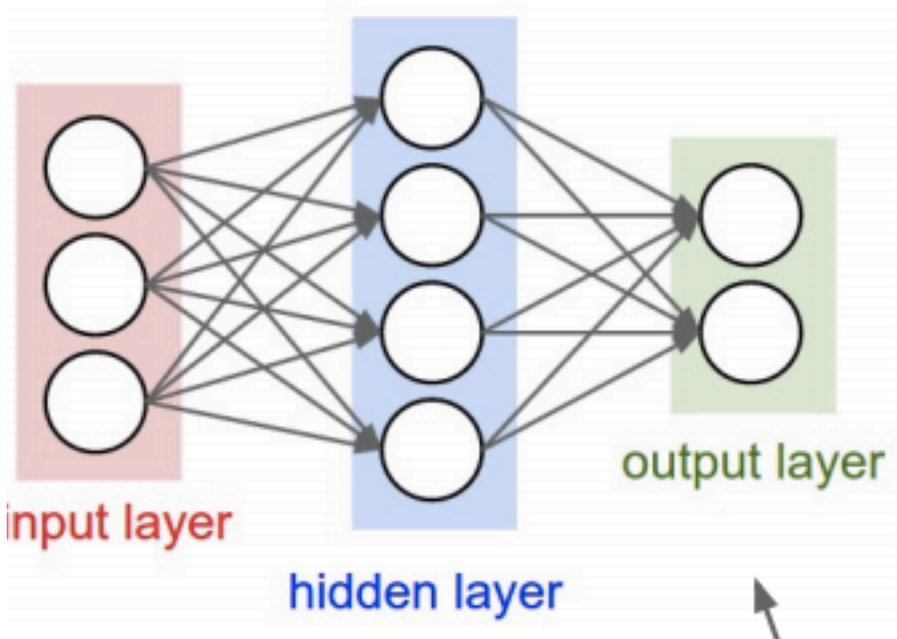
- A deep learner **has more than one** hidden unit,

$$\mathbf{W}^{(3)} h(\mathbf{W}^{(2)} h(\mathbf{W}^{(1)} \mathbf{x}))$$

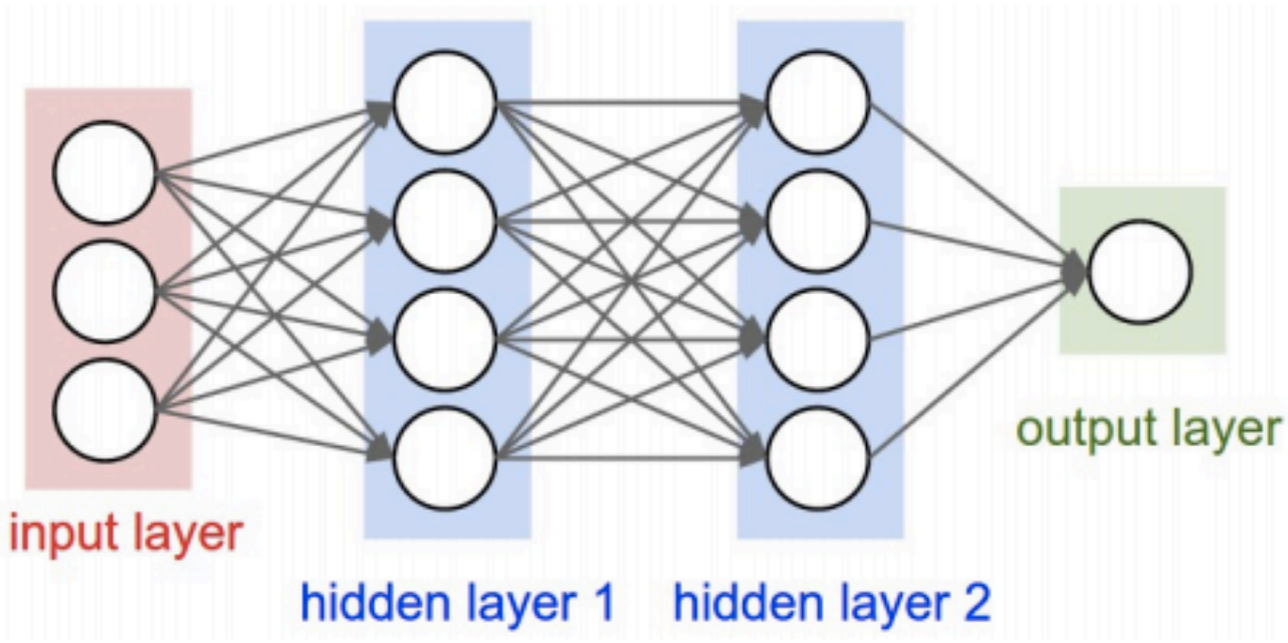
"Geoffrey Hinton"

Shallow vs. Deep Layers

“shallow”



“deep”



“2-layer neural net,” or
“1-hidden-layer neural net”

“Fully-connected” layers

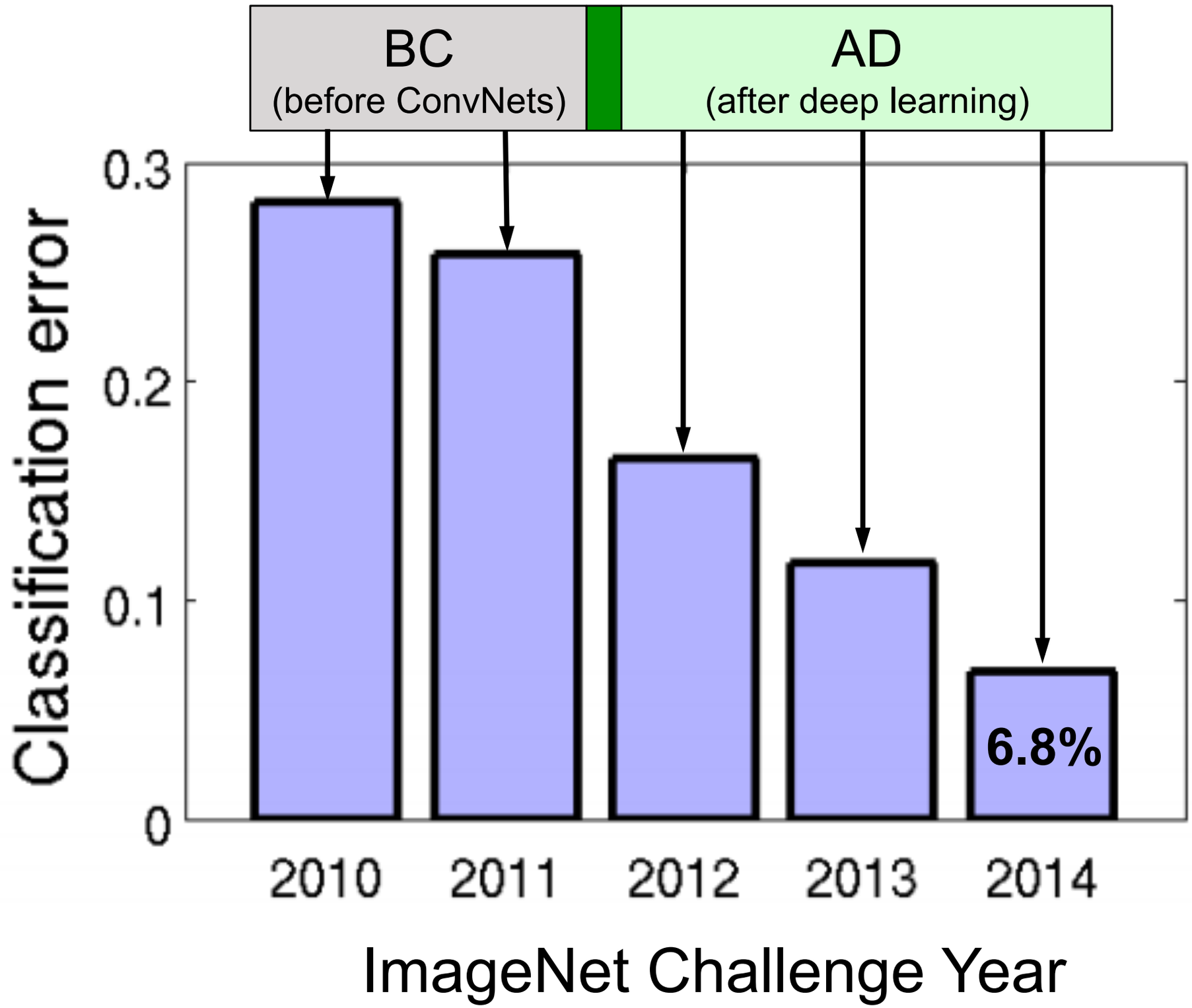
“3-layer neural net,” or
“2-hidden-layer neural net”

Why deepness?

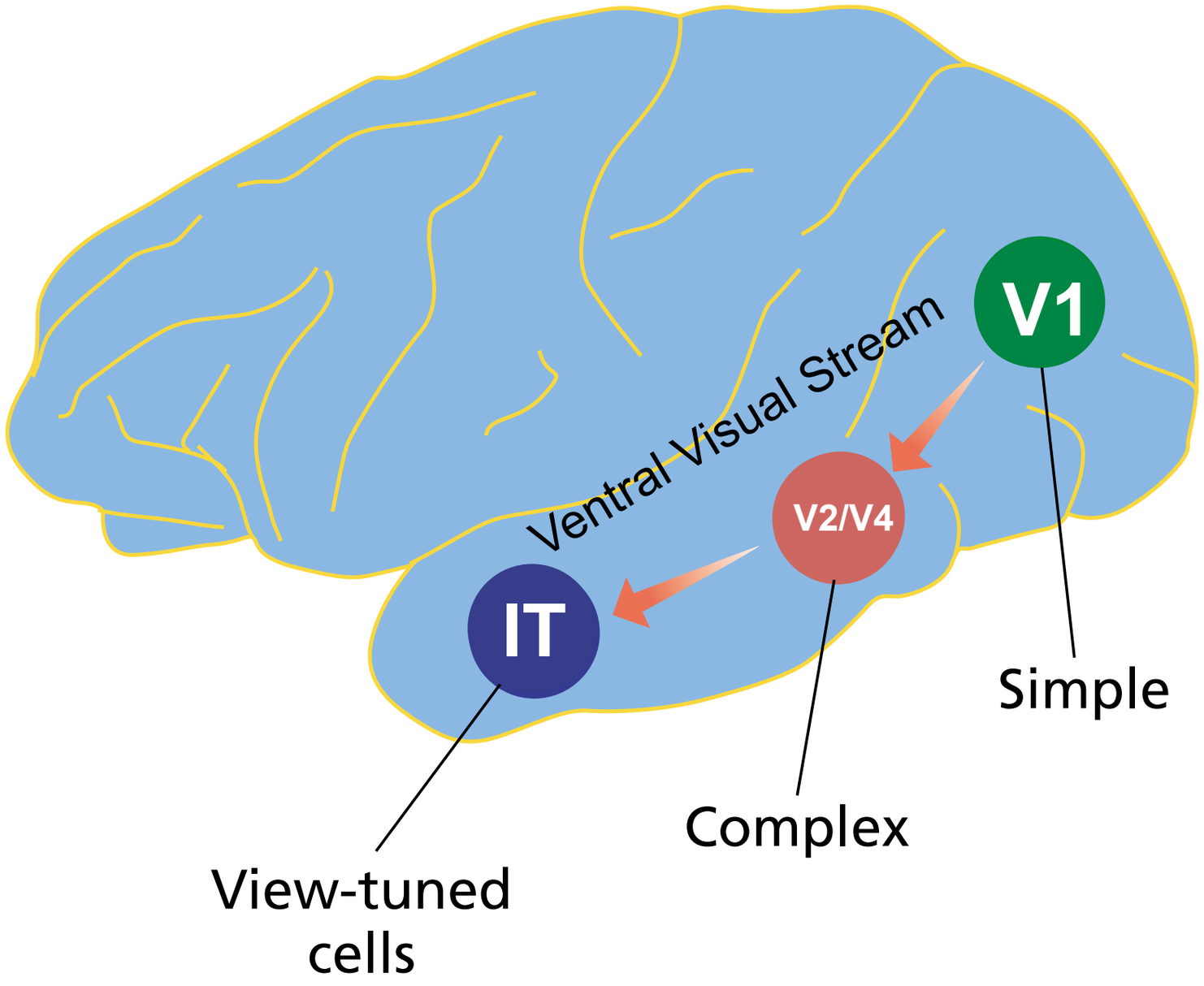
Today

- Shallow Visual Learning
- Deep Visual Learning
- Convolutional Neural Network

Deepness - Impact on Object Recognition

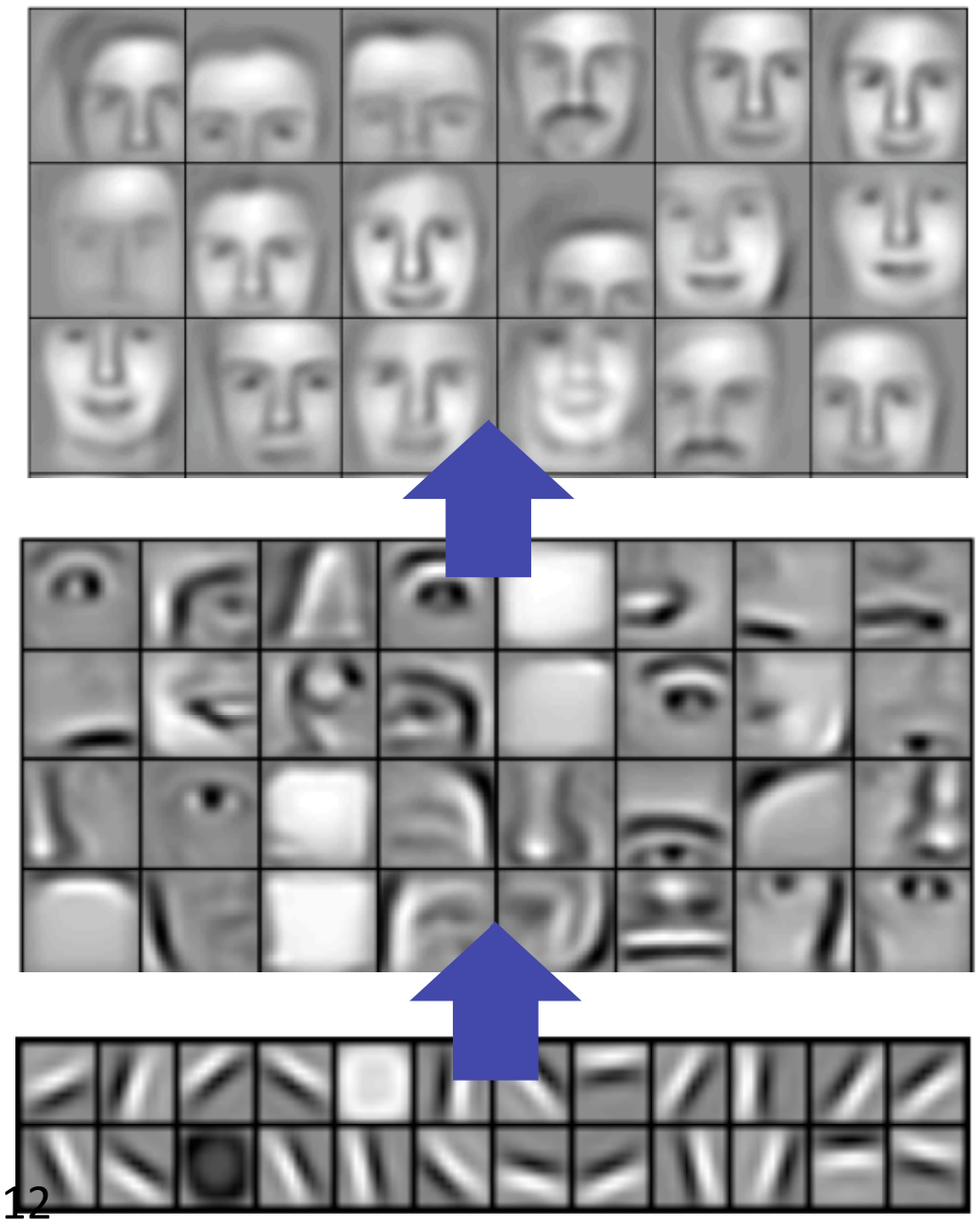


Reminder: Hierarchical Learning



Hierarchical Learning

Successive model layers learn deeper intermediate representations

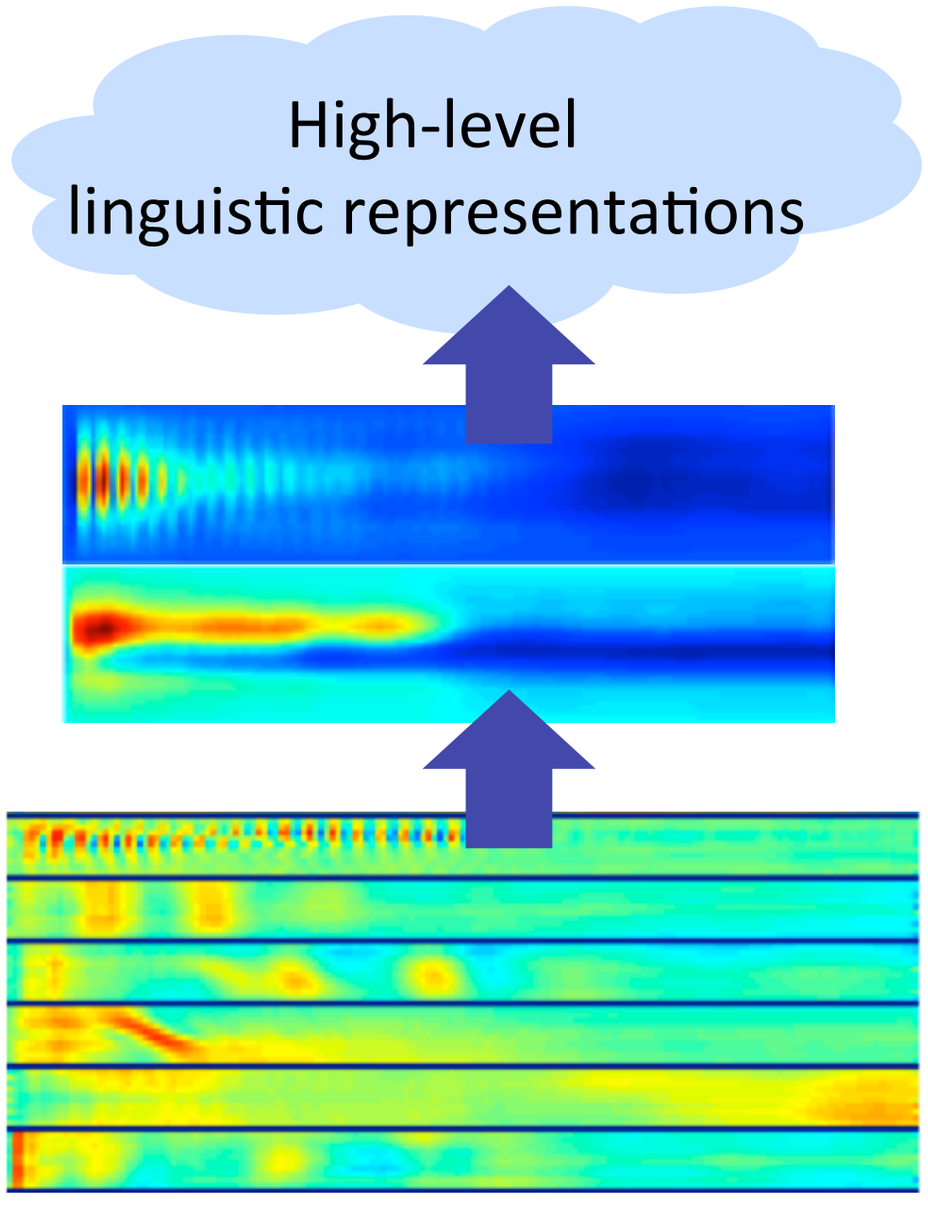


Layer 3

Parts combine to form objects

Layer 2

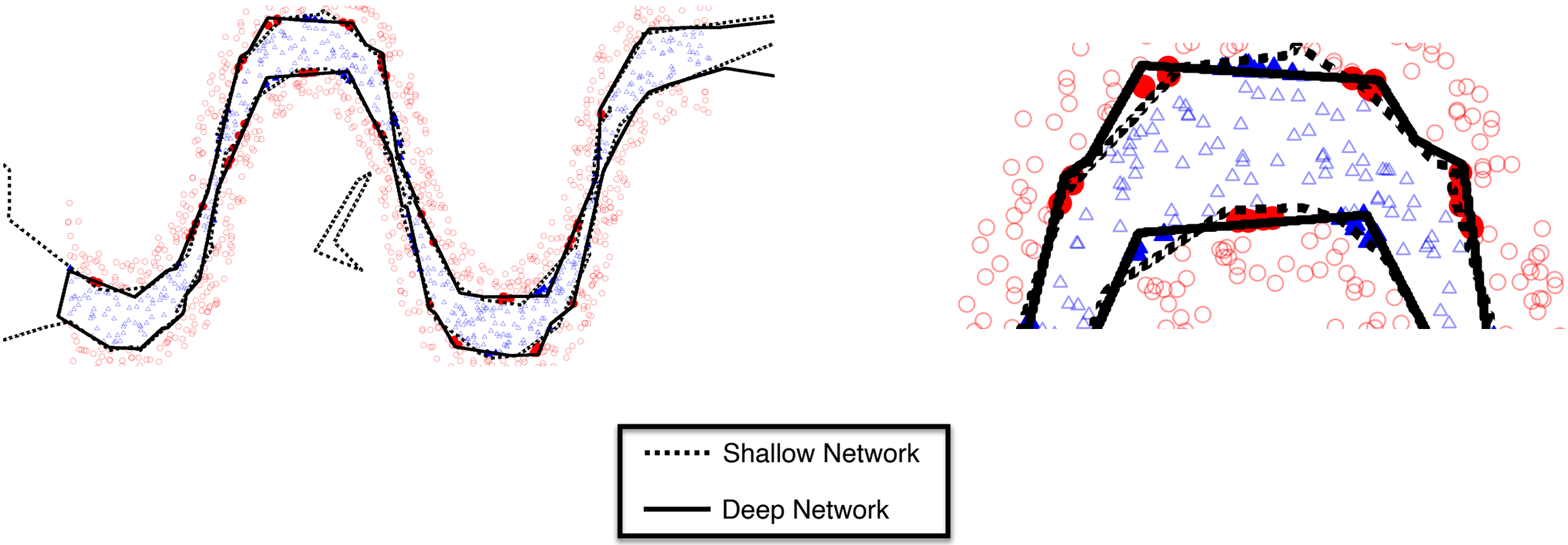
Layer 1



Prior: underlying factors & concepts compactly expressed w/ multiple levels of abstraction

Why Deep?

- Deep network can be considered as an MLP with several or more hidden layers.
- Deeper nets are exponentially more expressive than shallow ones.

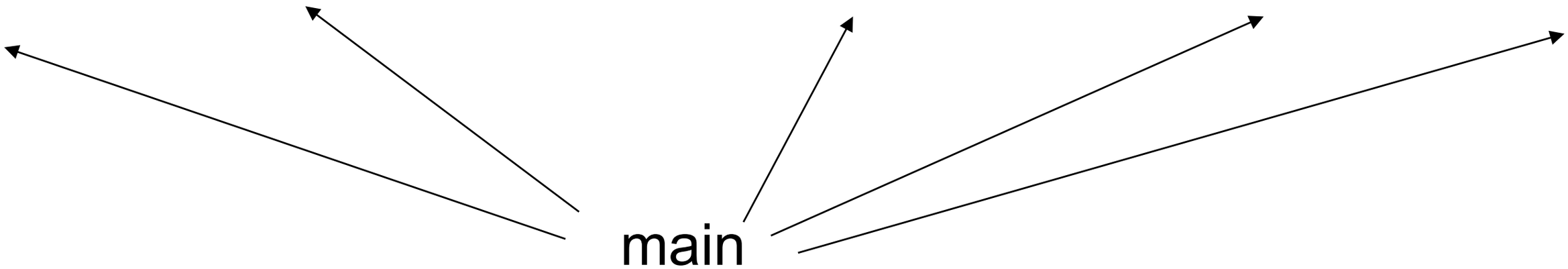


Montufar, Guido F., et al. "On the number of linear regions of deep neural networks." NIPS 2014.

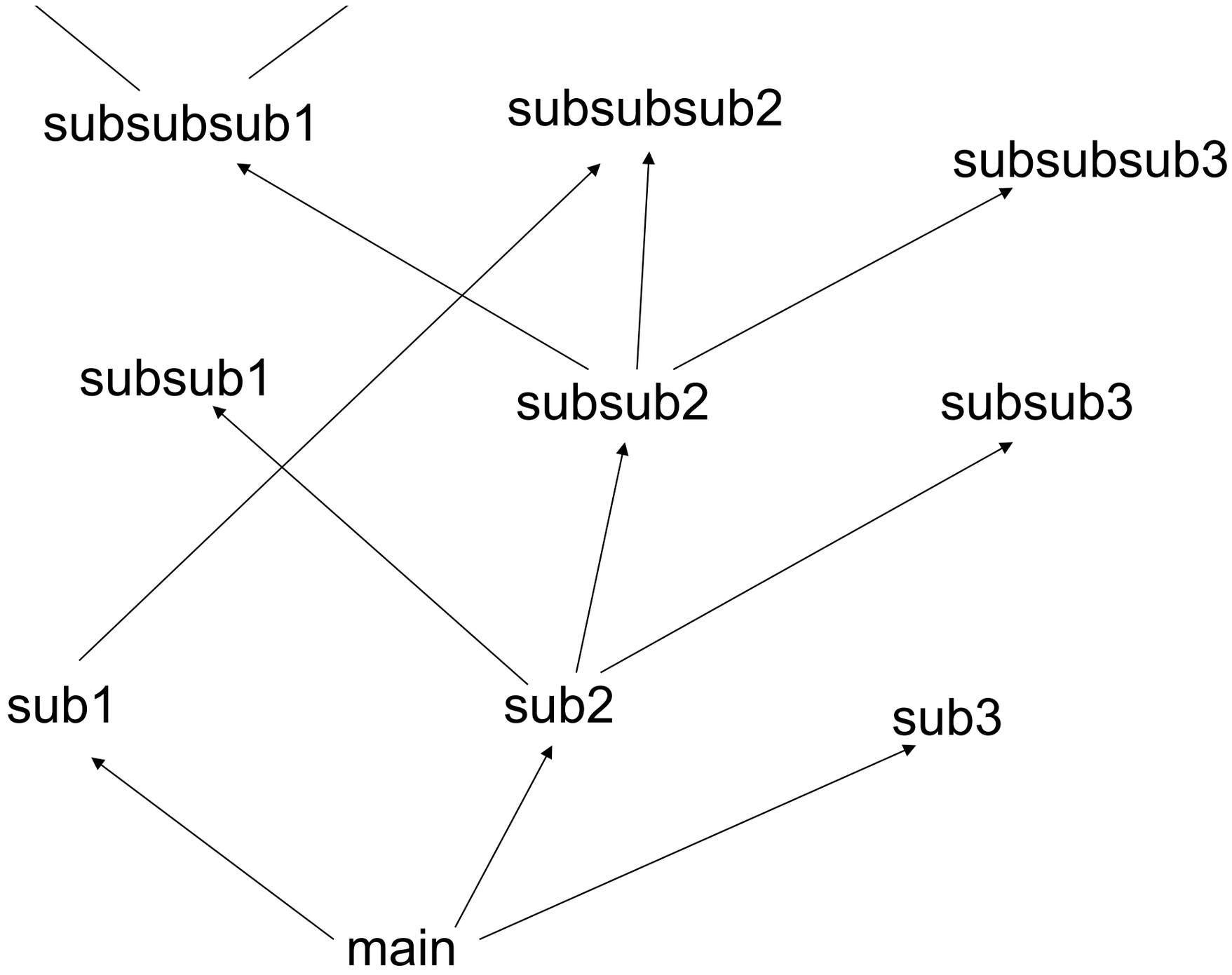
Shallow Computer Program

subroutine1 includes
subsub1 code and
subsub2 code and
subsubsub1 code

subroutine2 includes
subsub2 code and
subsub3 code and
subsubsub3 code and …



Deep Computer Program



Mysteries still remain?

DOI:10.1145/3446776

Understanding Deep Learning (Still) Requires Rethinking Generalization

By **Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht,** and **Oriol Vinyals**

Abstract

Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small gap between training and test performance. Conventional wisdom attributes small generalization error either to properties of the model family or to the regularization techniques used during training.

Through extensive systematic experiments, we show how these traditional approaches fail to explain why large neural networks generalize well in practice. Specifically, our experiments establish that state-of-the-art convolutional networks for image classification trained with stochastic gradient methods easily fit a random labeling of the training data. This phenomenon is qualitatively unaffected by explicit regularization and occurs even if we replace the true images by completely unstructured random noise. We corroborate these experimental findings with a theoretical construction showing that simple depth two neural networks already have perfect finite sample expressivity as soon as the number of parameters exceeds the number of data points as it usually does in practice.

We interpret our experimental findings by comparison with traditional models.

We supplement this republication with a new section at the end summarizing recent progresses in the field since the original version of this paper.

underwrites the generalization ability of a model has occupied the machine learning research community for decades.

There are a variety of theories proposed to explain generalization.

Uniform convergence, margin theory, and algorithmic stability are but a few of the important conceptual tools to reason about generalization. Central to much theory are different notions of *model complexity*. Corresponding generalization bounds quantify how much data is needed as a function of a particular complexity measure. Despite much significant theoretical work, the prescriptive and descriptive value of these theories remains debated.

This work takes a step back. We do not offer any new theory of generalization. Rather, we offer a few simple experiments to interrogate the empirical import of different purported theories of generalization. With these experiments at hand, we broadly investigate what practices do and do not promote generalization, what does and does not measure generalization?

1.1. The randomization test

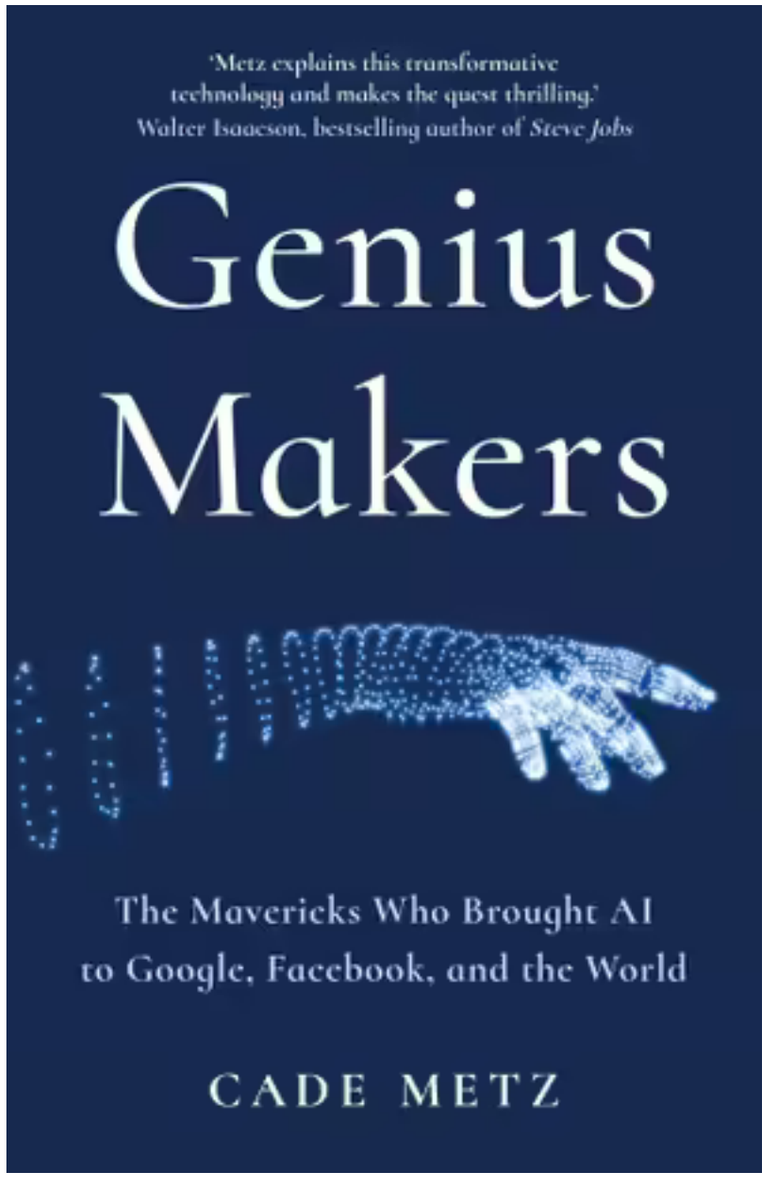
In our primary experiment, we create a copy of the training data where we replace each label independently by a random label chosen from the set of valid labels. A dog picture labeled "dog" might thus become a dog picture labeled "air-

Today

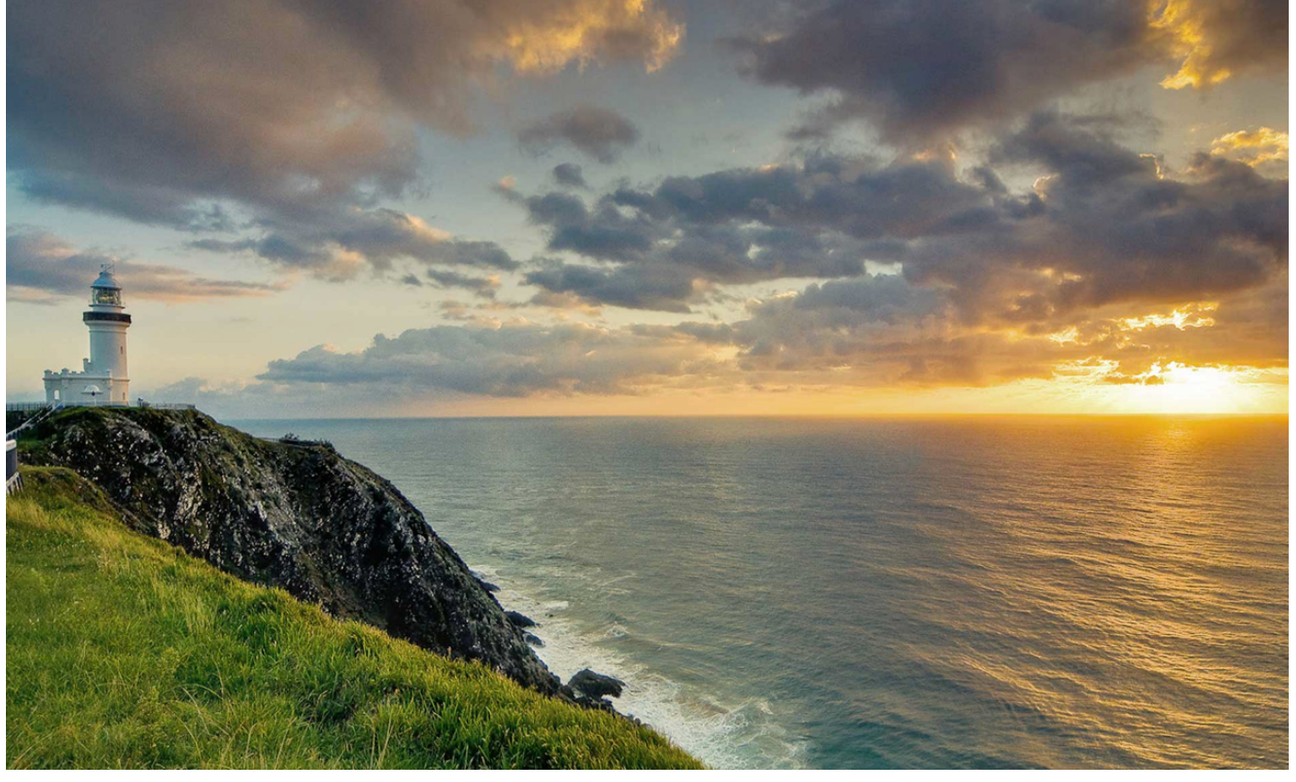
- Single-Layer Perceptron
- Multi-Layer Perceptron
- Convolutional Neural Network

Reminder: Innate vs. Experience

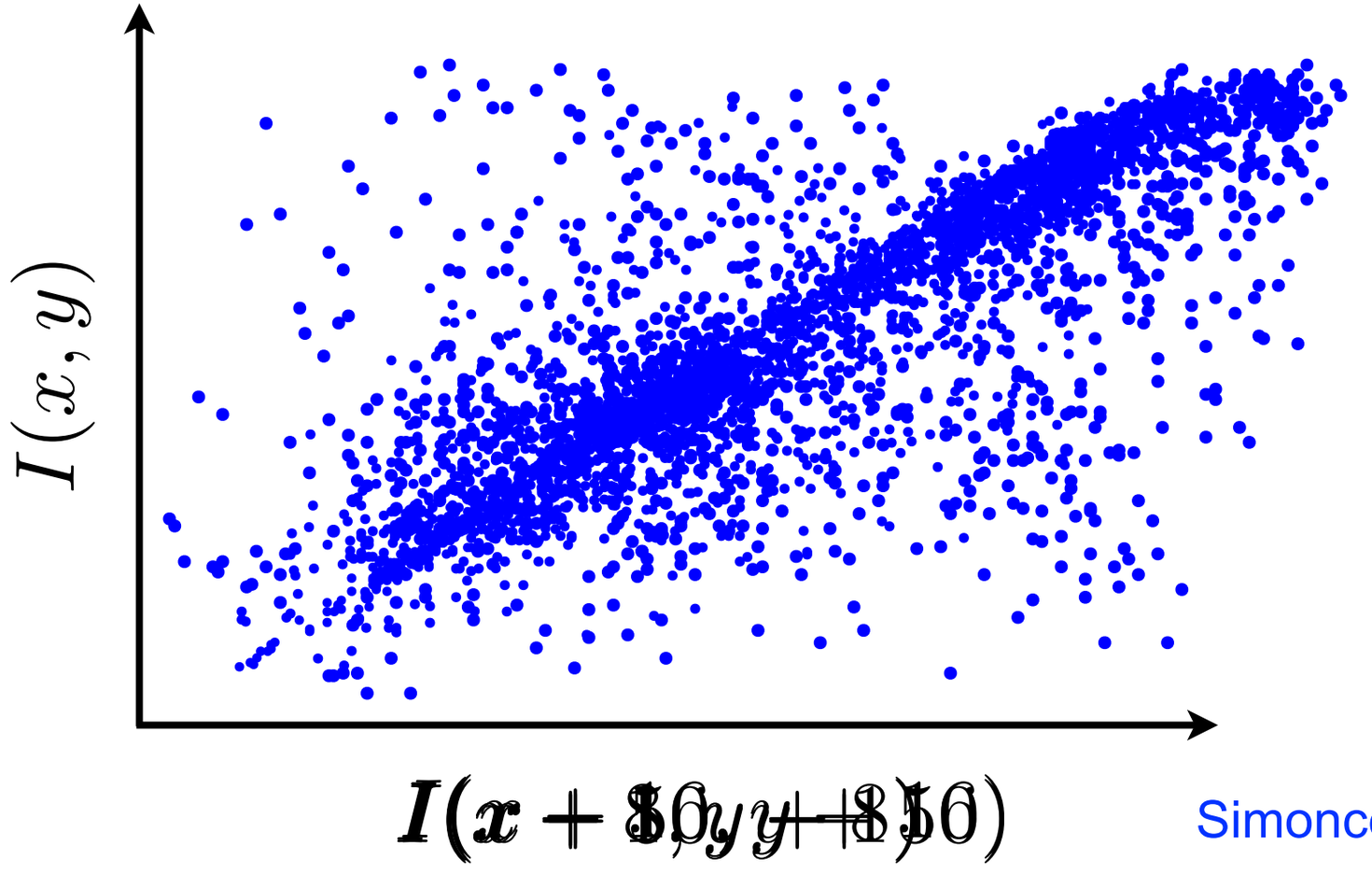
Yoshua Bengio



Gary Marcus

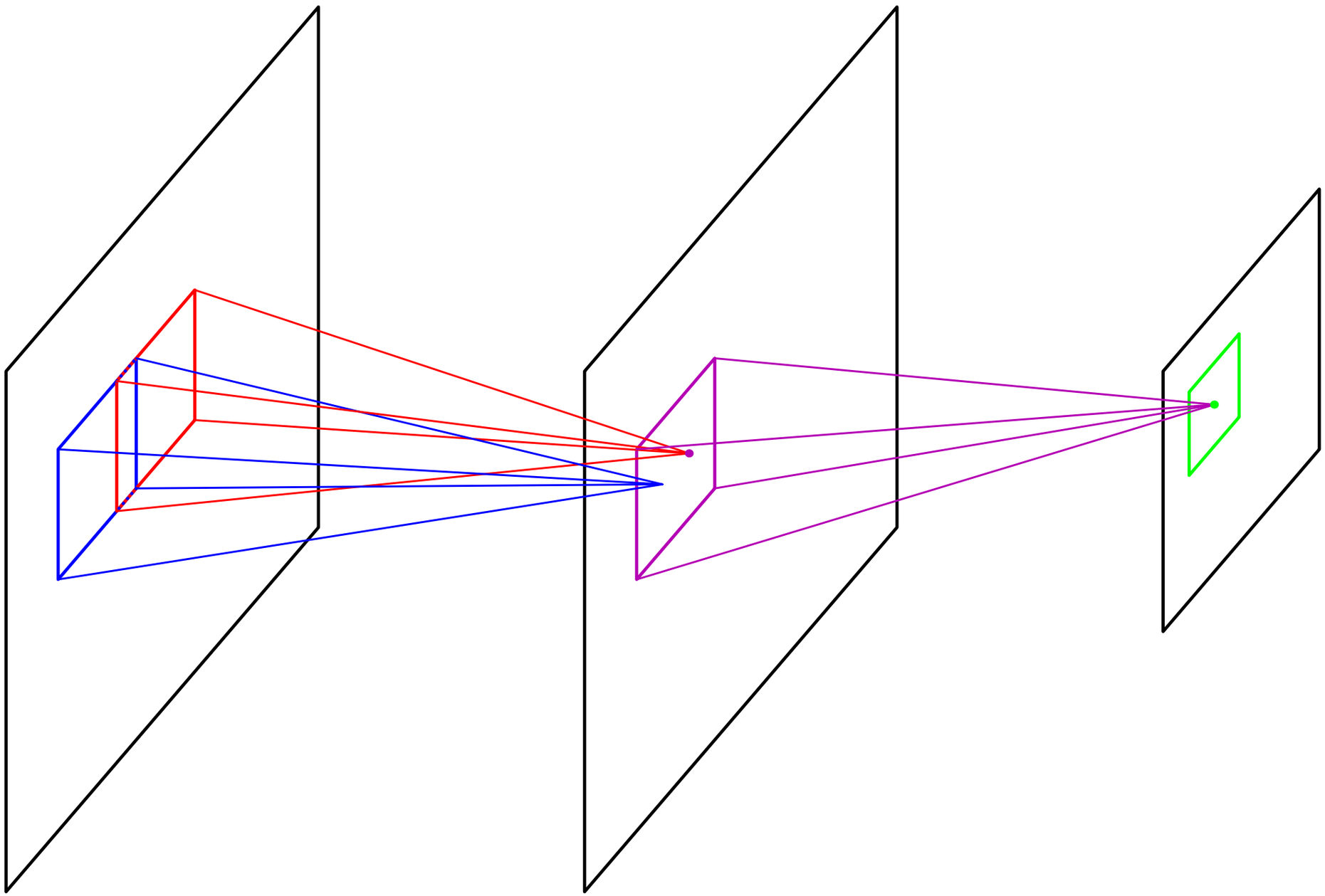


I



Simoncelli & Olshausen 2001

Convolutional Neural Network

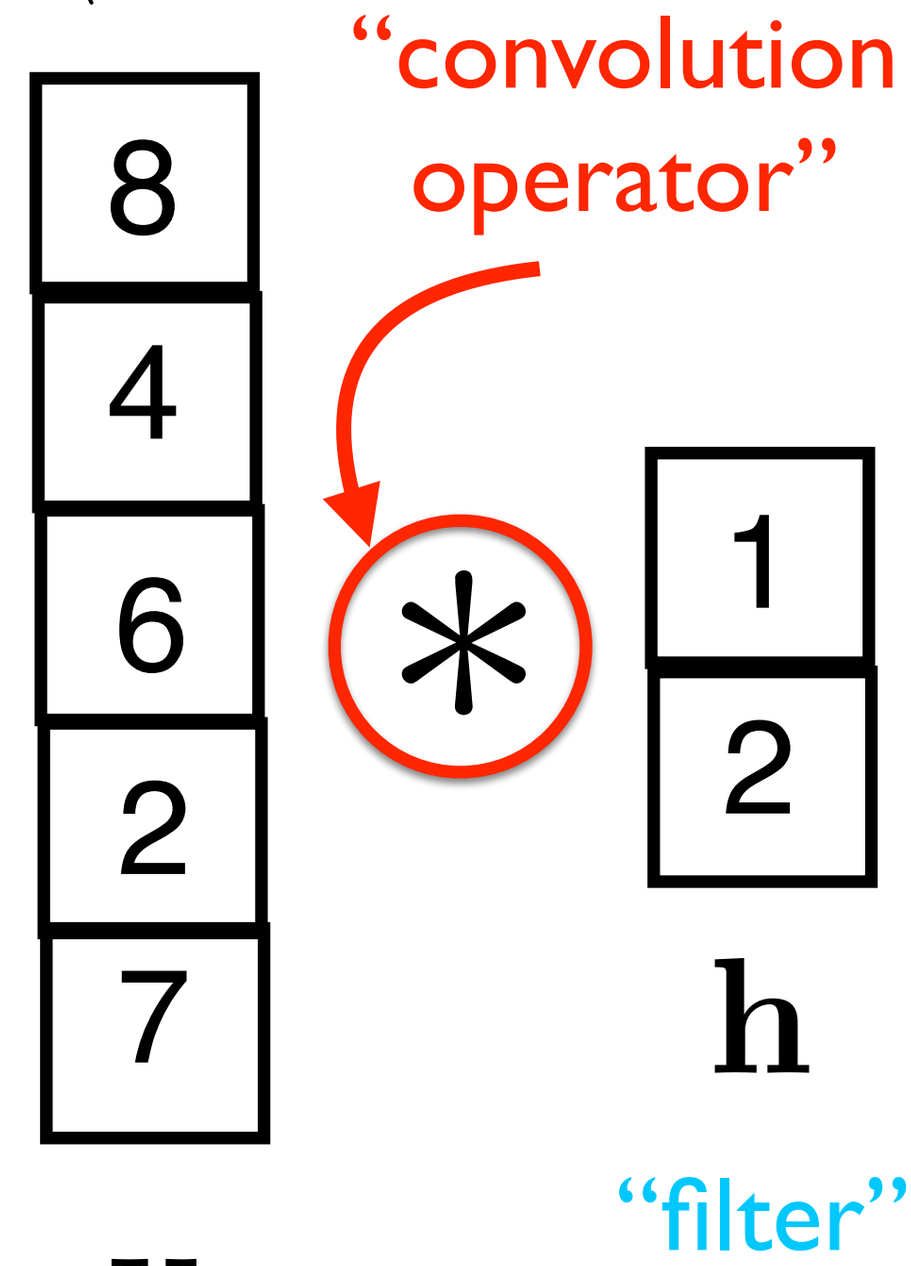


LeCun 1980

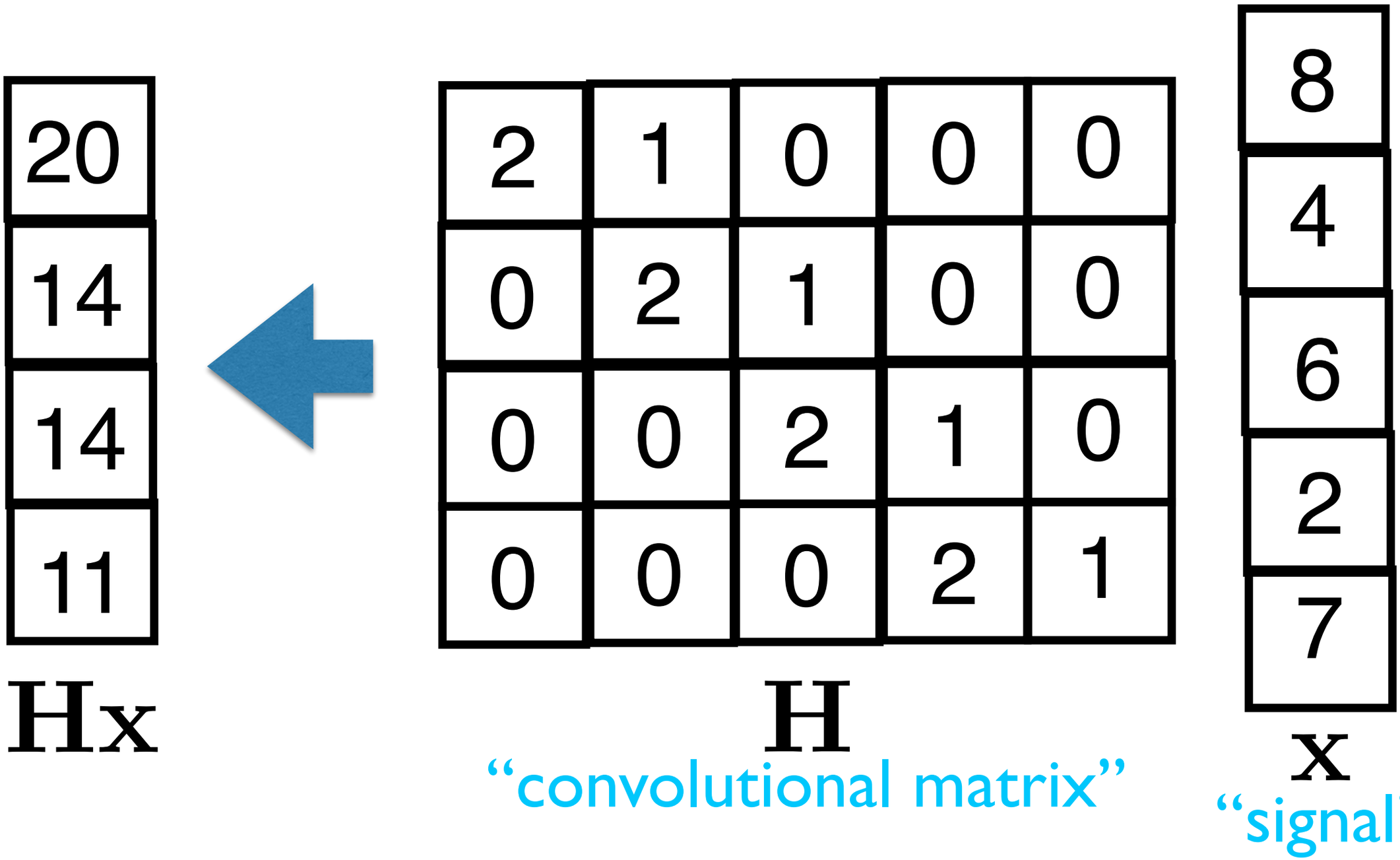
Reminder: Convolution

```
>>> from scipy.signal import \
    convolve as conv
>>> conv(x,h,'valid')
array([20, 14, 14, 11])
```

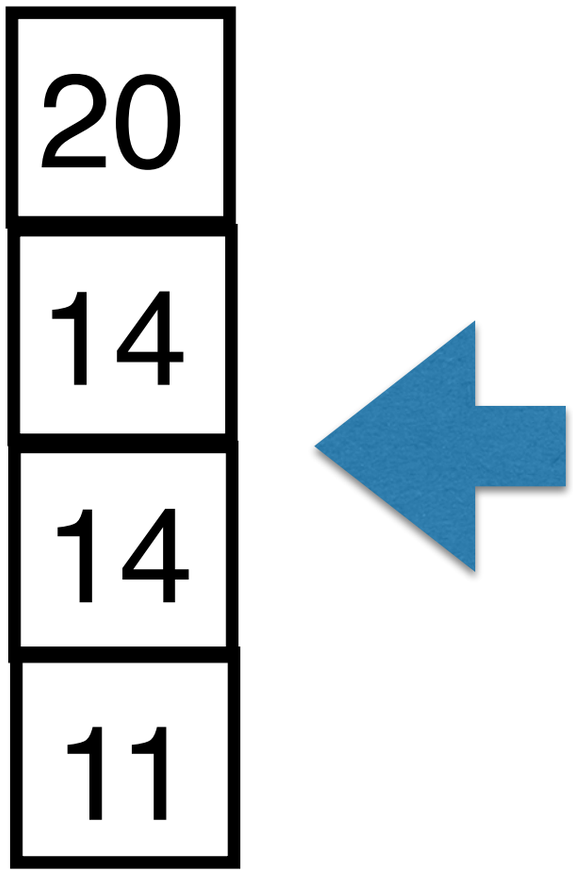
$$\frac{\partial[\mathbf{x} * \mathbf{h}]}{\mathbf{h}^T} = ???$$



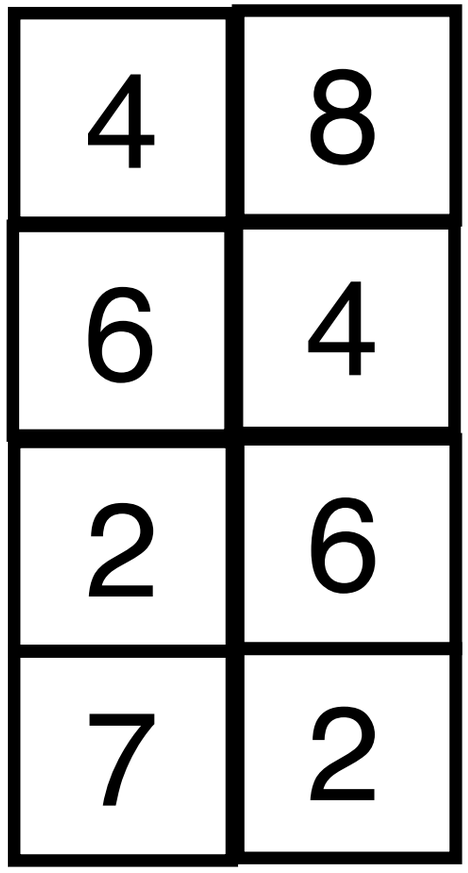
Reminder: Convolution



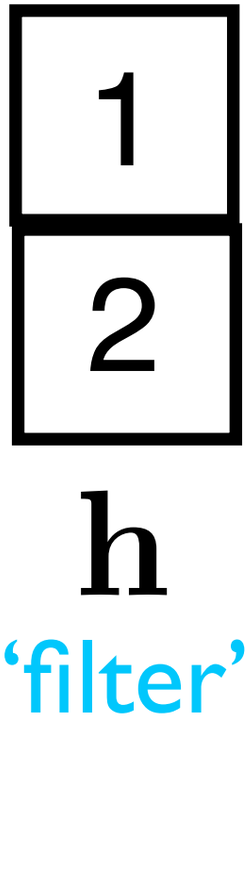
Reminder: Convolution



$\mathbf{Xh}$



$\mathbf{X}$



$\mathbf{h}$

"filter"

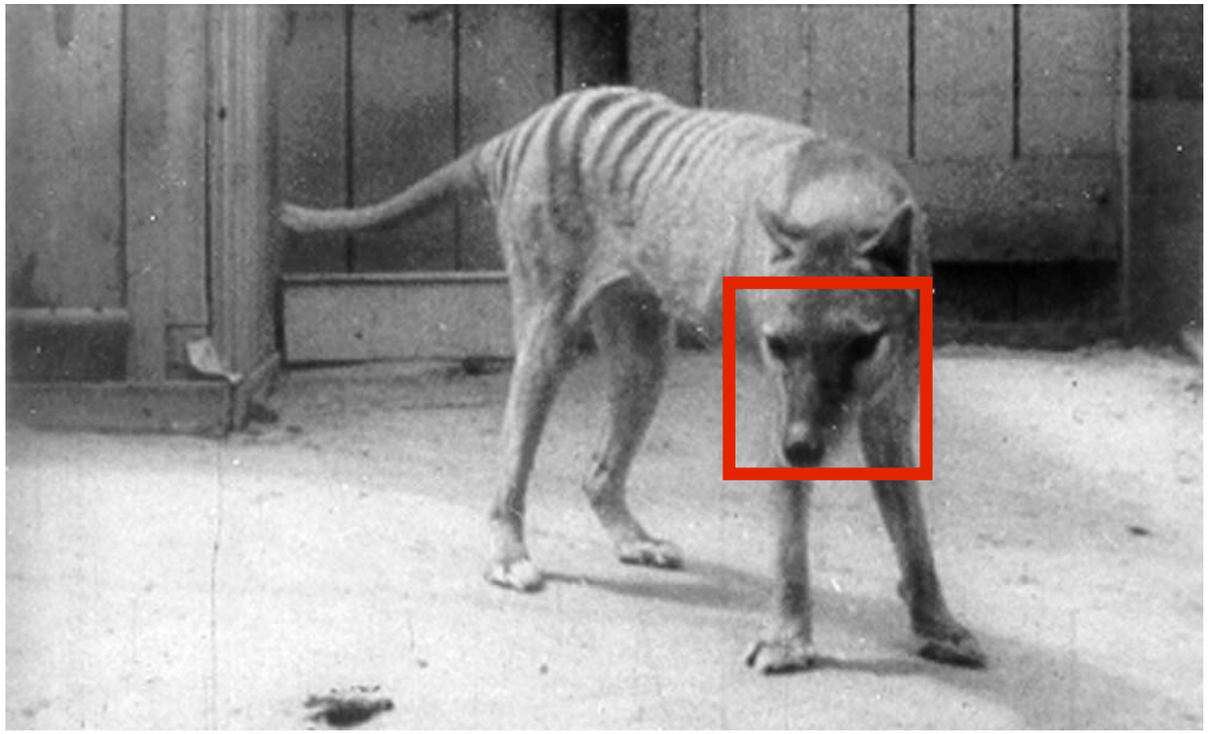
"convolutional signal"

$$\frac{\partial[\mathbf{x} * \mathbf{h}]}{\mathbf{h}^T} = \frac{\partial[\mathbf{X}\mathbf{h}]}{\mathbf{h}^T} = \mathbf{X}^T$$

Efficiency of Convolution

Input size: 320 by 280
Filter size: 2 by 1
Output size: 319 by 280

	Convolution	Dense matrix	Sparse matrix
Stored floats	2	319*280*320*280 > 8e9	2*319*280 = 178,640
Float muls or adds	319*280*3 = 267,960	> 16e9	Same as convolution (267,960)



X

*

H

Y

Correlation vs. Convolution

- Convolution is preferred mathematically over correlation as it is,

$$\mathbf{g} * (\mathbf{h} * \mathbf{x}) = (\mathbf{g} * \mathbf{h}) * \mathbf{x}$$ **(associative)**

$$\mathbf{h} * \mathbf{x} = \mathbf{x} * \mathbf{h}$$ **(communicative)**

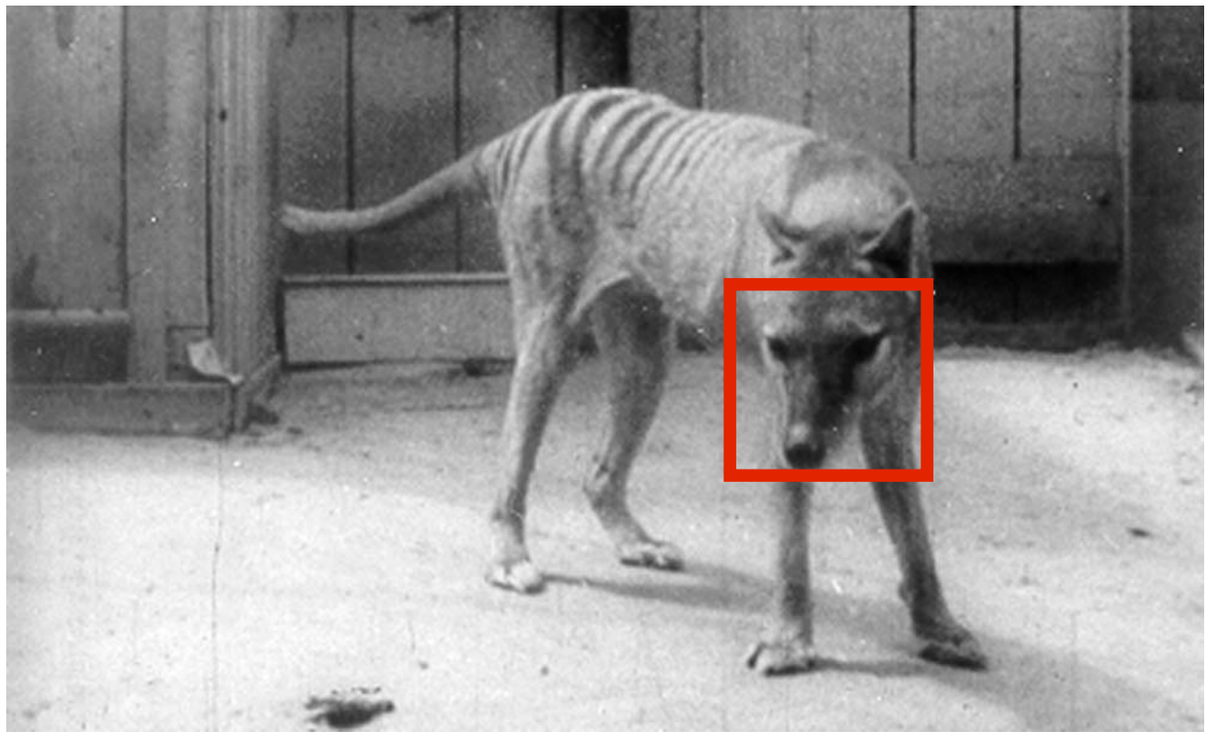
- Correlation is neither!!!

$$\mathbf{g} \otimes (\mathbf{h} \otimes \mathbf{x}) \neq (\mathbf{g} \otimes \mathbf{h}) \otimes \mathbf{x}$$

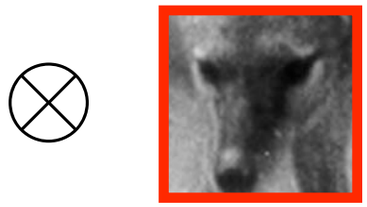
$$\mathbf{h} \otimes \mathbf{x} \neq \mathbf{x} \otimes \mathbf{h}$$

- Correlation, however, preferred for signal matching/ detection.

2D Convolution vs. Correlation



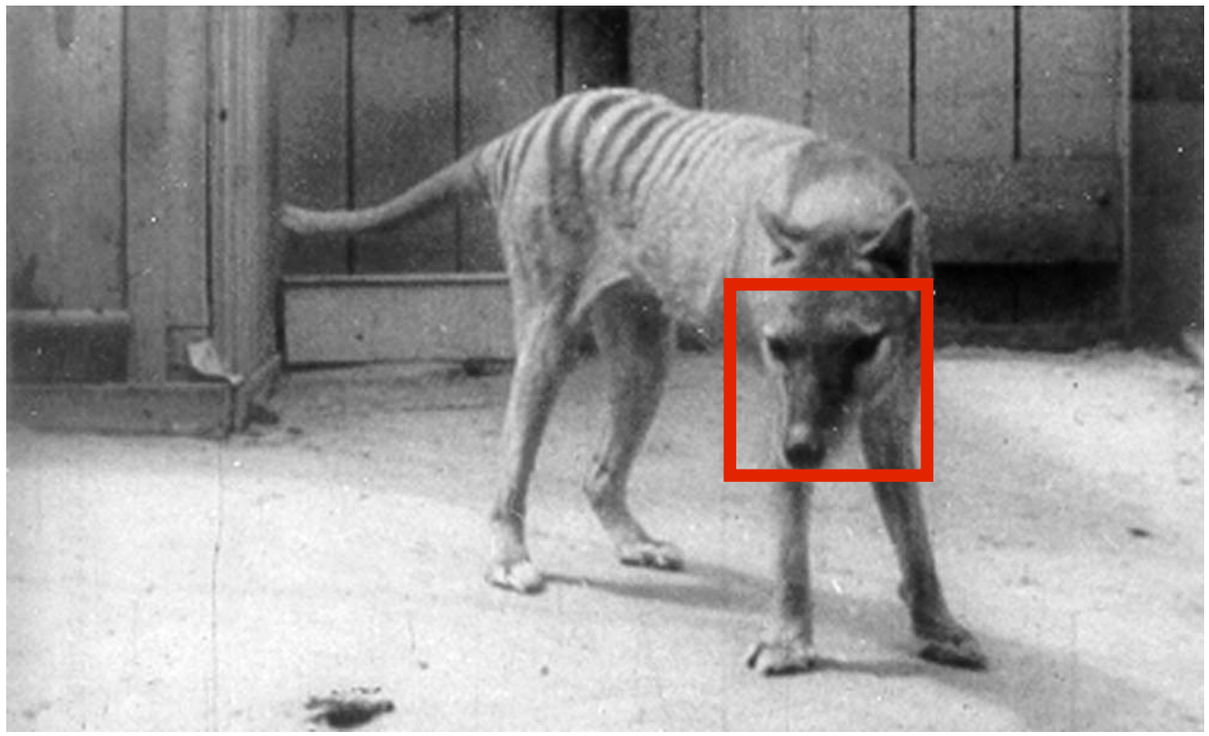
X



G

```
>>> from scipy.ndimage import \
correlation as imcorr
>>> Y = imcorr(X,G)
```

2D Convolution vs. Correlation



$*$

H

X

```
>>> H = flipud(fliplr(G))
>>> Y = imconv(X,H)
```

Vectorizing 2D-Convolution

$$\text{vec}(\mathbf{Y}) = \text{vec}(\mathbf{X} * \mathbf{H})$$

```
def forward(self, x):
    # Flattens the image like structure into vect
    x = torch.flatten(x, start_dim=1)
```

Vectorizing 2D-Convolution

$$\mathrm{vec}(\mathbf{Y}) = \mathbf{H}'\mathrm{vec}(\mathbf{X})$$

Multiple Filters

$$\underbrace{\begin{bmatrix} \mathbf{H}_1' \\ \vdots \\ \mathbf{H}_M' \end{bmatrix}}_{(D \cdot M \times D)} \underbrace{\mathbf{X}}_{(D \times 1)} \Leftarrow \underbrace{\begin{bmatrix} \mathrm{vec}(\mathbf{X} * \mathbf{H}_1) \\ \vdots \\ \mathrm{vec}(\mathbf{X} * \mathbf{H}_M) \end{bmatrix}}_{(D \cdot M \times 1)}$$

"convolution matrix"

Removing Redundancy - Striding

```
Stride: [4,4]
```

Every 4th element

Every 4th element

207	245	77	21	247	211	240	1
219	41	58	179	161	154	184	98
215	145	187	71	251	249	65	100
192	2	189	247	166	63	232	213
105	94	66	190	156	61	89	145
159	154	87	184	101	105	72	71
192	111	6	94	60	70	65	226
175	120	210	226	80	183	168	184
134	56	36	240	159	178	76	135
239	244	199	9	132	104	188	185
245	210	78	199	0	92	9	246
5	121	187	122	107	47	12	119
230	171	135	36	82	54	65	37
61	140	79	19	161	96	127	187
56	223	46	6	180	186	142	244
28	20	61	2	178	187	98	220

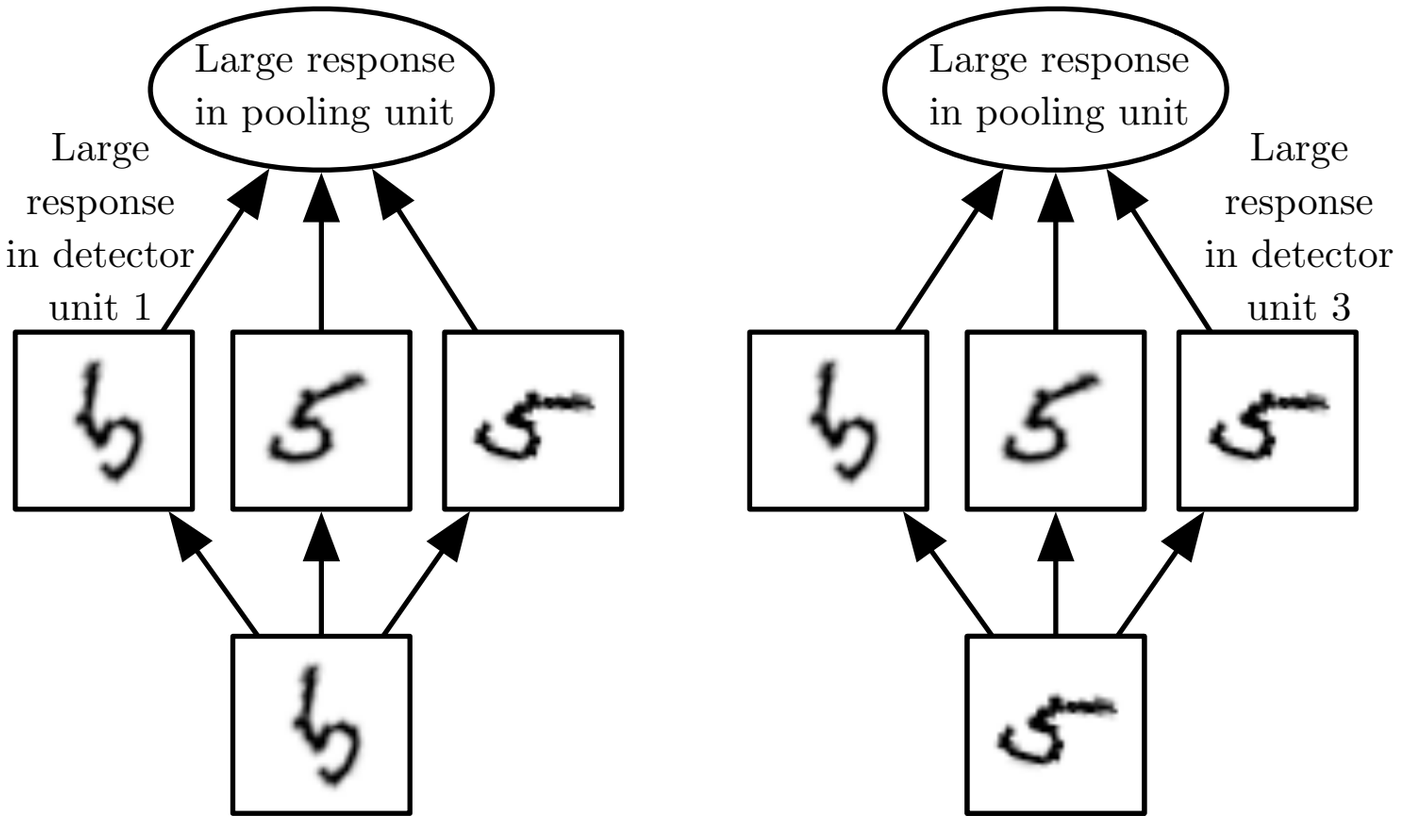
Removing Redundancy - Striding

$$\mathbf{Y}[m, n] = \mathbf{X}[s_m \cdot m, s_n \cdot n]$$

"stride in rows"

"stride in columns"

Removing Redundancy - Max Pooling



$$\mathbf{Y}[m,n] = \max_{\Delta \in \mathbb{N}} \{\mathbf{X}[m+\Delta_m, n+\Delta_n]\}$$

n

207	245	77	21	247	211	240	1
219	41	58	179	161	154	184	98
215	145	187	71	251	249	65	100
192	2	189	247	166	63	232	213
105	94	66	190	156	61	89	145
159	154	87	184	101	105	72	71
192	111	6	94	60	70	65	226
175	120	210	226	80	183	168	184
134	56	36	240	159	178	76	135
239	244	199	9	132	104	188	185
245	210	78	199	0	92	9	246
5	121	187	122	107	47	12	119
230	171	135	36	82	54	65	37
61	140	79	19	161	96	127	187
56	223	46	6	180	186	142	244
28	20	61	2	178	187	98	220

m

$\mathbf{X}$

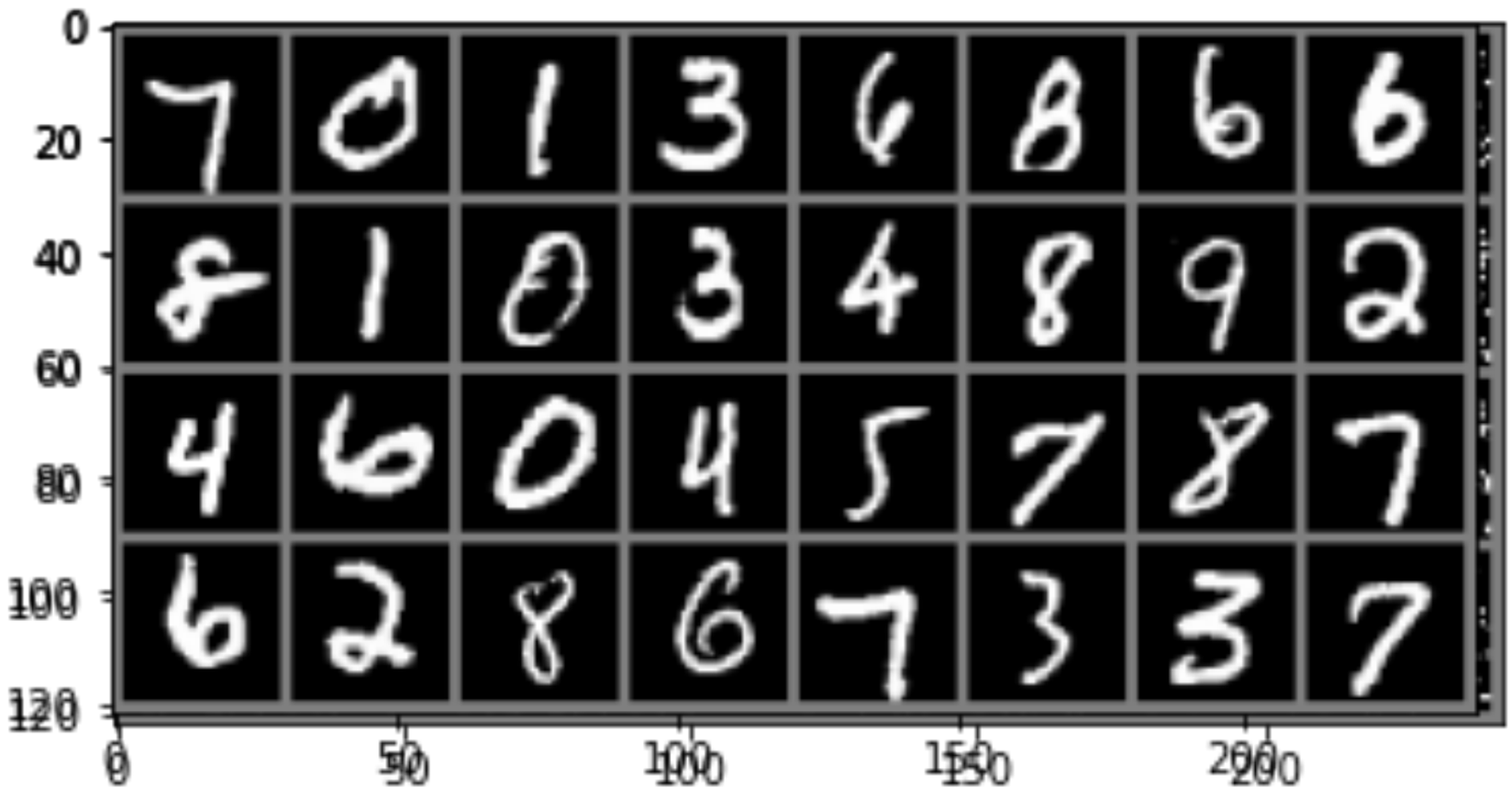
Let’s have another play!!!

https://colab.research.google.com/github/slucey-cs-cmu-edu/RVSS2022/blob/main/Visual_Learning/Session2/LeNetClassificationExcercise_2021.ipynb

Some things to try!!!

- What happens to performance if you use a linear function?
- What happens when you permute the pixels?

```
from numpy.random import permutation
idx_permute = torch.from_numpy(permutation(784))
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Lambda(lambda x: x.view(-1)[idx_permute].view(1, 28, 28) ),
                               transforms.Normalize((0.5,), (0.5,)),
                               ])
```



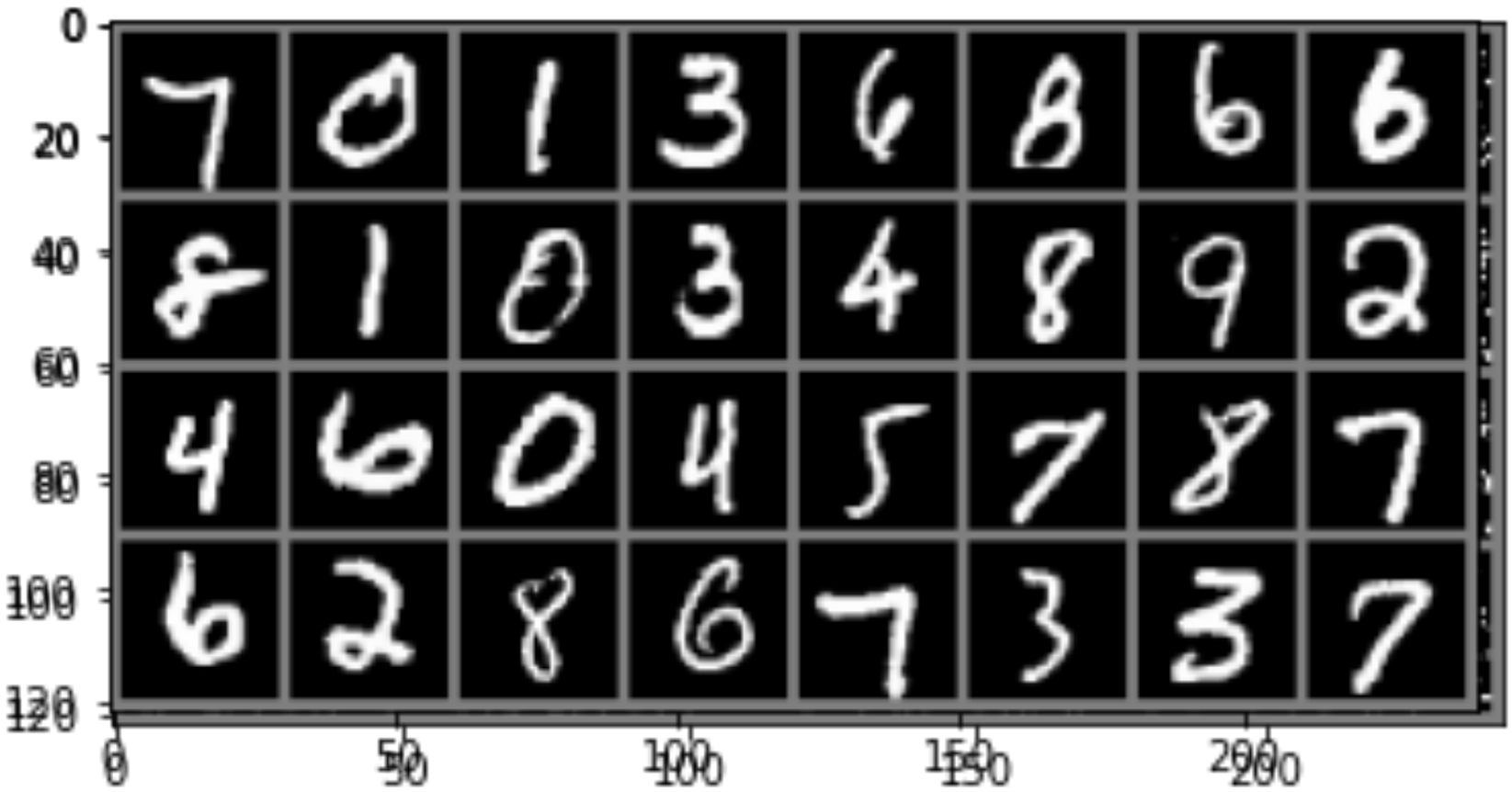
Adding Striding - CNN

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, 2)
        self.fc1 = nn.Linear(256, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        # Input goes to convolution so no need t
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
```

Permuting Pixels - CNN

- What happens when you permute the pixels in a CNN?

```
from numpy.random import permutation
idx_permute = torch.from_numpy(permutation(784))
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Lambda(lambda x: x.view(-1)[idx_permute].view(1, 28, 28) ),
                               transforms.Normalize((0.5,), (0.5,)),
                               ])
```



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

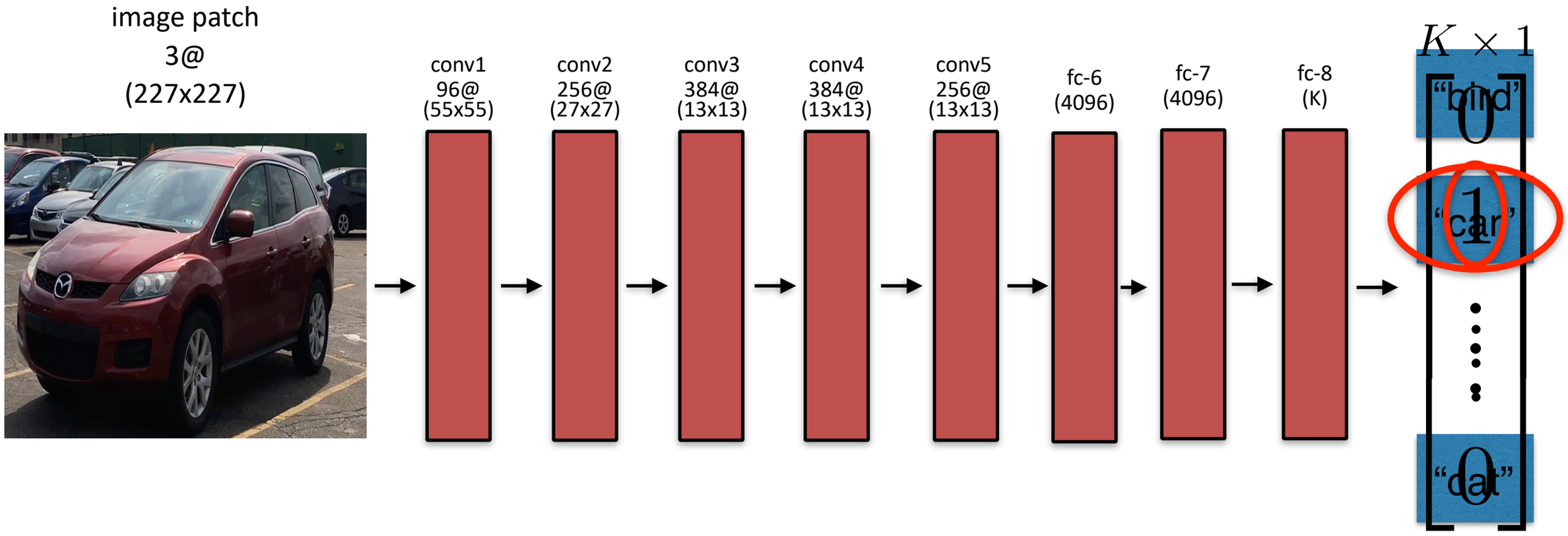
Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

AlexNet

- AlexNet won the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%. (Second best was 26.2 %).
- Network has 25 layers, but only 8 layers with learnable weights.
  - 5 convolutional weights.
  - 3 fully connected weights.



A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

AlexNet in PyTorch

```
>>> from torchvision import models
>>> net = models.alexnet()
```

AlexNet in PyTorch

```
>>> net
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace)
    (3): Dropout(p=0.5)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

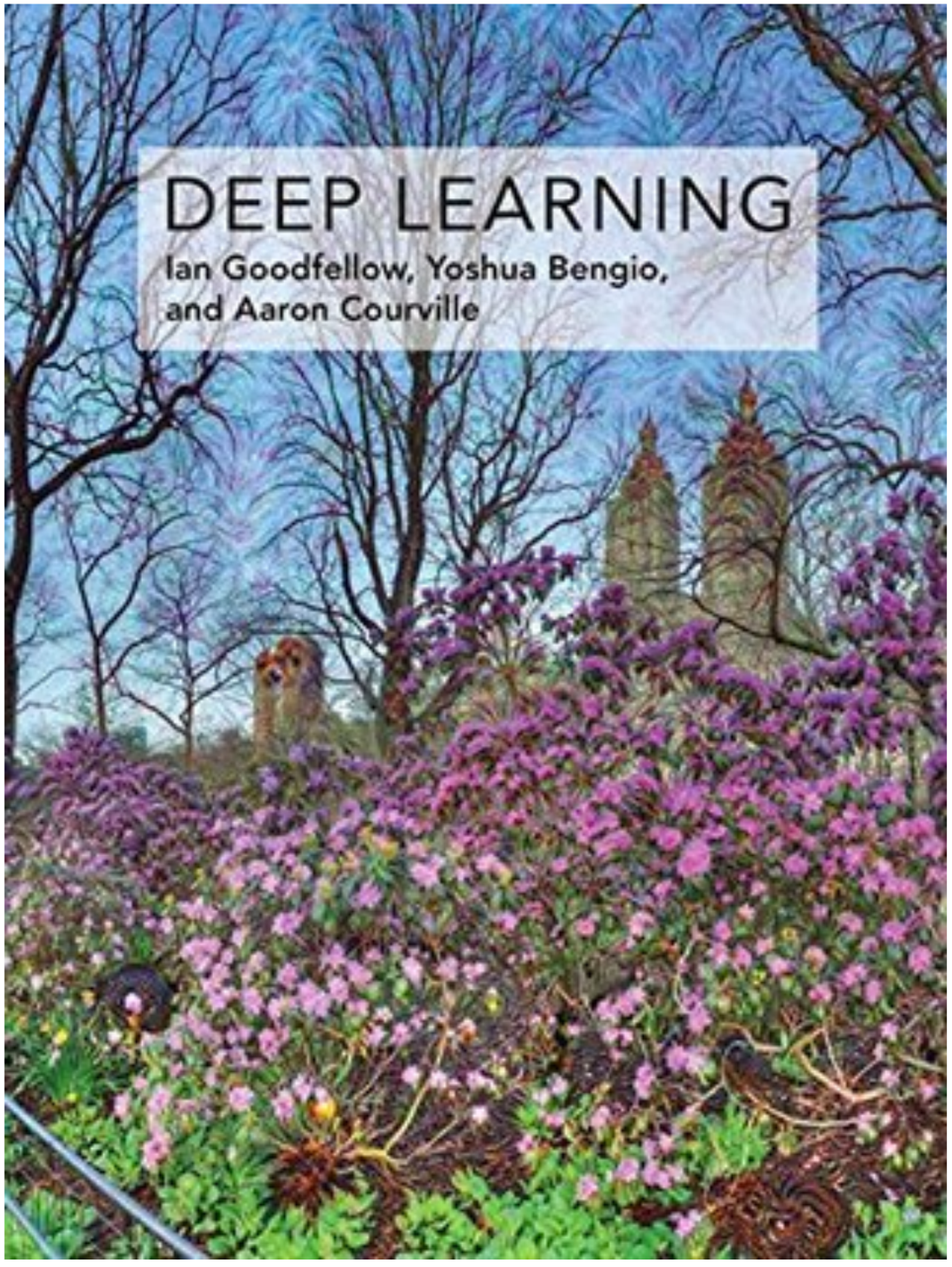
AlexNet in PyTorch

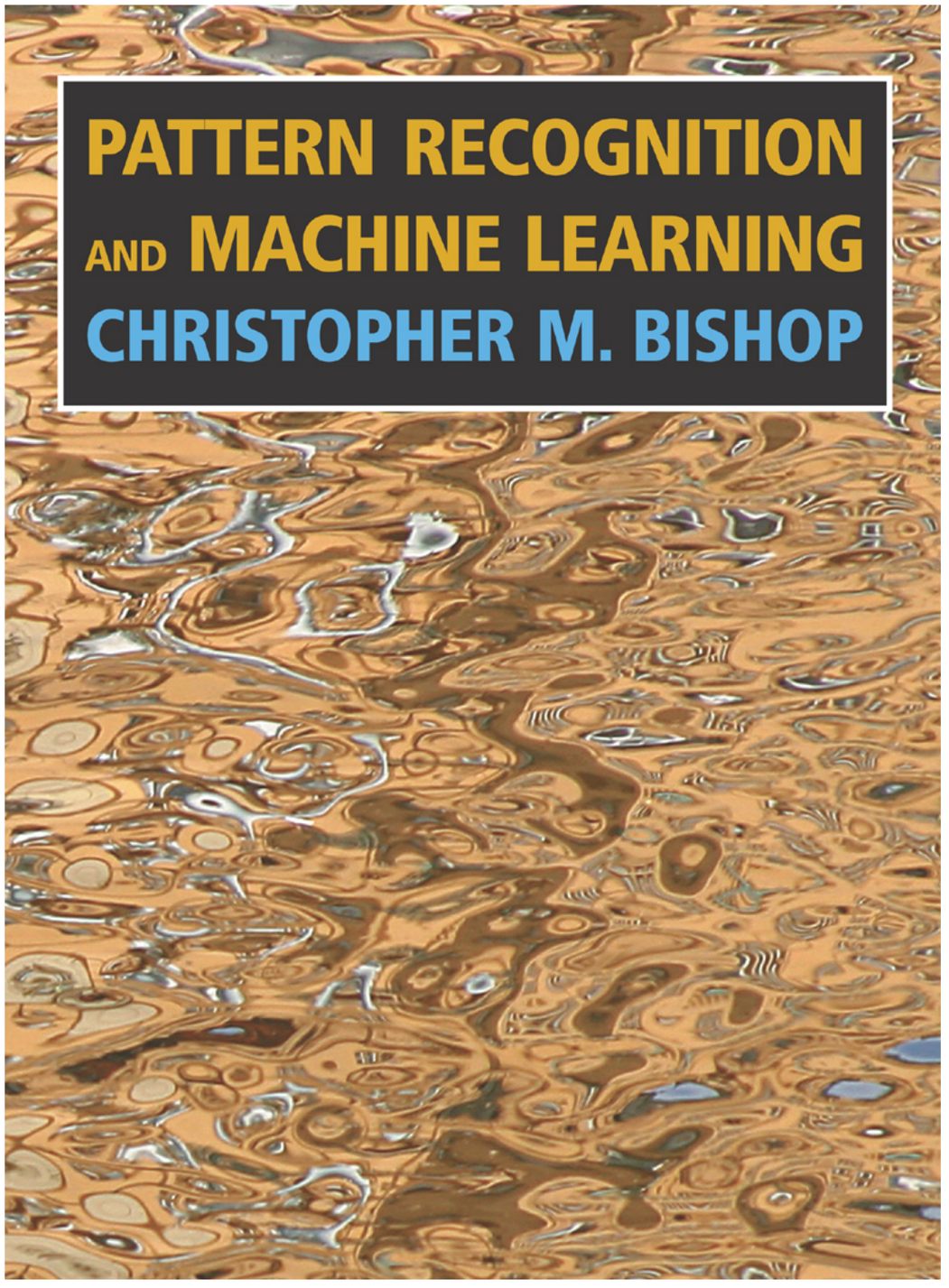
```
>>> from torchvision import models
>>> net = models.alexnet()
```

AlexNet in PyTorch

```
>>> net
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace)
    (3): Dropout(p=0.5)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

More to read…


DEEP LEARNING
Ian Goodfellow, Yoshua Bengio,
and Aaron Courville


PATTERN RECOGNITION
AND MACHINE LEARNING
CHRISTOPHER M. BISHOP