# Geometry of Neuroscience

Matilde Marcolli & Doris Tsao

## Mar 2: Deep learning

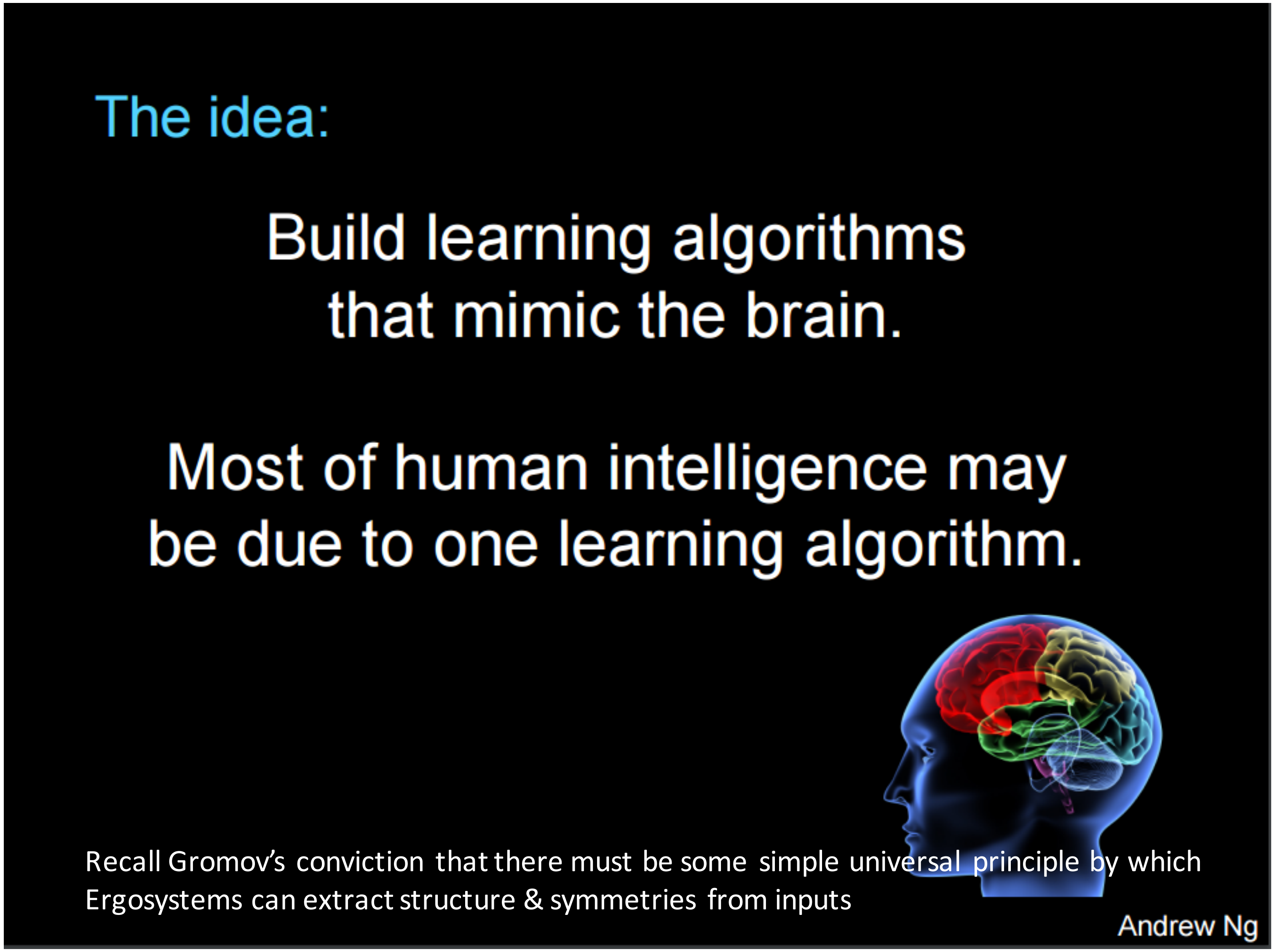
## References

- Deep Learning by Goodfellow, Bengio, Courville
- http://neuralnetworksanddeeplearning.com

## History of Artificial Intelligence

## Knowledge base approach: (computer can reason automatically about statements in formal languages using logical inference rules)

Machine learning: AI systems acquire their own knowledge by extracting patterns from raw data



The idea:

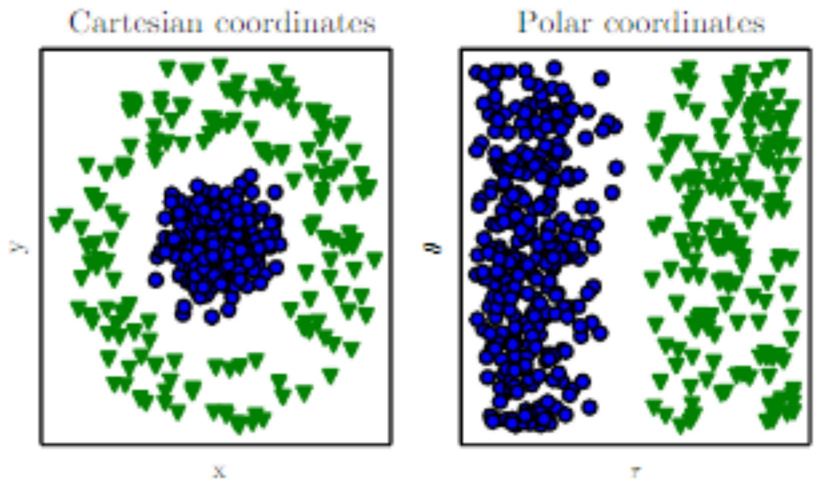
Build learning algorithms that mimic the brain.

Most of human intelligence may be due to one learning algorithm.

Recall Gromov's conviction that there must be some simple universal principle by which Ergosystems can extract structure & symmetries from inputs

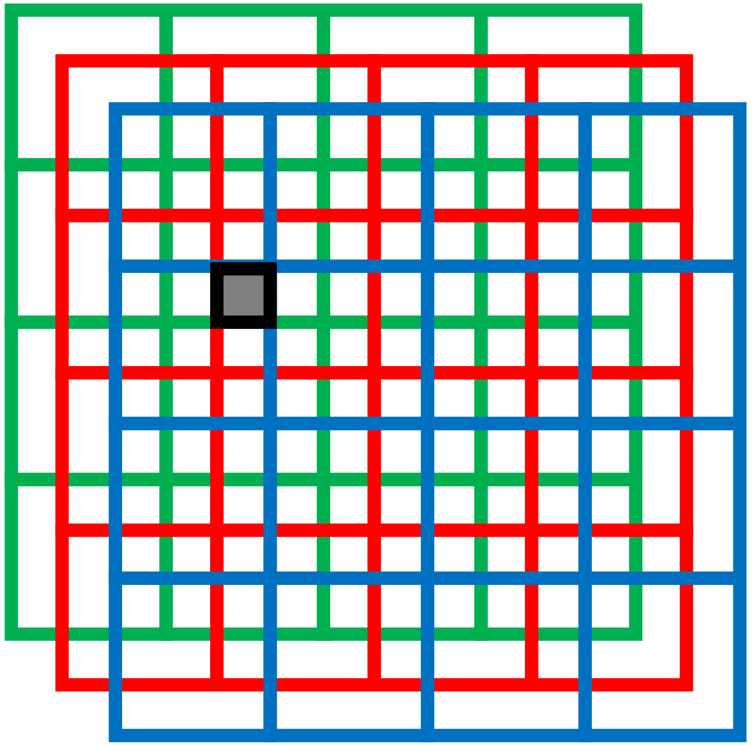
Andrew Ng

# Representation is critical



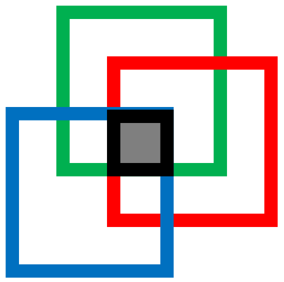
Distributed representations are powerful (each input represented by many features, each feature involved in representation of many inputs)

# Coarse coding of fine spatial detail



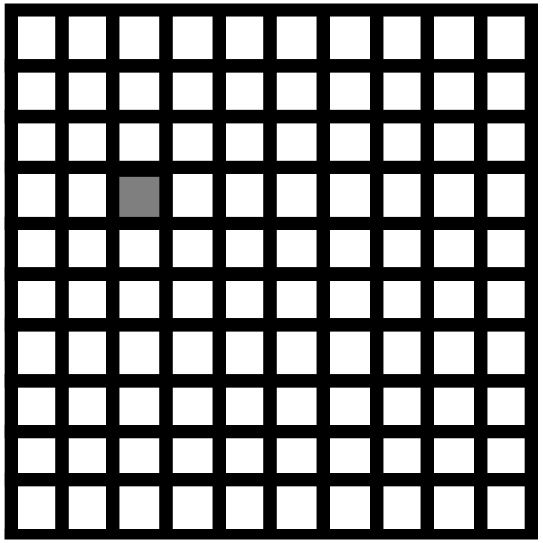
(argument due to Goeff Hinton)

## Coarse coding of fine spatial detail



(argument due to Goeff Hinton)

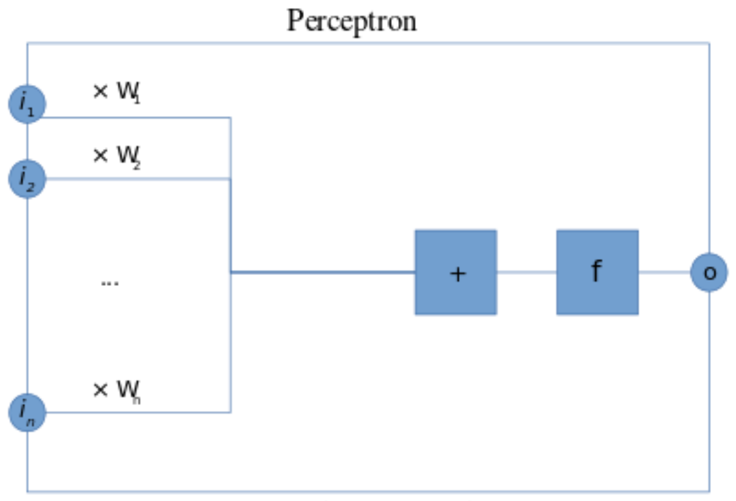
## Coarse coding of fine spatial detail



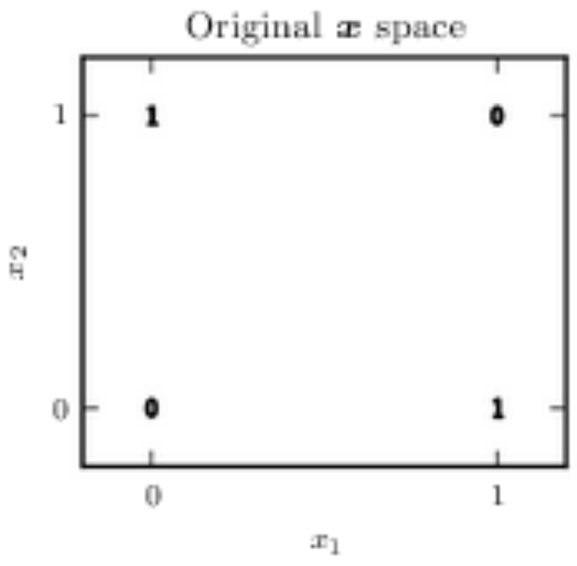
(argument due to Goeff Hinton)

# Perceptron

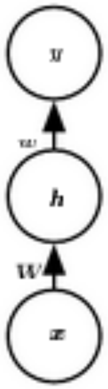
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$



$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$



$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$



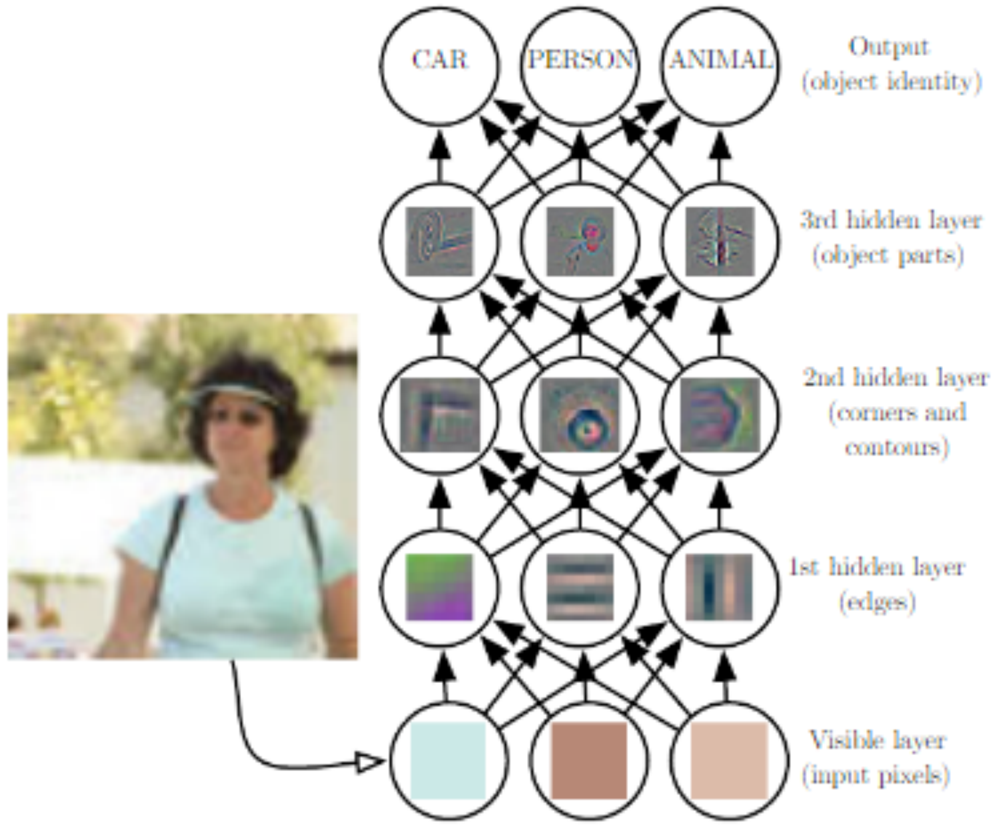
We can now specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

and $b = 0$.



Deep learning: *learns representations* that are expressed in terms of other, simpler representations (or alternatively: sequence of steps with intermediate buffers).

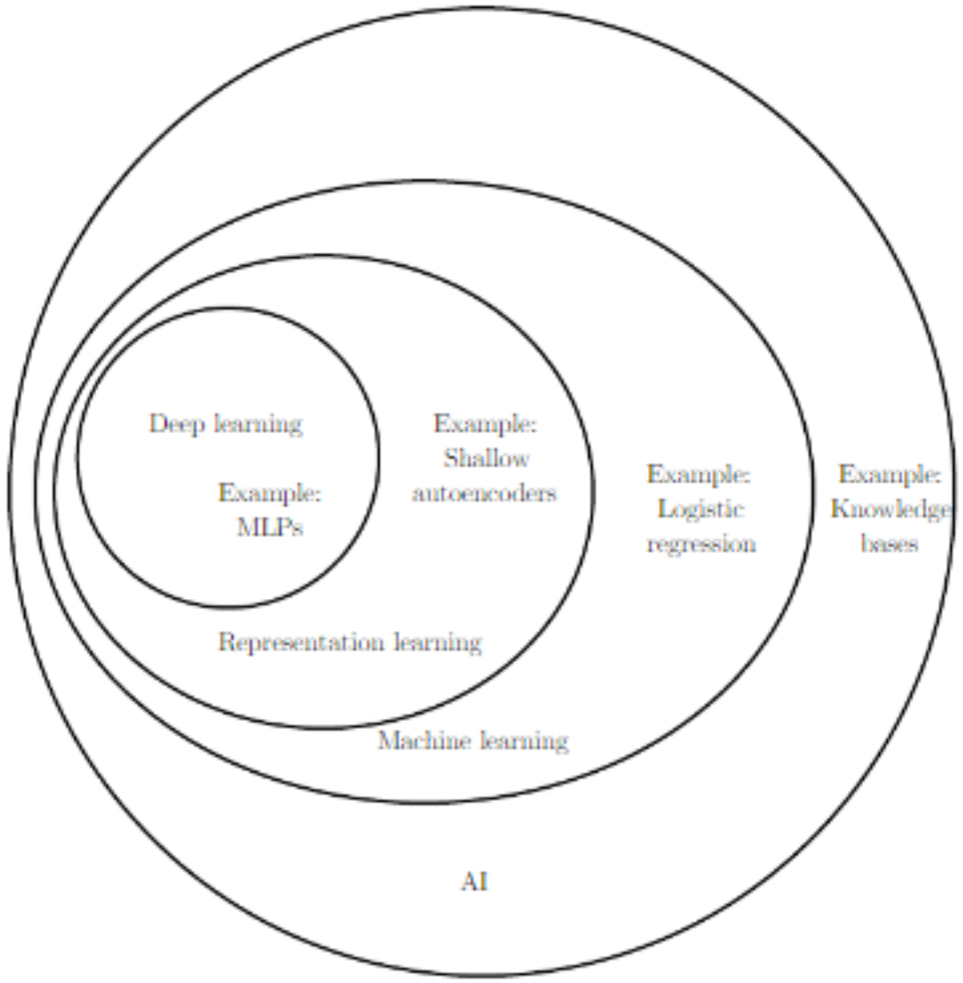


Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

## Why is deep learning so successful?

- Large data sets
- Fast computers enabling training of large networks

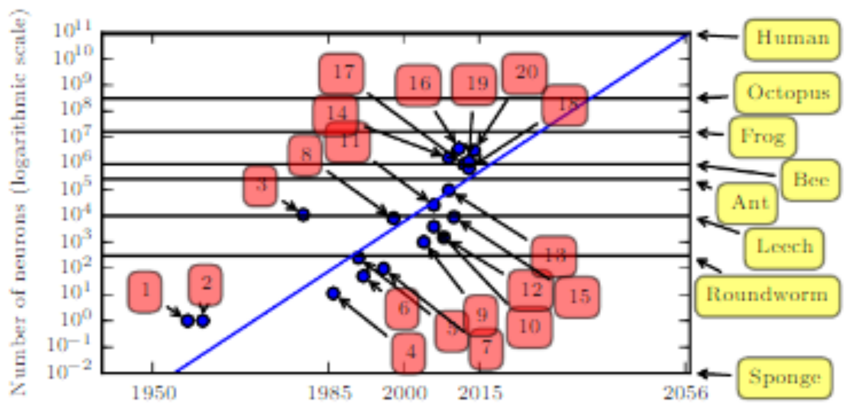


Figure 1.11: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

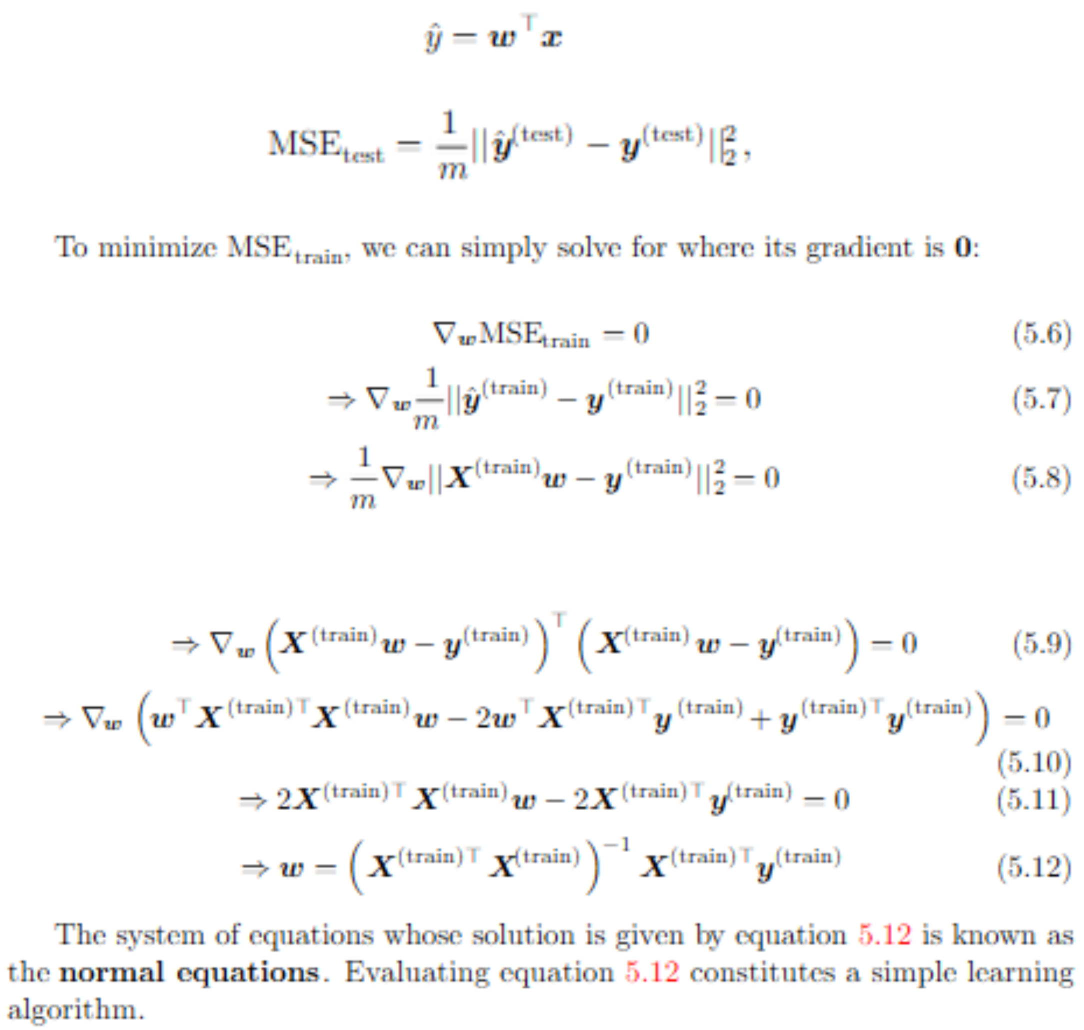
# Typical types of tasks for machine learning

- Classification (1…k)
- Regression (predict numerical value given some input)
- Transcription (image -> text)
- Machine translation
- Structured output (e.g., segmentation map)
- Synthesis
- Denoising

# Networks can perform complex tasks

- Neural turing machine: can learn to read from memory cells and write arbitrary content to memory cells; can learn simple programs from examples of desired behavior (e.g., learn to sort lists given examples of scrambled and sorted lists).
- Reinforcement learning: Autonomous agent learns to perform task by trial and error without any guidance from human (DeepMind, Atari and Go).

# Linear regression: A simple example of machine learning



$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

$$\text{MSE}_{\text{test}} = \frac{1}{m} || \hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})} ||_2^2,$$

To minimize $\text{MSE}_{\text{train}}$, we can simply solve for where its gradient is $\mathbf{0}$:

$$\nabla_{\boldsymbol{w}} \text{MSE}_{\text{train}} = 0 \tag{5.6}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \frac{1}{m} || \hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})} ||_2^2 = 0 \tag{5.7}$$

$$\Rightarrow \frac{1}{m} \nabla_{\boldsymbol{w}} || \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} ||_2^2 = 0 \tag{5.8}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \left( \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right)^\top \left( \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.9}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \left( \boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2 \boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.10}$$

$$\Rightarrow 2 \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2 \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} = 0 \tag{5.11}$$

$$\Rightarrow \boldsymbol{w} = \left( \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \right)^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.12}$$

The system of equations whose solution is given by equation 5.12 is known as the **normal equations**. Evaluating equation 5.12 constitutes a simple learning algorithm.

# Capacity

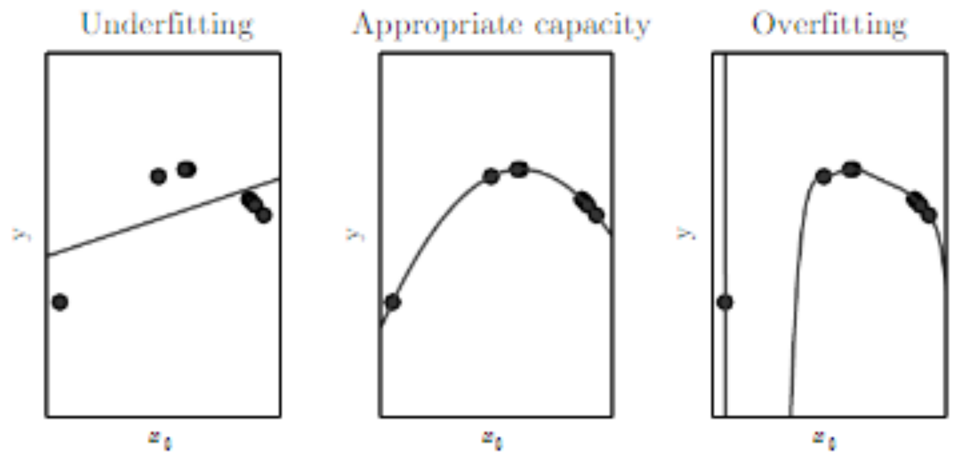


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling $x$ values and choosing $y$ deterministically by evaluating a quadratic function. *(Left)*A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. *(Center)*A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. *(Right)*A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudoinverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

## Regularization: modification to learning algorithm to reduce generalization error but not training error

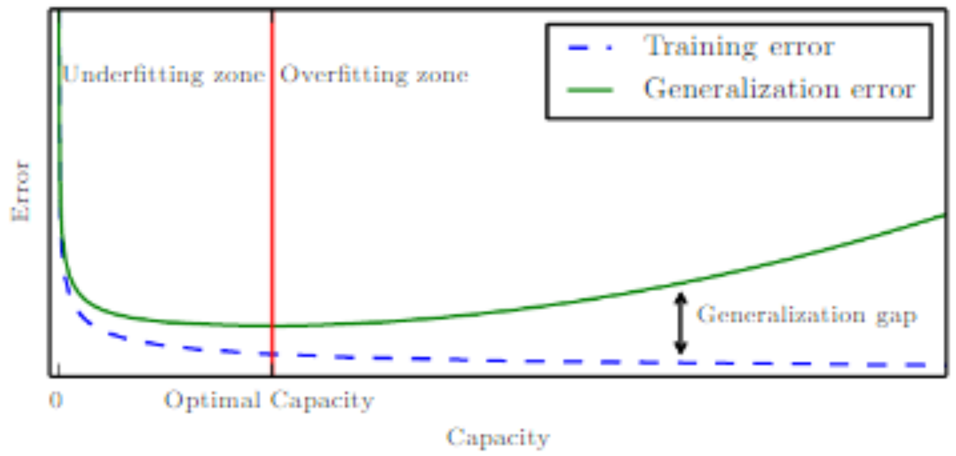
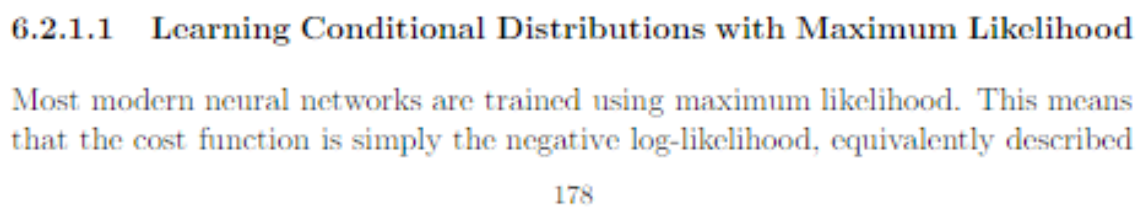


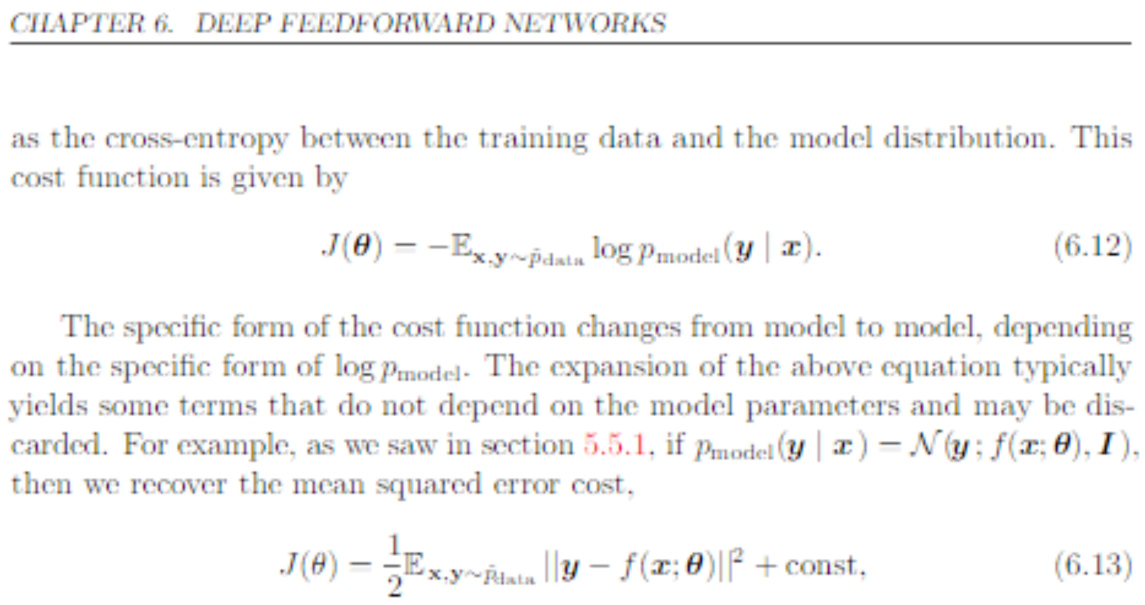
Figure 5.3: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the **overfitting regime**, where capacity is too large, above the **optimal capacity**.

# Cost function



### 6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described



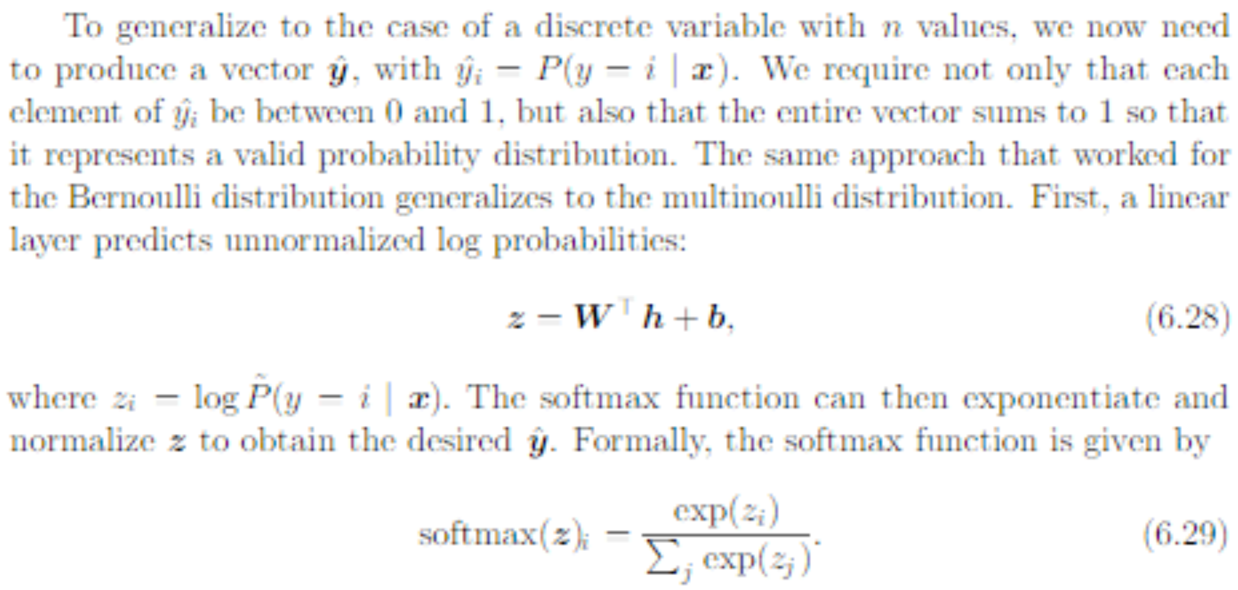
as the cross-entropy between the training data and the model distribution. This cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$. The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded. For example, as we saw in section 5.5.1, if $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}\,; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$, then we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}, \tag{6.13}$$

# Classifier output



To generalize to the case of a discrete variable with $n$ values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We require not only that each element of $\hat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

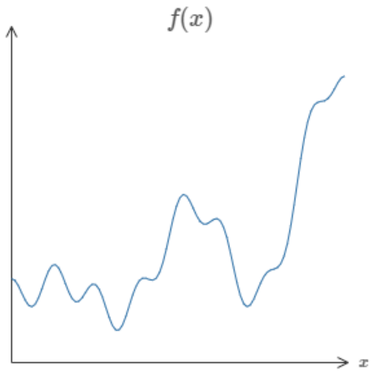
where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$. The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax function is given by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.29}$$

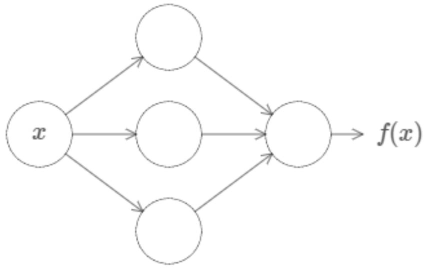
## Universal approximation theorem

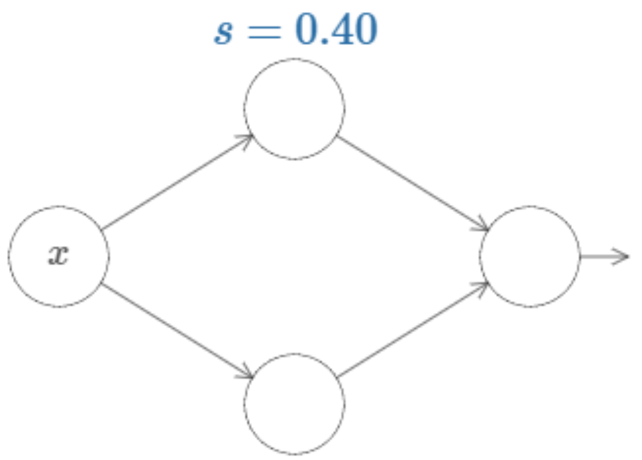
- A feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as logistic sigmoid) can approximate any continuous function from one finite-dim space to another with any desired nonzero error.

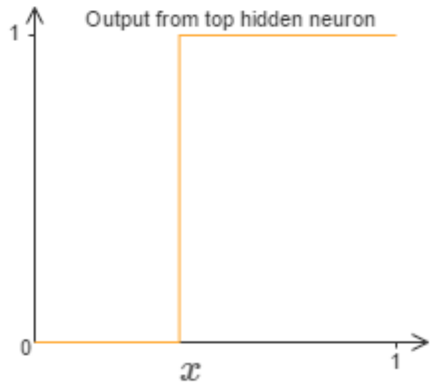
One of the most striking facts about neural networks is that they can compute any function at all. That is, suppose someone hands you some complicated, wiggly function, $f(x)$:



No matter what the function, there is guaranteed to be a neural network so that for every possible input, $x$, the value $f(x)$ (or some close approximation) is output from the network, e.g.:

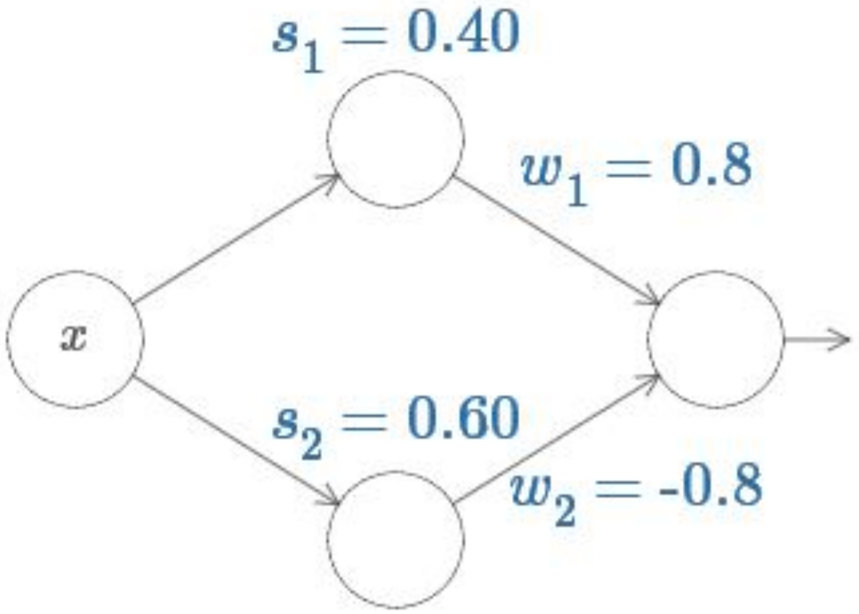


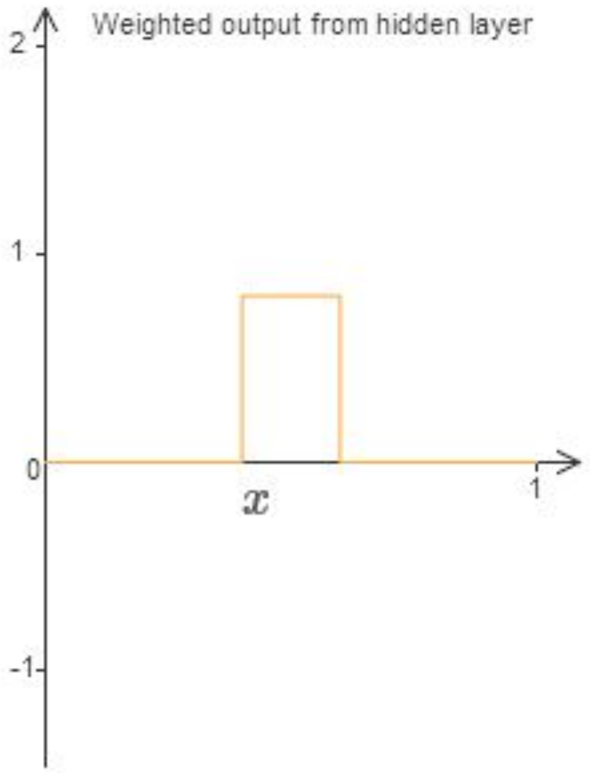


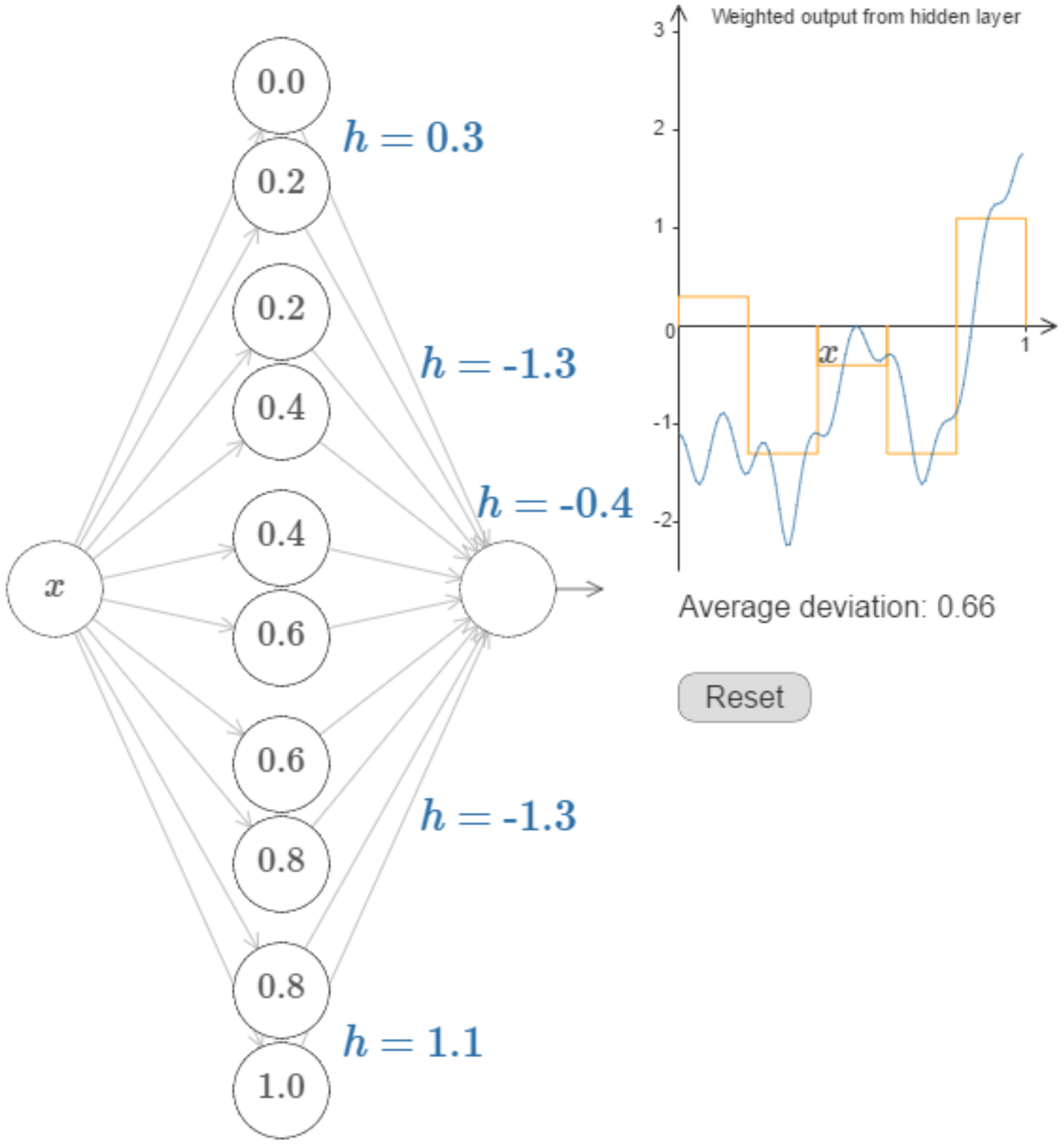


s=-b/w, big w and b results in step

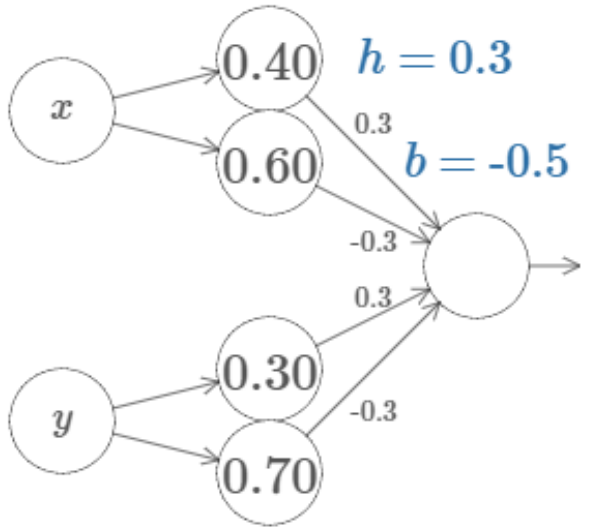
As we learnt earlier in the book, what's being computed by the hidden neuron is $\sigma(wx + b)$, where $\sigma(z) \equiv 1/(1 + e^{-z})$ is the sigmoid function.


$s_1 = 0.40$
$w_1 = 0.8$
$x$
$s_2 = 0.60$
$w_2 = $ -0.8

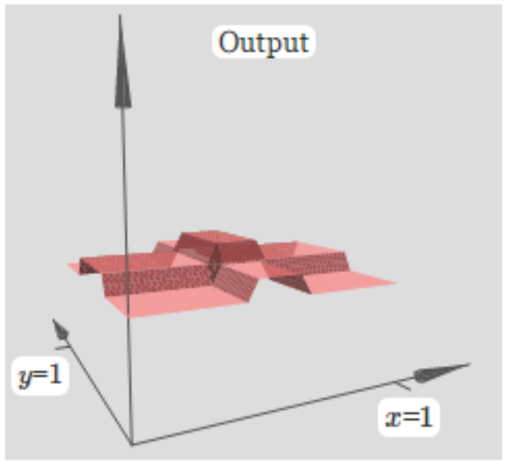

Weighted output from hidden layer
2
1
0
-1
$x$
1



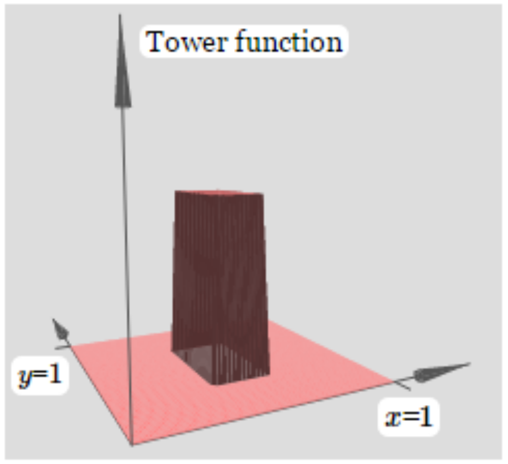
Average deviation: 0.66



$x$
$y$
0.40
0.60
0.30
0.70
$h = 0.3$
$b = -0.5$
0.3
-0.3
0.3
-0.3



Output
$y$=1
$x$=1

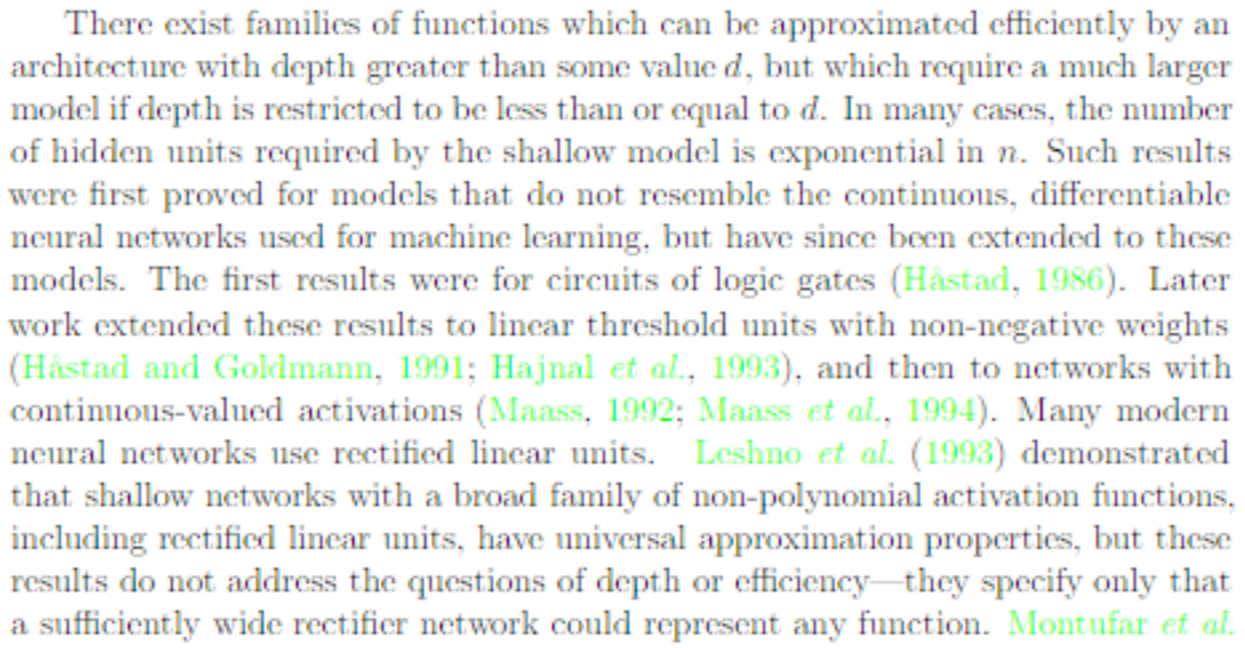


Tower function
$y$=1
$x$=1

## Universal approximation theorem

- In worse case, exponential number of hidden units (possibly one for each input config that needs to be distinguished) is required.
- In binary case, easy to see: number of possible binary vectors on v in $\{0,1\}^n$ is 2^2^n, selecting one such function requires 2^n bits.
- While single hidden layer is sufficient to represent any function, the layer may be unfeasibly large and may fail to learn and generalize correctly.

# Exponential increase in efficiency with depth



There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value $d$, but which require a much larger model if depth is restricted to be less than or equal to $d$. In many cases, the number of hidden units required by the shallow model is exponential in $n$. Such results were first proved for models that do not resemble the continuous, differentiable neural networks used for machine learning, but have since been extended to these models. The first results were for circuits of logic gates (Håstad, 1986). Later work extended these results to linear threshold units with non-negative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), and then to networks with continuous-valued activations (Maass, 1992; Maass *et al.*, 1994). Many modern neural networks use rectified linear units. Leshno *et al.* (1993) demonstrated that shallow networks with a broad family of non-polynomial activation functions, including rectified linear units, have universal approximation properties, but these results do not address the questions of depth or efficiency—they specify only that a sufficiently wide rectifier network could represent any function. Montufar *et al.*

(2014) showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network. More precisely, they showed that piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a number of regions that is exponential in the depth of the network. Figure 6.5 illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value nonlinearity). By composing these folding operations, we obtain an exponentially large number of piecewise linear regions which can capture all kinds of regular (e.g., repeating) patterns.

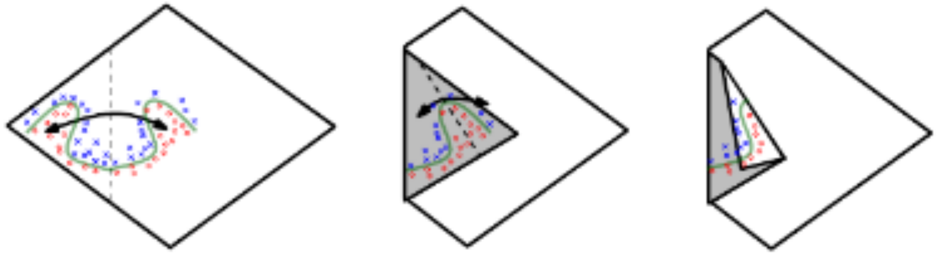


Figure 6.5: An intuitive, geometric explanation of the exponential advantage of deeper rectifier networks formally by Montufar *et al.* (2014). *(Left)*An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit. A function computed on top of that unit (the green decision surface) will be a mirror image of a simpler pattern across that axis of symmetry. *(Center)*The function can be obtained by folding the space around the axis of symmetry. *(Right)*Another repeating pattern can be folded on top of the first (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers). Figure reproduced with permission from Montufar *et al.* (2014).

More precisely, the main theorem in Montufar *et al.* (2014) states that the number of linear regions carved out by a deep rectifier network with $d$ inputs, depth $l$, and $n$ units per hidden layer, is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right), \tag{6.42}$$

# Computational experiment demonstrating increase in efficiency with depth

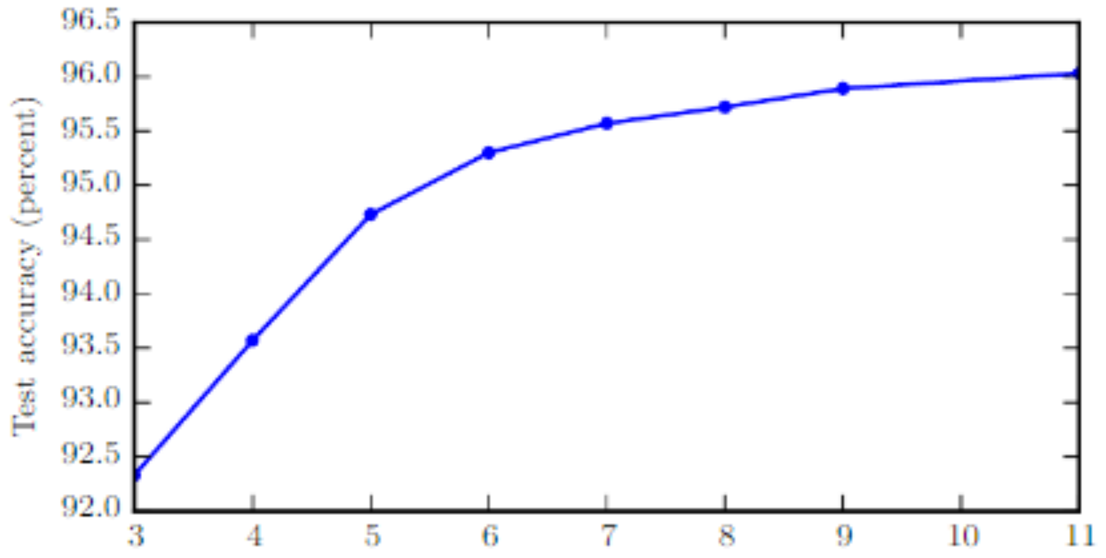


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

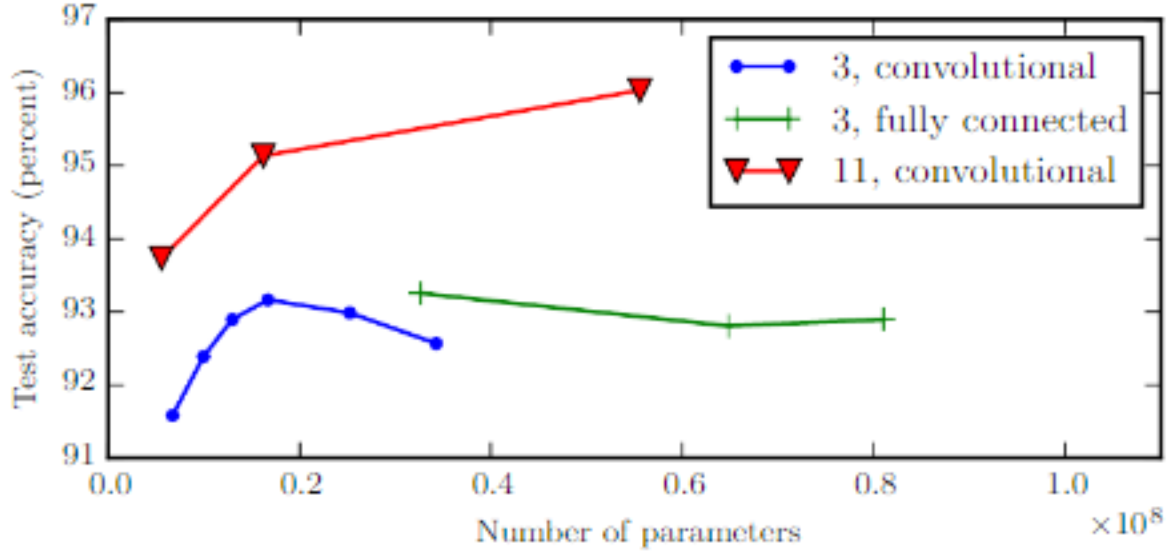
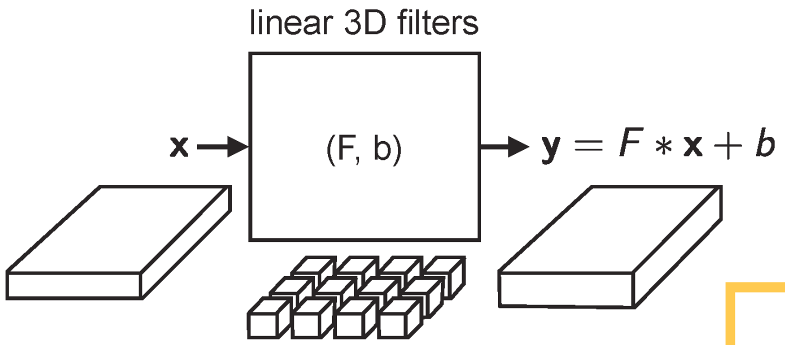


Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

## Convolutional neural network (special type of multilayer perceptron)

## Linear convolution

**A bank of "3D" linear filters**



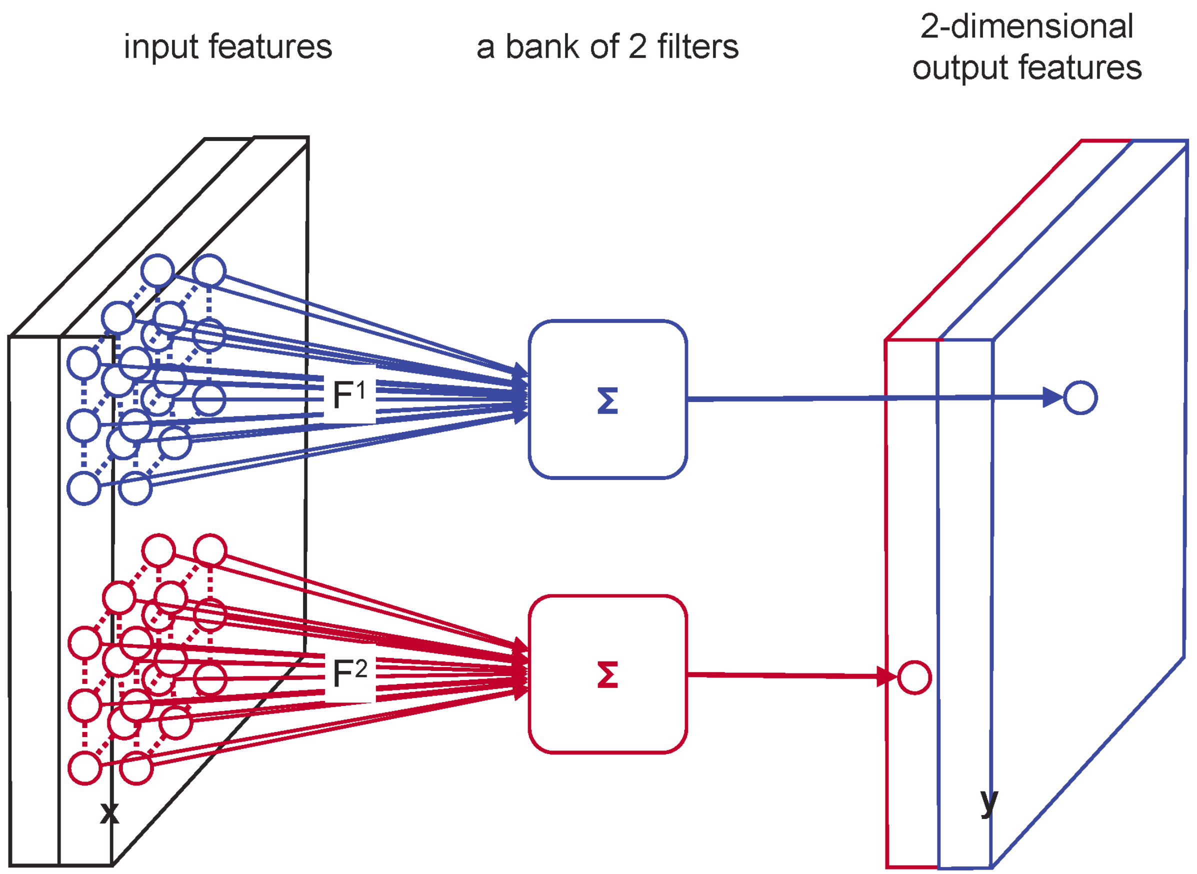
$$y_{ijq} = b_q + \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \sum_{k=1}^{K} x_{u+i,v+j,k} f_{u,v,k,q}$$

Linear, translation invariant, local:

- Input **x** = H × W × K array
- Filter bank F = H' × W' × K × Q array
- Output **y** = (H - H' + 1) × (W - W' + 1) × Q array

# Linear convolution

**As a neural network**



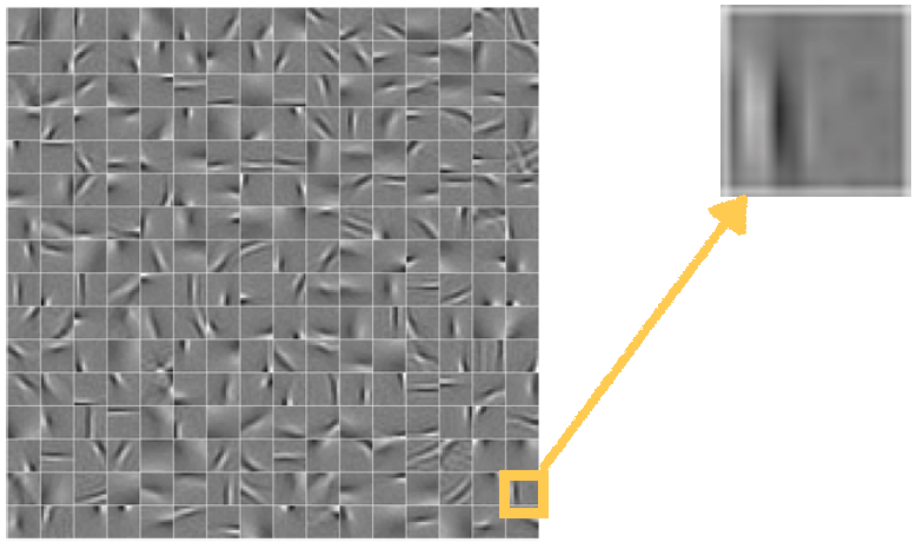
# Linear convolution

## Filter bank example

A bank of 256 filters (learned from data)

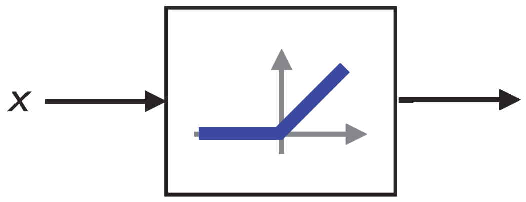
Each filter is 1D (it applies to a grayscale image)

Each filter is $16 \times 16$ pixels



# Activation functions

**Scalar non-linearity**



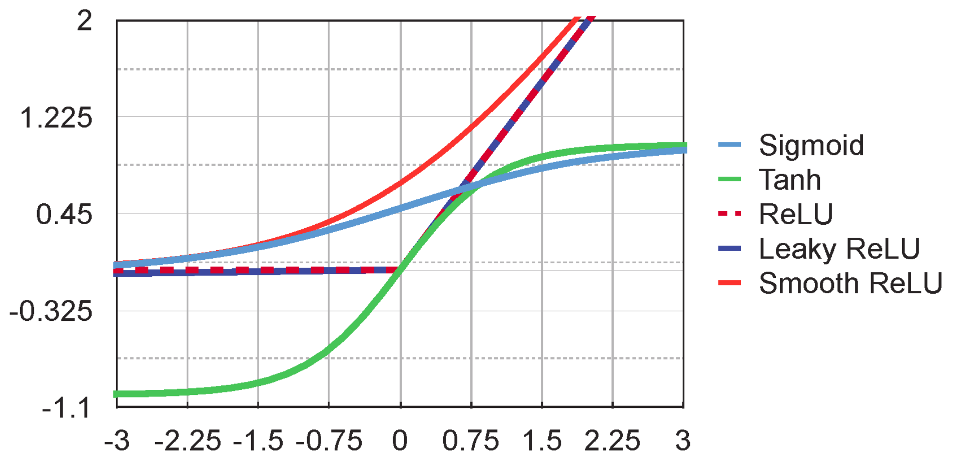
$$y = \frac{1}{1+e^{-x}}$$ sigmoid

$$y = \tanh(x)$$ hyperb. tan

$$y = \max\{0, x\}$$ ReLU

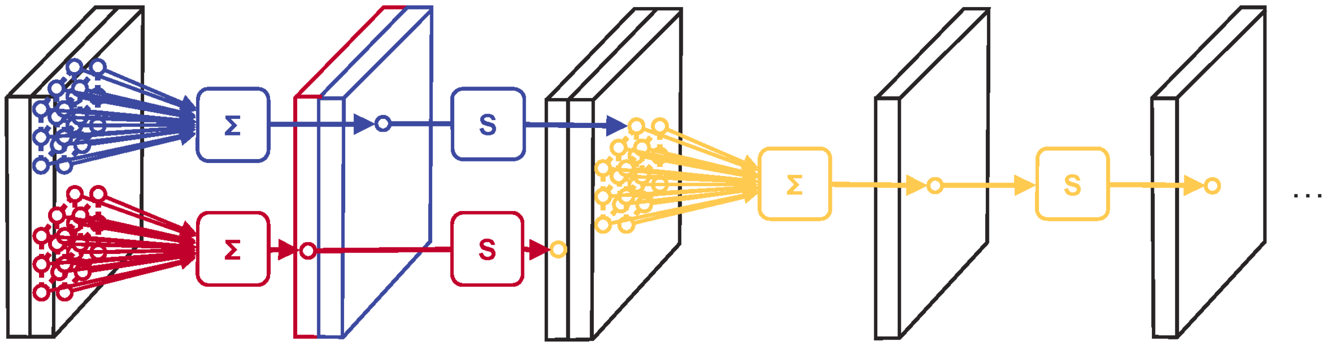
$$y = \log(1+e^{x})$$ Soft ReLU

$$y = \epsilon x + (1-\epsilon)\max\{0, x\}$$ Leaky ReLU



## Multiple layers

**Convolution, gating, convolution, …**

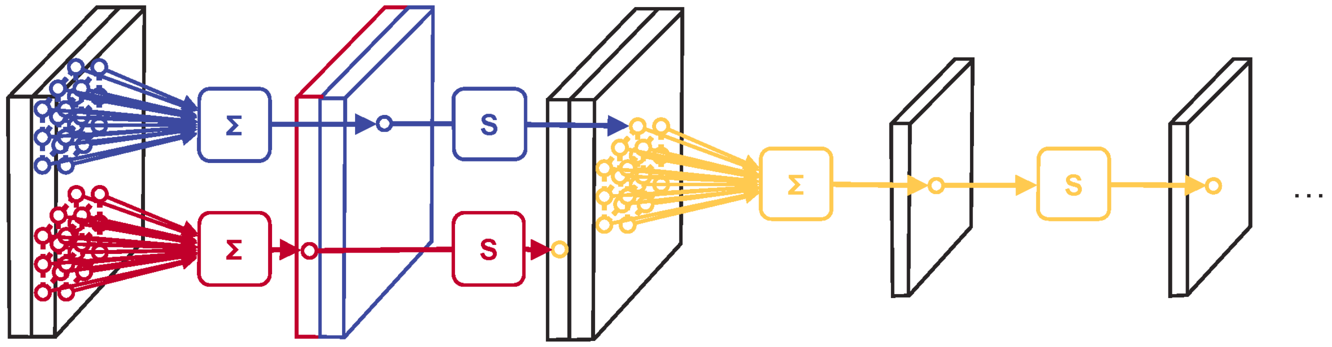


Filters are followed by non-linear operators (e.g. gating, but see later)

Multiple such layers are chained together

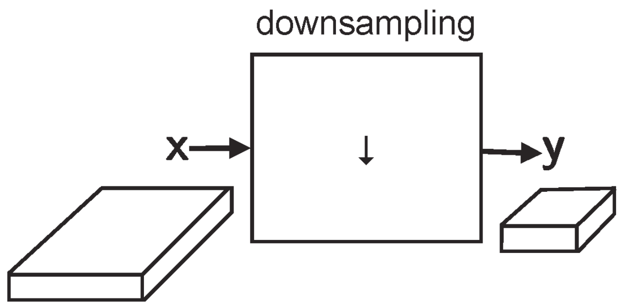
## Multiple layers

### Downsampling



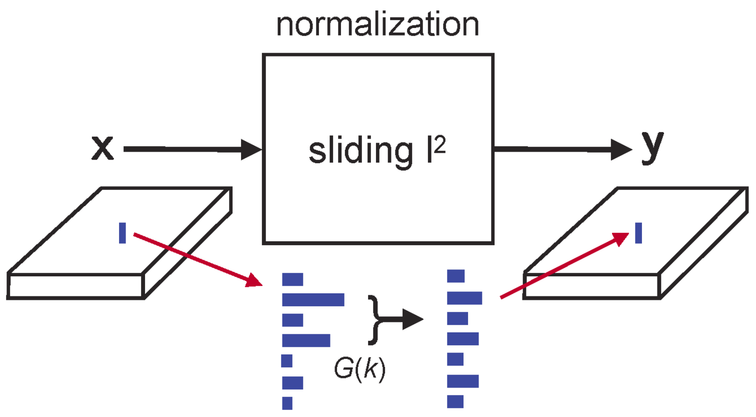
Filters are often followed (or incorporate) downsampling

This is often compensated by an increase in the number of feature channels (not shown)



## Local feature normalisation

**Across feature channels rather than spatially**



$$y_{ijk} = x_{ijk} \left( \kappa + \alpha \sum_{q \in G(k)} x_{ijq}^2 \right)^{-\beta}$$

Operates at each spatial location independently

Normalize groups G(k) of feature channels

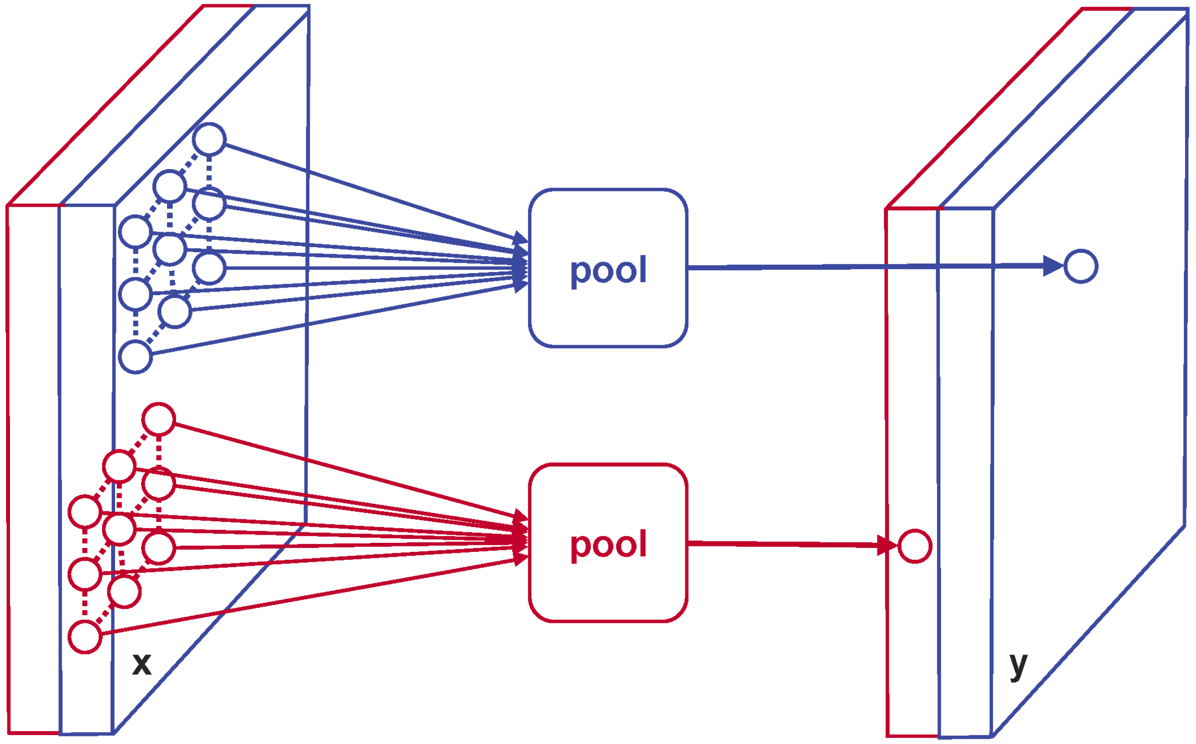
Groups are usually defined in a **sliding window manner**

## Spatial pooling

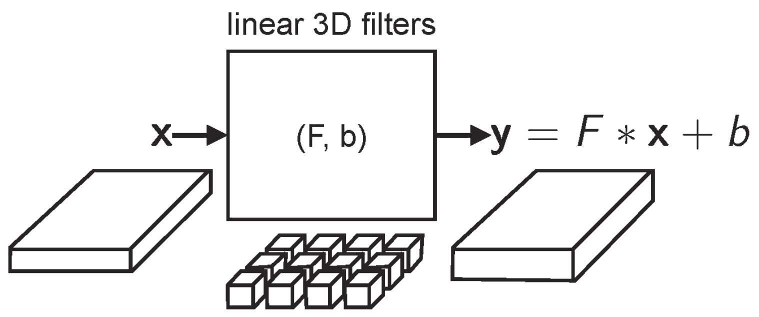
**Reduce dependency on precise location**

Pooling compute the average / max of the features in a neighbourhood.

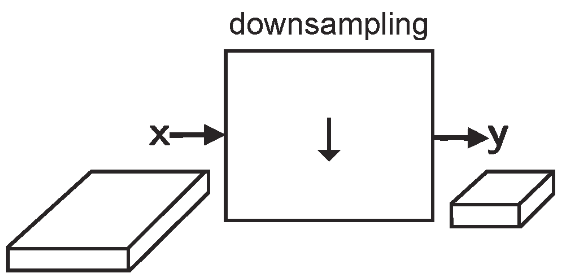
It is applied channel-by-channel.

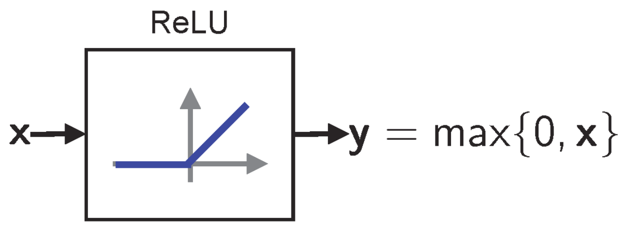


## Components summary

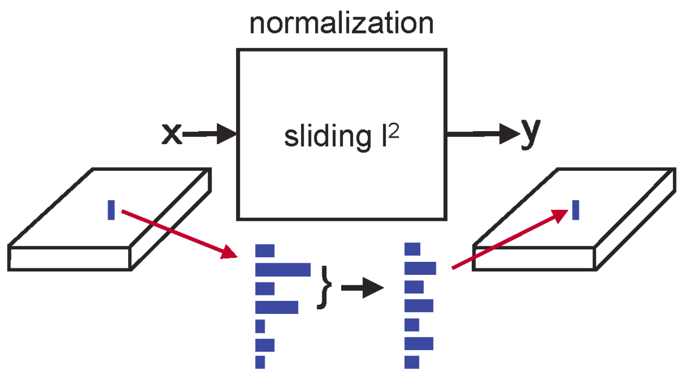


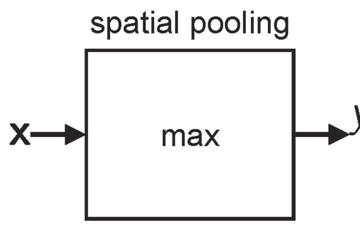
$$\mathbf{y} = F * \mathbf{x} + b$$





$$\mathbf{y} = \max\{0, \mathbf{x}\}$$





$$y_{ijk} = \max_{pq \in \Omega_{ij}} x_{pqk}$$

# Motivation from Visual Cortex

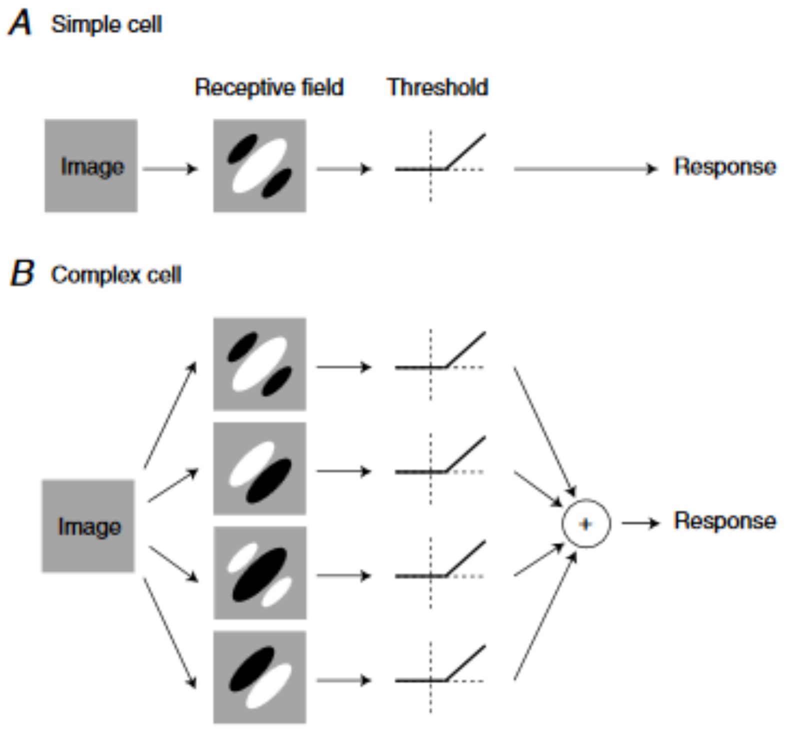
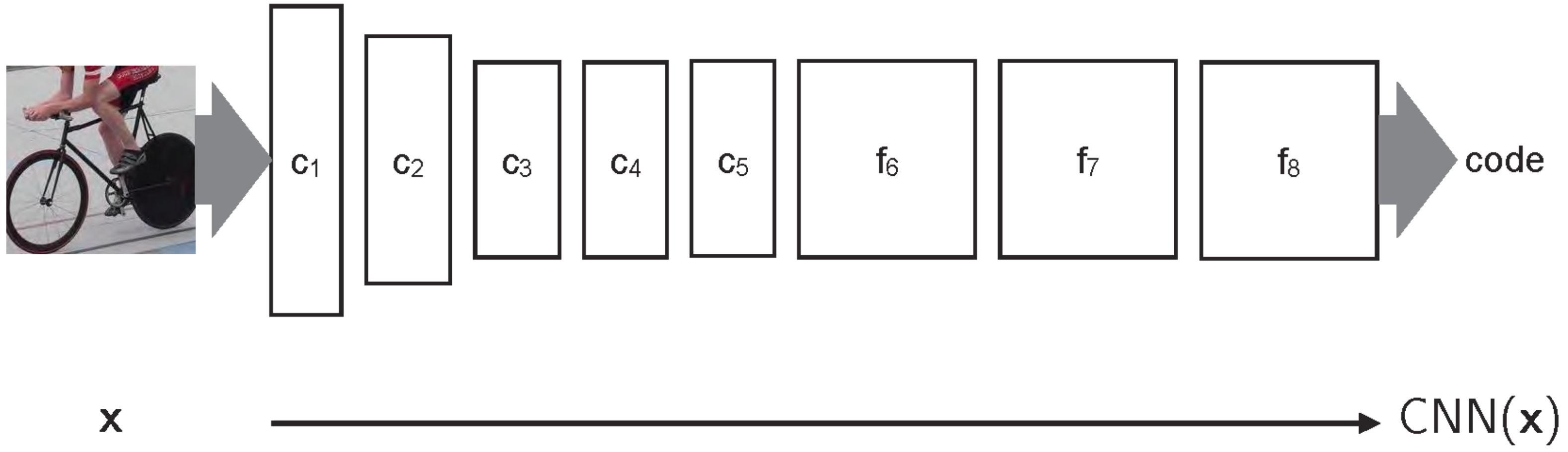


**Figure 1. The models of simple and complex cells proposed by Movshon, Thompson and Tolhurst (Movshon *et al.* 1978*a,b*)**
*A*, linear model of simple cells. The first stage is linear filtering, i.e. aweighted sum of the image intensities, with weights given by the receptive field. The second stage isrectification: only the part of the responses that is larger than a threshold is seen in the firing rate response. *B*, subunit model of complex cells. The first stage is linear filtering by a number of receptive fields such asthose of simple cells (here we show four of them with spatial phases offset by 90 deg). The subsequent stages involve rectification, and then summation.

## Convolutional neural networks (CNNs)



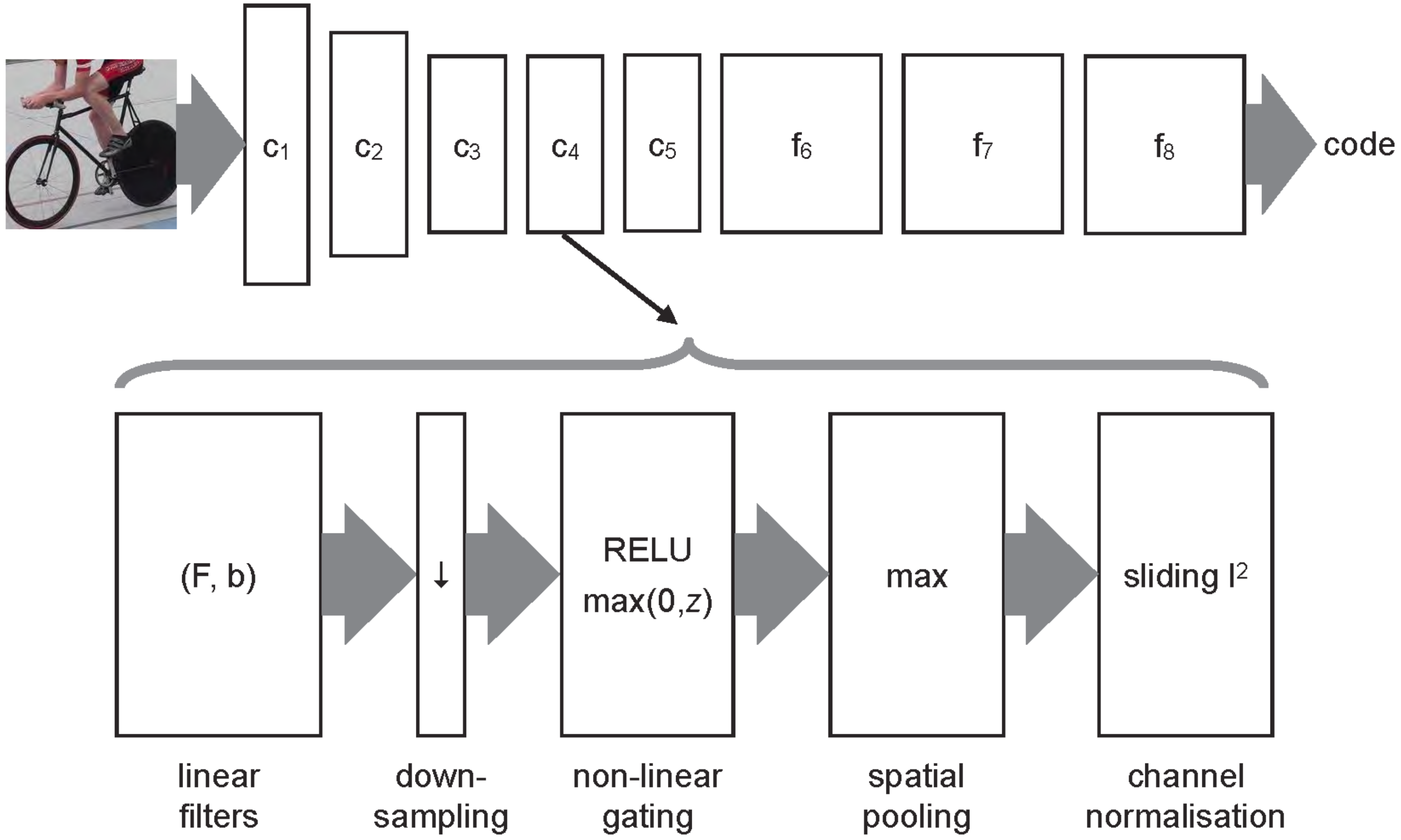
**From left to right**

- decreasing spatial resolution
- increasing feature dimensionality

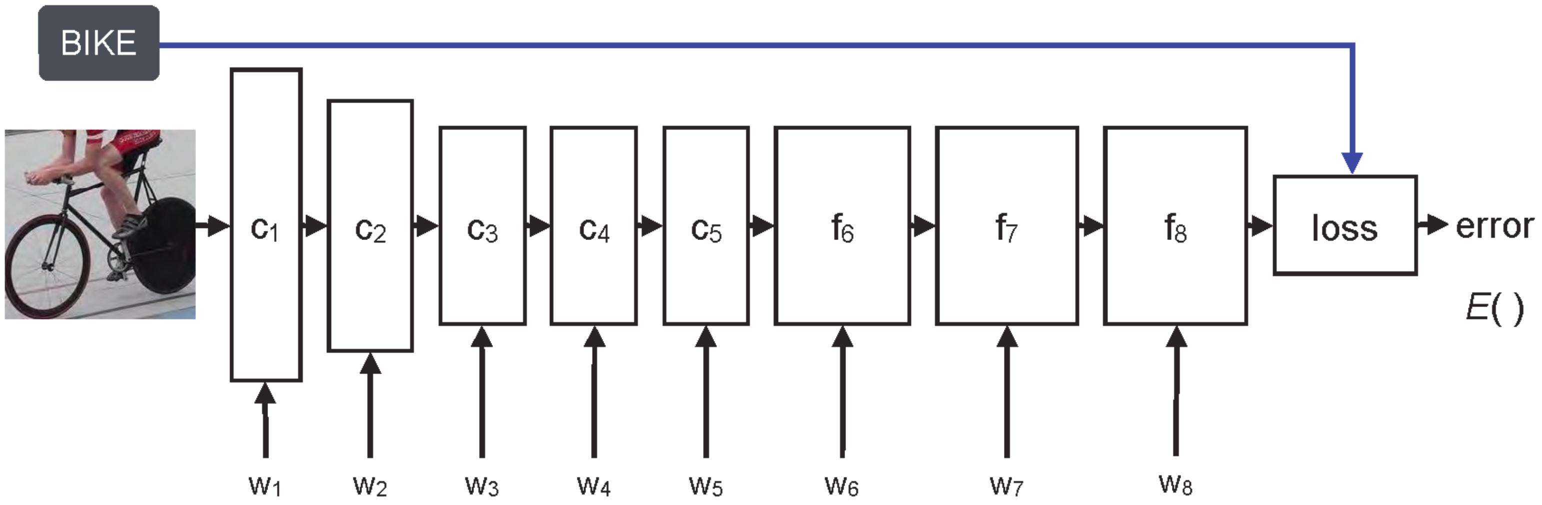
**Fully-connected layers**

- same as convolutional, but with $1 \times 1$ spatial resolution
- contain most of the parameters

## Convolutional layers



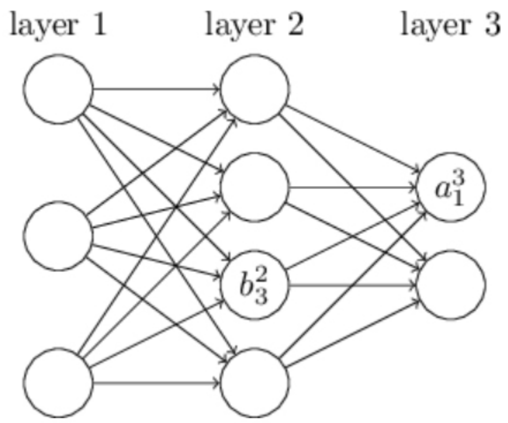
## Learning a CNN



$$\boxed{\operatorname{argmin} E(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_8)}$$

Stochastic gradient descent
(with momentum, dropout, ...)

# Stochastic Gradient Descent via Back propagation



$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \qquad C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2$$

**Summary: the equations of backpropagation**

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{BP4}$$

# Stochastic Gradient Descent via Back propagation

1. **Input a set of training examples**

2. **For each training example $x$:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:

    - **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.

    - **Output error $\delta^{x,L}$:** Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.

    - **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

## Is back propagation biological?

## Learning CNNs classifiers

**Challenge**

- many parameters, prone to overfitting

**Key ingredients**

- very large annotated data
- heavy regularisation (dropout)
- stochastic gradient descent
- GPU(s)

IMAGENET

1K classes
~ 1K training images per class
~ 1M training images

**Training time**

- ~ 90 epochs
- days—weeks of training
- requires processing ~150 images/sec

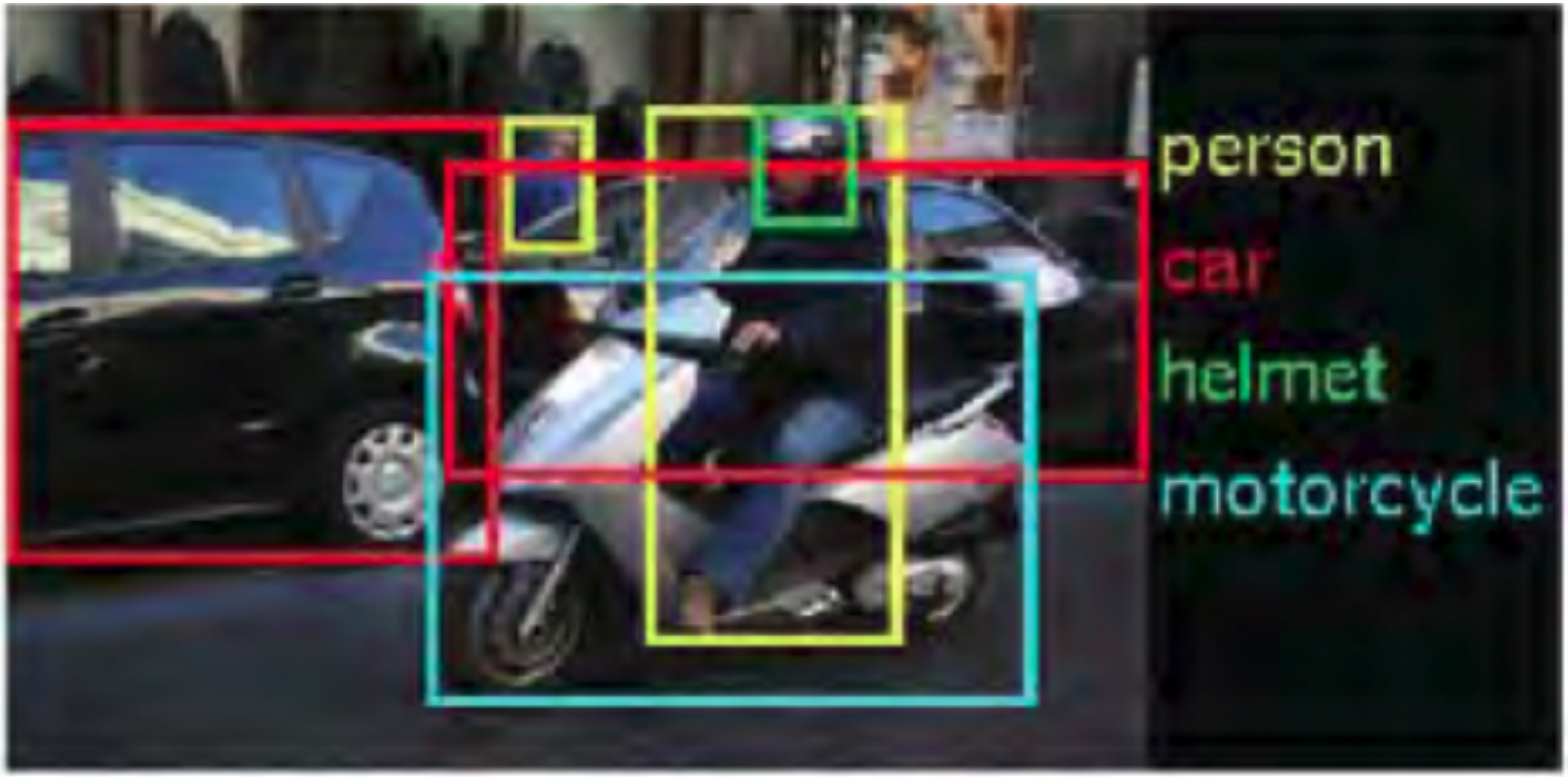
## IMAGENET [Deng et al. 2009]

**www.image-net.org**

**22K** categories and **15M** images

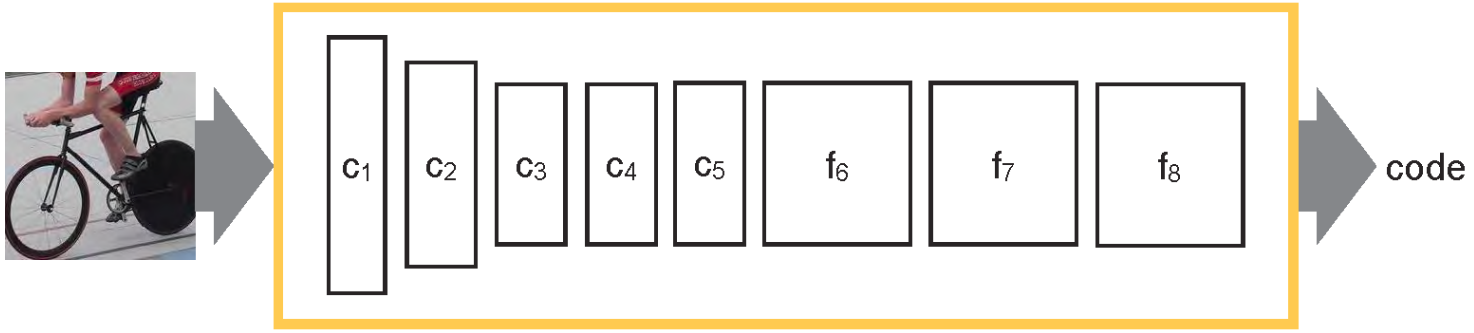
- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
- Food
- Materials
- Structures
- Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
- Sport Activities

## Human level performance



Slide credit: Joshua Bengio

## CNNs as general purpose representations



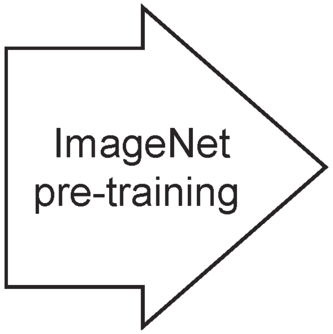
**Step 1:** pre-train a large CNN on a very large dataset

**Step 2:** reuse the CNN to tackle smaller problems

- chop off last layer
- the output is a code vector for the image

## Feature generality

**Pre-trained features are quite general**



**Object classification** (PASCAL VOC)

- [Chatifield et al. 2014, Razavian et al. 2014, Zeiler et al. 2014]

**Object detection** (PASCAL VOC)

- R-CNN [Girshick et al. 2014]

**Fine-grained classification** (UCSD birds)

- Part-R-CNN  [Zhang et al. 2014]

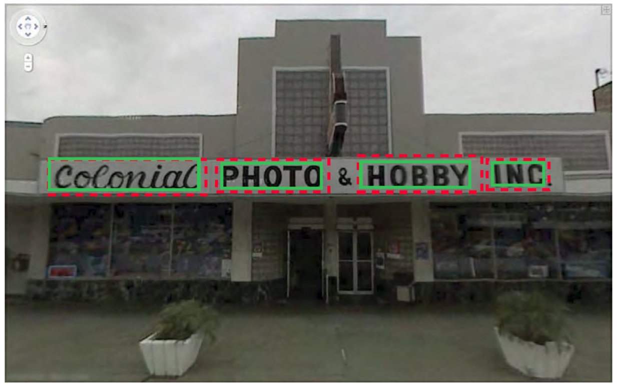
**MIT 67 scene classification**

- [Razavin et al. 2014]

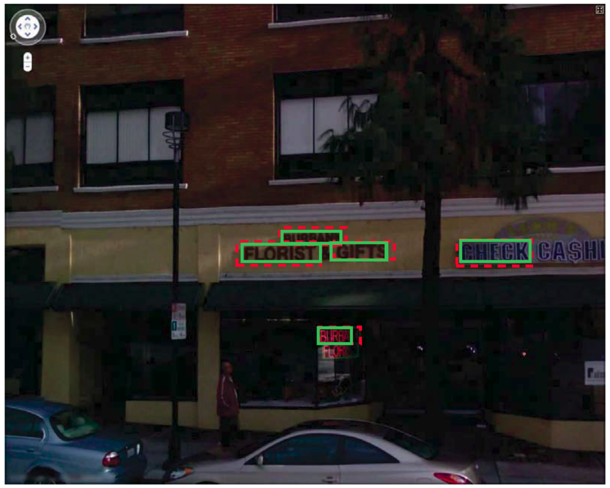
…

# Qualitative results: text spotting

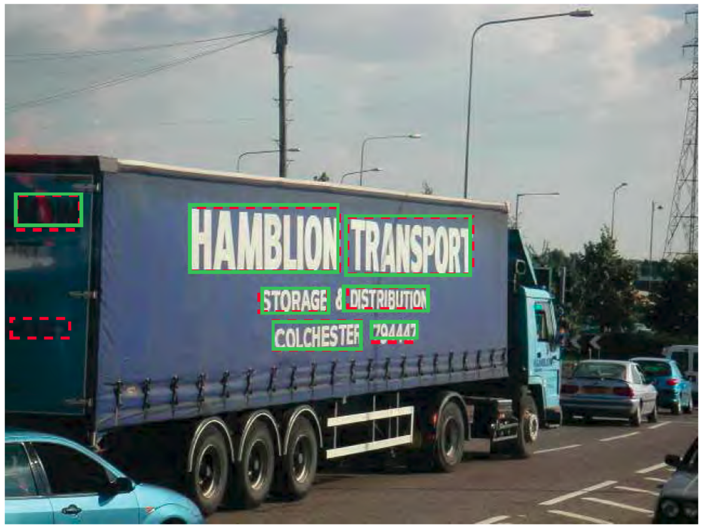
1.00/1.00/1.00



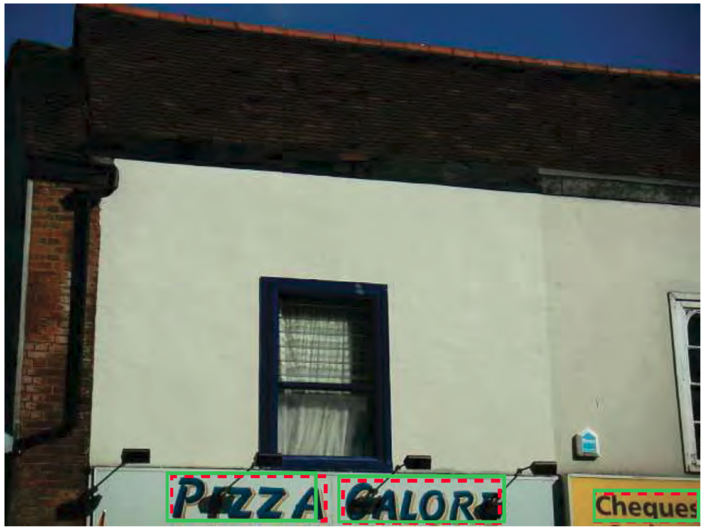
1.00/1.00/1.00



1.00/0.88/0.93

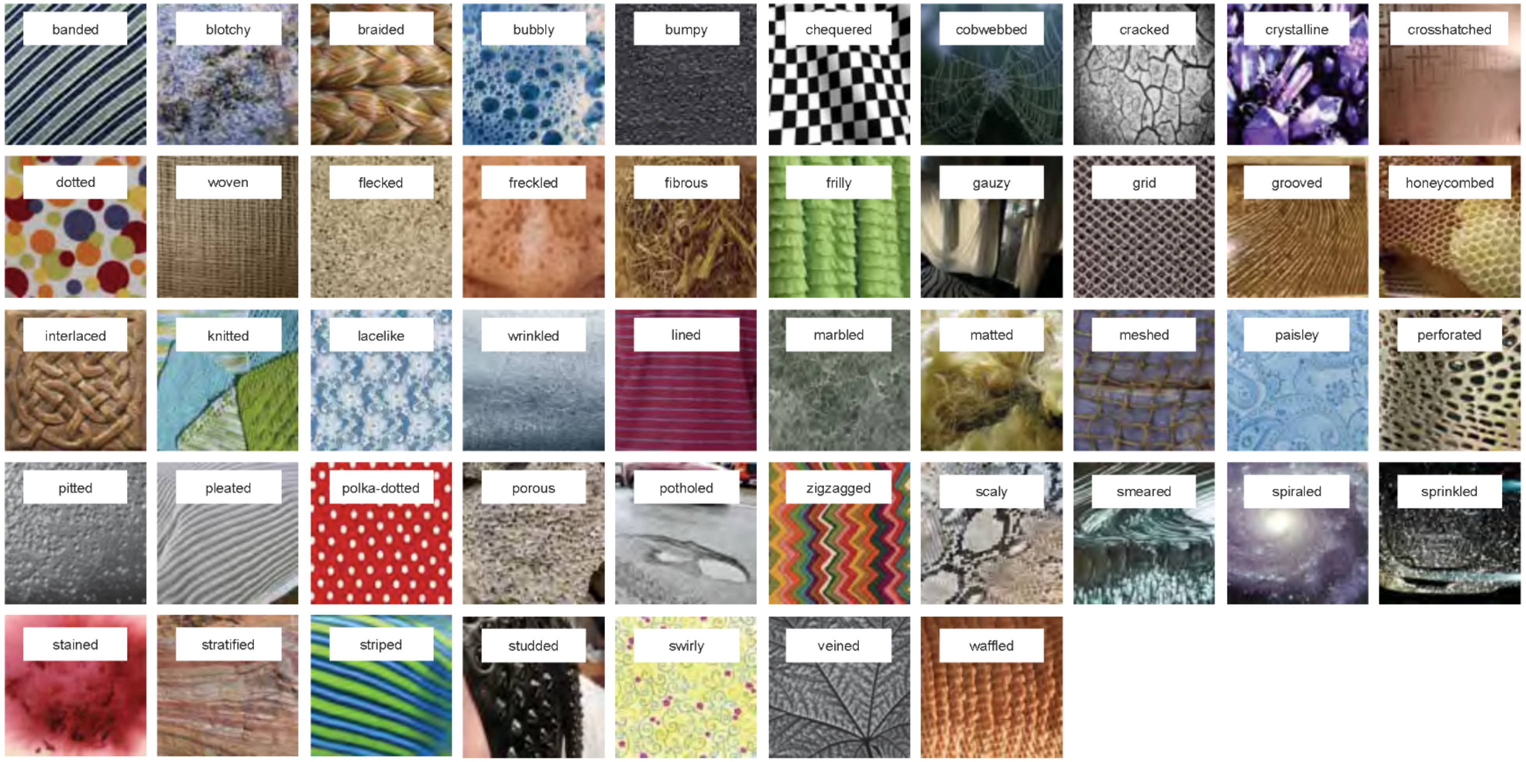


1.00/1.00/1.00



# Feature generality

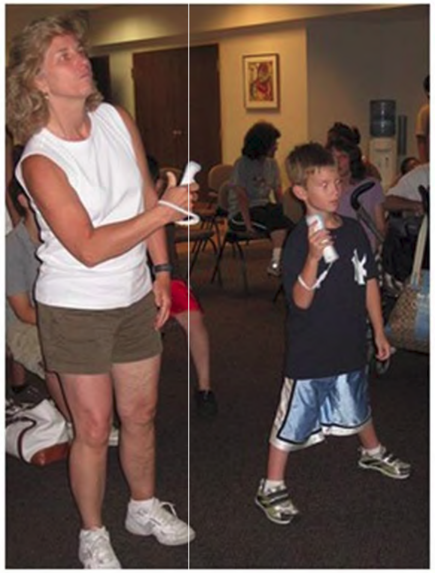
ImageNet pre-trained features achieve **state-of-the-art material recognition** and **texture naming** (but similar to Fisher Vector) [Cimpoi et al. 2014, 2015a, 2015b]



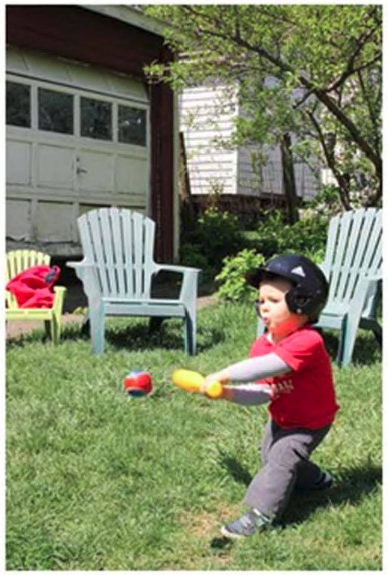
[Describable textures dataset]

# Verbal descriptions of pictures

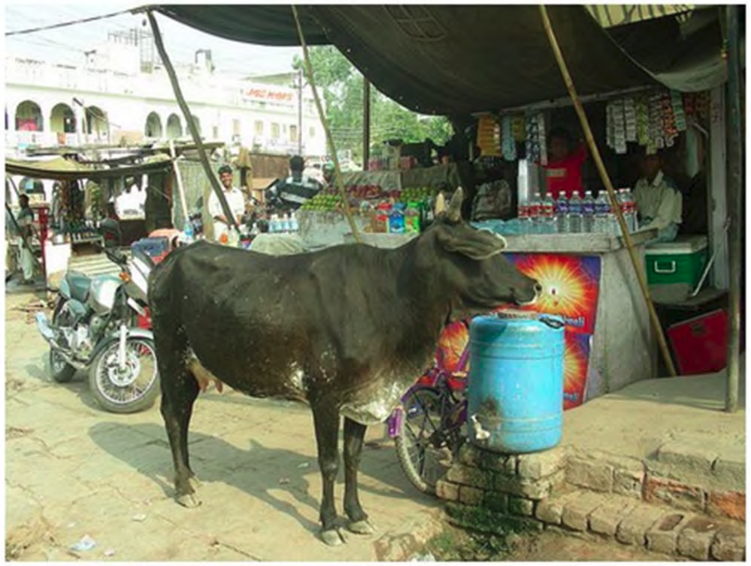
Wow I can't believe that worked



a group of people standing around a room with remotes
`logprob: -9.17`



a young boy is holding a baseball bat
`logprob: -7.61`



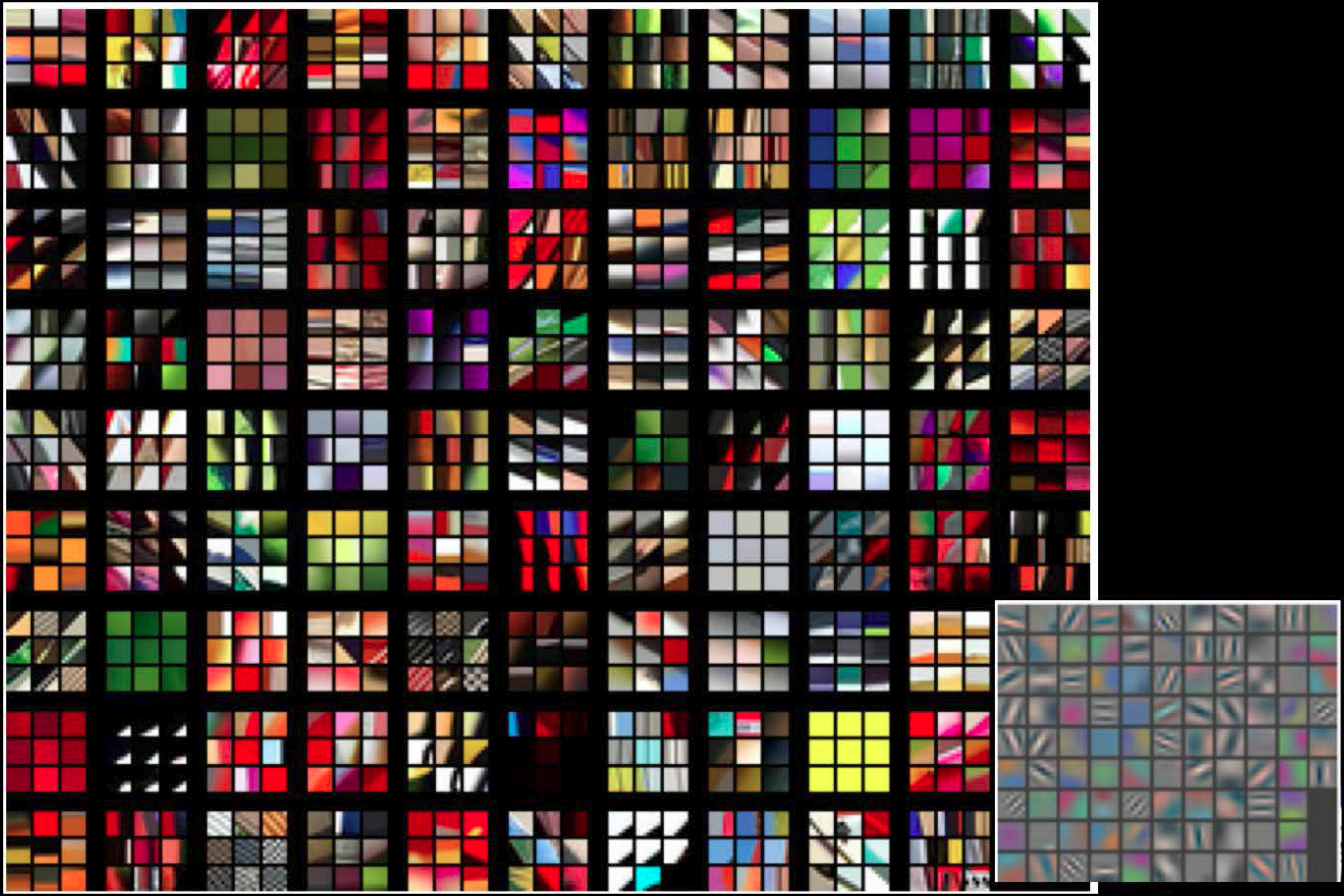
a cow is standing in the middle of a street
`logprob: -8.84`

Karpathy & Fei-Fei, CVPR 2015

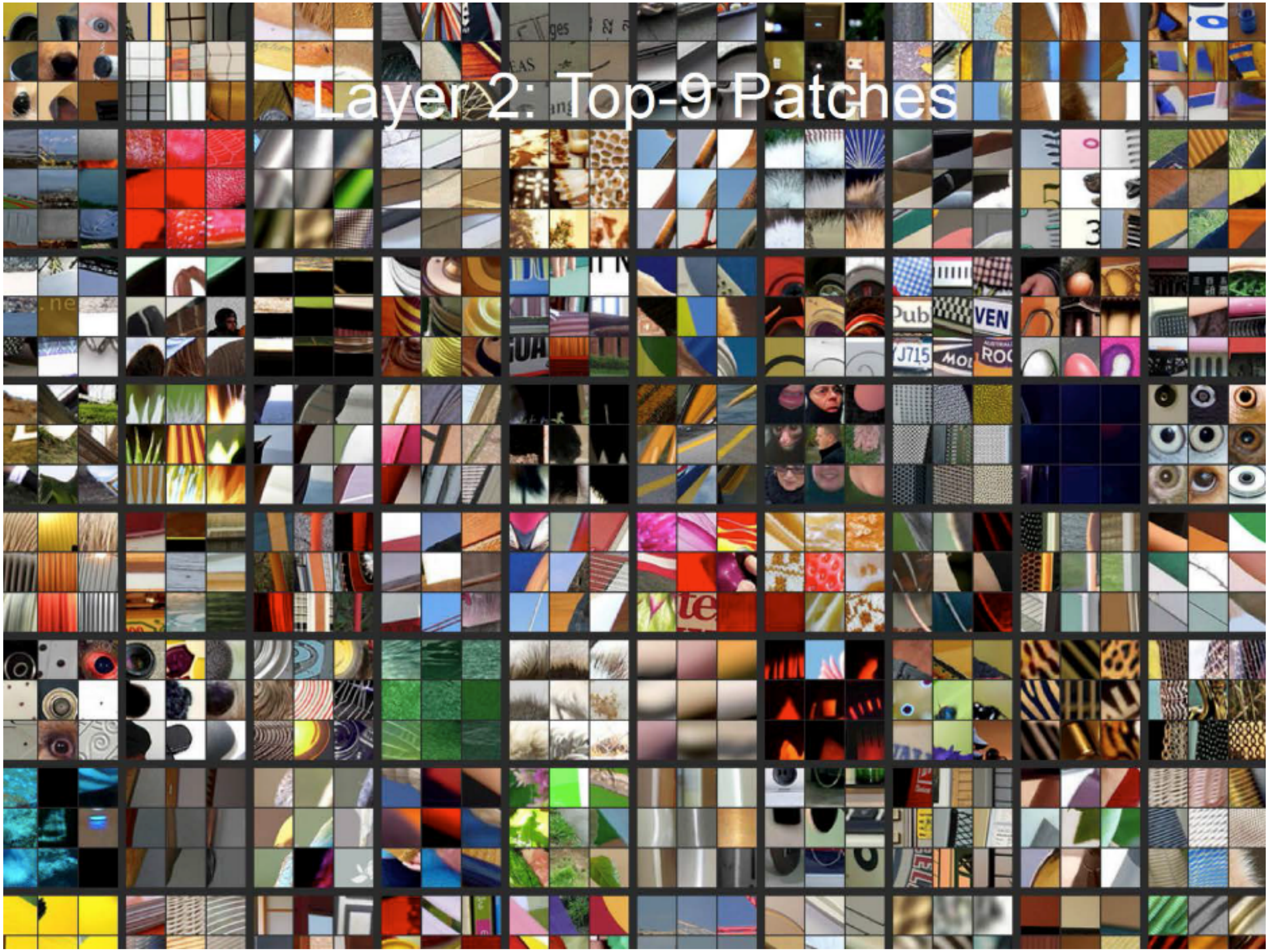
Slide credit: Fei-Fei Li

Slide credit: Matthew Zeiler

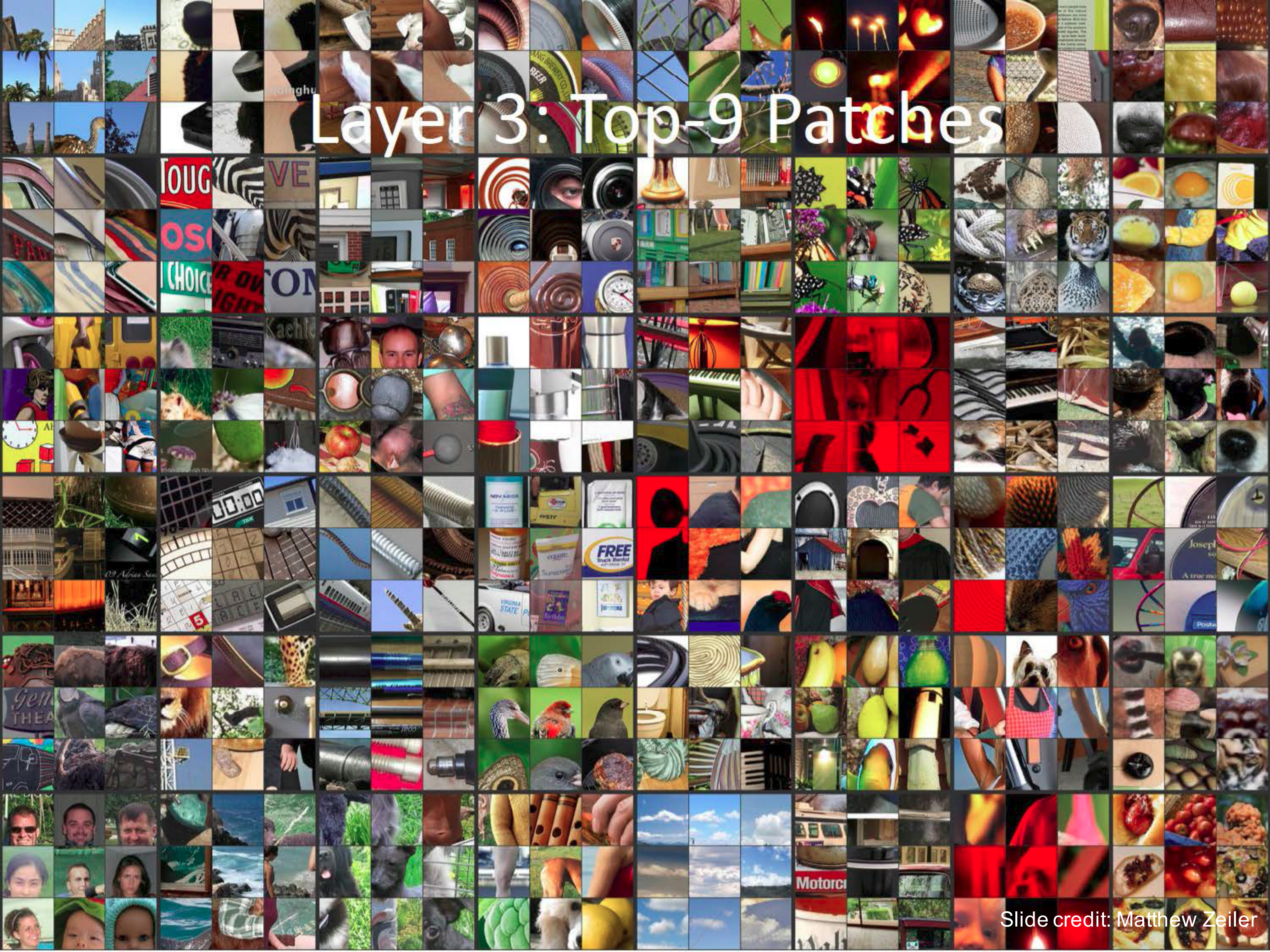
# Layer 1: Top-9 Patches



Slide credit: Matthew Zeiler

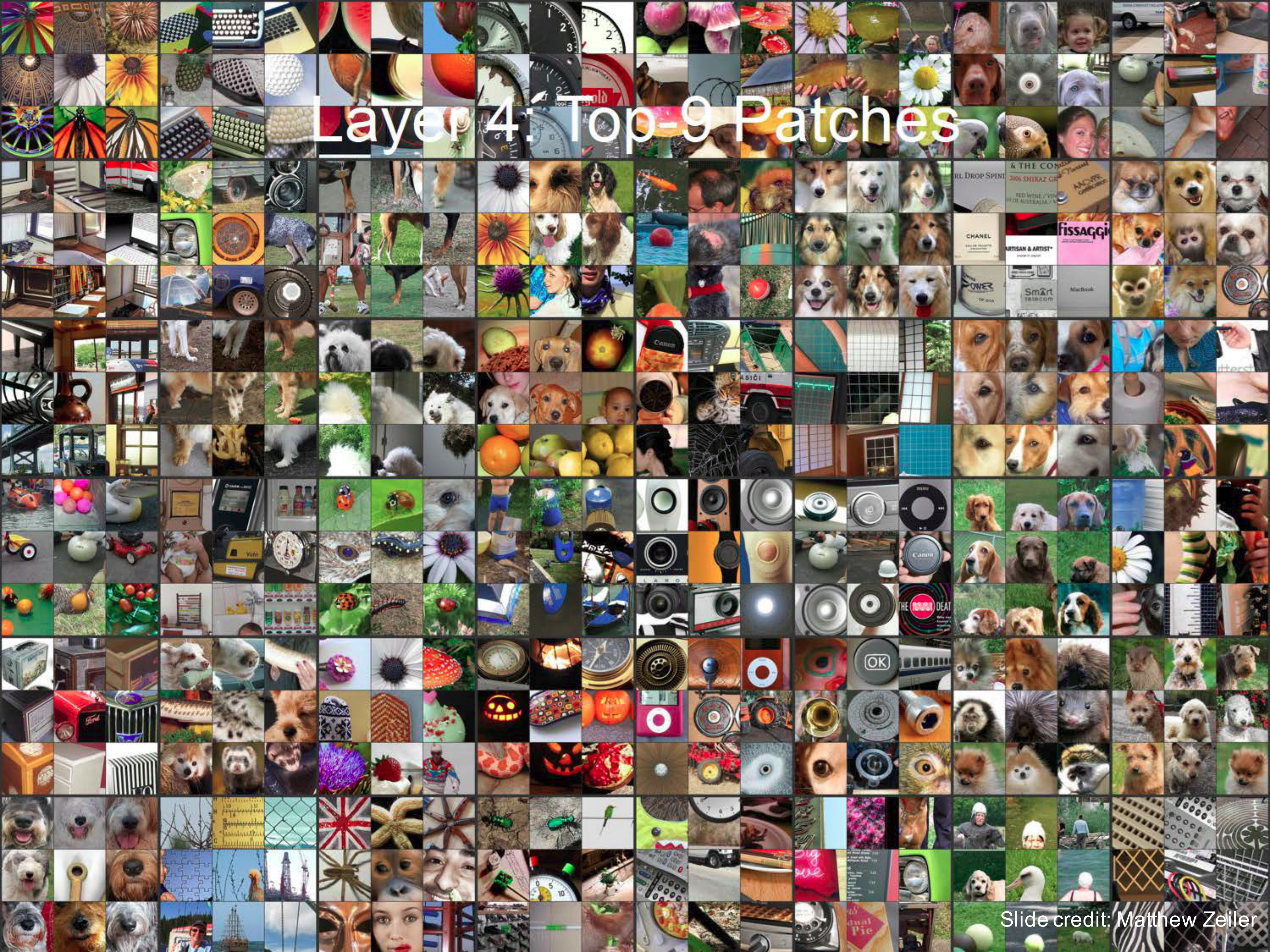


# Layer 3: Top-9 Patches



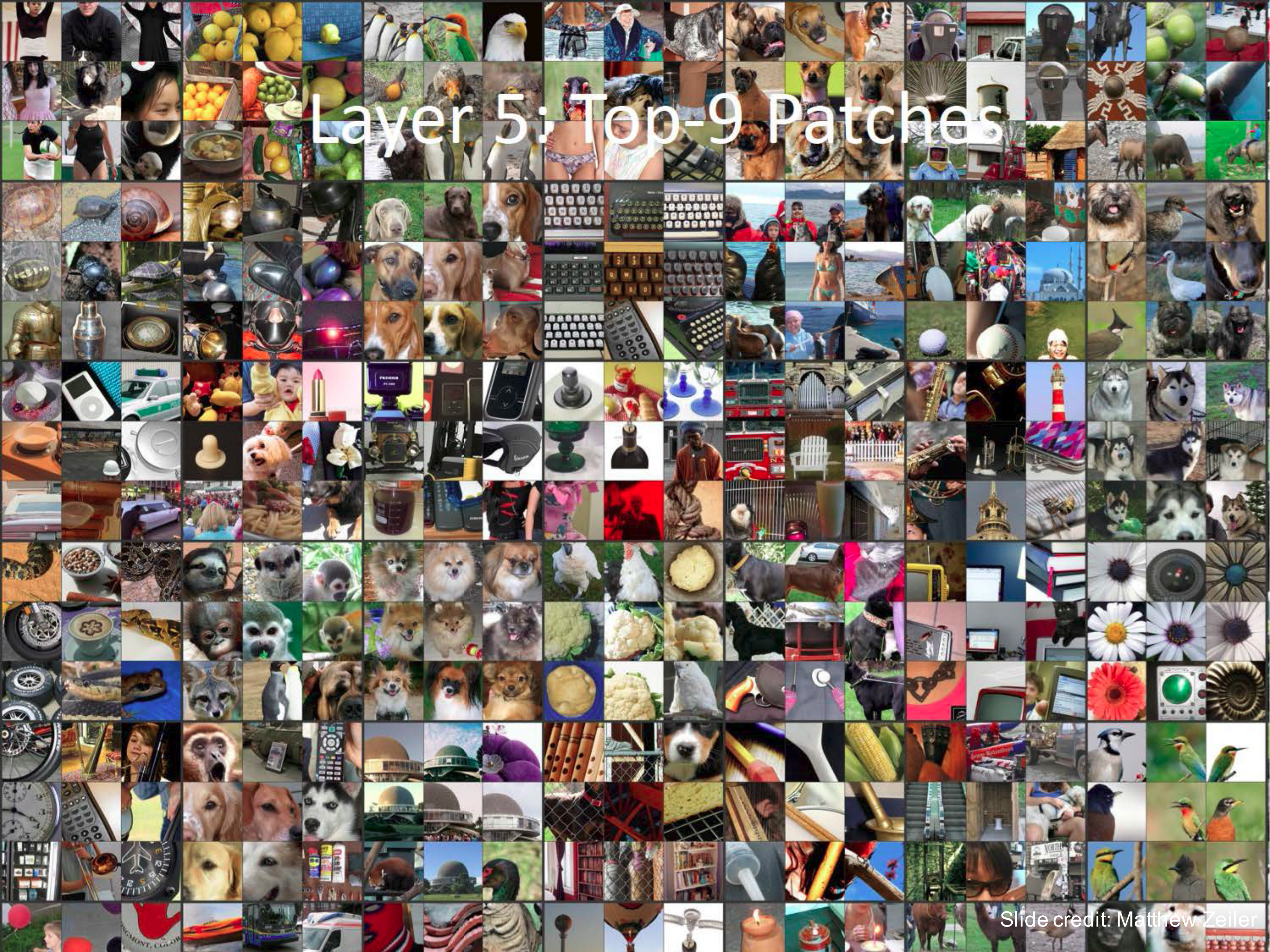
Slide credit: Matthew Zeiler

# Layer 4: Top-9 Patches



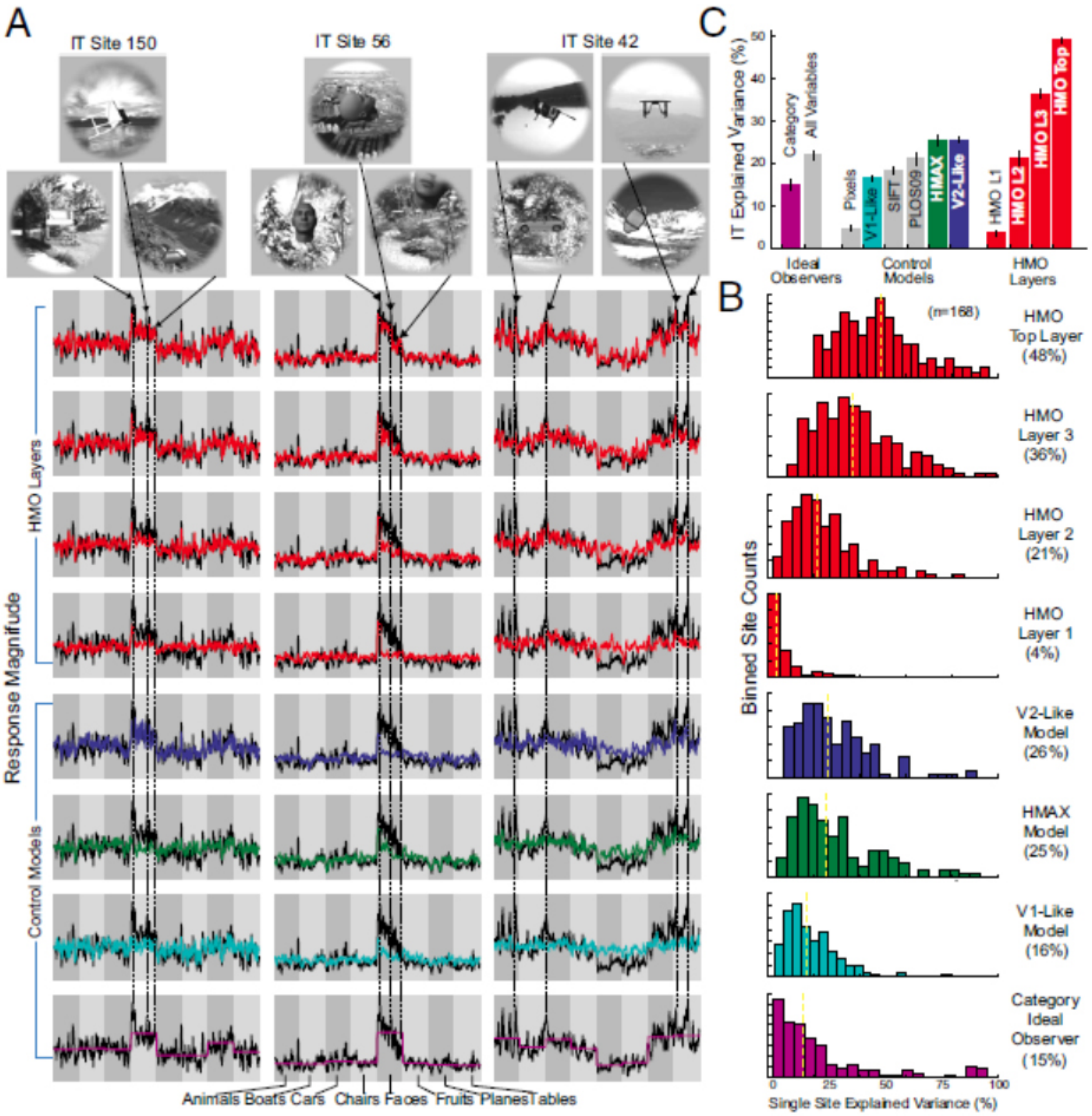
Slide credit: Matthew Zeiler

# Layer 5: Top-9 Patches

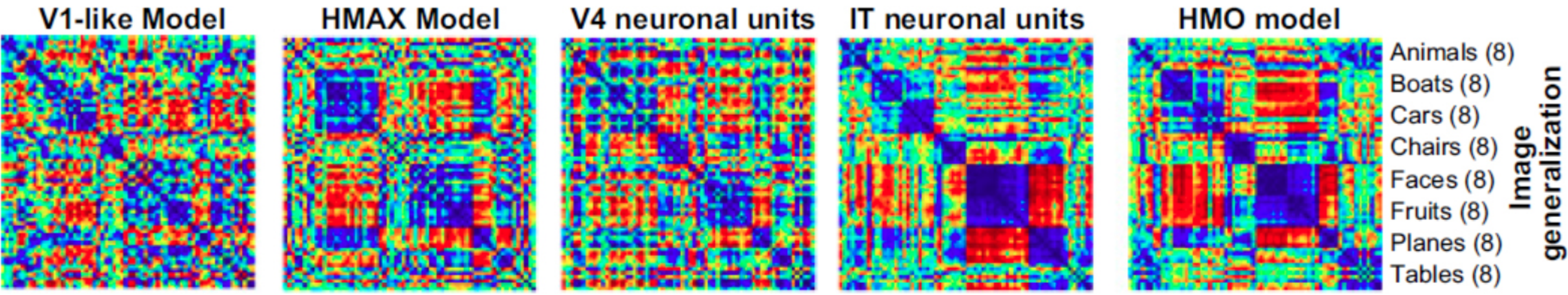


Slide credit: Matthew Zeiler

## Do deep networks model the brain?

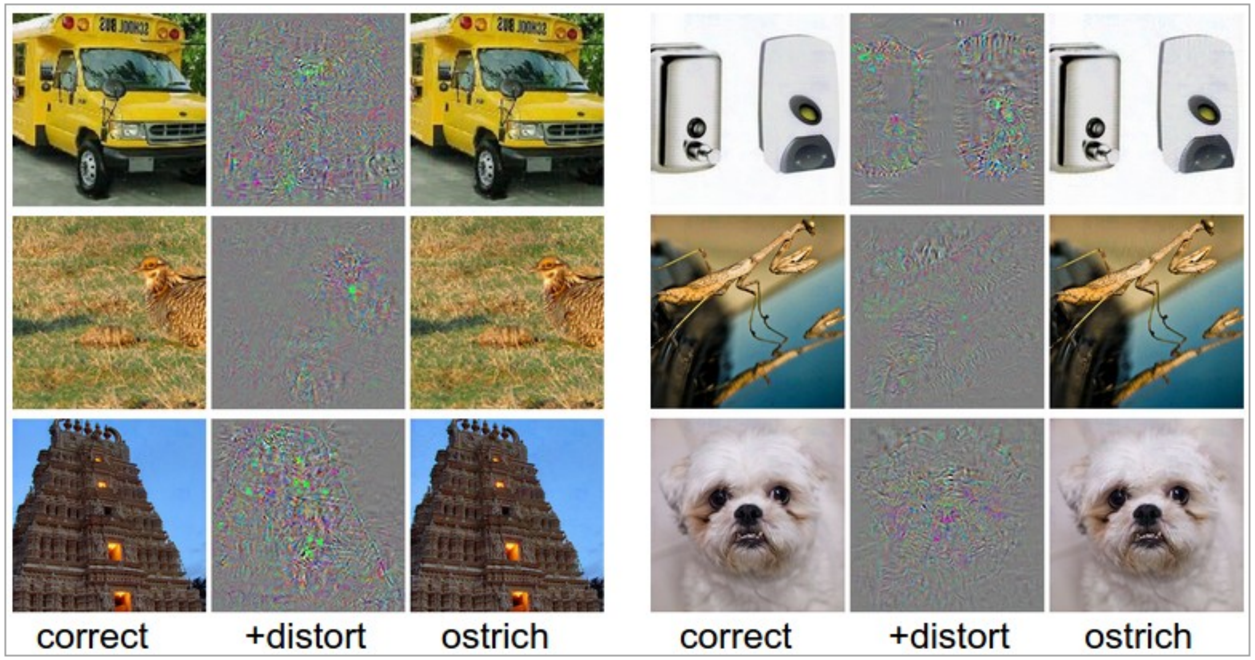


Yamins and DiCarlo, PNAS 2014



Yamins and DiCarlo, PNAS 2014

# Adversarial examples



Take a correctly classified image (left image in both columns), and add a tiny distortion (middle) to fool the ConvNet with the resulting image (right).

In short, to create a fooling image we start from whatever image we want (an actual image, or even a noise pattern), and then use backpropagation to compute the gradient of the image pixels on any class score, and nudge it along.

## Recurrent networks (RNN)

- Hopfield network
- Boltzmann machine
- Networks for sequence modeling

## 27.1 HOPFIELD NETWORKS

The state of a Hopfield network with $N$ cells is specified by $\mathbf{s} \in \mathbb{R}^N$ where each $s_i \in \{-1,1\}$. These two values could represent, e.g., high and low activity states of the corresponding neurons. We advance, from time $j$ to time $j+1$, for $j=1,2,\ldots$, by thresholding a linear combination of state elements. In particular, state $\mathbf{s}^j$ is advanced to

$$\mathbf{s}^{j+1} = \text{Hop}(\mathbf{W}\mathbf{s}^j) \quad \text{where} \quad \text{Hop}(x) \equiv \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \le 0 \end{cases} \tag{27.1}$$

is applied to each component of $\mathbf{W}\mathbf{s}^j$ in the Hopfield net. Here $\mathbf{W} \in \mathbb{R}^{N\times N}$ is the synaptic weight matrix. This net can be trained to remember an input pattern $\mathbf{p} \in \{-1,1\}^N$ by setting the weights to $\mathbf{W} = \mathbf{p}\mathbf{p}^T$. In this case, proceeding from an arbitrary state $\mathbf{s}$, we find

$$\mathbf{W}\mathbf{s} = \mathbf{p}\mathbf{p}^T\mathbf{s} = \mathbf{p}(\mathbf{p}^T\mathbf{s}) = (\mathbf{p}^T\mathbf{s})\mathbf{p}$$

and so

$$\text{Hop}(\mathbf{W}\mathbf{s}) = \begin{cases} \mathbf{p} & \text{if } \mathbf{p}^T\mathbf{s} > 0 \\ -\mathbf{e} & \text{if } \mathbf{p}^T\mathbf{s} = 0, \\ -\mathbf{p} & \text{if } \mathbf{p}^T\mathbf{s} < 0. \end{cases} \quad \text{where} \quad \mathbf{e} \equiv \texttt{ones(N,1)}.$$

In particular, both $\mathbf{p}$ and $-\mathbf{p}$ are *fixed points* of the associated Hopfield net in the sense that

$$\mathrm{Hop}(\mathbf{W}\mathbf{p}) = \mathbf{p} \quad \text{and} \quad \mathrm{Hop}(\mathbf{W}(-\mathbf{p})) = -\mathbf{p}.$$

Furthermore, these are the only fixed points unless $\mathbf{p}$ is balanced in the sense that $\mathbf{p}^T\mathbf{e} = 0$, in which case, $-\mathbf{e}$ is the only other fixed point. These fixed points are *attractors* in the sense that the Hopfield trajectory, Eq. (27.1), will terminate (rapidly) in one of these fixed points regardless of the initial state.

All of this generalizes nicely to multiple training patterns. In fact, if $\mathbf{p}_1$ and $\mathbf{p}_2$ are two such patterns, we set $\mathbf{P} = (\mathbf{p}_1\ \mathbf{p}_2)$ and $\mathbf{W} = \mathbf{P}\mathbf{P}^T$. Arguing as above, we find

$$\mathbf{W}\mathbf{s} = \mathbf{P}\mathbf{P}^T\mathbf{s} = (\mathbf{s}^T\mathbf{p}_1)\mathbf{p}_1 + (\mathbf{s}^T\mathbf{p}_2)\mathbf{p}_2.$$

Evaluating Hop of this is now a much more interesting affair. If $\mathbf{p}_1$ and $\mathbf{p}_2$ are orthogonal, i.e., $\mathbf{p}_1^T\mathbf{p}_2 = 0$, then it is not hard to see that both $\pm\mathbf{p}_1$ and $\pm\mathbf{p}_2$ will be fixed points. In the nonorthogonal case the input patterns may combine to form phantom fixed points.

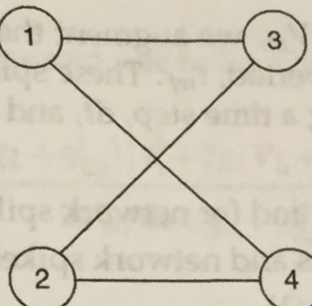
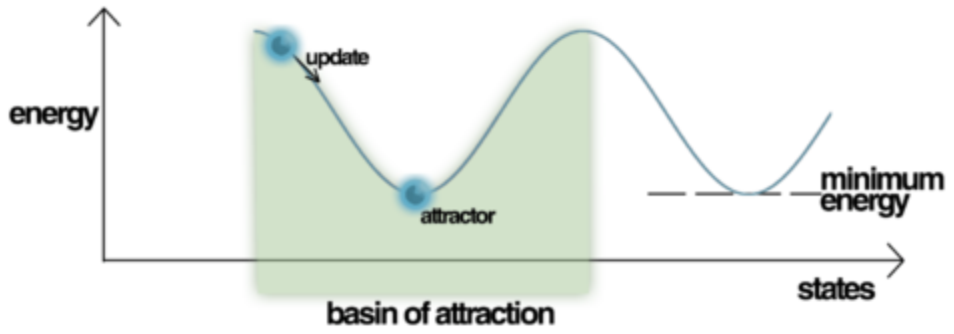


FIGURE 27.3 A four-cell network with bidirectional synapses between nodes 1 and 3, 1 and 4, 2 and 3, and 2 and 4.

We should note that fixed points are not the only possible attractors. Indeed, it is quite possible that the network may "oscillate" by periodically bouncing between several states. As a concrete example we consider the network of Figure 27.3. If we assume reciprocal unit weights along each of the edges in Figure 27.3 then we arrive at the symmetric weight matrix

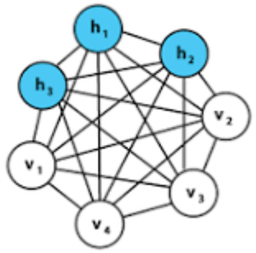
$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}.$$

If initially we excite cells 1 and 2 then $\mathbf{s}^1 = (1\ 1\ -1\ -1)$. It then follows that $\mathbf{s}^2 = -\mathbf{s}^1$ and $\mathbf{s}_3 = -\mathbf{s}_2 = \mathbf{s}_1$ and we say that the network has an attractor of period 2. We shall see in Exercise 2 that this example captures the general result, in the sense that no undirected Hopfield net may have an attractor with period greater than 2.



$$E = -\frac{1}{2}\sum_{ij} w_{ij}x_i x_j = -\frac{1}{2}\sum_{ij} x_i x_j x_i x_j = -\frac{1}{2}\sum_{ij} 1 = -\frac{1}{2}(N-1)^2 \qquad (6)$$

# Boltzmann machine



A Boltzmann machine, like a Hopfield network, is a network of units with an "energy" defined for the network. It also has binary units, but unlike Hopfield nets, Boltzmann machine units ar stochastic. The global energy, $E$, in a Boltzmann machine is identical in form to that of a Hopfield network:

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right)$$

Where:

- $w_{ij}$ is the connection strength between unit $j$ and unit $i$.
- $s_i$ is the state, $s_i \in \{0, 1\}$, of unit $i$.
- $\theta_i$ is the bias of unit $i$ in the global energy function. ($-\theta_i$ is the activation threshold for the unit.)

Often the weights are represented in matrix form with a symmetric matrix $W$, with zeros along the diagonal.

The difference in the global energy that results from a single unit $i$ being 0 (off) versus 1 (on), written $\Delta E_i$, assuming a symmetric matrix of weights, is given by:

$$\Delta E_i = \sum_{j>i} w_{ij}\, s_j + \sum_{j<i} w_{ji}\, s_j + \theta_i$$

We can now finally solve for $p_{\text{i=on}}$, the probability that the $i$-th unit is on.

$$p_{\text{i=on}} = \frac{1}{1 + \exp(-\frac{\Delta E_i}{T})}$$

There are two phases to Boltzmann machine training, and we switch iteratively between them. One is the "positive" phase where the visible units' states are clamped to a particular bin (according to $P^+$). The other is the "negative" phase where the network is allowed to run freely, i.e. no units have their state determined by external data. Surprisingly enough, the gra given by the very simple equation (proved in Ackley et al.[3]):

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{R}[p_{ij}^{+} - p_{ij}^{-}]$$

where:

- $p_{ij}^{+}$ is the probability of units *i* and *j* both being on when the machine is at equilibrium on the positive phase.
- $p_{ij}^{-}$ is the probability of units *i* and *j* both being on when the machine is at equilibrium on the negative phase.
- $R$ denotes the learning rate

# RNNs for sequence modeling

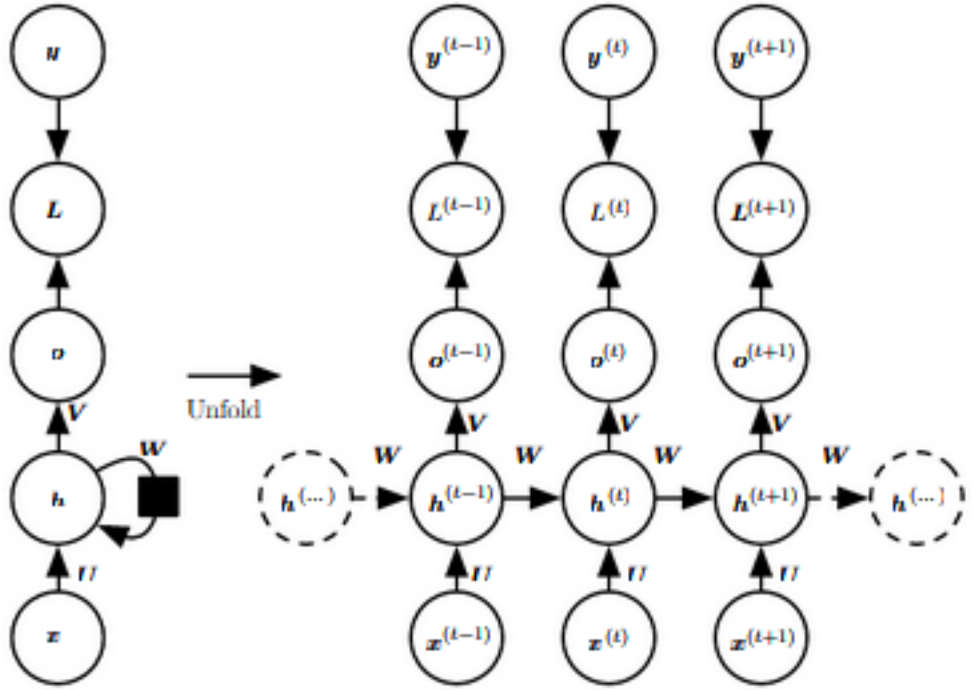


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss $L$ measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss $L$ internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)*The RNN and its loss drawn with recurrent connections. *(Right)*The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

# Autoencoder

- An autoencoder is a neural network that is trained to attempt to copy its input to its output.

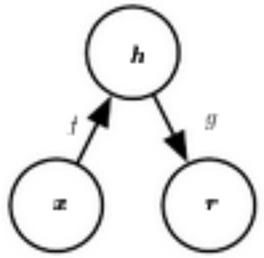


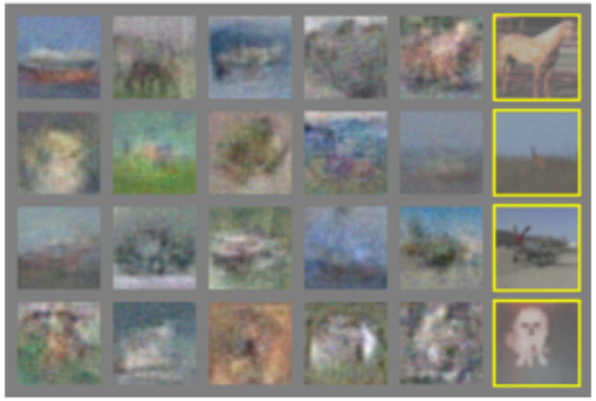
Figure 14.1: The general structure of an autoencoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The autoencoder has two components: the encoder $f$ (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder $g$ (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

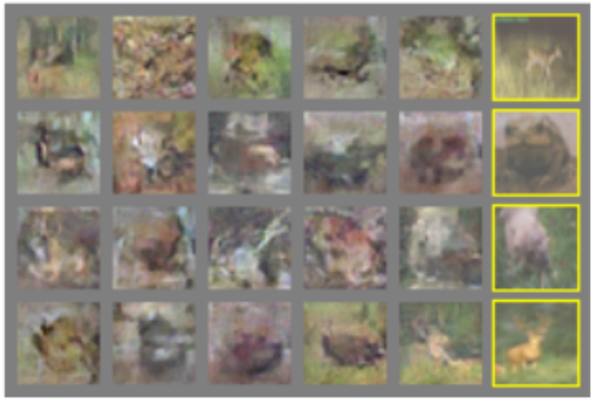
- Can learn representations that are sparse, denoised, low-dimensional etc.

# Generator Adversarial Networks (GANS)

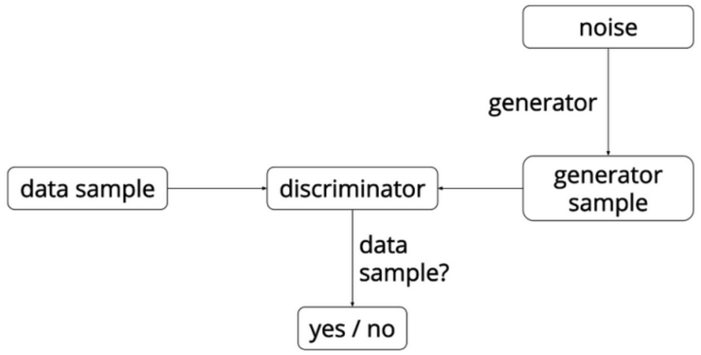
a)

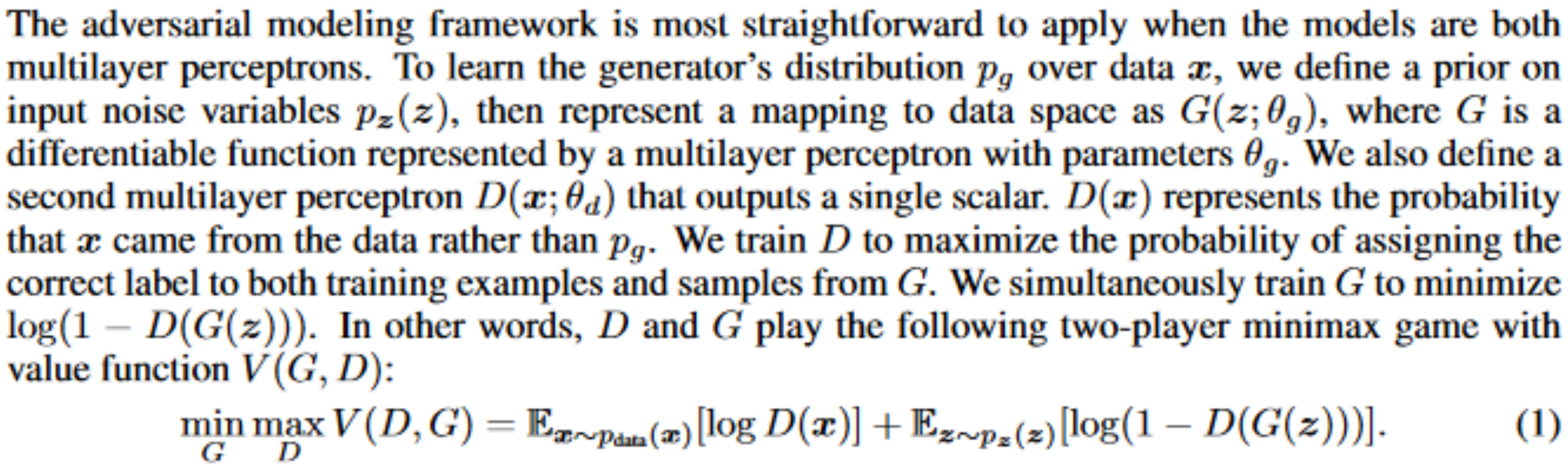
b)


c)


d)

# Generator Adversarial Networks (GANS)



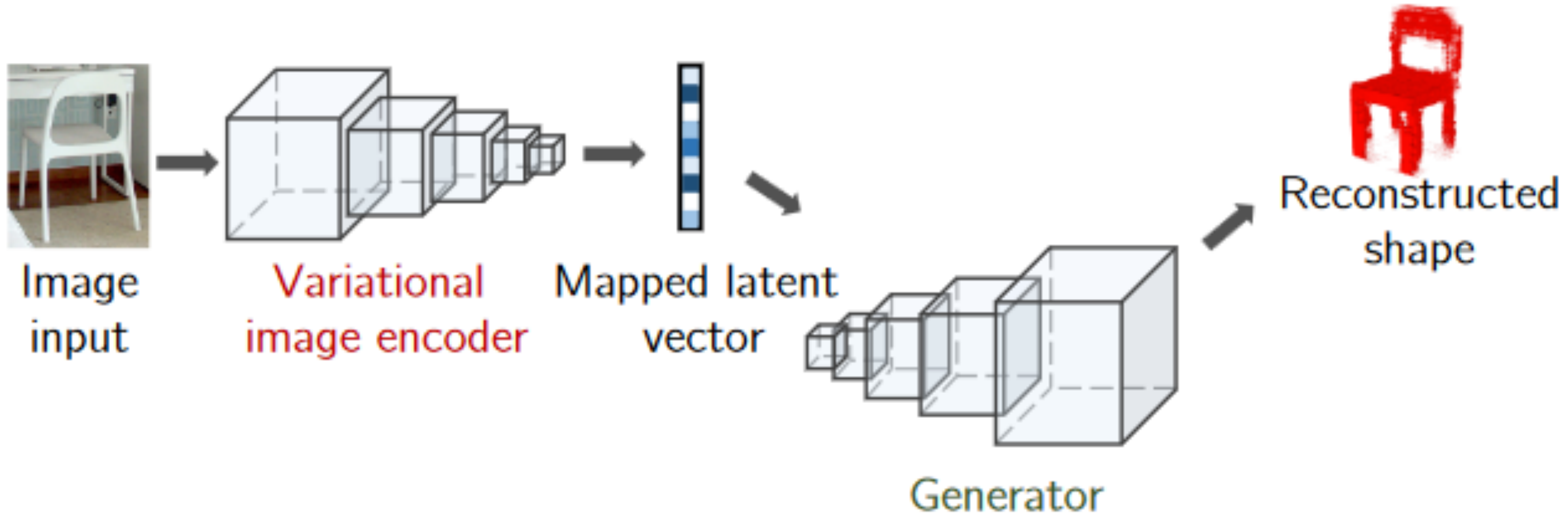


The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution $p_g$ over data $\boldsymbol{x}$, we define a prior on input noise variables $p_{\boldsymbol{z}}(\boldsymbol{z})$, then represent a mapping to data space as $G(\boldsymbol{z};\theta_g)$, where $G$ is a differentiable function represented by a multilayer perceptron with parameters $\theta_g$. We also define a second multilayer perceptron $D(\boldsymbol{x};\theta_d)$ that outputs a single scalar. $D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from the data rather than $p_g$. We train $D$ to maximize the probability of assigning the correct label to both training examples and samples from $G$. We simultaneously train $G$ to minimize $\log(1 - D(G(\boldsymbol{z})))$. In other words, $D$ and $G$ play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]. \tag{1}$$

# Generator Adversarial Networks (GANS)

**Model: 3D-VAE-GAN**



Wu et al. NIPS 2016

# Unsupervised Learning

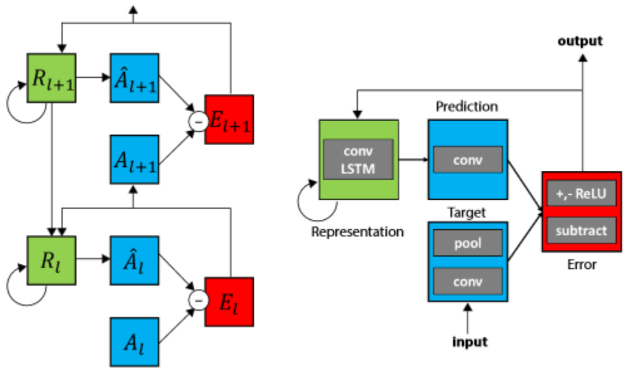
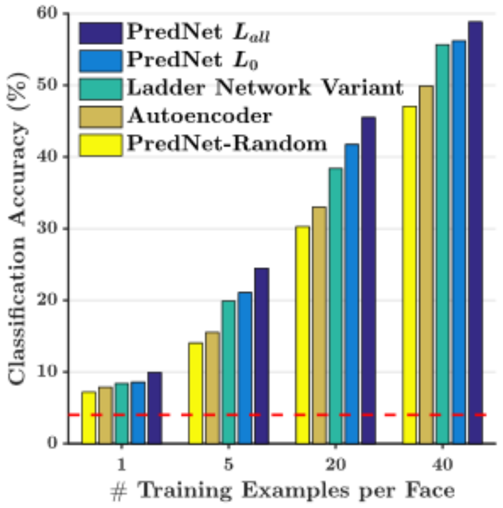
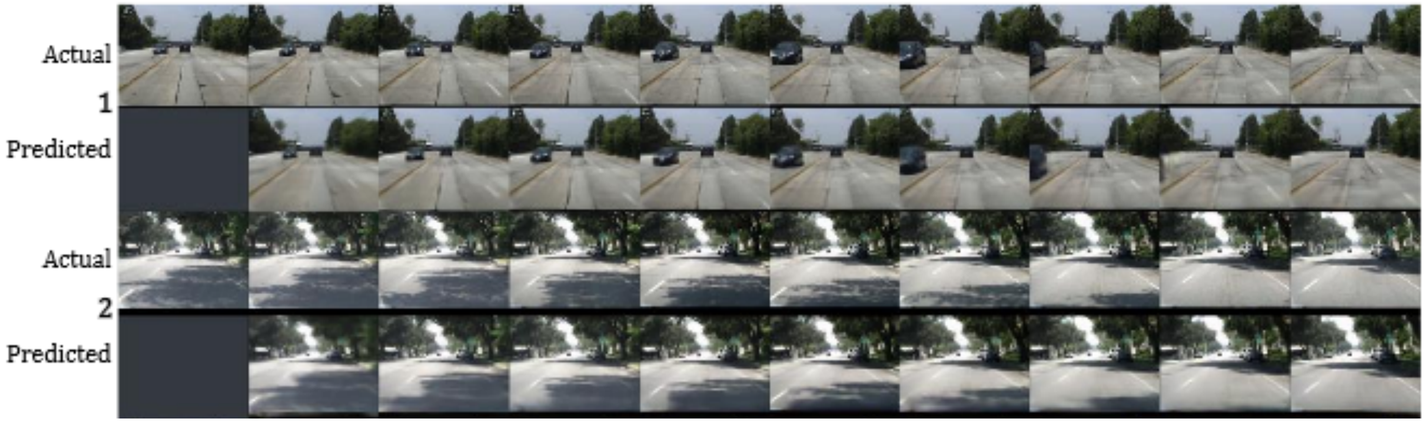
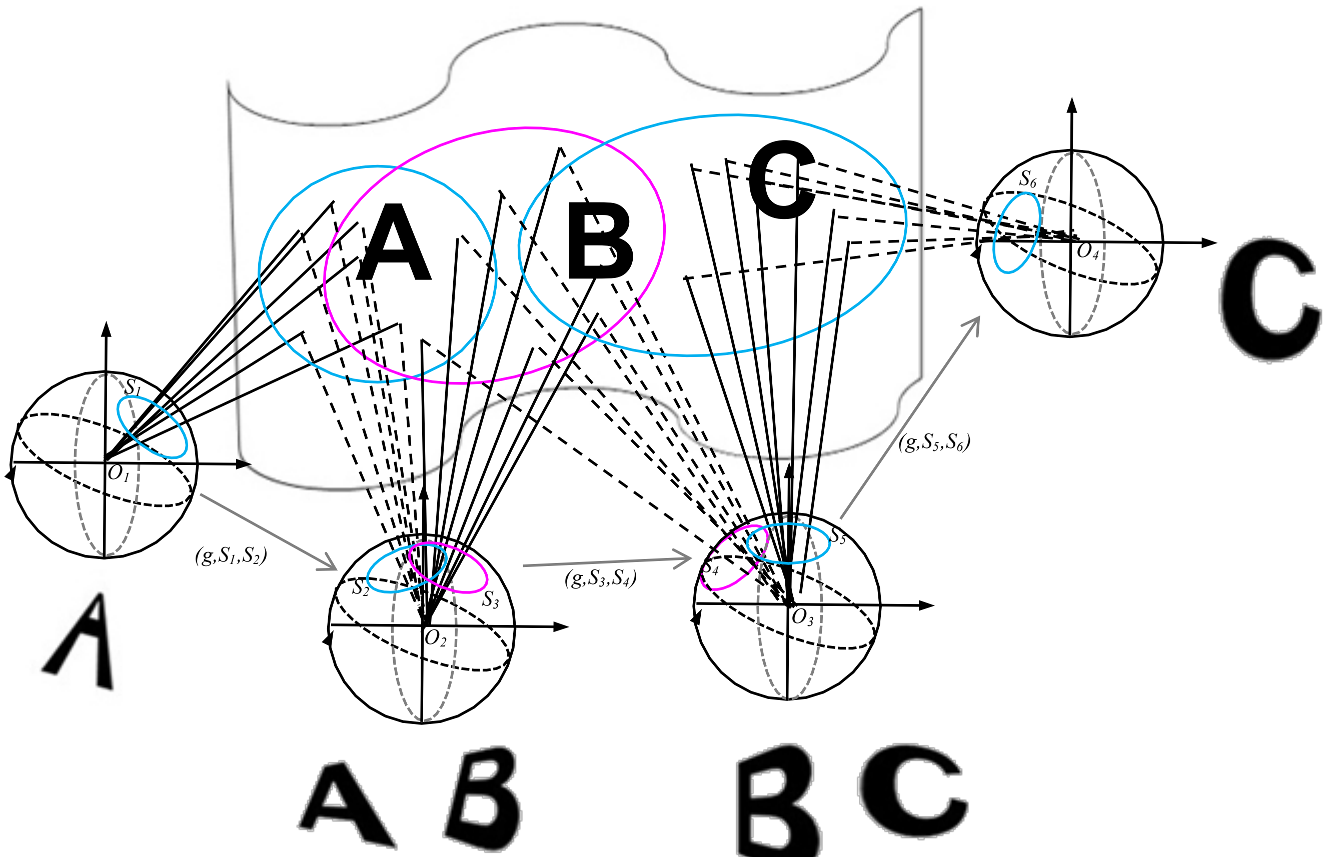


Figure 1: Predictive Coding Network (PredNet). Left: Illustration of information flow within two layers. Each layer consists of representation neurons ($R_l$), which output a layer-specific prediction at each time step ($\hat{A}_l$), which is compared against a target ($A_l$) (Bengio, 2014) to produce an error term ($E_l$), which is then propagated laterally and vertically in the network. Right: Module operations for case of video sequences.





Lotter et al, ICLR 2017


A
B
C
S1
O1
(g,S1,S2)
S2
S3
O2
(g,S3,S4)
S4
S5
O3
(g,S5,S6)
S6
O4
A
A B
B C
C

## What is the nature of the representations learned by deep networks?

*“An essential ingredient of ergo-learning strategy is a search for symmetry--repetitive patterns--in flows of signals. Even more signicantly, an ergo system creates/identies such patterns by reducing/compressing "information" and by structuralizing "redundancies" in these flows.” --Gromov*

A connection between category theory & machine learning?