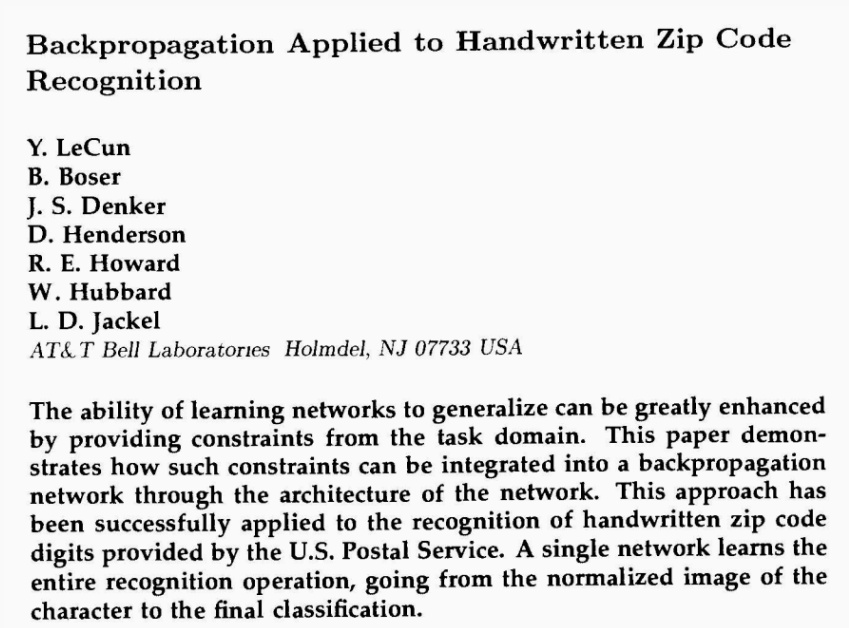
CS-GY 6923: Lecture 10
Back-propagation, Convolution + Feature Extraction

NYU Tandon School of Engineering, Prof. Christopher Musco

EARLY NEURAL NETWORK EXPLOSION

Around 1985 several groups (re)-discovered the **backpropagation algorithm** which allows for efficient training of neural nets via **(stochastic) gradient descent**. Along with increased computational power this lead to a resurgence of interest in neural network models.



Backpropagation Applied to Handwritten Zip Code Recognition

Y. LeCun
B. Boser
J. S. Denker
D. Henderson
R. E. Howard
W. Hubbard
L. D. Jackel
AT&T Bell Laboratories Holmdel, NJ 07733 USA

The ability of learning networks to generalize can be greatly enhanced by providing constraints from the task domain. This paper demonstrates how such constraints can be integrated into a backpropagation network through the architecture of the network. This approach has been successfully applied to the recognition of handwritten zip code digits provided by the U.S. Postal Service. A single network learns the entire recognition operation, going from the normalized image of the character to the final classification.

Very good performance on problems like digit recognition.

NEURAL NETWORK DECLINE

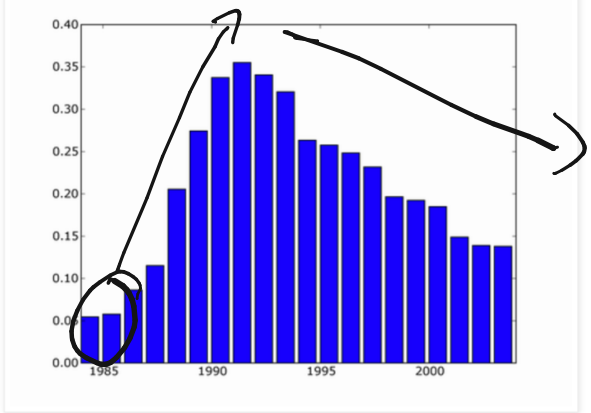
From 1990s - 2010, kernel methods, SVMs, and probabilistic methods began to dominate the literature in machine learning:

- Work well “out of the box”.
- Relatively easy to understand theoretically.
- Not too computationally expensive for moderately sized datasets.

Fun blog post to check out from 2005:
`http://yaroslavvb.blogspot.com/2005/12/trends-in-machine-learning-according.html`

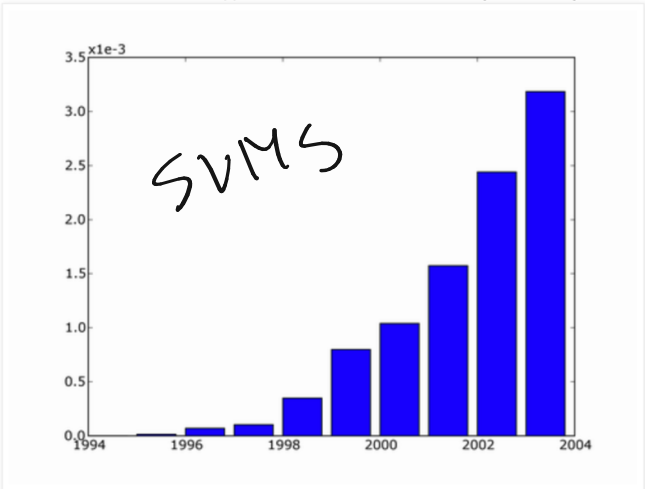
NEURAL NETWORK DECLINE

Finding trends in machine learning by search papers in Google Scholar that match a certain keyword:



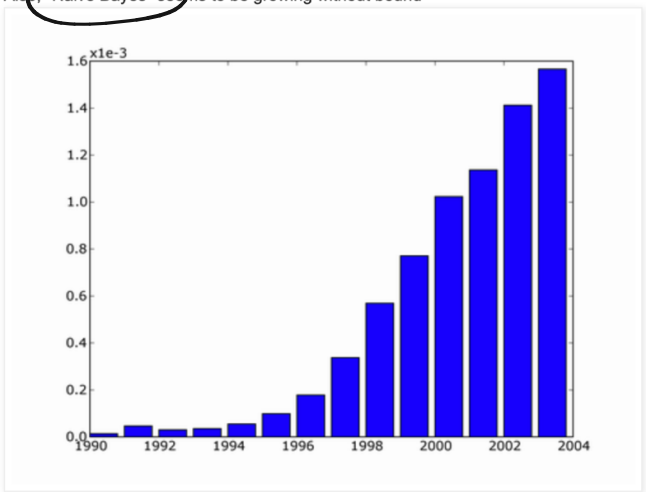
You can see a major upward trend starting around 1985 (that's when Yann LeCun and several others independently rediscovered backpropagation algorithm), peaking in 1992, and going downwards from then.

On other hand, search for "support vector machine" shows no sign of slowing down



(1995 is when Vapnik and Cortez proposed the algorithm)

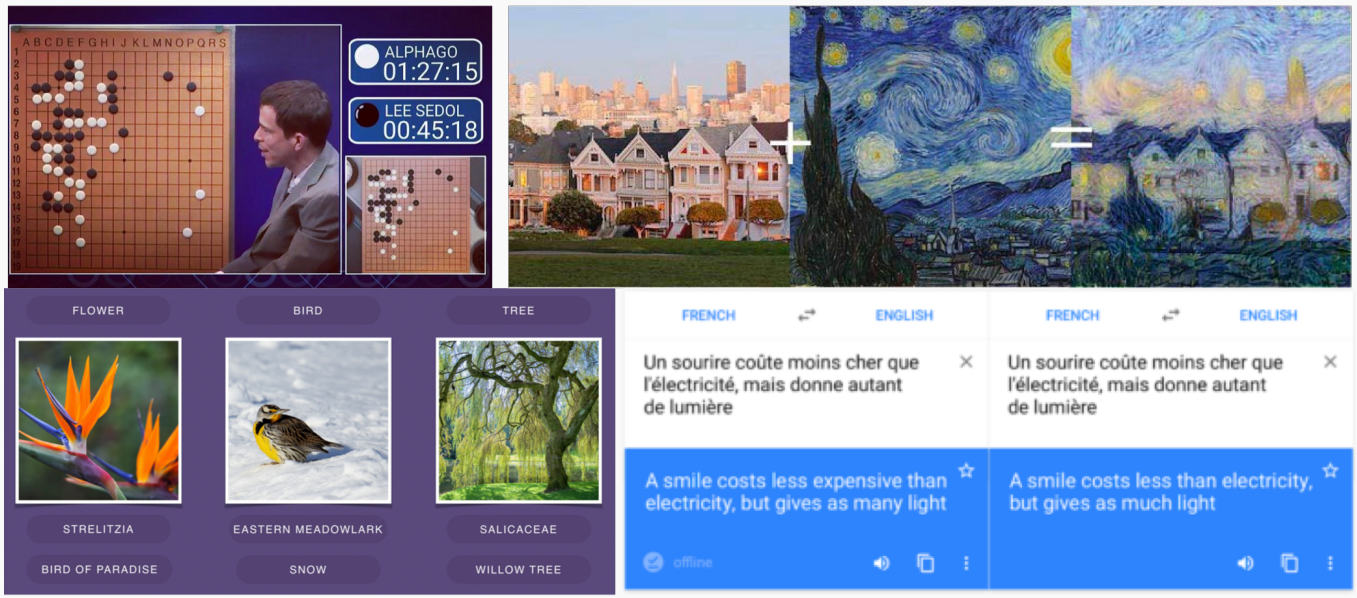
Also, "Naive Bayes" seems to be growing without bound



If I were to trust this, I would say that Naive Bayes research the hottest machine learning area right now

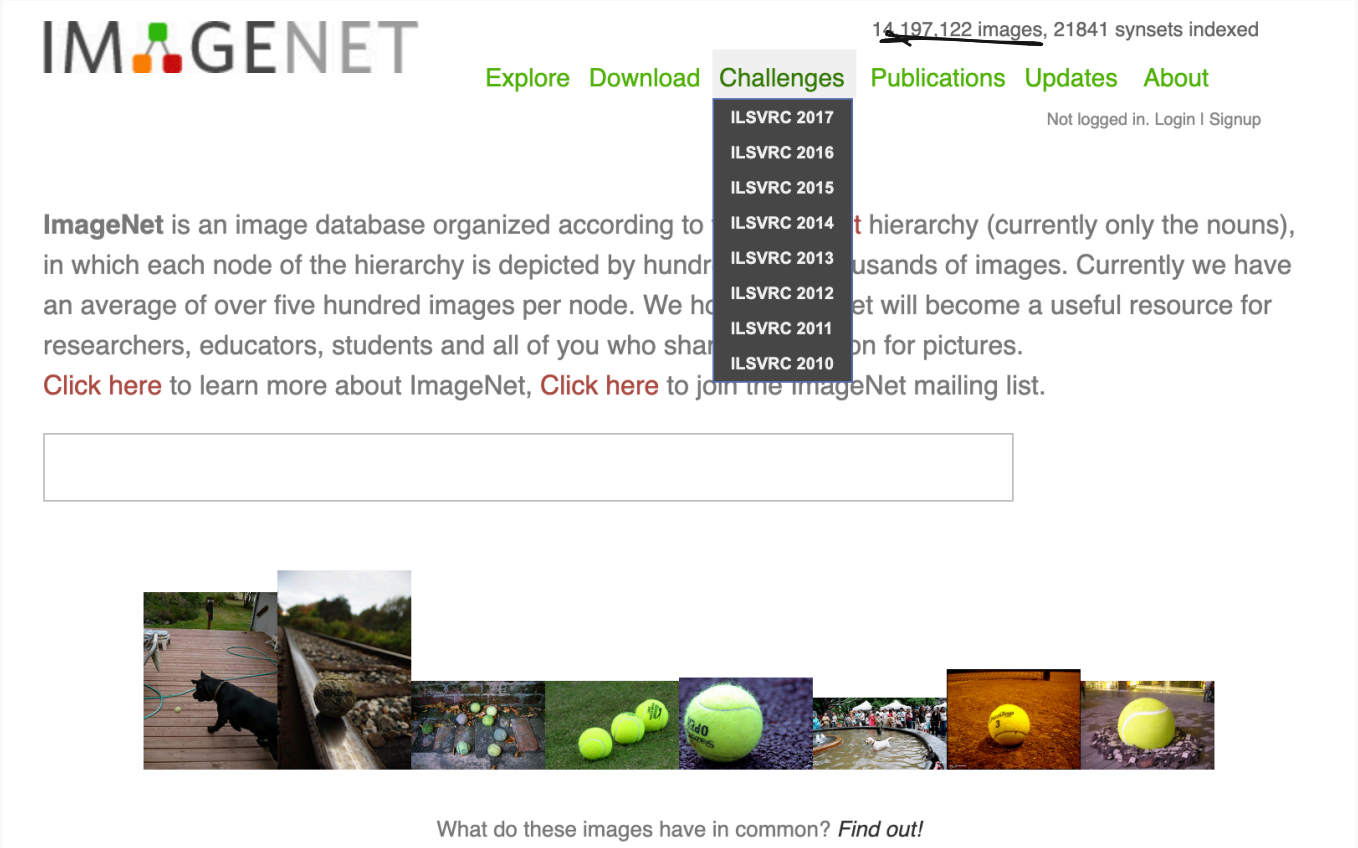
MODERN NEURAL NETWORK RESURGENCE

In recent years this trend completely turned around:



Recent state-of-the-art results in game playing, image recognition, content generation, natural language processing, machine translation, many other areas.

All changed with the introduction of AlexNet and the 2012 ImageNet Challenge...



Very general image classification task.

All changed with AlexNet and the 2012 ImageNet Challenge...

team name	team members	filename	flat cost	hie cost	description
NEC-UIUC	NEC: Yuanqing Lin, Fengjun Lv, Shenghuo Zhu, Ming Yang, Timothee Cour, Kai Yu UIUC: LiangLiang Cao, Zhen Li, Min-Hsuan Tsai, Xi Zhou, Thomas Huang Rutgers: Tong Zhang	flat_opt.txt	0.28191	2.1144	using sift and lbp feature with two non-linear coding representations and stochastic SVM, optimized for top-5 hit rate

2010 Results

Team name	Filename	Error (5 guesses)	Description
SuperVision	test-preds-141-146.2009-131-137-145-146.2011-145f.	0.15315	Using extra training data from ImageNet Fall 2011 release
SuperVision	test-preds-131-137-145-135-145f.txt	0.16422	Using only supplied training data
ISI	pred_FVs_wLACs_weighted.txt	0.26172	Weighted sum of scores from each classifier with SIFT+FV, LBP+FV, GIST+FV, and CSIFT+FV, respectively.

2012 Results

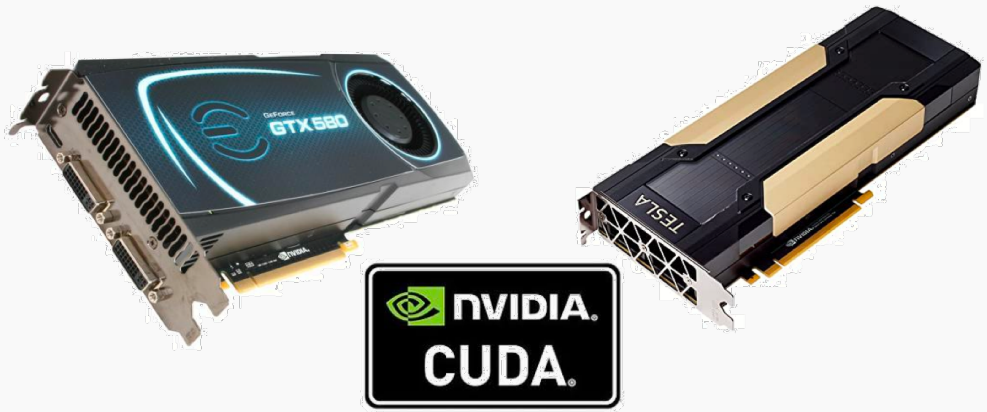
Why 2012?

- Clever ideas in changing neural network architectures. E.g. convolutional units baked into the neural net.
- Wide-spread access to GPU computing power).

GRAPHICS PROCESSING UNIT

Hardware innovation: Widely available, inexpensive GPUs allowing for cheap, highly parallel linear algebra operations.

- 2007: Nvidia released CUDA platform, which allows GPUs to be easily programmed for general purposed computation.



AlexNet architecture used 60 million parameters. Could not have been trained using CPUs alone (except maybe on a government super computer).

Two main algorithmic tools for training neural network models:

1. Stochastic gradient descent.
2. Backpropogation.

For a function $f(x)$ we write the derivative with respect to x as:

$$f'(x) = \frac{df}{dx} = \lim_{t \to 0} \frac{f(x+t) - f(x)}{t}.$$

For a function $f(x, y, z)$ we write the partial derivative with respect to x as:

$$\frac{\partial f}{\partial x} = \lim_{t \to 0} \frac{f(x+t, y, z) - f(x, y, z)}{t}.$$

Let $y(x)$ be a function of x and let $f(y)$ be a function of y. The chain rule says that:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x}$$

$$\frac{\partial f}{\partial x} = \lim_{t \to 0} \frac{f(y(x+t)) - f(y(x))}{t}$$

$$= \lim_{t \to 0} \left(\frac{f(y(x+t)) - f(y(x))}{y(x+t) - y(x)} \right) \cdot \left(\frac{y(x+t) - y(x)}{t} \right)$$

As long at $\lim_{t \to 0} y(x+t) - y(x) = 0$ then the first term equals $\frac{\partial f}{\partial y}$ and the second equals $\frac{\partial y}{\partial x}$.

Let $y(x), z(x), w(x)$ be functions of x and let $f(y, z, w)$ be a function of y,z,w.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x} + \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x} + \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial x}$$

Example: Let $y(x) = x^3$ and $z(x) = x^2$. Let $f(y, z) = y \cdot z$. Then:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x} + \left(\frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x}\right) \\
&= z \cdot (3x^2) + y \cdot 2x \\
&= x^2 \cdot (3x^2) + x^3 \cdot 2x \\
&= 5x^4.
\end{aligned}$$

$f(x) = x^3 \cdot x^2$

$= x^5$

$\frac{df}{dx} = 5x^4$

Let $f(\boldsymbol{\theta}, \mathbf{x})$ be our neural network. A typical ℓ-layer feed forward model has the form:

$$g_\ell\left(\mathbf{W}_\ell\left(\ldots \mathbf{W}_3 \cdot g_2\left(\mathbf{W}_2 \cdot g_1\left(\mathbf{W}_1\mathbf{x} + \boldsymbol{\beta}_1\right) + \boldsymbol{\beta}_2\right) + \boldsymbol{\beta}_3 \ldots\right) + \boldsymbol{\beta}_\ell\right).$$

$\mathbf{W}_i$ and $\boldsymbol{\beta}_i$ are the weight matrix and bias vector for layer i and g_i is the non-linearity (e.g. sigmoid). $\boldsymbol{\theta} = [\mathbf{W}_0, \boldsymbol{\beta}_0, \ldots, \mathbf{W}_\ell, \boldsymbol{\beta}_\ell]$ is a vector of all entries in these matrices.

Goal: Given training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ minimize the loss

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right)$$

Example: We might use the binary cross-entropy loss for binary classification. f is the output class probability.

$$L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right) = y_i \log\left(f(\boldsymbol{\theta}, \mathbf{x}_i)\right) + (1 - y_i) \log(1 - f(\boldsymbol{\theta}, \mathbf{x}_i))$$

GRADIENT OF THE LOSS

Most common approach: minimize the loss by using gradient descent. Which requires us to compute the gradient of the loss function, $\nabla \mathcal{L}$. Note that this gradient has an entry for every value in $[\mathbf{W}_0, \beta_0, \ldots, \mathbf{W}_\ell, \beta_\ell]$.

As usual, our loss function has finite sum structure, so:

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla L(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i))$$

So we can focus on computing:

$$\nabla_{\boldsymbol{\theta}} L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right)$$

for a single training example $(\mathbf{x}_i, y_i)$.

GRADIENT OF THE LOSS

Applying chain rule to loss:

$$\nabla_{\boldsymbol{\theta}} L\left(y, f(\boldsymbol{\theta}, \mathbf{x})\right) = \frac{\partial L}{\partial f(\boldsymbol{\theta}, \mathbf{x})} \cdot \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{x})$$

number

vector

Binary cross-entropy example:

$$L\left(y, f(\boldsymbol{\theta}, \mathbf{x})\right) = y \log(f(\boldsymbol{\theta}, \mathbf{x})) + (1 - y) \log(1 - f(\boldsymbol{\theta}, \mathbf{x}))$$

$$\frac{y}{f(\theta, x)} + (1 - y) \cdot \frac{1}{1 - f(\theta, x)} \cdot (-1)$$

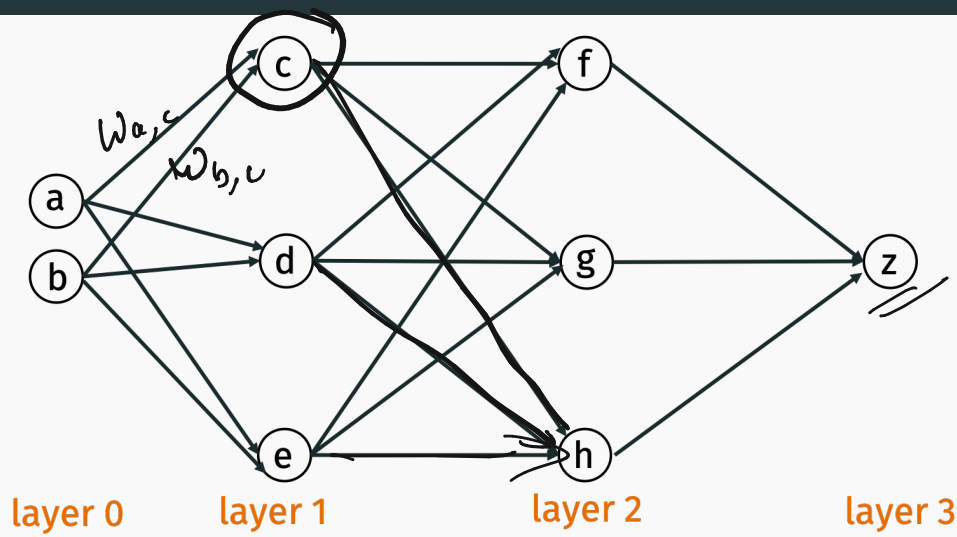
$$\begin{bmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \theta_1} \\ \\ \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \theta_m} \end{bmatrix}$$

GRADIENT OF THE LOSS

We have reduced our goal to computing $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{x})$, where the gradient is with respect to the parameters $\boldsymbol{\theta}$.

Back-propagation is a natural and efficient way to compute $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{x})$. It derives its name because we compute gradient from back to front: starting with the parameters closest to the output of the neural net.

BACKPROP EXAMPLE



Notation for next few slides:

- $a, b, \ldots, z$ are the node names, and used to denote values at nodes after applying non-linearity.
- $\bar{a}, \bar{b}, \ldots, \bar{z}$ denote value before applying non-linearity.
- $W_{i,j}$ is the weight of edge from node i to node j.
- $s(\cdot) : \mathbb{R} \to \mathbb{R}$ is the non-linear activation function.
- β_j is the bias for node j.

Example: $h = s(\bar{h}) = s(c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + \beta_h)$

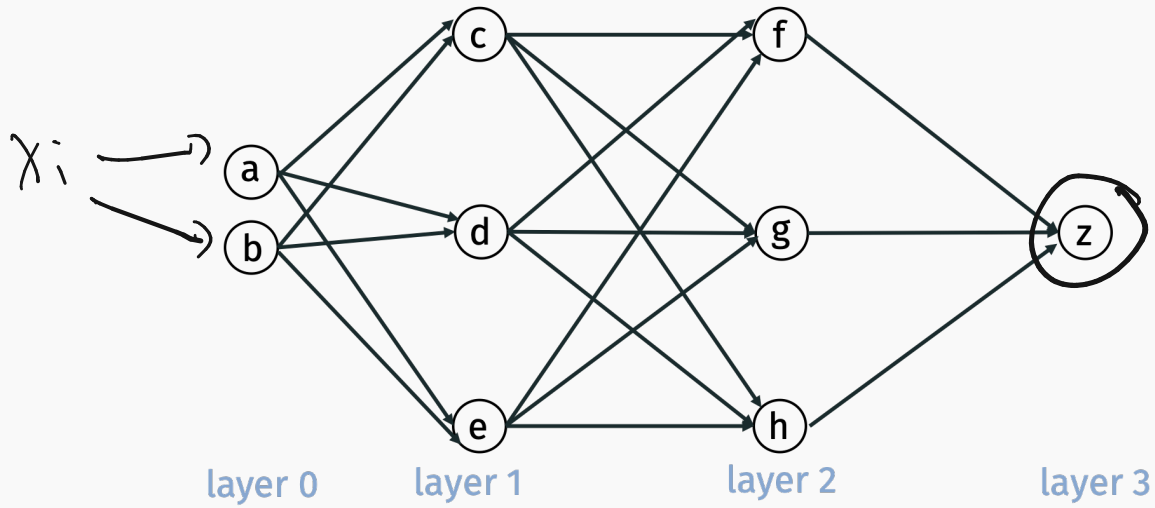
BACKPROP EXAMPLE

Goal: Compute the gradient $\nabla f(\boldsymbol{\theta}, \mathbf{x})$, which contains the partial derivatives with respect to every parameter:

- $\partial z / \partial \beta_z$
- $\partial z / \partial W_{f,z}$, $\partial z / \partial W_{g,z}$, $\partial z / \partial W_{h,z}$
- $\partial z / \partial W_{c,f}$, $\partial z / \partial W_{c,g}$, $\partial z / \partial W_{c,h}$
- $\partial z / \partial W_{d,f}$, $\partial z / \partial W_{d,g}$, $\partial z / \partial W_{d,h}$
- $\vdots$
- $\partial z / \partial W_{a,c}$, $\partial z / \partial W_{a,d}$, $\partial z / \partial W_{a,e}$

Two steps: Forward pass to compute function value.
Backwards pass to compute gradients.

Step 1: Forward pass.

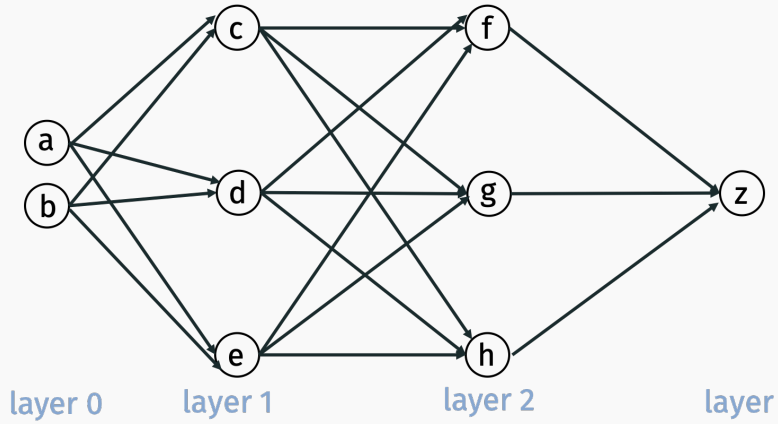


- Using **current parameters**, compute the output z by moving from left to right.
- Store all intermediate results:

$$\bar{c}, \bar{d}, \bar{e}, c, d, e, \bar{f}, \bar{g}, \bar{h}, f, g, h, \bar{z}, z.$$

BACKPROP EXAMPLE

Step 1: Forward pass.



N nodes

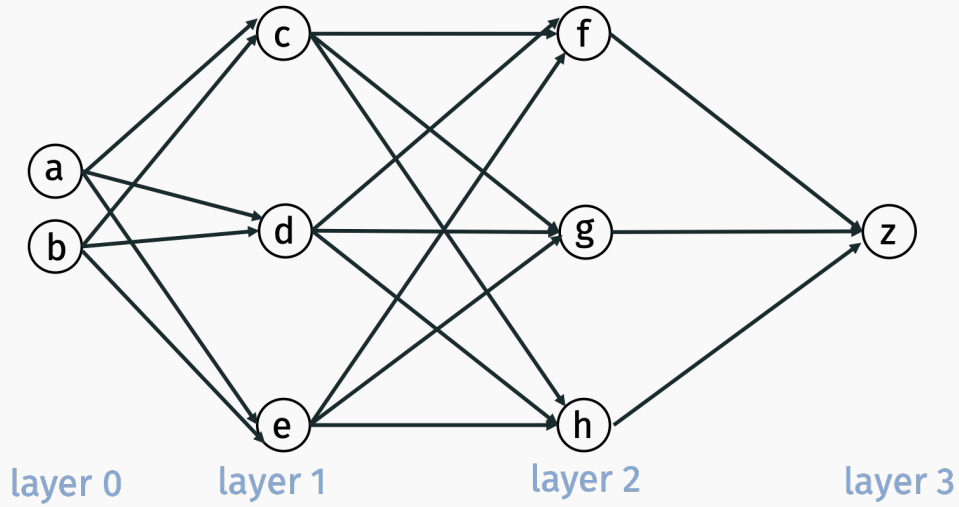
W weights

$O(W + N) = O(W)$

$$\bar{c} = W_{a,c} \cdot a + W_{b,c} \cdot b + \beta_c \qquad c = s(\bar{c})$$
$$\bar{d} = W_{a,d} \cdot a + W_{b,d} \cdot b + \beta_d \qquad d = s(\bar{d})$$
$$\bar{e} = W_{a,e} \cdot a + W_{b,e} \cdot b + \beta_e \qquad e = s(\bar{e})$$
$$\bar{f} = W_{c,f} \cdot c + W_{d,f} \cdot d + W_{e,f} \cdot e + \beta_f \qquad f = s(\bar{f})$$
$$\vdots \qquad\qquad \vdots$$
$$\bar{z} = W_{f,z} \cdot f + W_{g,z} \cdot g + W_{h,z} \cdot h + \beta_z \qquad z = s(\bar{z})$$

→ N

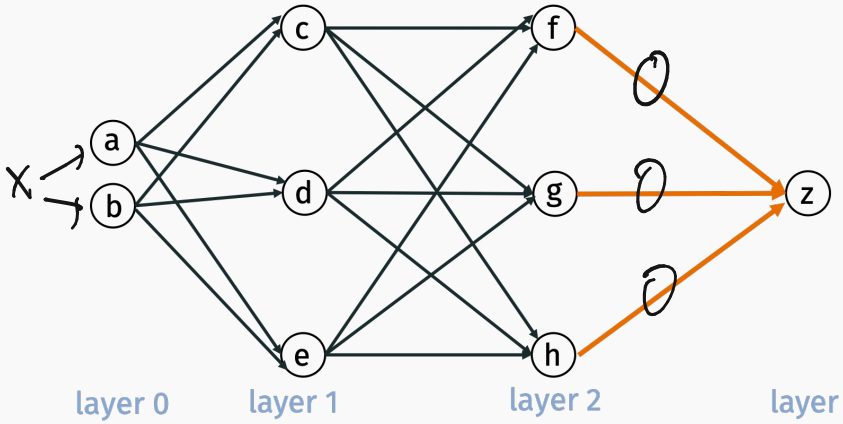
Step 2: Backward pass.



- Using **current parameters** and **computed node values**, compute the partial derivatives of all parameters by moving from right to left.

BACKPROP EXAMPLE

Step 2: Backward pass. Deepest layer.



$$z = f(\theta, x)$$

$$z = s\left(w_{fz} \cdot f + w_{gz} \cdot g + w_{hz} \cdot h + b_z\right) = s(\bar{z})$$

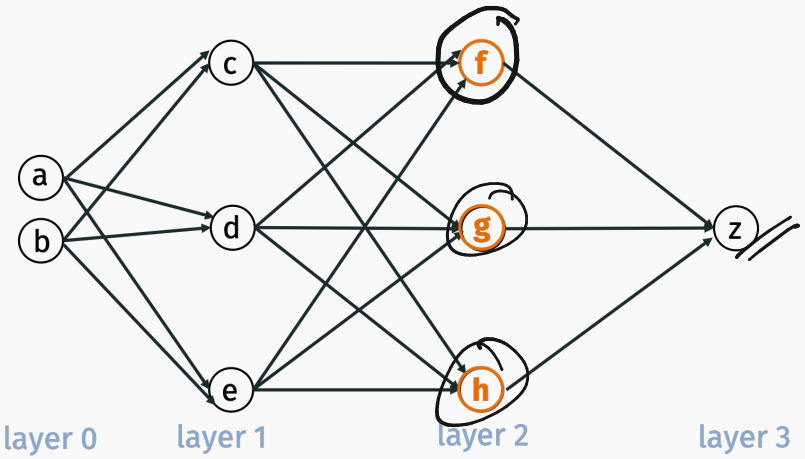
$$\frac{\partial z}{\partial b_z} = \frac{\partial \bar{z}}{\partial b_z} \cdot \frac{\partial z}{\partial \bar{z}} = 1 \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{f,z}} = \frac{\partial \bar{z}}{\partial W_{f,z}} \cdot \frac{\partial z}{\partial \bar{z}} = f \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{g,z}} = \frac{\partial \bar{z}}{\partial W_{g,z}} \cdot \frac{\partial z}{\partial \bar{z}} = g \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{h,z}} = \frac{\partial \bar{z}}{\partial W_{h,z}} \cdot \frac{\partial z}{\partial \bar{z}} = h \cdot s'(\bar{z})$$

Step 2: Backward pass.



$$z = s\left(w_{fz} \cdot f + w_{gz} \cdot g + w_{hz} \cdot h + b_z\right), \quad \bar{z}$$

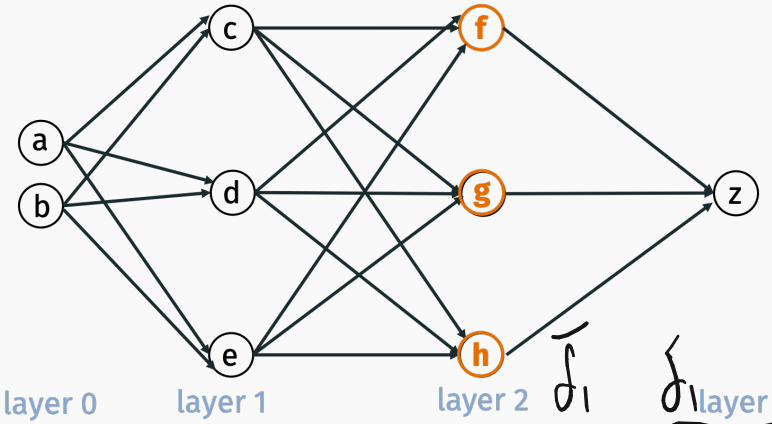
$$\frac{\partial z}{\partial f} = \frac{\partial \bar{z}}{\partial f} \cdot \frac{\partial z}{\partial \bar{z}} = W_{f,z} \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial g} = \frac{\partial \bar{z}}{\partial g} \cdot \frac{\partial z}{\partial \bar{z}} = W_{g,z} \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial h} = \frac{\partial \bar{z}}{\partial h} \cdot \frac{\partial z}{\partial \bar{z}} = W_{h,z} \cdot s'(\bar{z})$$

Compute partials with respect to nodes, even though not needed for gradient.

Step 2: Backward pass.



$$f = s(\bar{f})$$

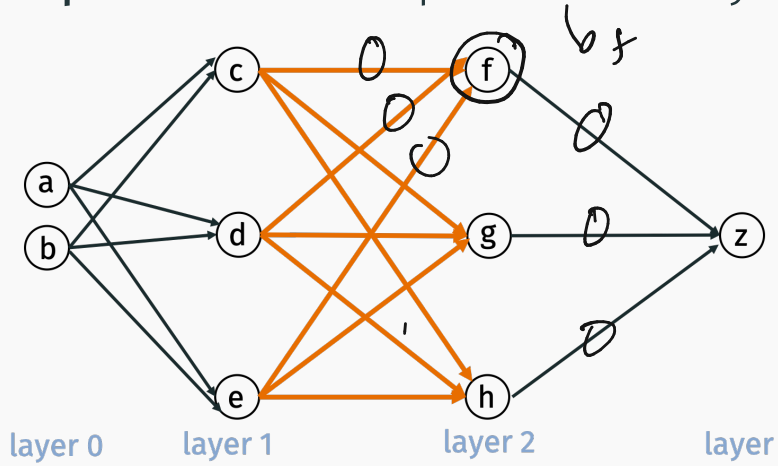
$$s'(\bar{f})$$

$$\bar{\delta}_1 \quad \delta_1 \rightarrow s'(\bar{v}_1)$$

$$\begin{bmatrix} \frac{\partial z}{\partial \bar{f}} \\ \frac{\partial z}{\partial \bar{g}} \\ \frac{\partial z}{\partial \bar{h}} \end{bmatrix} = \begin{bmatrix} \frac{\partial z}{\partial f} \\ \frac{\partial z}{\partial g} \\ \frac{\partial z}{\partial h} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial f}{\partial \bar{f}} \\ \frac{\partial g}{\partial \bar{g}} \\ \frac{\partial h}{\partial \bar{h}} \end{bmatrix} = \begin{bmatrix} \frac{\partial z}{\partial f} \cdot s'(\bar{f}) \\ \frac{\partial z}{\partial g} \cdot s'(\bar{g}) \\ \frac{\partial z}{\partial h} \cdot s'(\bar{h}) \end{bmatrix}$$

And for nodes pre-nonlinearity

Step 2: Backward pass. Next layer.



$$\bar{f} = w_{cf} \cdot c + w_{df} \cdot d + w_{ef} \cdot e + b_f$$

2 w

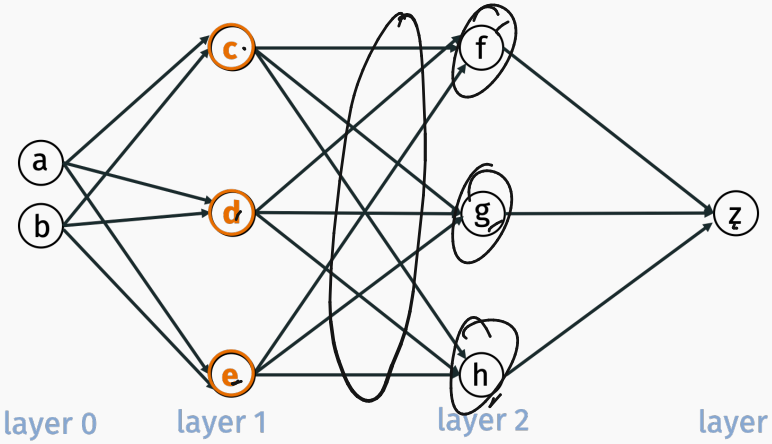
$$\frac{\partial z}{\partial b_f} = \frac{\partial z}{\partial \bar{f}} \frac{\partial \bar{f}}{\partial b_f} = \frac{\partial z}{\partial \bar{f}} \cdot 1$$

$$\frac{\partial z}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot c$$

$$\frac{\partial z}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot d$$

$$\frac{\partial z}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot e$$

Step 2: Backward pass. Next set of nodes.



$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial c} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial c} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial c}$$

$$= \frac{\partial z}{\partial \bar{f}} \cdot W_{c,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{c,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{c,h}$$

$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial \bar{f}} \cdot W_{d,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{d,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{d,h}$$

$$\frac{\partial z}{\partial e} = \frac{\partial z}{\partial \bar{f}} \cdot W_{e,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{e,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{e,h}$$

Linear algebraic view.

Let $\mathbf{v}_i$ be a vector containing the value of all nodes j in layer i.

$$\mathbf{v}_3 = \begin{bmatrix} z \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \qquad \mathbf{v}_1 = \begin{bmatrix} c \\ d \\ e \end{bmatrix}$$

Let $\bar{\mathbf{v}}_i$ be a vector containing $\bar{j}$ for all nodes j in layer i.

$$\bar{\mathbf{v}}_3 = \begin{bmatrix} \bar{z} \end{bmatrix} \qquad \bar{\mathbf{v}}_2 = \begin{bmatrix} \bar{f} \\ \bar{g} \\ \bar{h} \end{bmatrix} \qquad \bar{\mathbf{v}}_1 = \begin{bmatrix} \bar{c} \\ \bar{d} \\ \bar{e} \end{bmatrix}$$

Note: $\mathbf{v}_i = s(\bar{\mathbf{v}}_i)$ where s is applied entrywise.

Linear algebraic view.

Let $\boldsymbol{\delta}_i$ be a vector containing $\partial z/\partial j$ for all nodes j in layer i.

$$\boldsymbol{\delta}_3 = \begin{bmatrix} 1 \end{bmatrix} \qquad \boldsymbol{\delta}_2 = \begin{bmatrix} \partial z/\partial f \\ \partial z/\partial g \\ \partial z/\partial h \end{bmatrix} \qquad \boldsymbol{\delta}_1 = \begin{bmatrix} \partial z/\partial c \\ \partial z/\partial d \\ \partial z/\partial e \end{bmatrix}$$

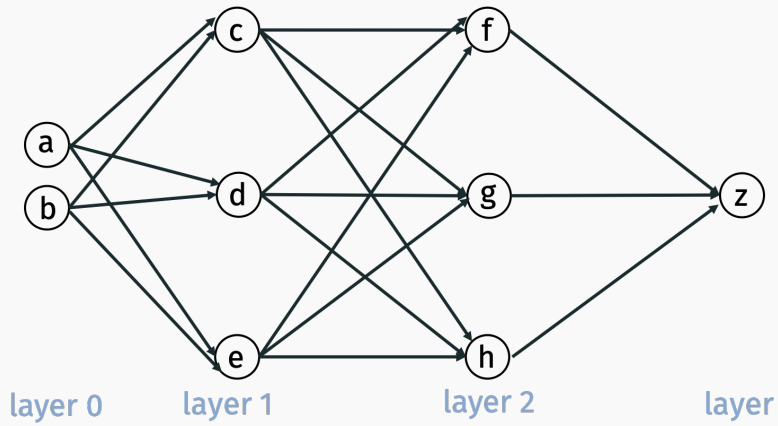
Let $\bar{\boldsymbol{\delta}}_i$ be a vector containing $\partial z/\partial \bar{j}$ for all nodes j in layer i.

$$\bar{\boldsymbol{\delta}}_3 = \begin{bmatrix} \partial z/\partial \bar{z} \end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_2 = \begin{bmatrix} \partial z/\partial \bar{f} \\ \partial z/\partial \bar{g} \\ \partial z/\partial \bar{h} \end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_1 = \begin{bmatrix} \partial z/\partial \bar{c} \\ \partial z/\partial \bar{d} \\ \partial z/\partial \bar{e} \end{bmatrix}$$

Note: $\bar{\boldsymbol{\delta}}_i = s'(\bar{\mathbf{v}}_i) \times \boldsymbol{\delta}_i$ where s' is the derivative of s and this function, as well as the $\times$ are applied entrywise.

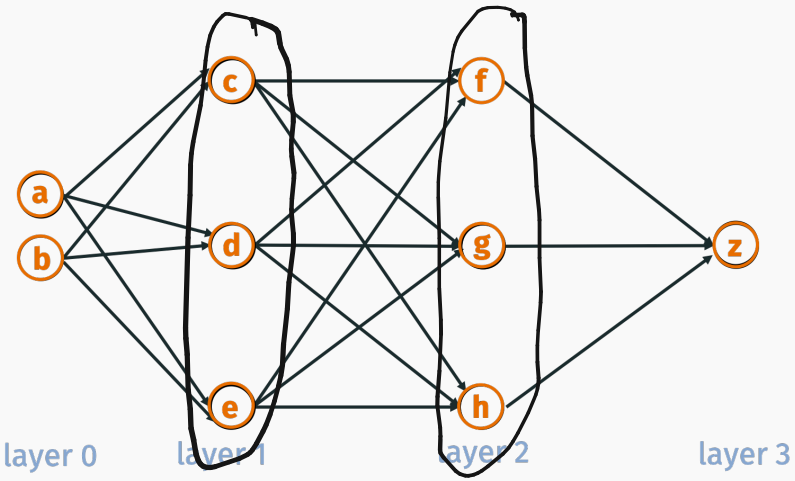
BACKPROP LINEAR ALGEBRA

Let $\mathbf{W}_i$ be a matrix containing all the weights for edges between layer i and layer $i+1$.



$$\mathbf{W}_2 = \begin{bmatrix} W_{f,z} & W_{g,z} & W_{h,z} \end{bmatrix} \quad \mathbf{W}_1 = \begin{bmatrix} W_{c,f} & W_{d,f} & W_{e,f} \\ W_{c,g} & W_{d,g} & W_{e,g} \\ W_{c,h} & W_{d,h} & W_{e,h} \end{bmatrix} \quad \mathbf{W}_0 = \begin{bmatrix} W_{a,c} & W_{b,c} \\ W_{a,d} & W_{b,d} \\ W_{a,e} & W_{b,e} \end{bmatrix}$$

BACKPROP LINEAR ALGEBRA

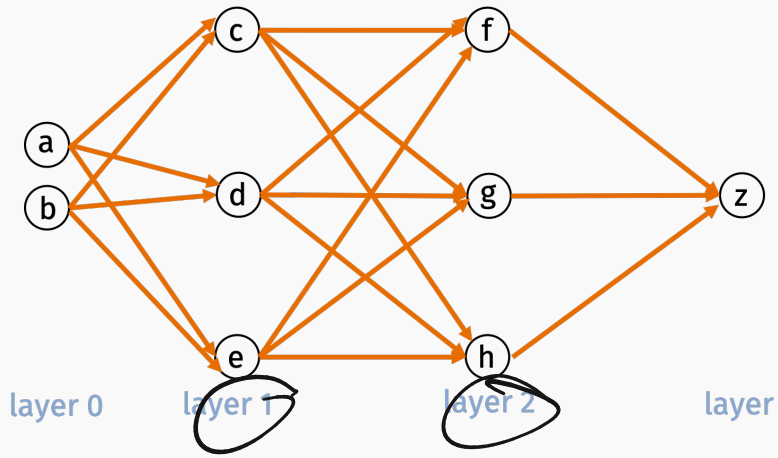


Claim 1: Node derivative computation is matrix multiplication.

$$\boldsymbol{\delta}_i = \mathbf{W}_i^T \bar{\boldsymbol{\delta}}_{i+1}$$

BACKPROP LINEAR ALGEBRA

Let $\boldsymbol{\Delta}_i$ be a matrix contain the derivatives for all weights for edges between layer i and layer $i+1$.

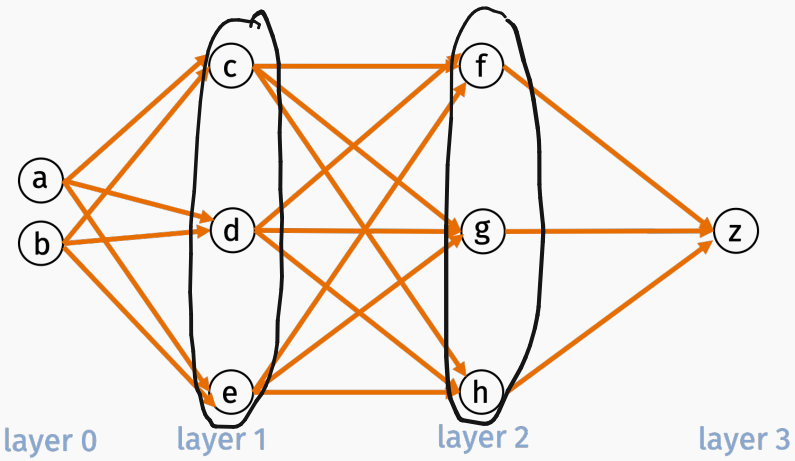


$$\boldsymbol{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

$$\boldsymbol{\Delta}_1 = \begin{bmatrix} \partial z/\partial W_{c,f} & \partial z/\partial W_{d,f} & \partial z/\partial W_{e,f} \\ \partial z/\partial W_{c,g} & \partial z/\partial W_{d,g} & \partial z/\partial W_{e,g} \\ \partial z/\partial W_{c,h} & \partial z/\partial W_{d,h} & \partial z/\partial W_{e,h} \end{bmatrix}$$

$$\boldsymbol{\Delta}_0 = \ldots$$

BACKPROP LINEAR ALGEBRA



Claim 2: Weight derivative computation is an outer-product.

$$\Delta_i = \mathbf{v}_i \bar{\delta}_{i+1}^T.$$

$$(3 \times 1) \cdot (1 \times 3) = (3 \times 3)$$

Takeaways:

- Backpropagation can be used to compute derivatives for all weights and biases for any feedforward neural network.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be parallelized and performed quickly on a GPU.

Backpropagation allows us to compute $\nabla L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right)$ for a single training example $(\mathbf{x}_i, y_i)$. Computing entire gradient requires computing:

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla L\left(y_i, f(\boldsymbol{\theta}, \mathbf{x}_i)\right)$$

Computing the entire sum would be very expensive.

$O\left((\text{time for backprop}) \cdot n\right)$ time.

Recall: Stochastic Gradient Descent (SGD).

Let $L_j(\boldsymbol{\theta})$ denote $L\left(y_j, f(\boldsymbol{\theta}, \mathbf{x}_j)\right)$.

Claim: If $j \in 1, \ldots, n$ is chosen uniformly at random. Then:

$$n \cdot \mathbb{E}\left[\nabla L_j(\boldsymbol{\theta})\right] = \nabla \mathcal{L}(\boldsymbol{\theta}).$$

$\nabla L_j(\boldsymbol{\theta})$ is called a stochastic gradient and just requires running backprop once.

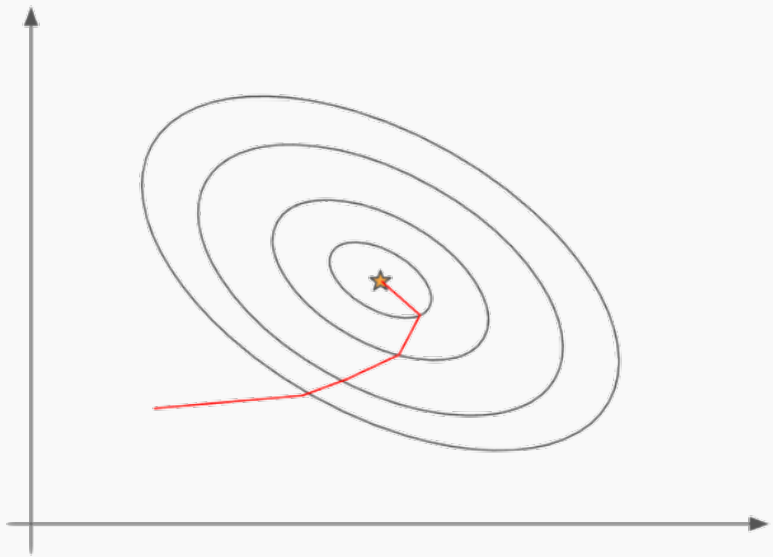
SGD iteration:

- Initialize $\boldsymbol{\theta}_0$ (typically randomly).
- For $t = 1, \ldots, T$:
  - Choose j uniformly at random.
  - Compute stochastic gradient $\mathbf{g} = \nabla L_j(\boldsymbol{\theta}_t)$.
    - For neural networks this is done using backprop with training example $(\mathbf{x}_j, y_j)$.
  - Update $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}$

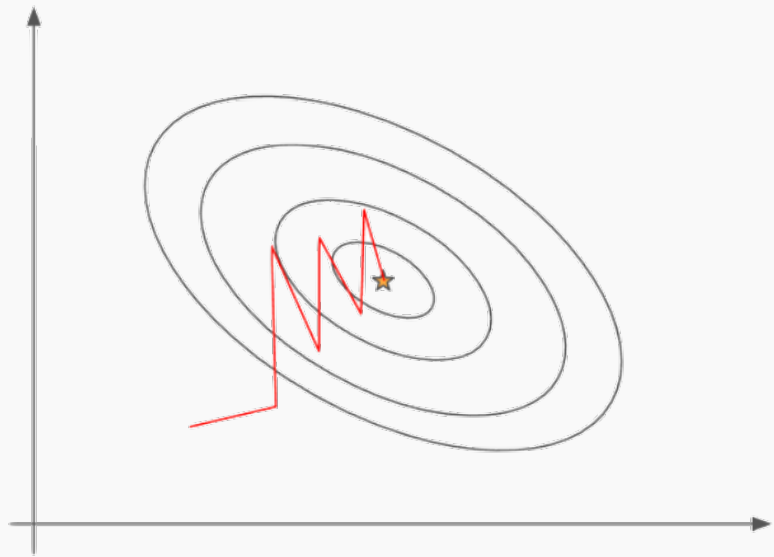
Move in direction of steepest descent in expectation.

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.

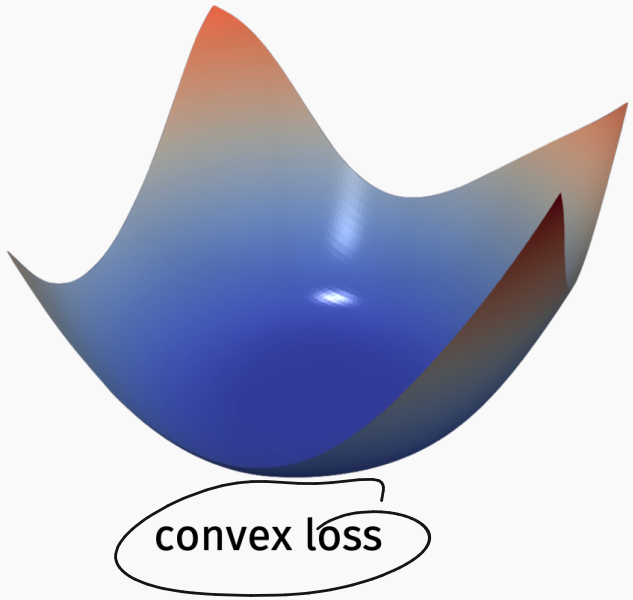


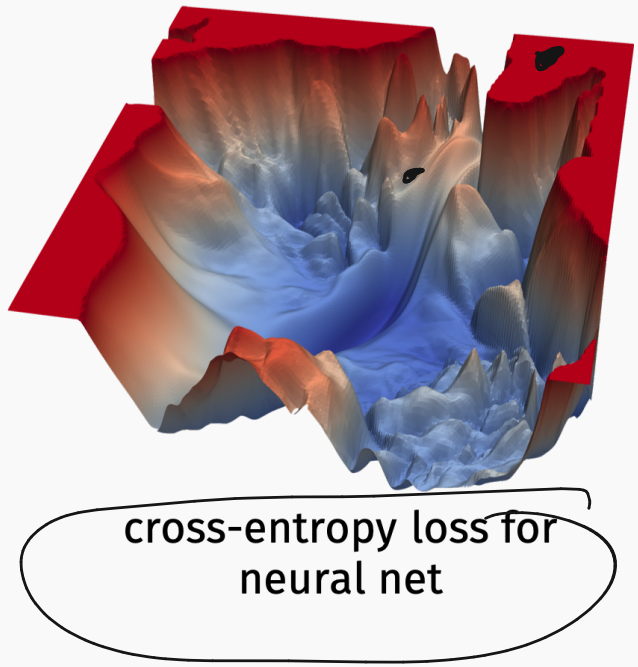
Gradient Descent



Stochastic Gradient Descent

Least squares regression, logistic regression, SVMs, even all of these with kernels lead to convex losses.



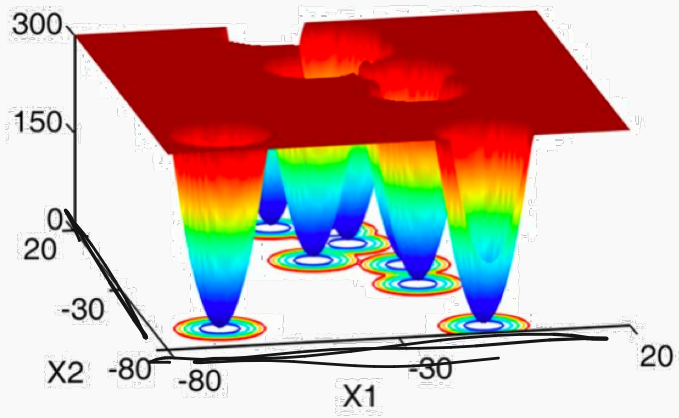


Neural networks very much do not...

CONVERGENCE

But SGD still performs remarkably well in practice. Understanding this phenomenon is a major open research question in machine learning and optimization.

- Initialization seems important (random uniform vs. random Gaussian vs. Xavier initialization vs. He initialization vs. etc.)
- SGD finds "good" local minima?



STOCHASTIC GRADIENT DESCENT IN PRACTICE

We already discussed a few practical modifications of SGD:

- Using "mini-batch" gradients. $\sum_{i=1}^{B} \nabla L_{j_i}(\boldsymbol{\theta})$.
- Shuffling then cycling through training data instead of picking a training data point at random each time.

Practical Modification: **Per-parameter adaptive learning rate.**

Let $\mathbf{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_p \end{bmatrix}$ be a stochastic or batch stochastic gradient. Our typical parameter update looks like:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}.$$

step size

We've already seen a simple method for adaptively choosing the learning rate/step size η. Worked well for convex functions.

Practical Modification: **Per-parameter adaptive learning rate.**

In practice, neural networks can often be optimized much faster by using "adaptive gradient methods" like Adagrad, Adadelta, RMSProp, and ADAM. These methods make updates of the form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \begin{bmatrix} \eta_1 \cdot g_1 \\ \vdots \\ \eta_p \cdot g_p \end{bmatrix}$$

So we have a separate learning rate for each entry in the gradient (e.g. parameter in the model). And each $\eta_1, \ldots, \eta_p$ is chosen adaptively.

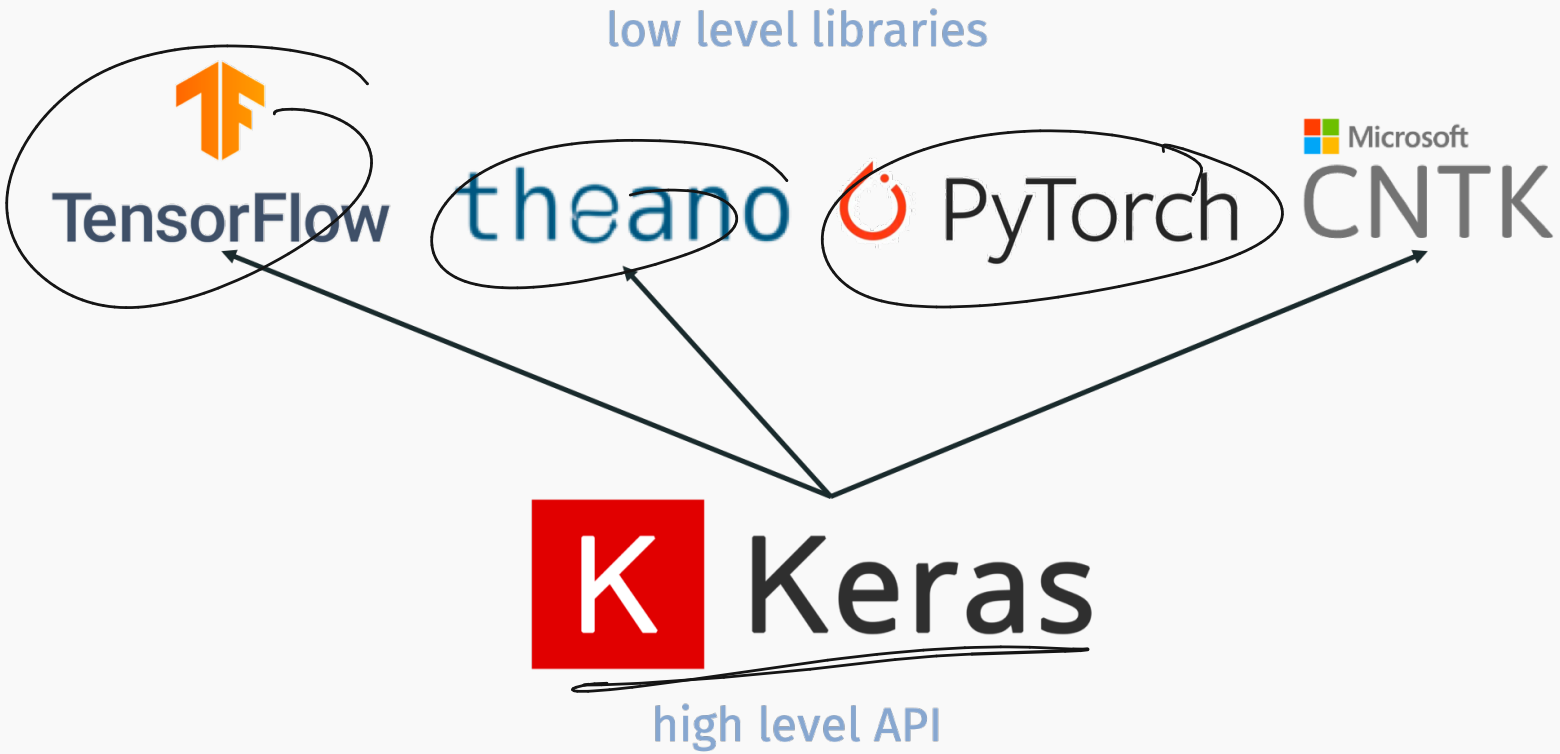
NEURAL NETWORK DEMOS

Two demos on neural networks:

- `keras_demo_synthetic.ipynb`
- `keras_demo_mnist.ipynb`

Please spend some time working through these!

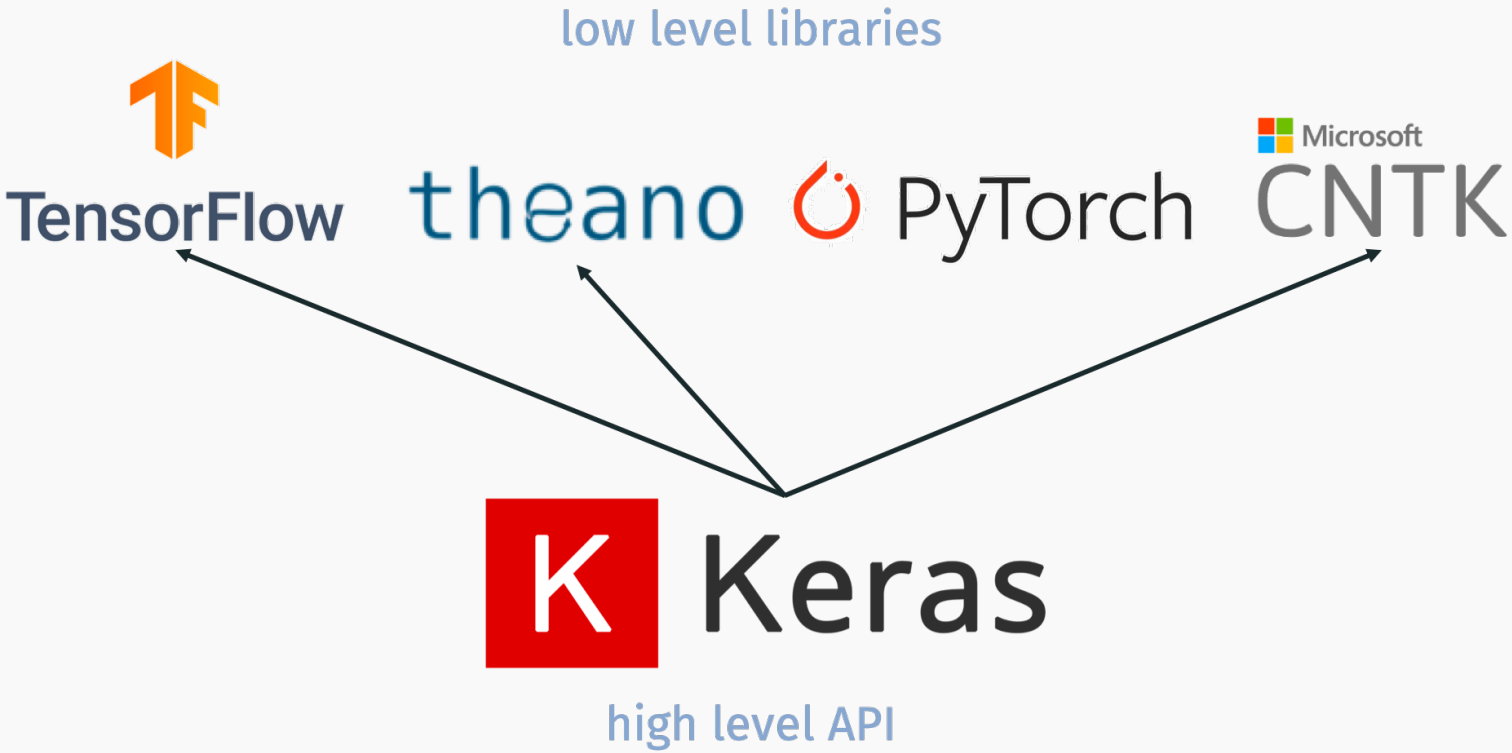
NEURAL NETWORK SOFTWARE



low level libraries

TensorFlow

NEURAL NETWORK SOFTWARE



Low-level libraries have built in optimizers (SGD and improvements) and can automatically perform backpropagation for arbitrary network structures. Also ptimize code for any available GPUs.

Keras has high level functions for defining and training a neural network architecture.

Define model:

```
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

Compile model:

```
opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train model:

```
hist = model.fit(Xtr, ytr, epochs=30, batch_size=100, validation_data=(Xts,yts))
```

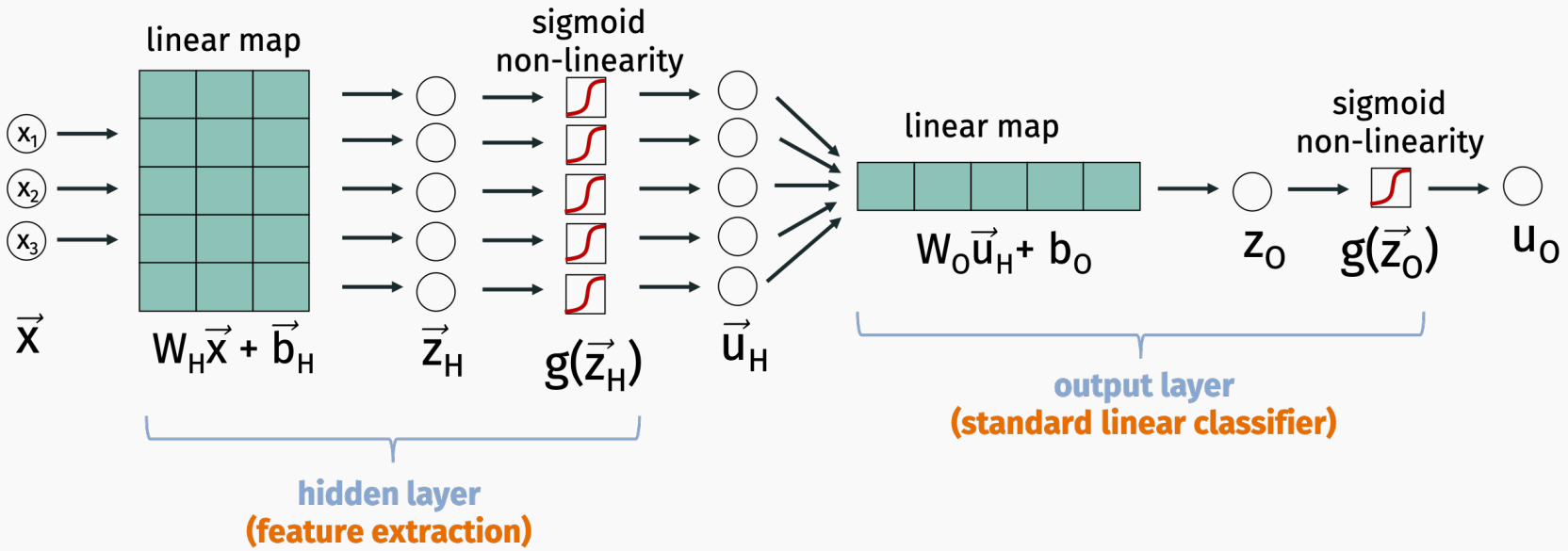
Convolutional neural networks (CNNs)

Why do neural networks work so well?

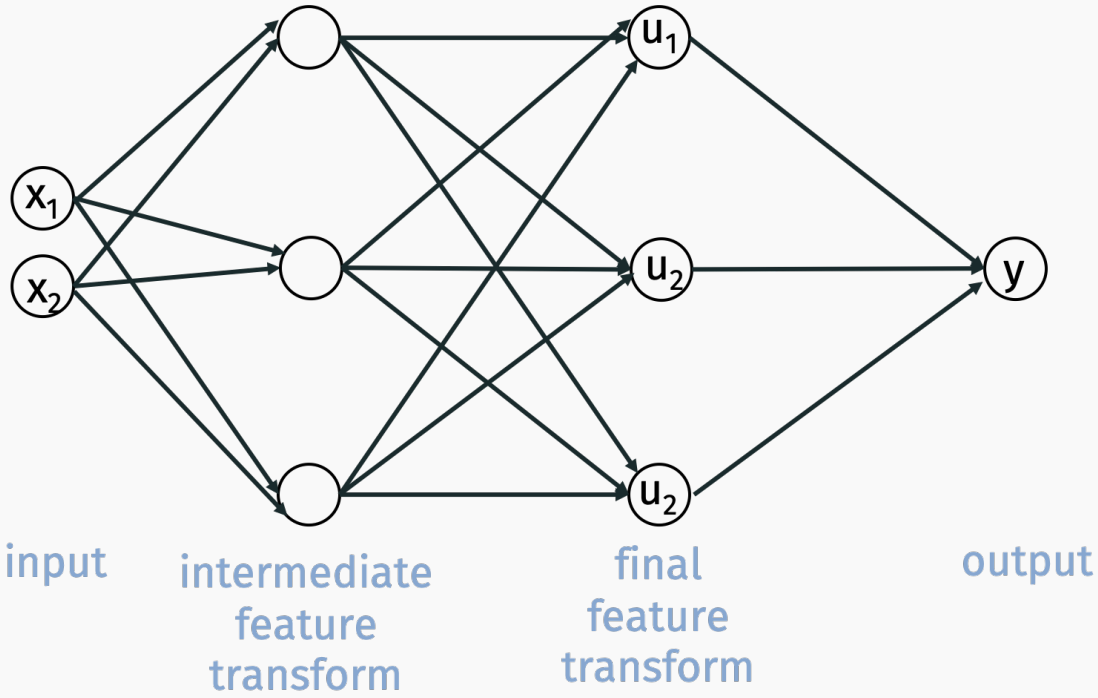
Treat feature transformation/extraction as part of the learning process instead of making this the users job.

But sometimes they still need a nudge in the right direction...

BASIC FEATURE EXTRACTION

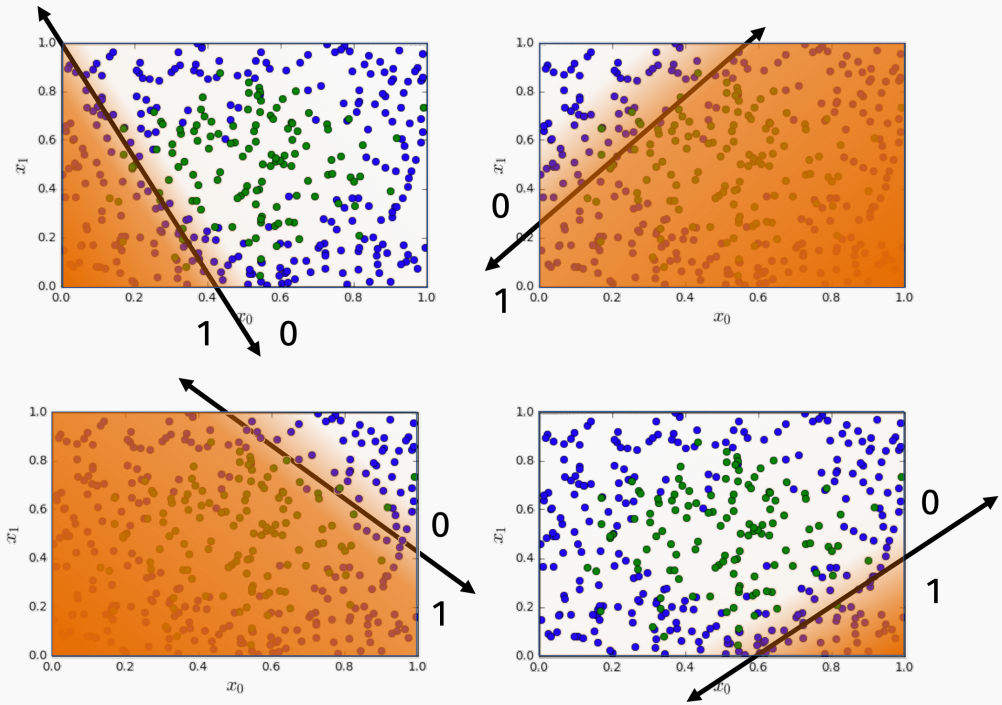


BASIC FEATURE EXTRACTION



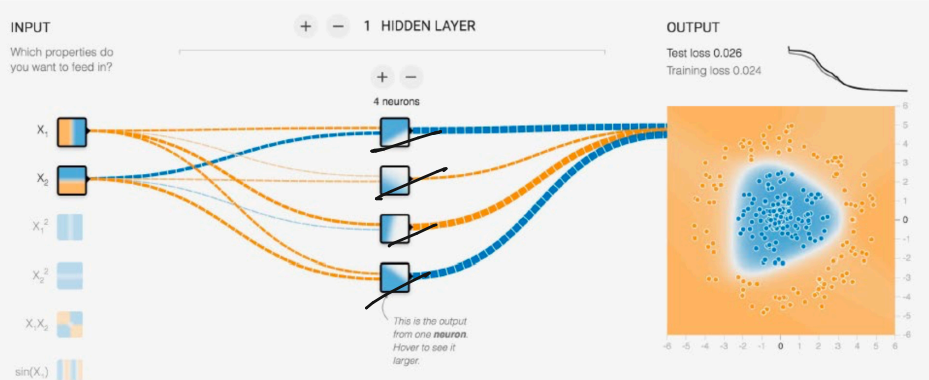
Final output or class label y is a linear function of the final layer variables $u_1, \ldots, u_k$. You could just as well have taken these variables and used them to predict y via linear regression, logistic regression, SVM, any other linear method.

Sigmoid activation: Each hidden variable z_i equal to $\frac{1}{1+e^{-\bar{z}_i}}$ where $z_i = \mathbf{w}^T\mathbf{x} + b$ for input $\mathbf{x}$.

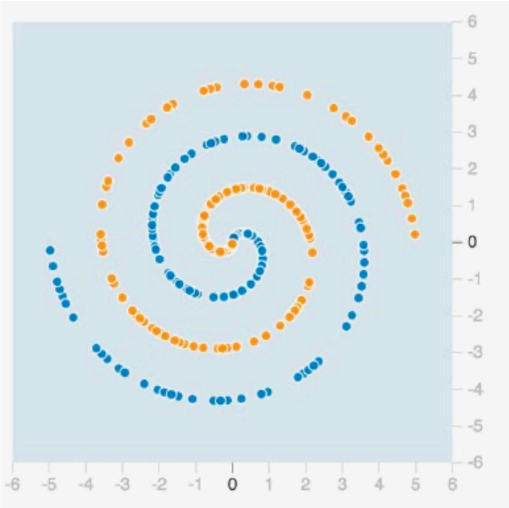


Other non-linearities yield similarly simple feature extractions.

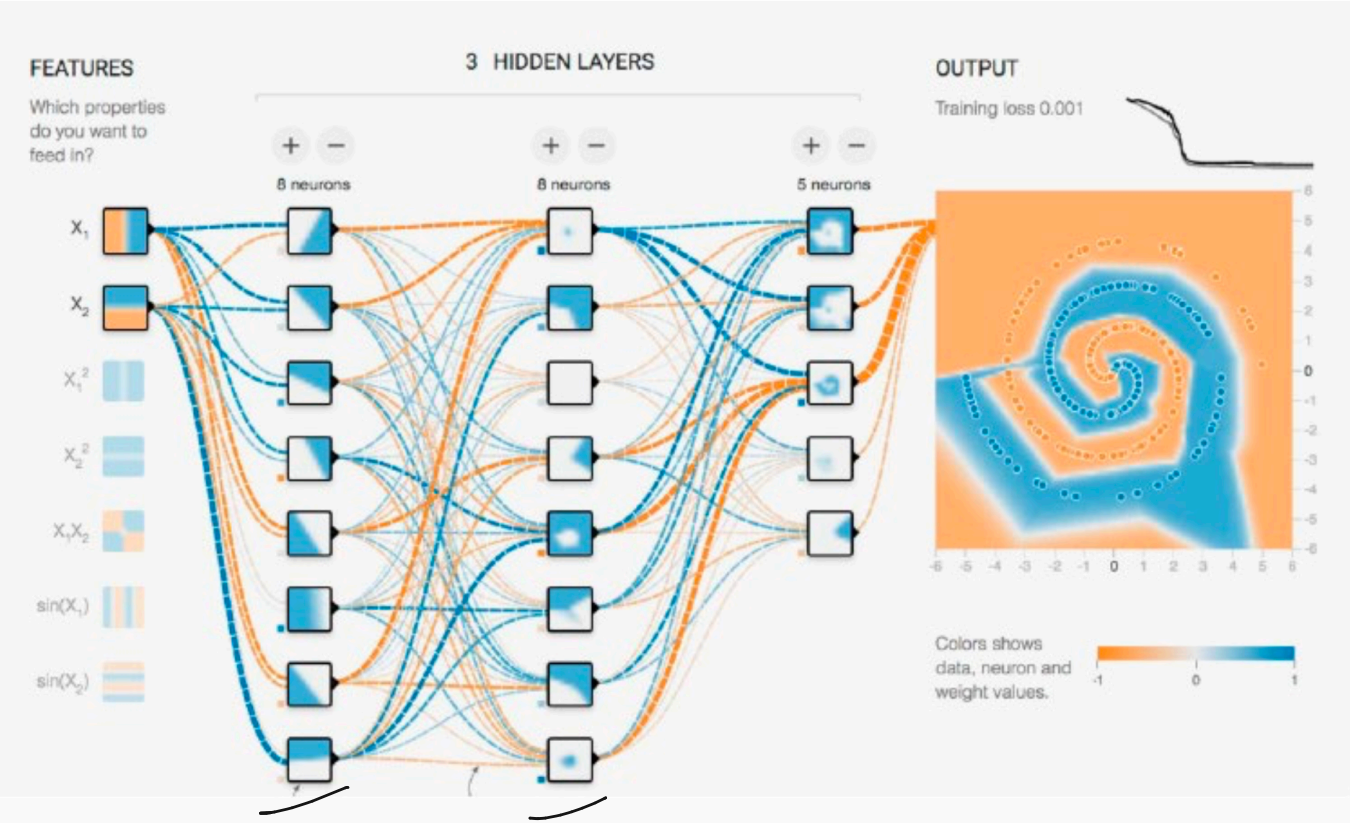
If you combine more hidden variables, you can start building more complicated classifiers.



How about for even more complex datasets?



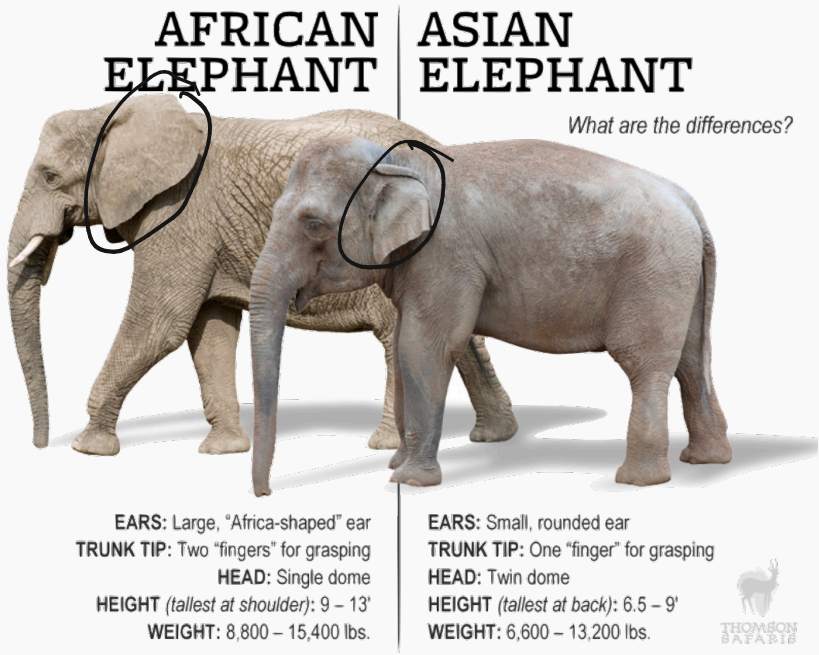
With more layers, complexity starts ramping up...



But there's a limit...

BASIC FEATURE EXTRACTION

Modern machine learning algorithms can differentiate between images of African and Asian elephants...



The features needed for a task like this are far more complex then we could expect a network to learn completely on its own using simple combinations of linear layers + non-linearities.

CONVOLUTIONAL FEATURE EXTRACTION

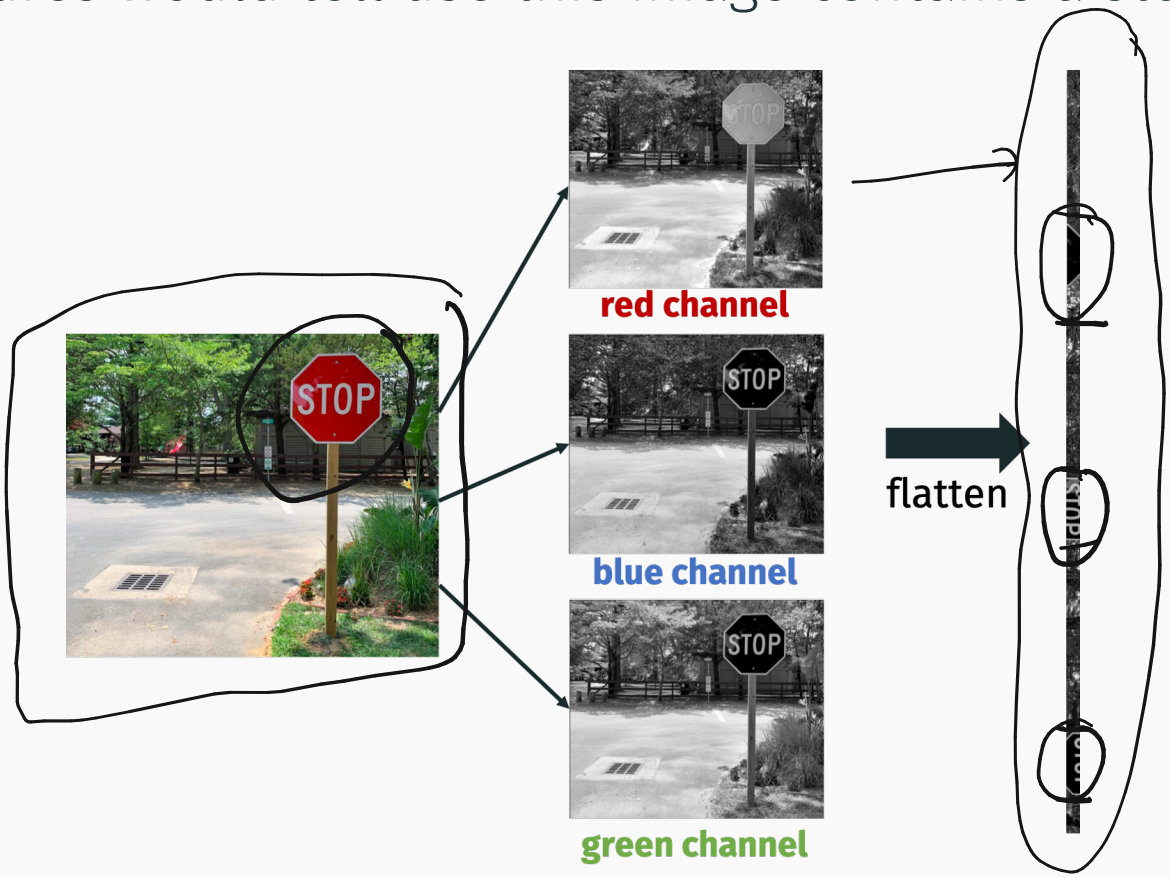
Today's topic: Understand why convolution is a powerful way of extracting features from:

- Image data.
- Audio data.
- Time series data.

Ultimately, can build convolutional networks that already have convolutional feature extraction pre-coded in. Just need to learn weights.

MOTIVATING EXAMPLE

What features would tell use this image contains a stop sign?



Typically way of vectorizing an image chops up and splits up any pixels in the stop sign. We need very complex features to piece these back together again...

CONVOLUTION

Objects or features of an image often involve pixels that are spatially correlated. Convolution explicitly encodes this.

Definition (Discrete 1D convolution[1])

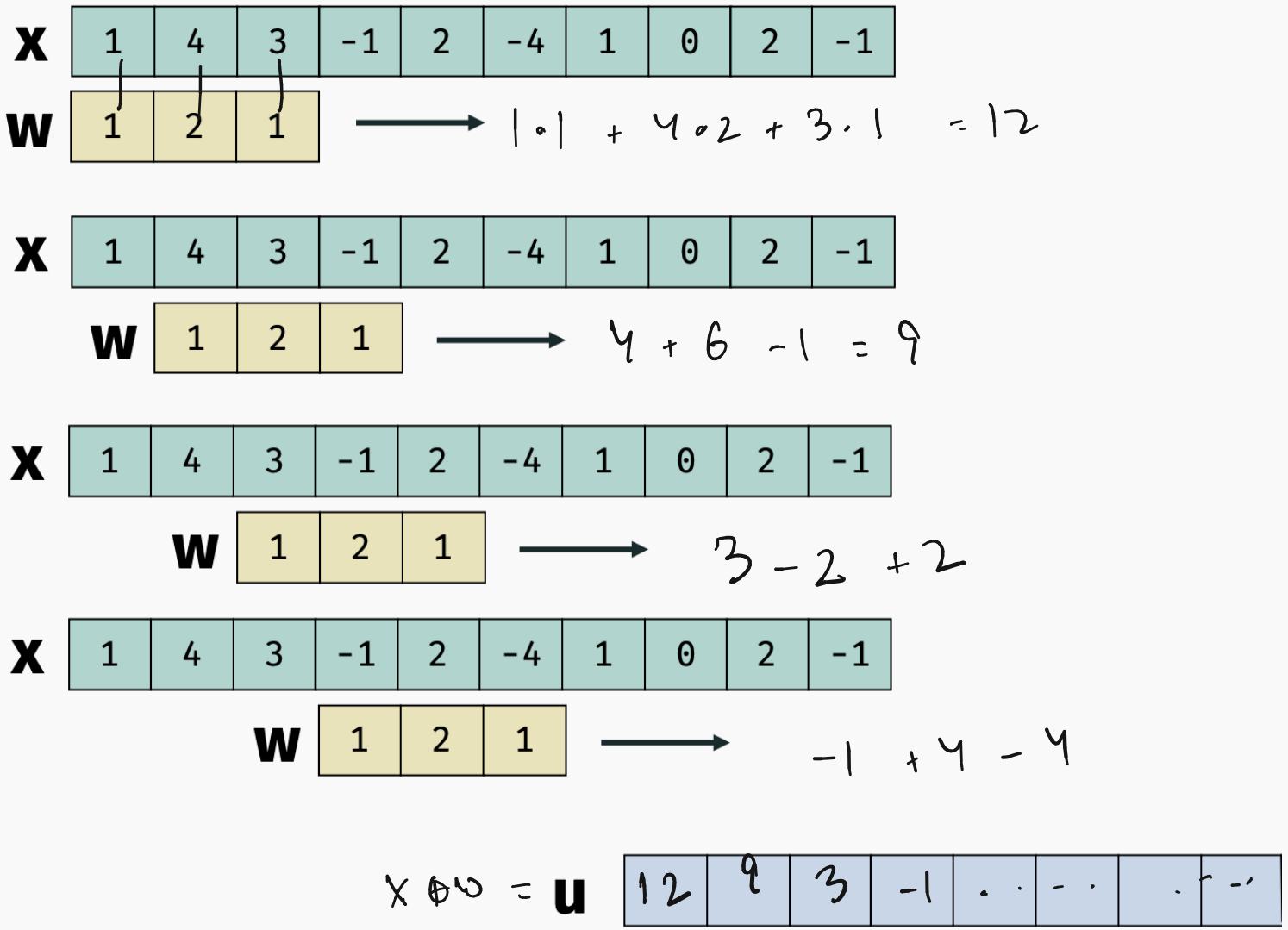
Given $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{w} \in \mathbb{R}^k$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $d - k + 1$ vector with:

$$[\mathbf{x} \circledast \mathbf{w}]_j = \sum_{j=1}^{k} \mathbf{x}_{(j+i-1)} \mathbf{w}_j$$

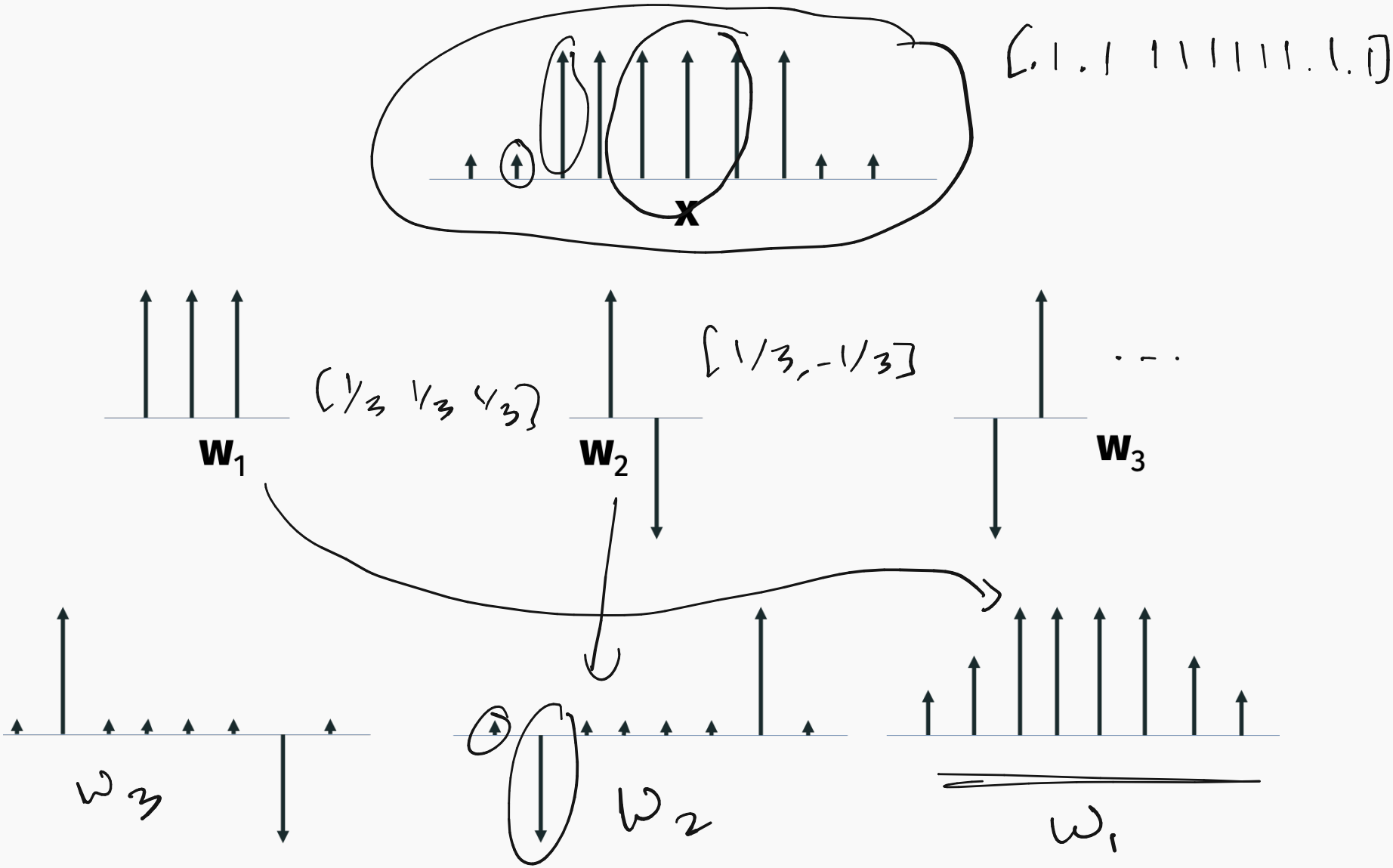
Think of $\mathbf{x} \in \mathbb{R}^d$ as long **data vector** (e.g. d = 512) and $\mathbf{w} \in \mathbb{R}^k$ as short **filter vector** (e.g. k = 8). $\mathbf{u} = [\mathbf{x} \circledast \mathbf{w}]$ is a feature transformation.

[1] This is slightly different from the definition of convolution you might have seen in a Digital Signal Processing class because $\mathbf{w}$ does not get "flipped". In signal processing our operation would be called correlation.

1D CONVOLUTION



MATCH THE CONVOLUTION



2D convolution

Definition (Discrete 2D convolution)

Given matrices $\mathbf{x} \in \mathbb{R}^{d_1 \times d_2}$ and $\mathbf{w} \in \mathbb{R}^{k_1 \times k_2}$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1)$ matrix with:

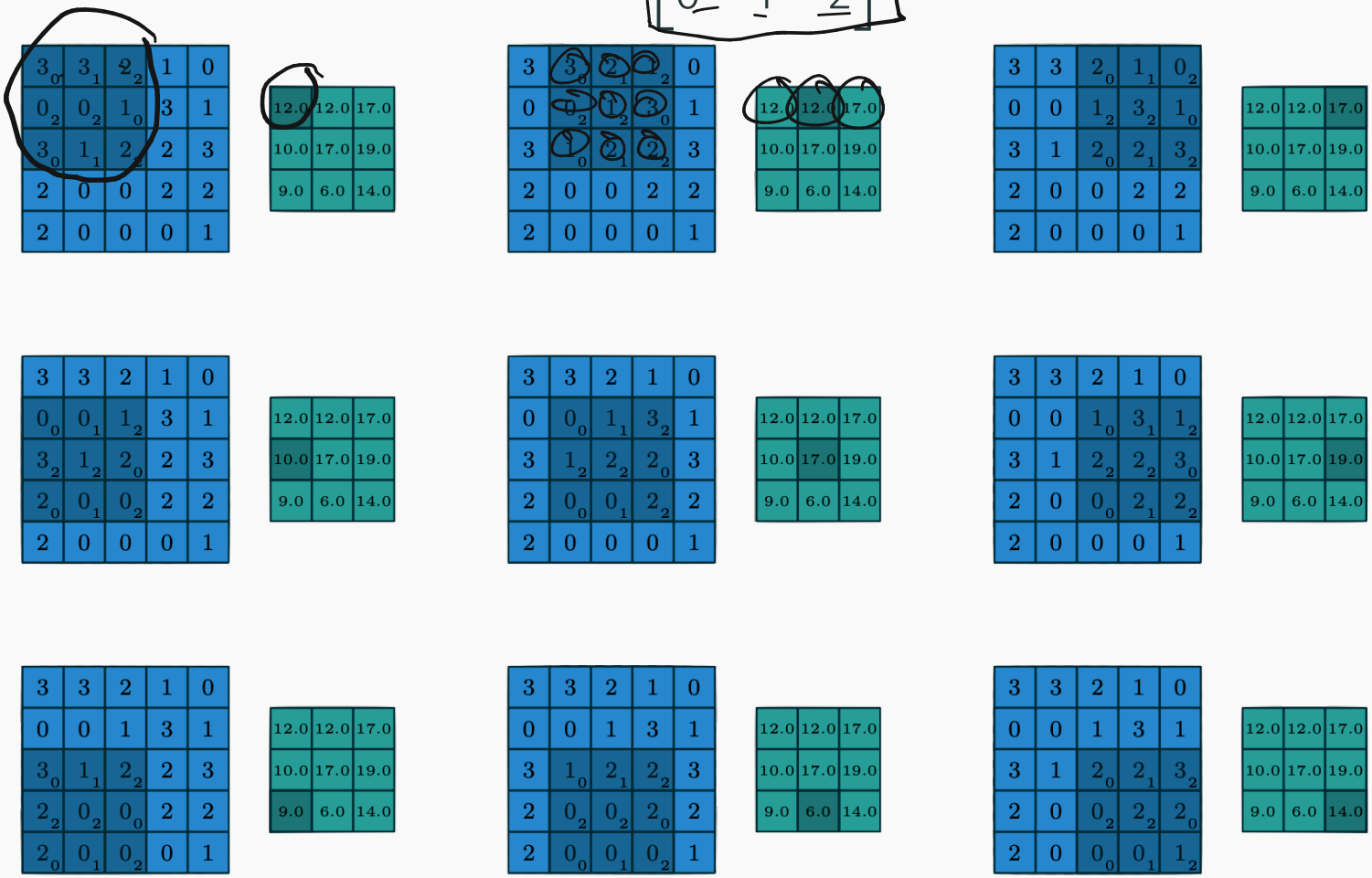
$$[\mathbf{x} \circledast \mathbf{w}]_{i,j} = \sum_{\ell=1}^{k_1} \sum_{h=1}^{k_2} \mathbf{x}_{(i+\ell-1),(j+h-1)} \cdot \mathbf{w}_{\ell,h}$$

Again technically this is "correlation" not "convolution". Should be performed in Python using `scipy.signal.correlate2d` instead of `scipy.signal.convolve2d`.

$\mathbf{w}$ is called the filter or convolution kernel and again is typically much smaller than $\mathbf{x}$.

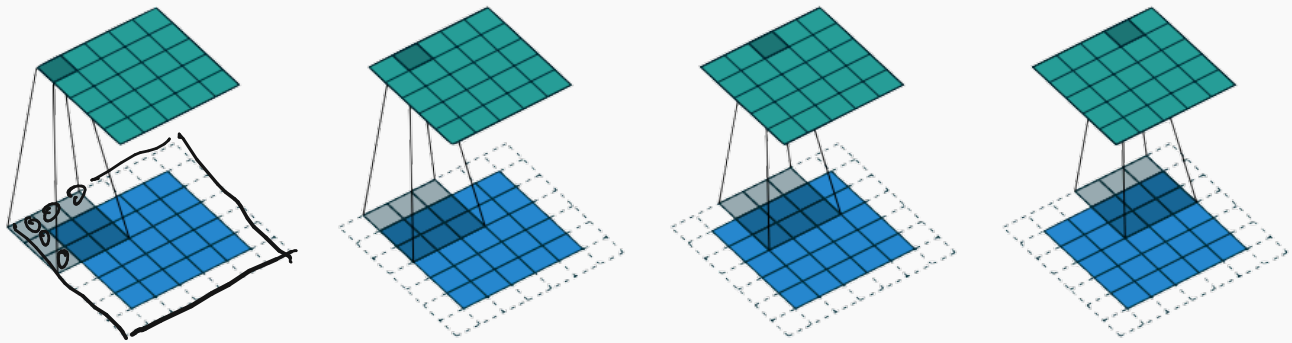
2D CONVOLUTION

$$w = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$



ZERO PADDING

Sometimes "zero-padding" is introduced so $\mathbf{x} \circledast \mathbf{w}$ is $d_1 \times d_2$ if $\mathbf{x}$ is $d_1 \times d_2$.



Need to pad on left and right by $(k_1 - 1)/2$ and on top and bottom by $(k_2 - 1)/2$.

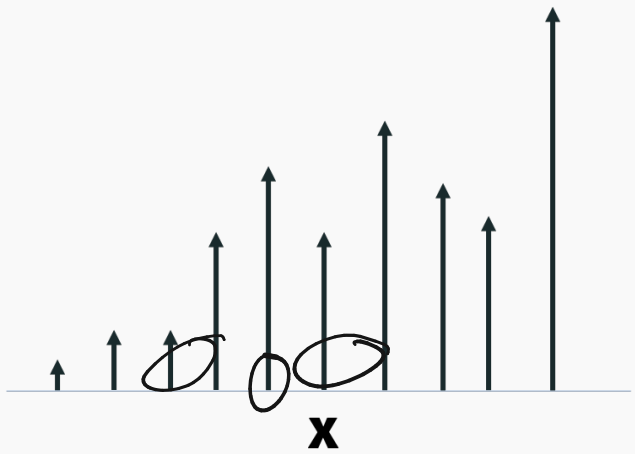
Examples code will be available in
`demo1_convolutions.ipynb`.

Application 1: Blurring/smooth.

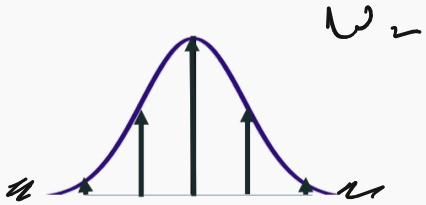
In one dimension:

- Uniform (moving average) filter: $\mathbf{w}_i = \frac{1}{k}$ for $i = 1, \ldots, k$.
- Gaussian filter: $\mathbf{w}_i \sim e^{-(i-k/2)^2/\sigma^2}$ for $i = 1, \ldots, k$.

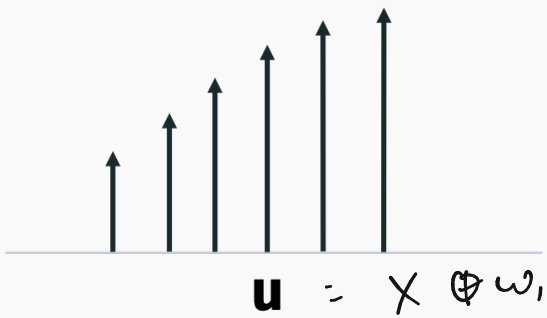
SMOOTHING FILTERS

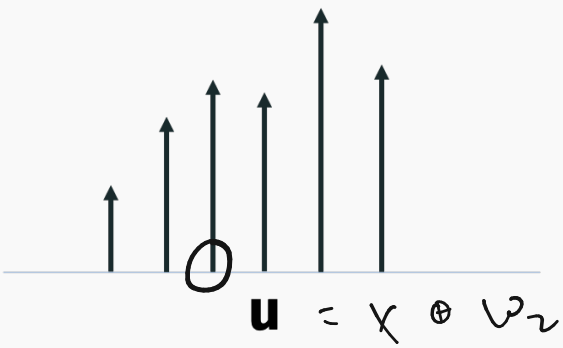


Uniform filter
(moving average)



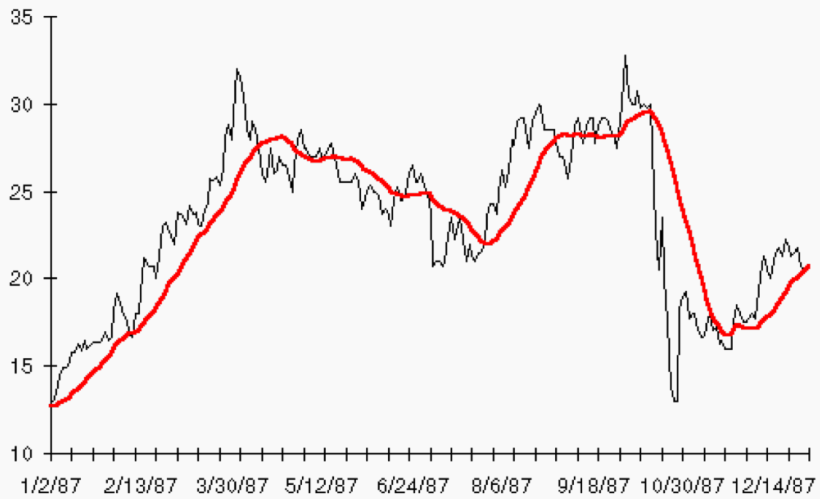
Gaussian filter





SMOOTHING FILTERS

Useful for smoothing time-series data, or removing noise/static from audio data.



Replaces every data point with a local average.

SMOOTHING IN TWO DIMENSIONS

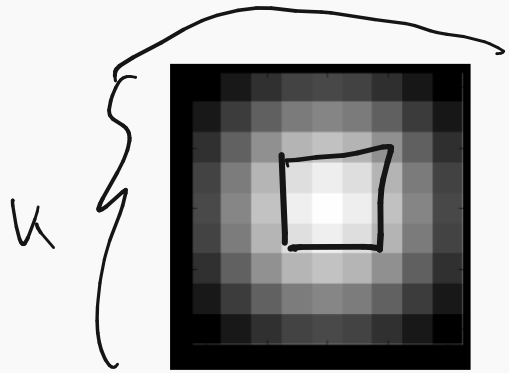
In two dimensions:

- Uniform filter: $\mathbf{w}_{i,j} = \frac{1}{k_1 k_2}$ for $i = 1, \ldots, k_1, j = 1, \ldots, k_2$.
- Gaussian filter: $\mathbf{w}_i \sim e^{-\frac{(i-k_1/2)^2+(j-k_2/2)^2}{\sigma^2}}$ for $i = 1, \ldots, k_1$, $j = 1, \ldots, k_2$.

Larger filter equates to more smoothing.

SMOOTHING IN TWO DIMENSIONS

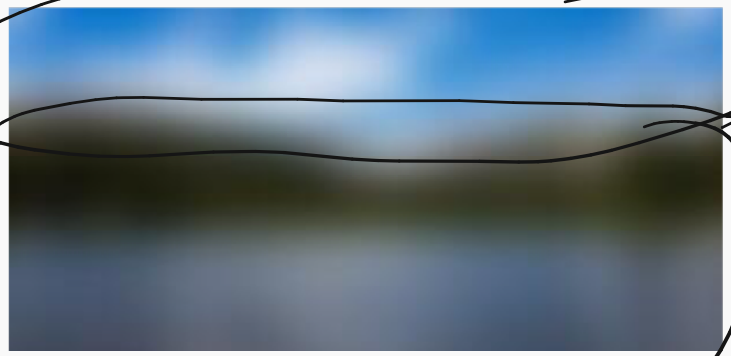
For Gaussian filter, you typically choose $k \gtrsim 2\sigma$ to capture the fall-off of the Gaussian.

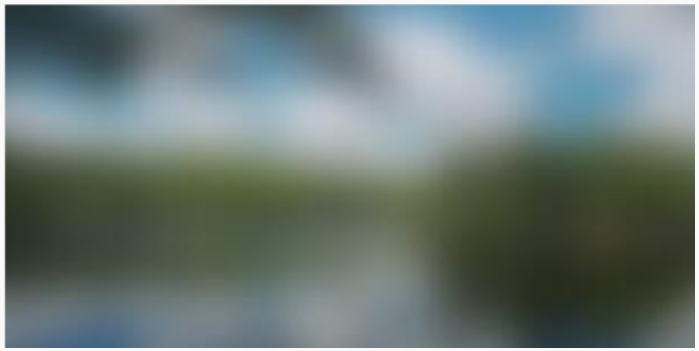


Both approaches effectively denoise and smooth images.

SMOOTHING FOR FEATURE EXTRACTION

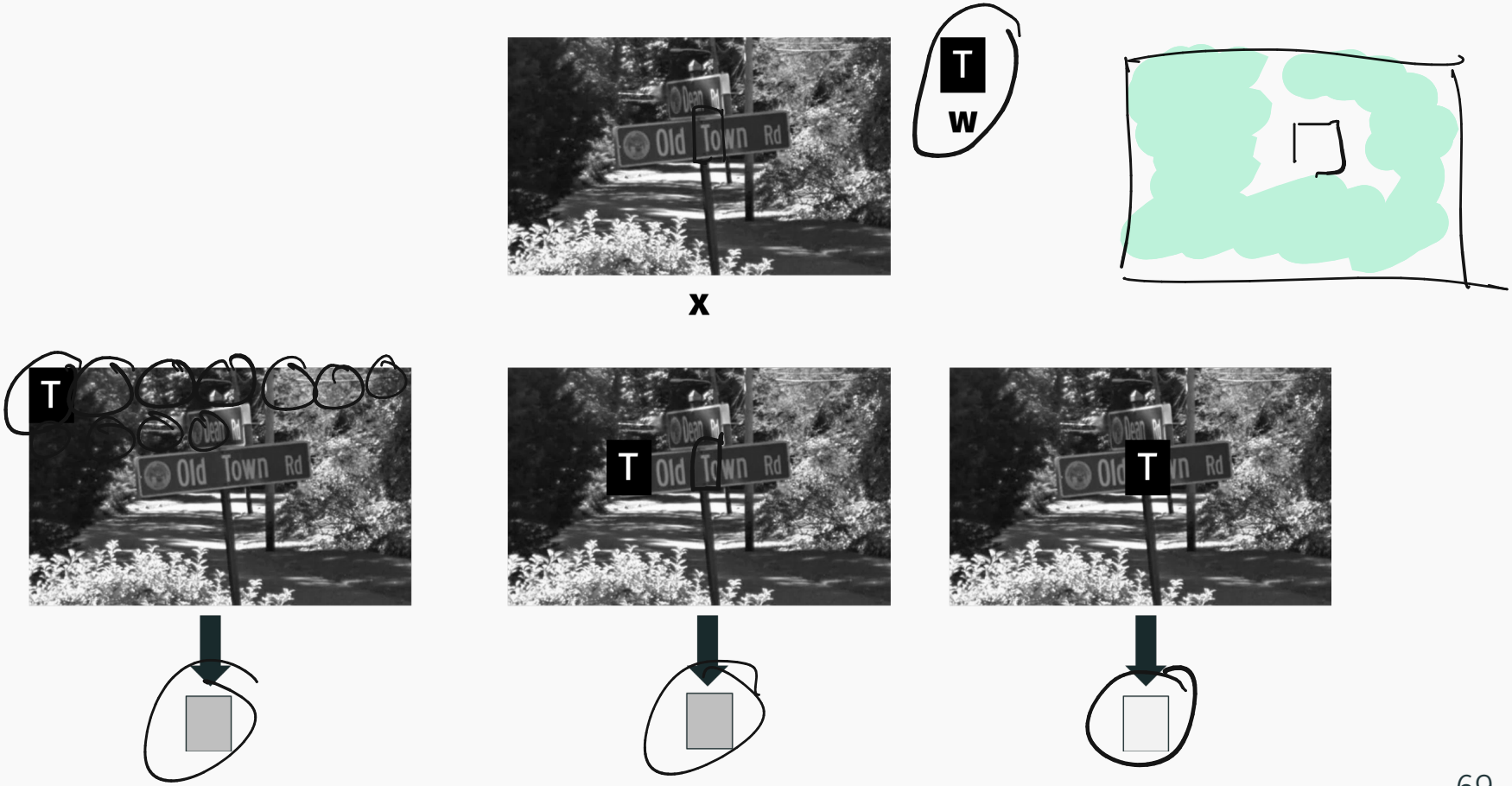
When combined with other feature extractors, smoothing at various levels allows the algorithm to focus on high-level features over low-level features.





Application 2: Pattern matching.

Slide a pattern over an image. Output of convolution will be higher when pattern correlates well with underlying image.



Applications of local pattern matching:

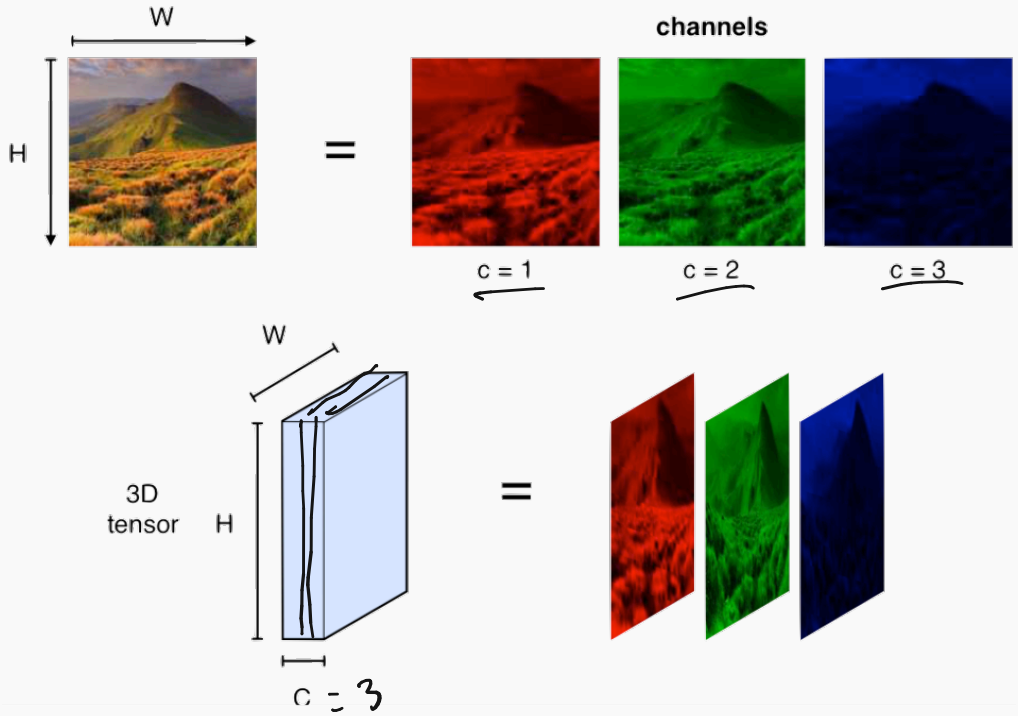
- Check if an image contains text.
- Look for specific sound in audio recording.
- Check for other well-structured objects

3D convolution

Recall that color images actually have three color channels for **red**, **green**, **blue**. Each pixel is represented by 3 values (e.g. in $0, \ldots, 255$) giving the intensity in each channel.

$[0, 0, 0]$ = black, $[0, 0, 0]$ = white, $[1, 0, 0]$ = pure red, etc.

View image as 3D tensor:



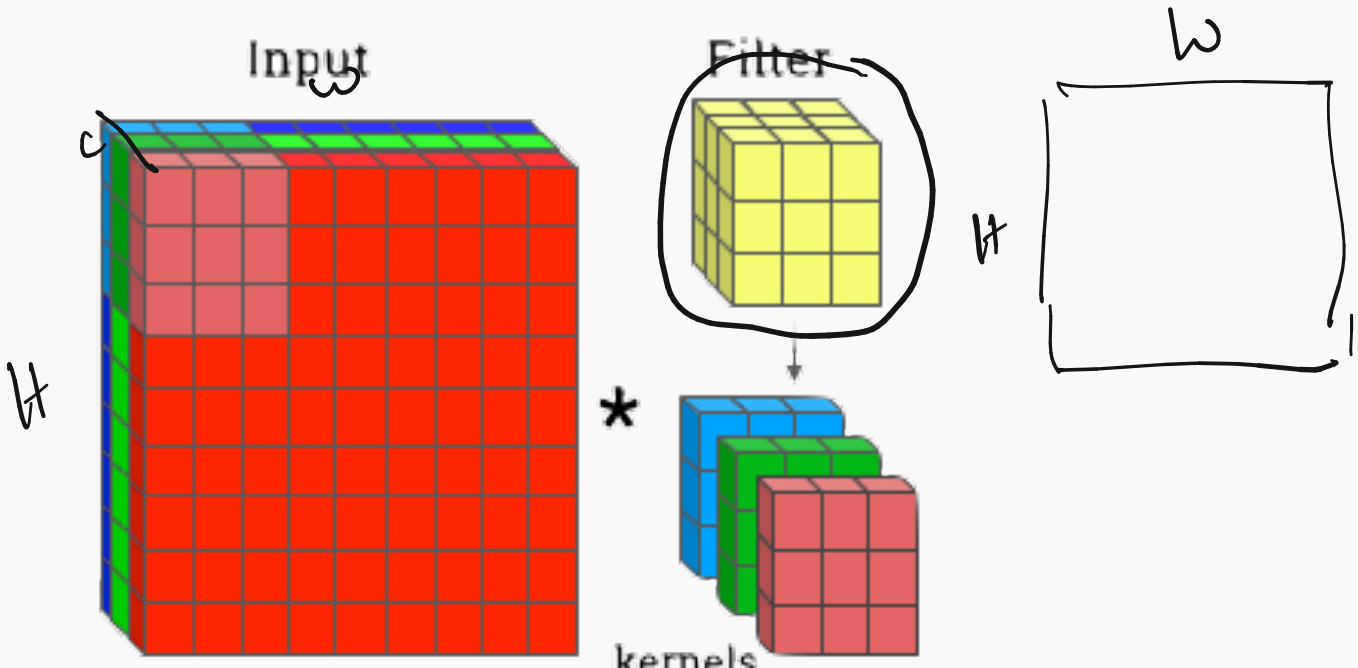
3D convolution

Can be convolved with 3D filter:

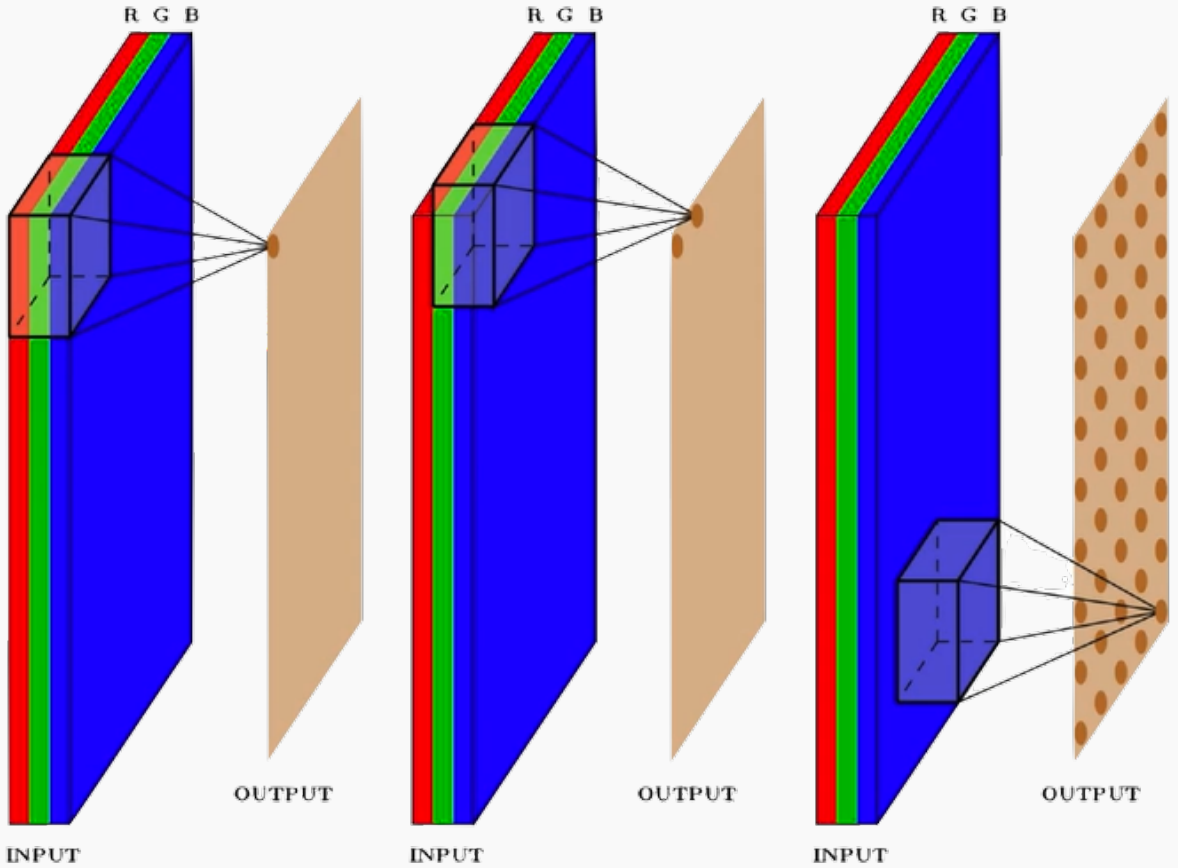
Definition (Discrete 3D convolution)

Given tensors $\mathbf{x} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $\mathbf{w} \in \mathbb{R}^{k_1 \times k_2 \times k_3}$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1) \times (d_3 - k_3 + 1)$ tensor with:

$$[\mathbf{x} \circledast \mathbf{w}]_{i,j,g} = \sum_{\ell=1}^{k_1} \sum_{m=1}^{k_2} \sum_{n=1}^{k_3} \mathbf{x}_{(i+\ell-1),(j+m-1),(g+n-1)} \cdot \mathbf{w}_{\ell,m,n}$$

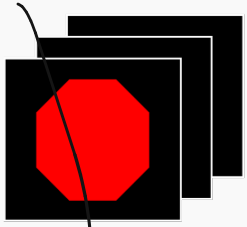


3D convolution

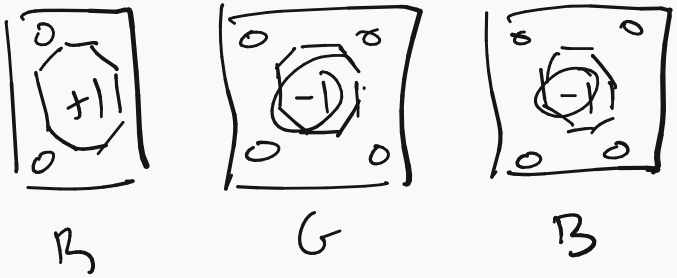


3D CONVOLUTION

x



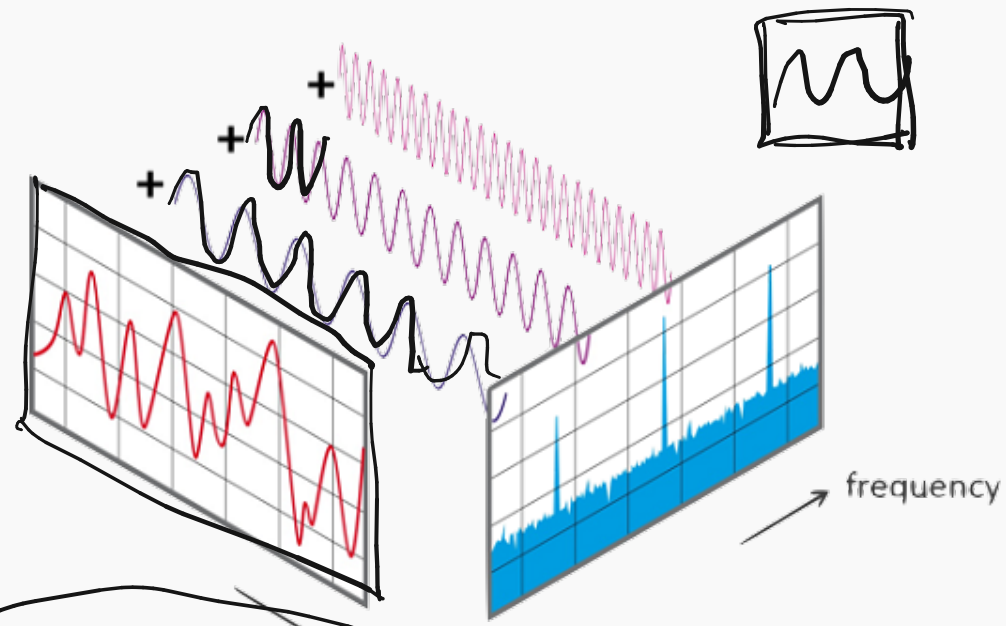
w



Relatively robust to imperfections, damage, occlusion, etc.

Less obvious example of pattern matching: Frequency detection in audio.

Any 1D signal (including a sound wave) can be decomposed into component frequencies:

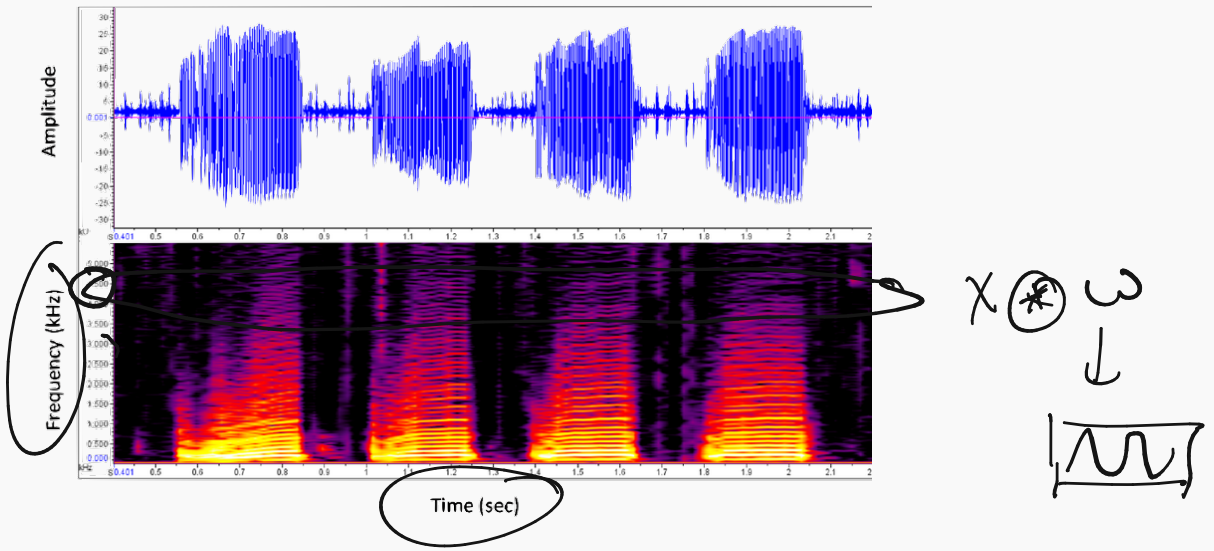


$$x(t) = \sin(f_1 t + s_1) + \sin(f_2 t + s_2) + \sin(f_3 t + s_3) + \ldots$$

FREQUENCY DETECTION

Convolve audio signal with snippet of pure frequency to determine where difference frequencies are prevalent. Detect things like:

- Common notes in a song.
- Different instruments.
- Human voices vs. other noise.

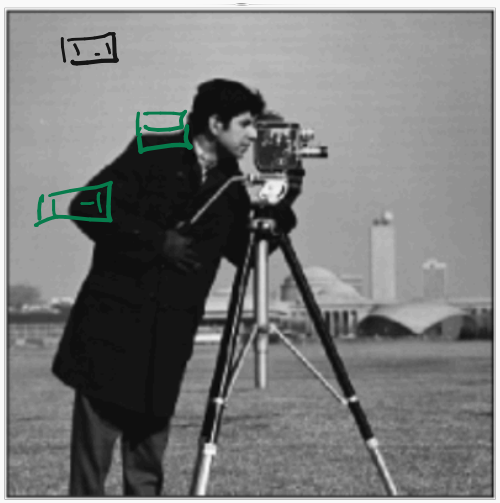


Main idea behind short-time Fourier transforms/spectrograms.

Application 3: Edge detection.

Consider a 2D edge detection filter:

$$W_1 = \begin{bmatrix} 1 & -1 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

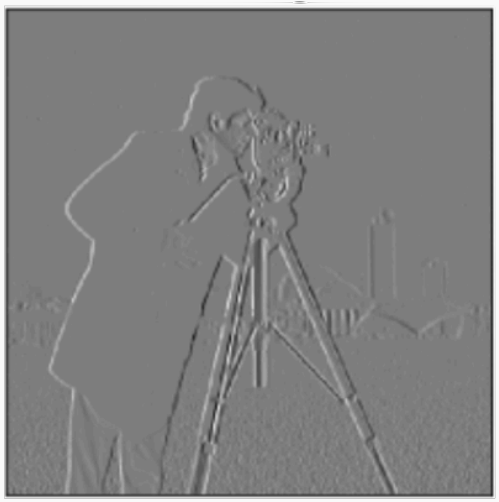


$x * ?$

$x * ?$

Sobel filter is more commonly used:

$$W_1 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

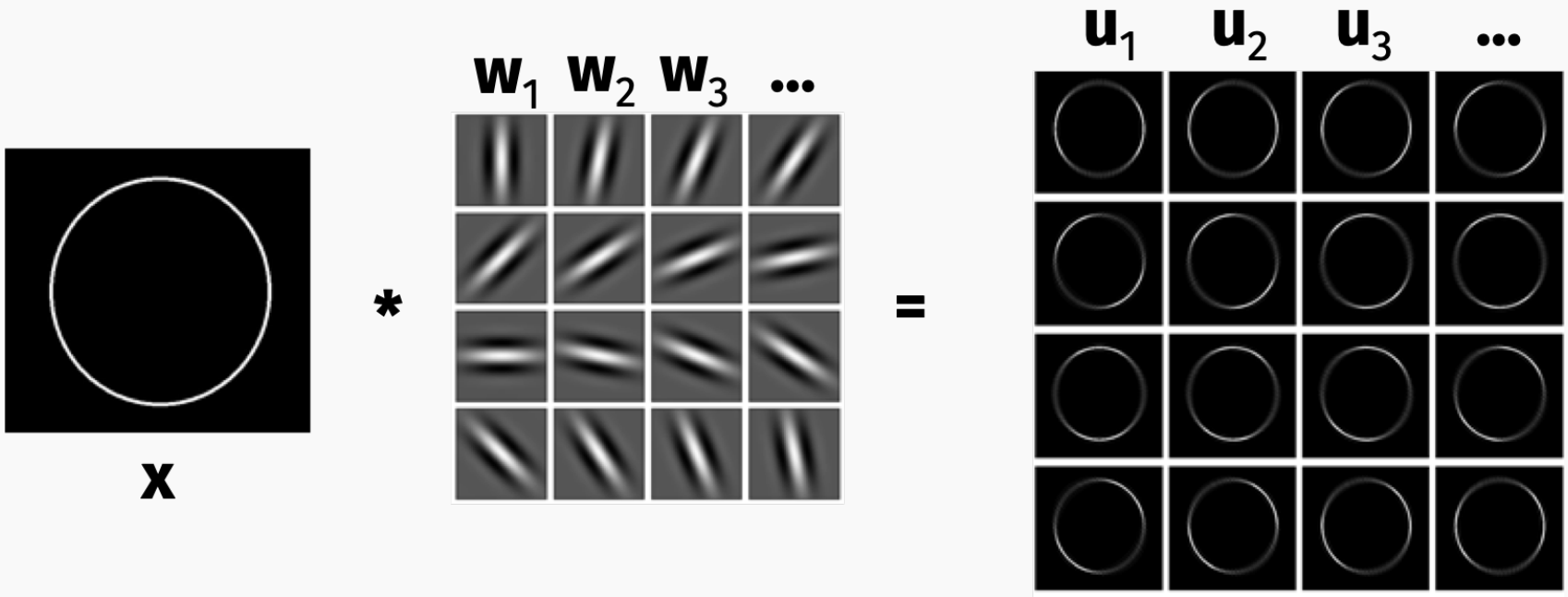


x * ?

x * ?

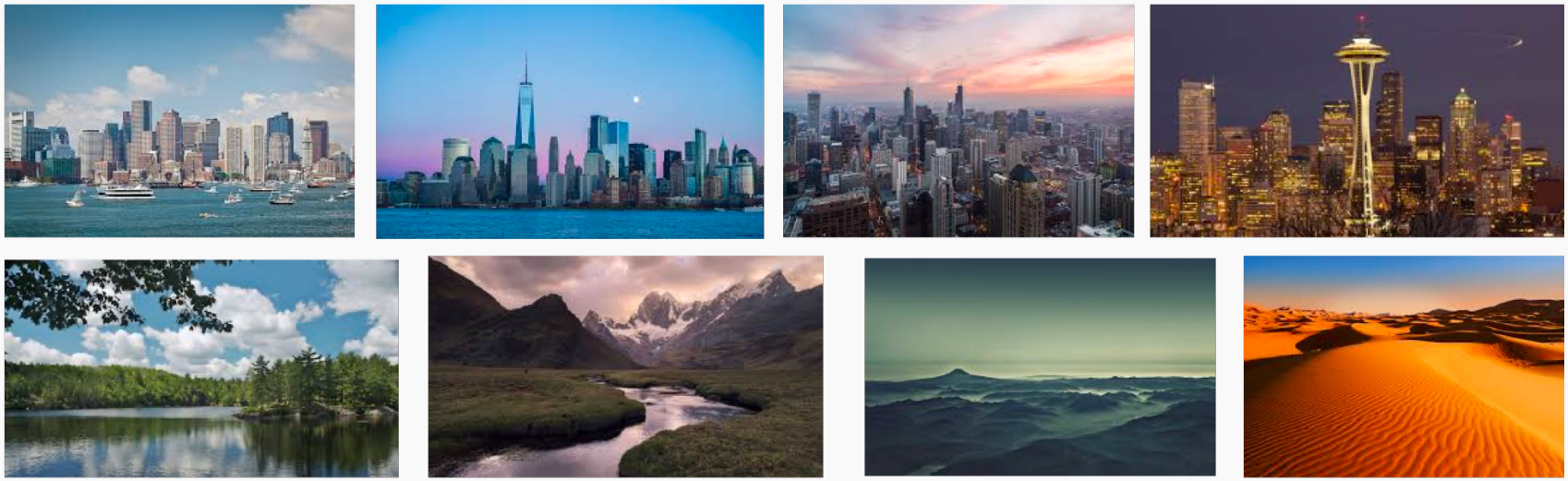
DIRECTIONAL EDGE DETECTION

Can define edge detection filters for any orientation.



EDGE DETECTION

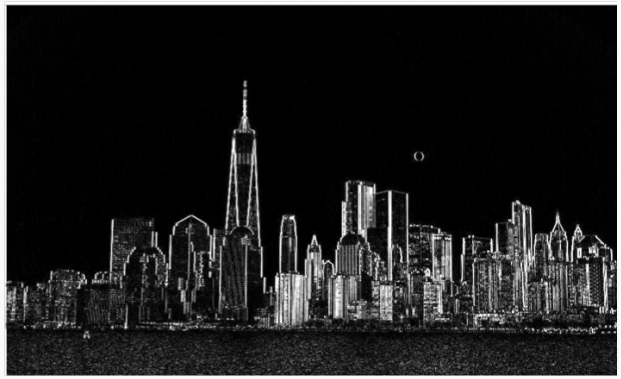
How would edge detection as a feature extractor help you classify images of city-scapes vs. images of landscapes?



EDGE DETECTION

$\mathbf{I}_C$

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$



$\mathbf{E}_C$

$\mathbf{I}_L$

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

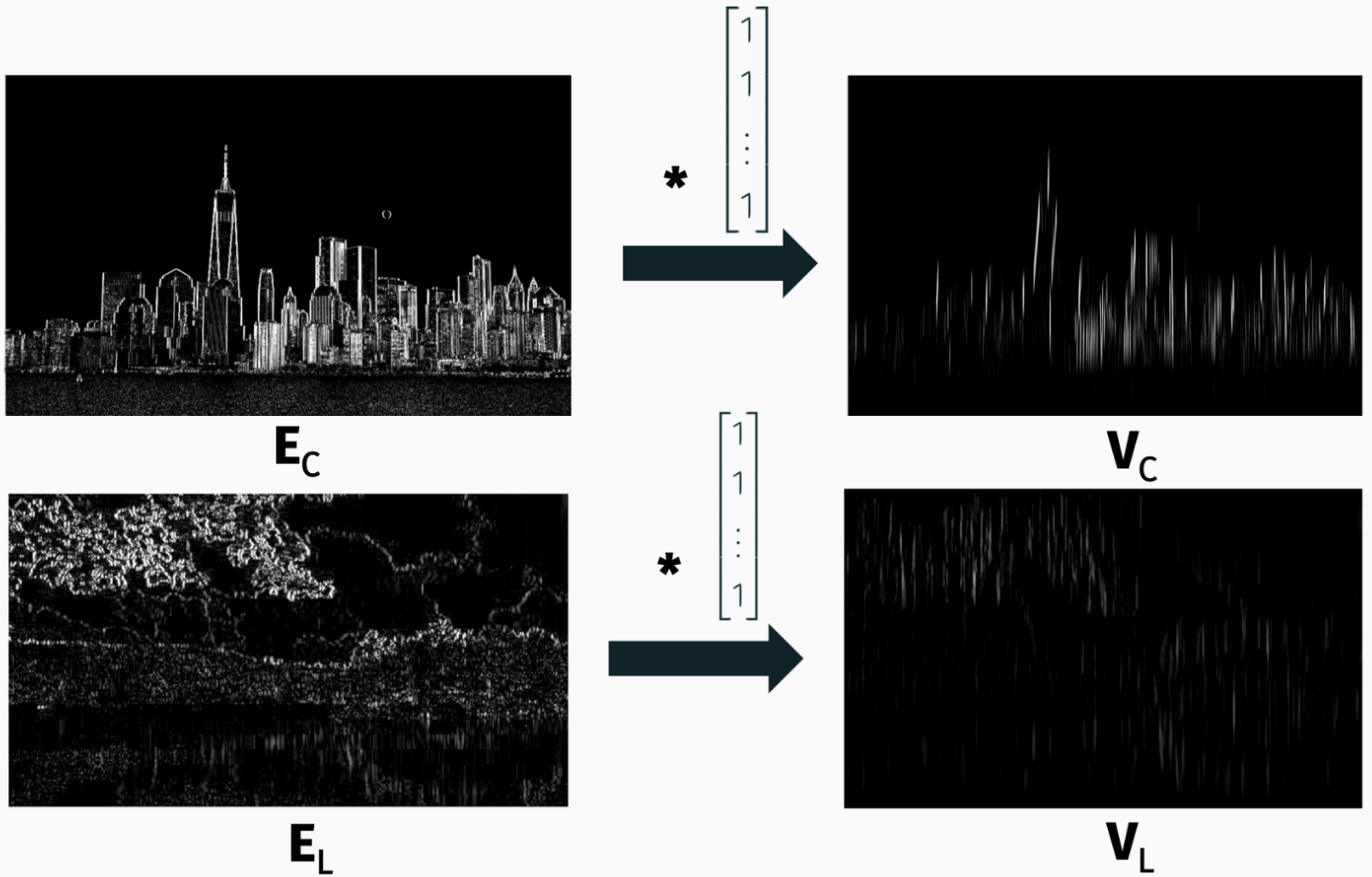
$\mathbf{E}_L$

$$\texttt{mean}(\mathbf{I}_C) = .108 \quad \text{vs.} \quad \texttt{mean}(\mathbf{I}_L) = .123$$

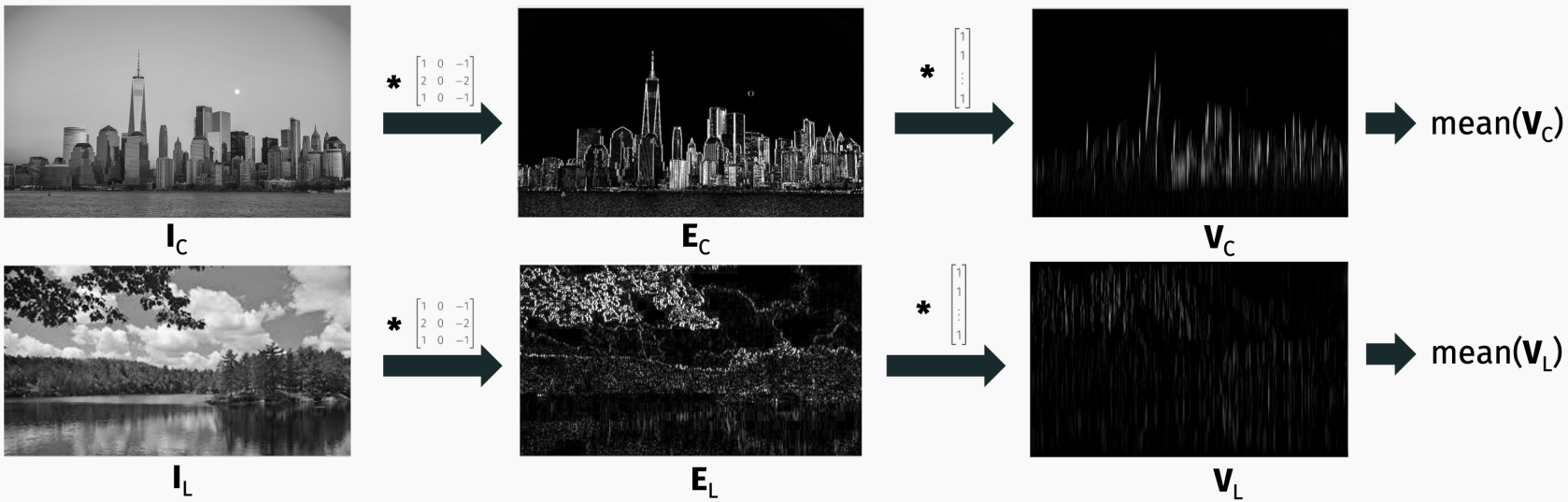
The image with highest vertical edge response isn't the city-scape.

EDGE DETECTION + PATTERN MATCHING

Feed edge detection result into pattern matcher that looks for long vertical lines.



HIERARCHICAL CONVOLUTIONAL FEATURES



$$\texttt{mean}(\mathbf{V}_C^2) = .042 \qquad \text{vs. } \texttt{mean}(\mathbf{V}_L^2) = .018$$

The image with highest average response to (edge detector) + (vertical pattern) is the city scape.

mean($\mathbf{V}$) is an extracted scalar feature which could be used for classifying cityscapes from landscapes using a linear classifier.

HIERARCHICAL CONVOLUTIONAL FEATURES

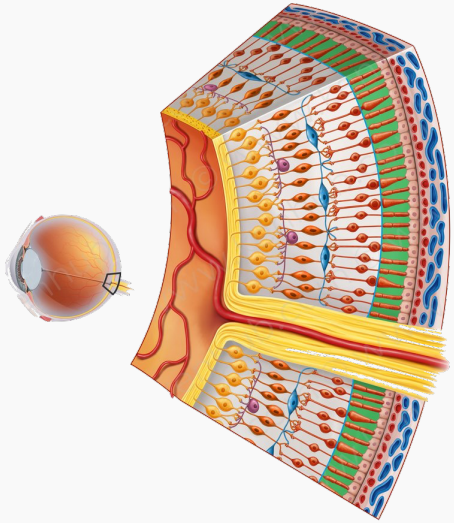
Hierarchical combinations of simple convolution filters are very powerful for understanding images.

In particular, edge detection seems like a critical first step.

Lots of evidence from biology.

VISUAL SYSTEM

Light comes into the eye through the lens and is detected by an array of photosensitive cells in the **retina**.



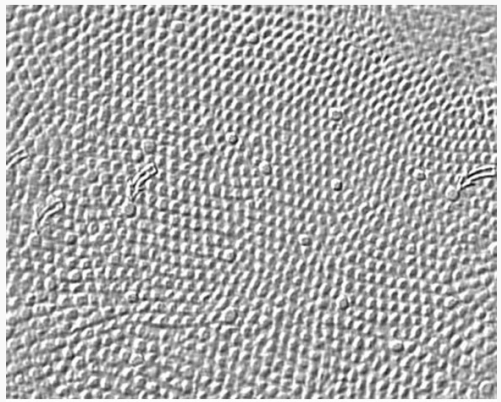
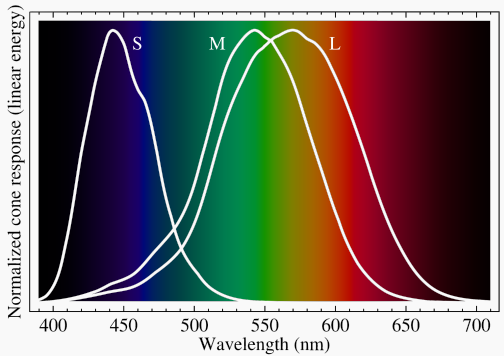


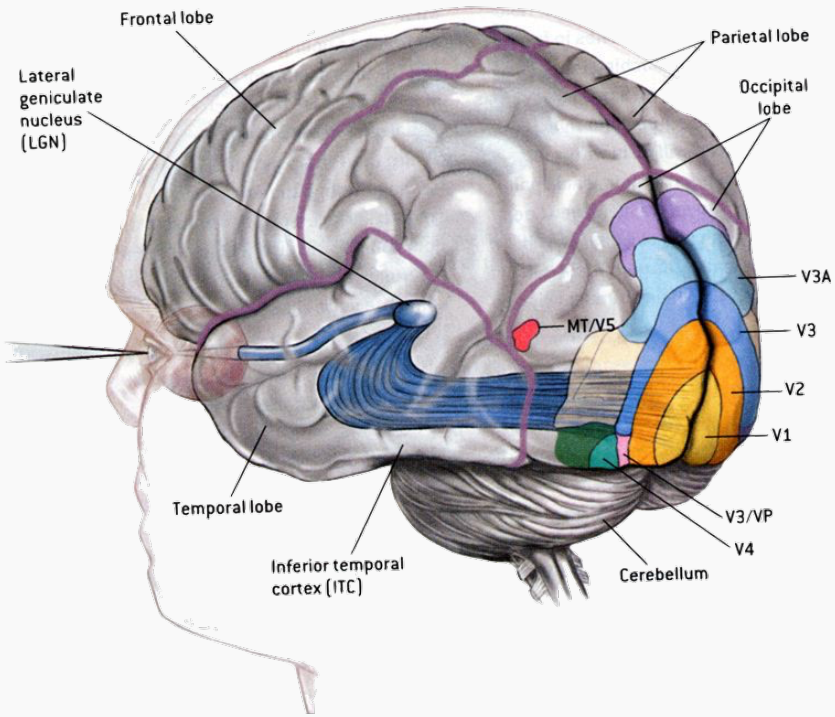
Fig. 13. Tangential section through the human fovea. Larger cones (arrows) are blue cones. From Ahnelt et al. 1987.

Rod cells are sensitive to all light, larger **cone** cells are sensitive to specific colors. We have three types of cones:



VISUAL SYSTEM

Signal passes from the retina to the primary (V1) visual cortex, which has neurons that connect to higher level parts of the brain.

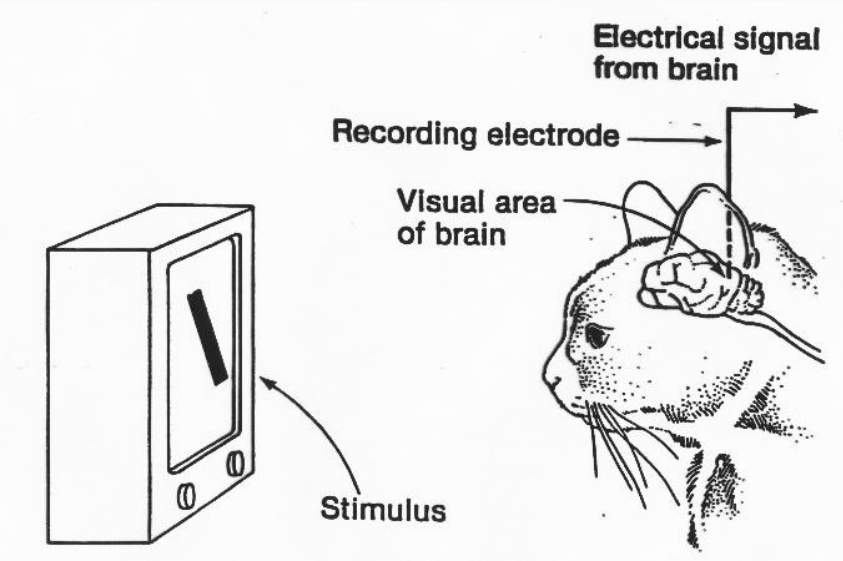


What sort of processing happens in the primary cortex?

Lots of edge detection!

EDGE DETECTORS IN CATS

Huber + Wiesel, 1959: “Receptive fields of single neurones in the cat’s striate cortex.” Won Nobel prize in 1981.



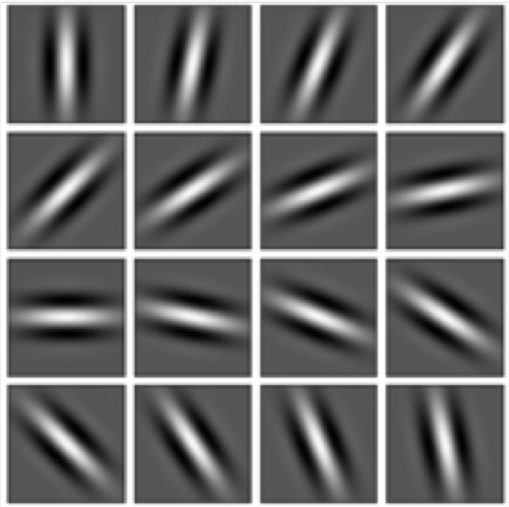
Different neurons fire when the cat is presented with stimuli at different angles. Cool video at `https://www.youtube.com/watch?v=OGxVfKJqX5E`.

”What the Frog’s Eye Tells the Frog’s Brain”, Lettvin et al. 1959. Found explicit edge detection circuits in a frogs visual cortex.

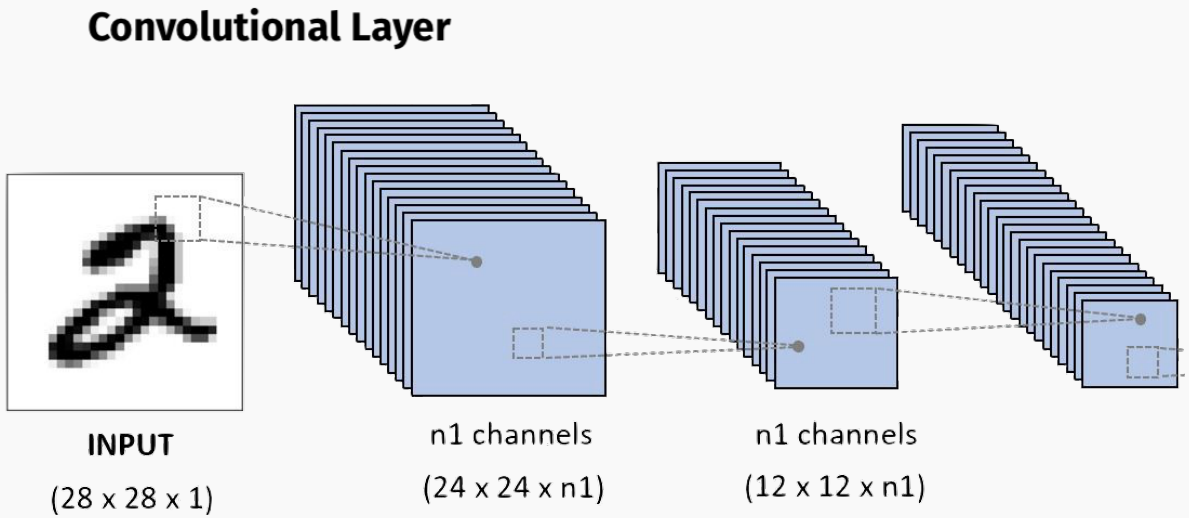
EXPLICIT FEATURE ENGINEERING

State of the art until $\sim$ 10 years ago:

- Convolve image with edge detection filters at many different angles.
- Hand engineer features based on the responses.
- **SIFT** and **HOG** features were especially popular.



Neural network approach: Learn the parameters of the convolution filters based on training data.

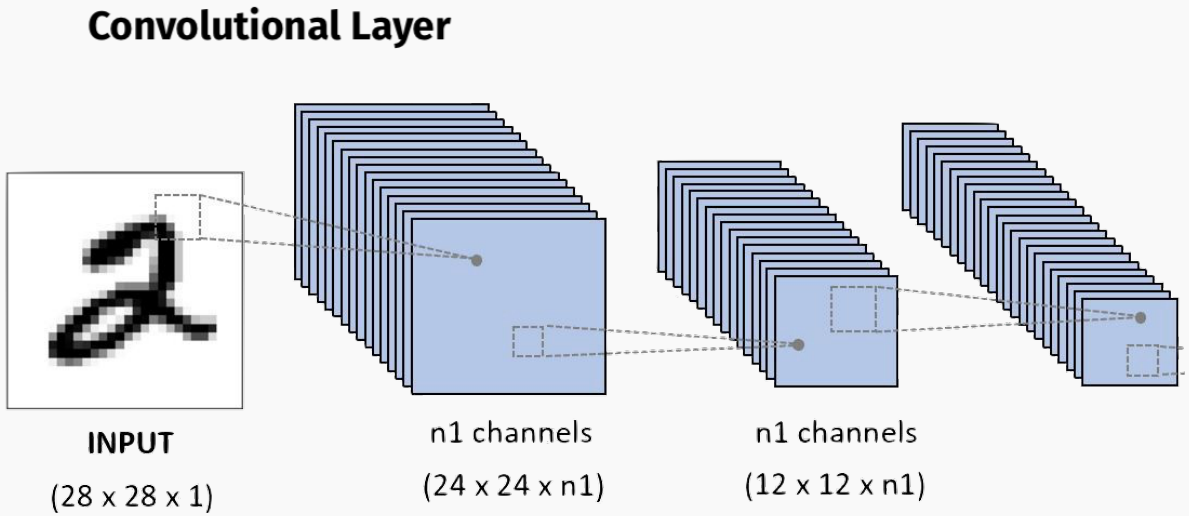


First convolutional layer involves $n1$ convolution filters $\mathbf{W}_1, \ldots, \mathbf{W}_{n1}$. Each is small, e.g. 5×5. Every entry in $\mathbf{W}_i$ is a free parameter: $\sim 25 \cdot n1$ parameters to learn.

Produces $n1$ matrices of hidden variables: i.e. a tensor with depth $n1$.

A fully connected layer that extracts the same feature would require $(28 \cdot 28 \cdot 24 \cdot 24) \cdot n1 = 451,584 \cdot n1$ parameters.

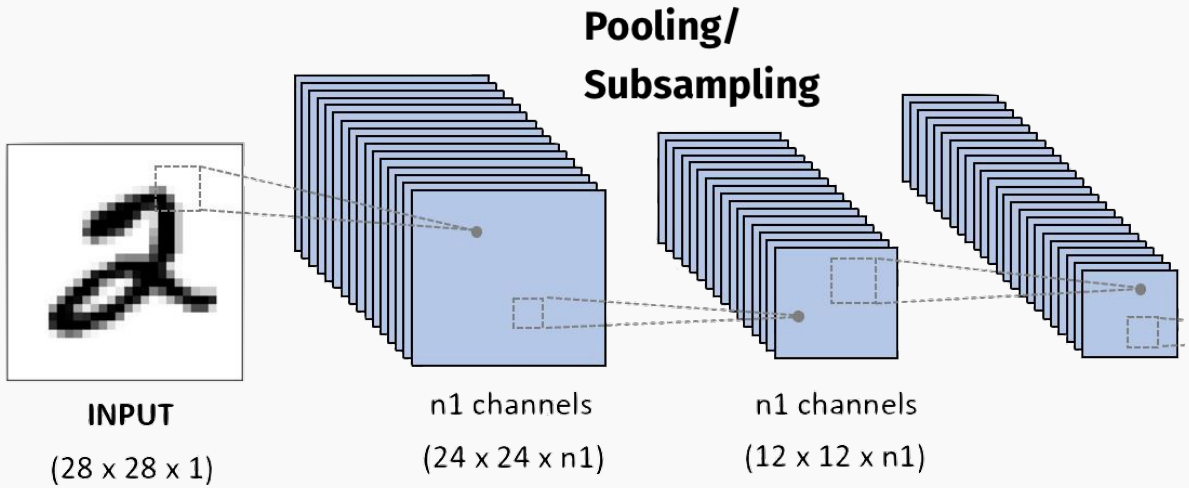
By "baking in" knowledge about what type of features matter, we greatly simply the network.



Each of the $n1$ ouputs is typically processed with a **non-linearity**. Most commonly a Rectified Linear Unity (ReLU): $x = \max(\bar{x}, 0)$.

POOLING AND DOWNSAMPLING

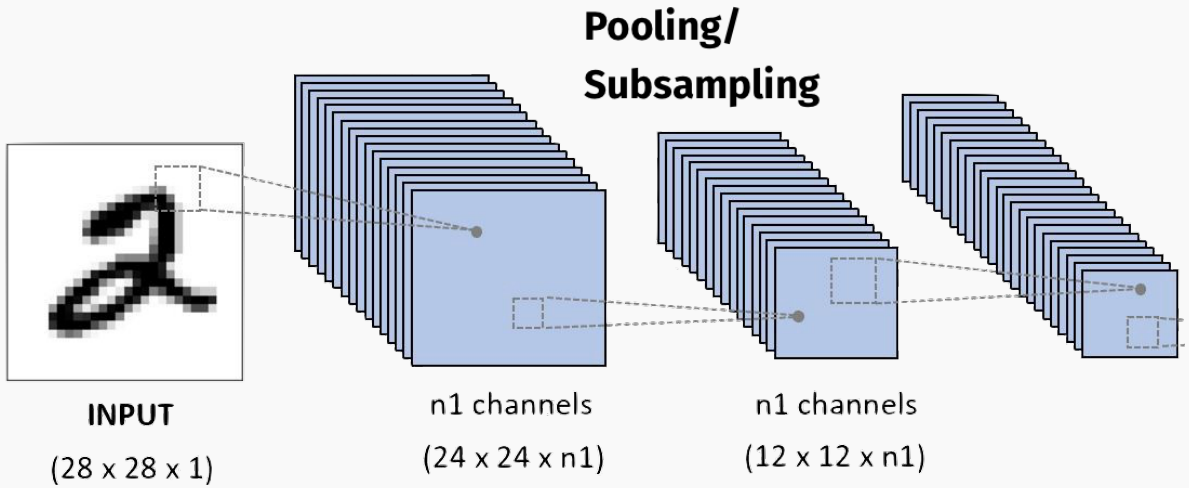
Convolution + non-linearity are typically followed by a layer which performs **pooling + down-sampling**.



Most common approach is max-pooling.

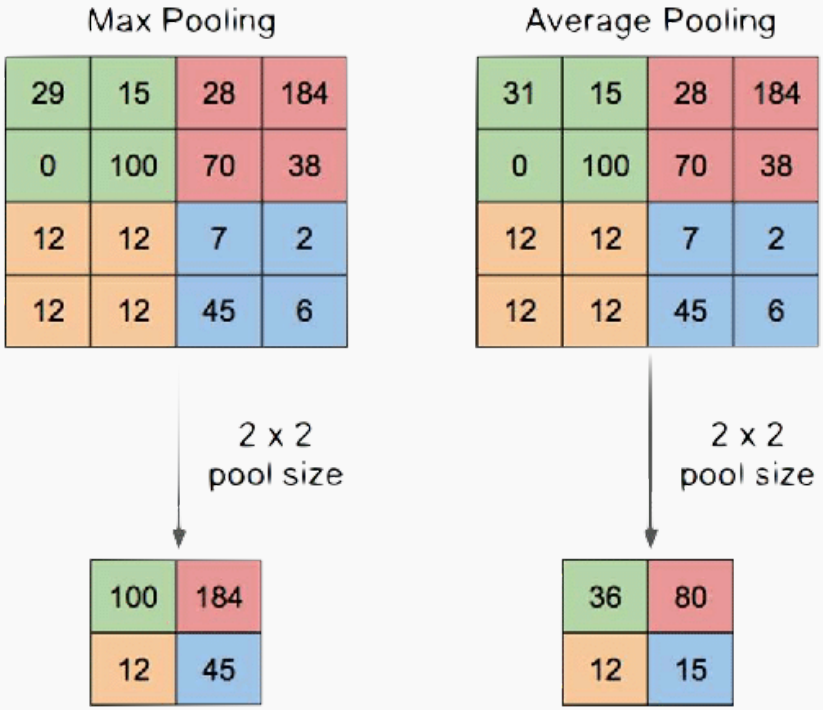
POOLING AND DOWNSAMPLING

Convolution + non-linearity are typically followed by a layer which performs **pooling + down-sampling**.

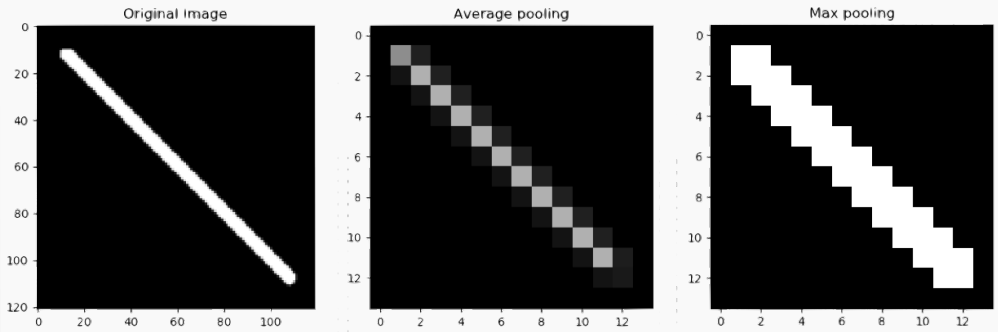


Most common approach is max-pooling.

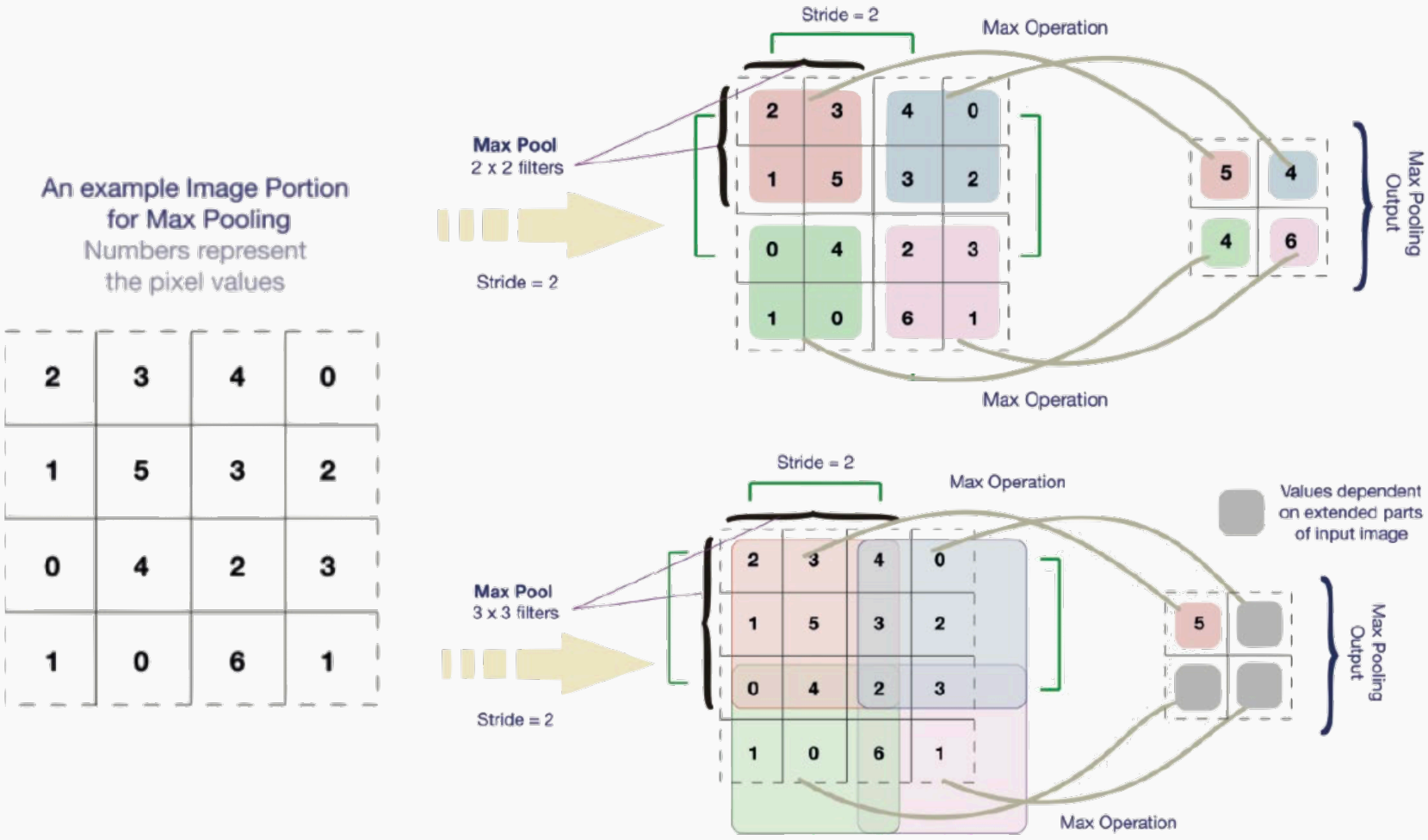
POOLING AND DOWNSAMPLING



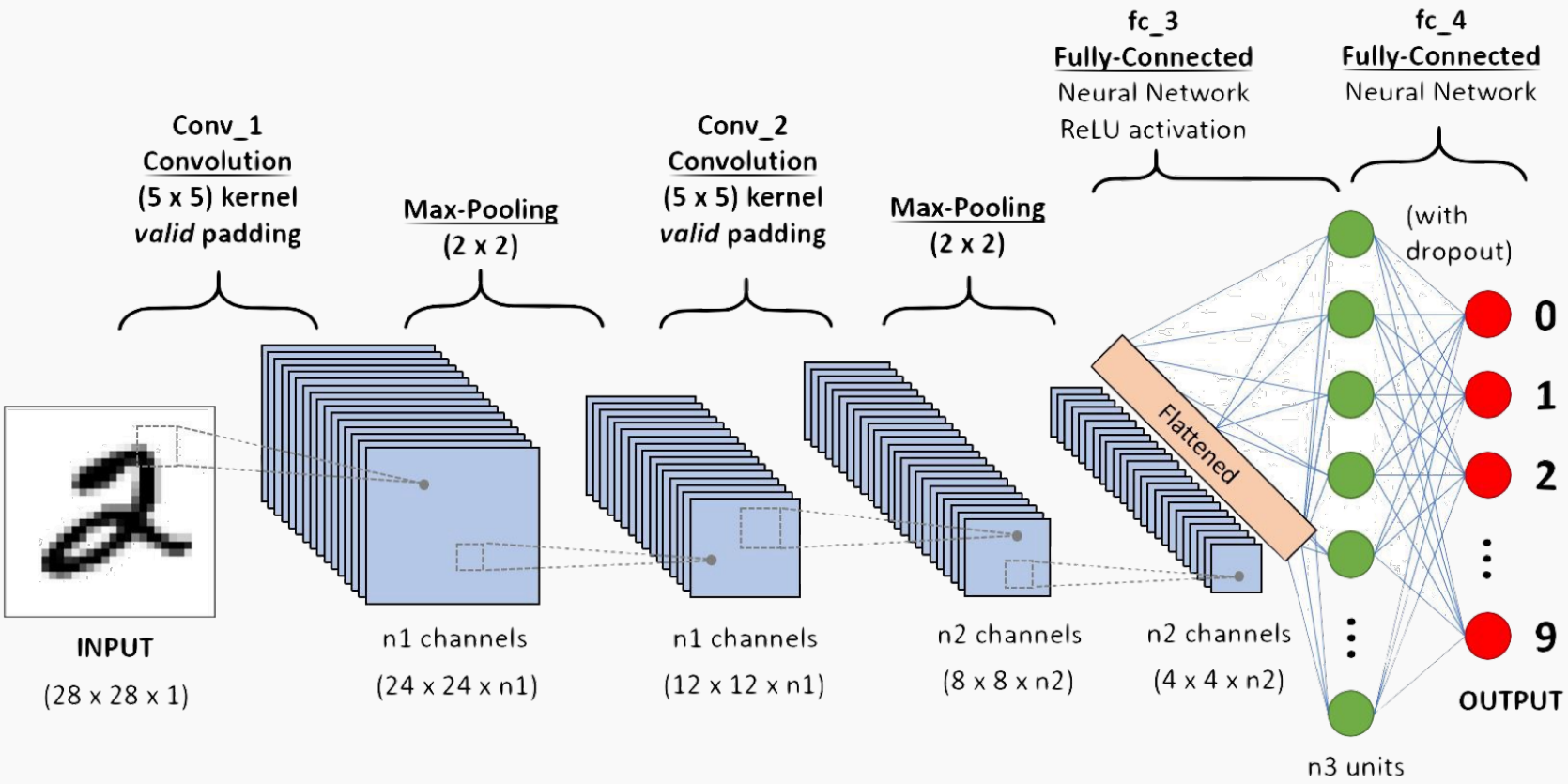
- Reduces number of variables, helps prevent over-fitting and speed up training.
- Helps “smooth” result of convolutional filters.
- Improves shift-invariance.



Many possible variations on standard 2x2 max-pooling.



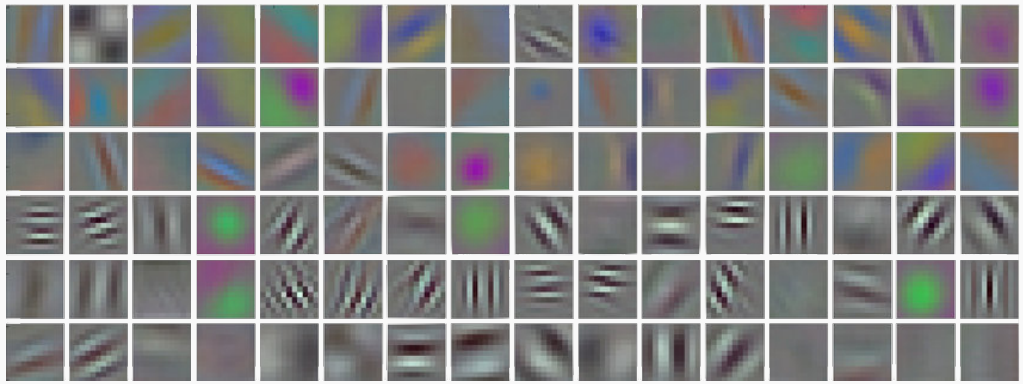
OVERALL NETWORK ARCHITECTURE



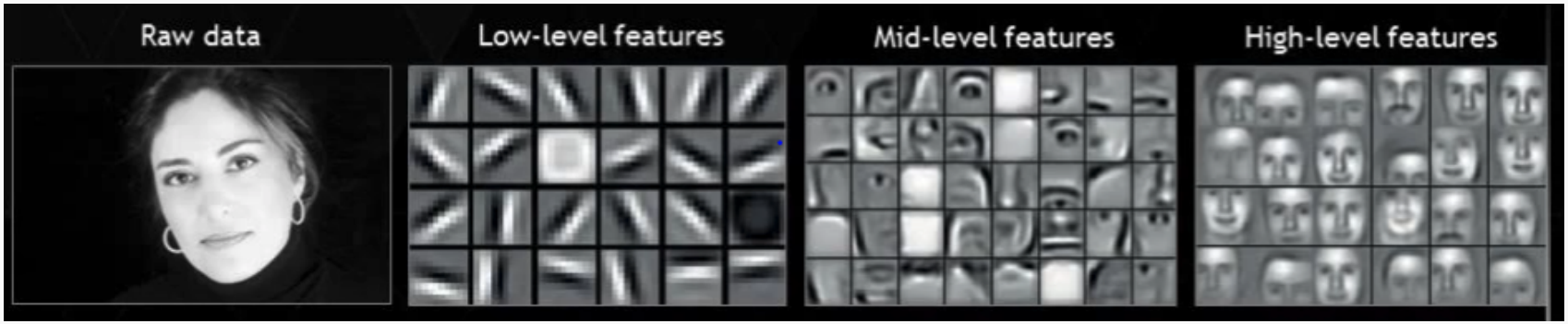
Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

UNDERSTANDING LAYERS

What type of convolutional filters do we learn from gradient descent?
Lots of edge detectors in the first layer!

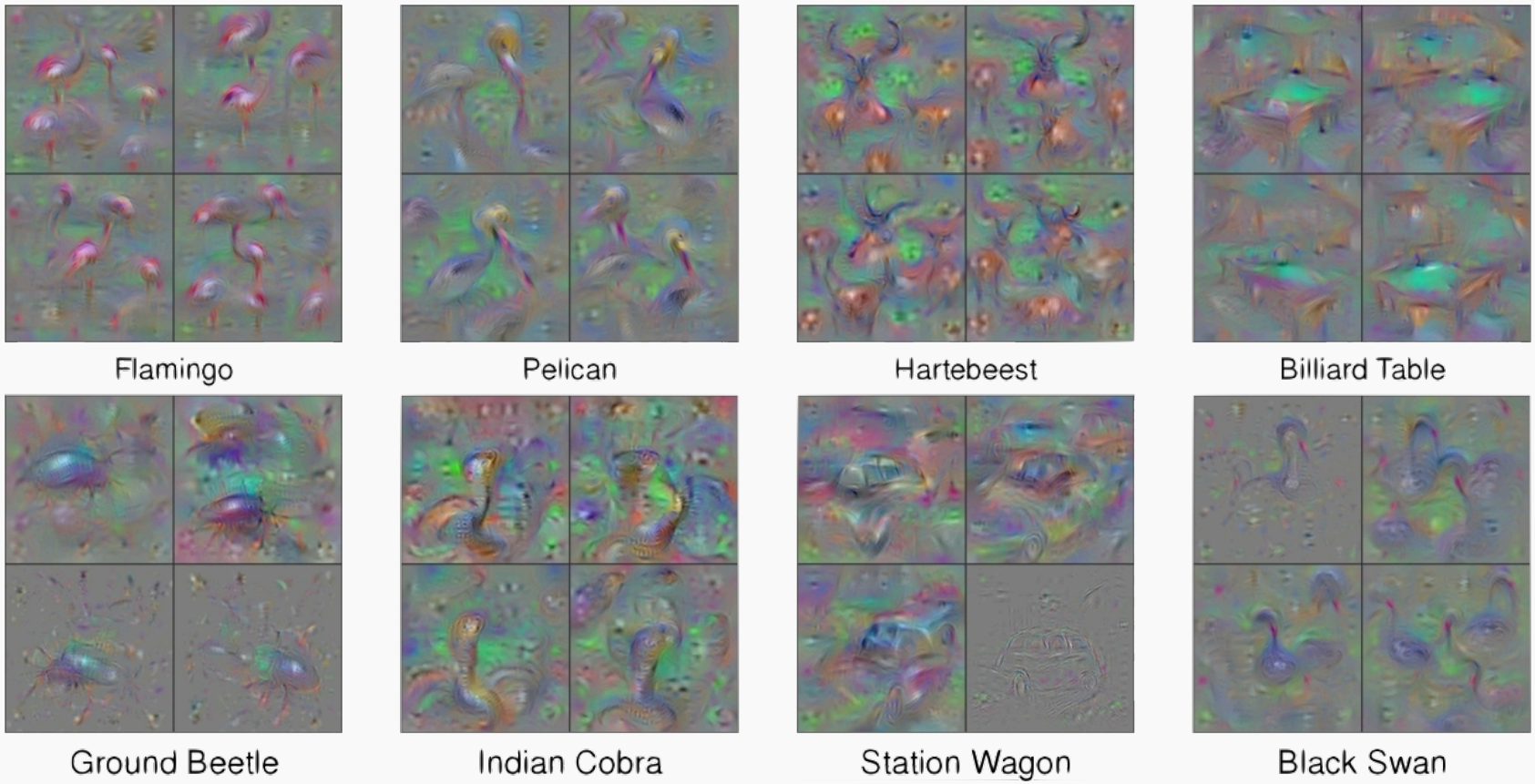


Other layers are harder to understand… but the hypothesis is that hidden variables later in the network encode for "higher level features":



UNDERSTANDING LAYERS

Technique to probe later neurons: Use optimization to find images that most strongly "activate" a given neuron deep in the network.



"Understanding Neural Networks Through Deep Visualization", Yosinski et al.

TRICKS OF THE TRADE

Beyond techinques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training convolutional networks requires a lot of “tricks”.

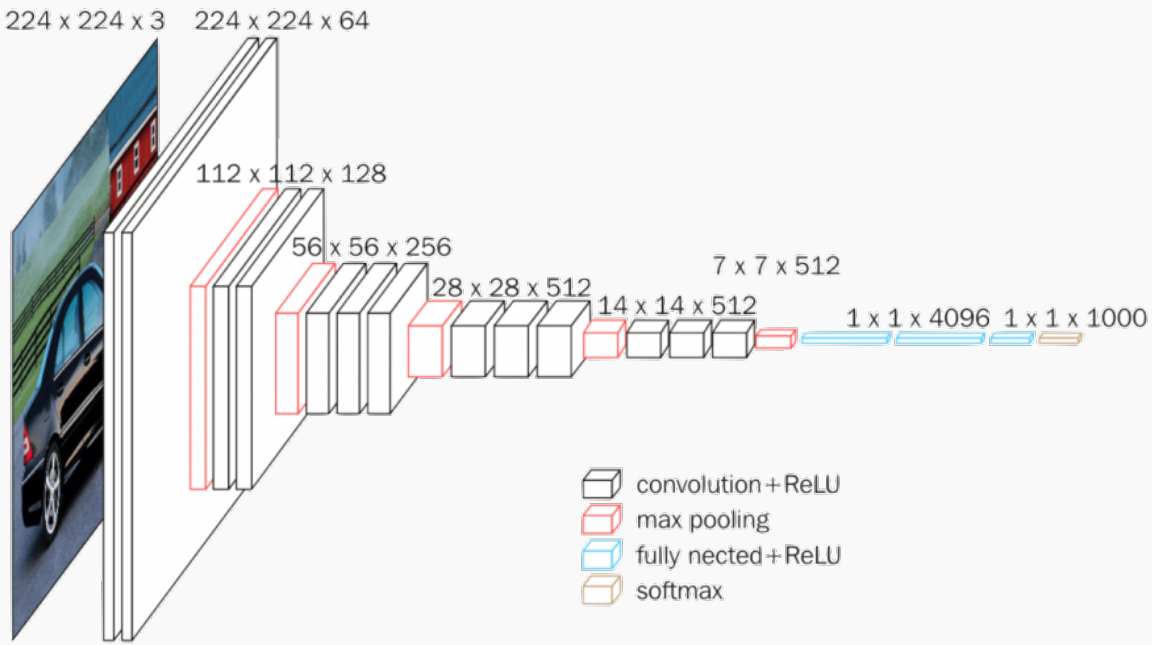
- Batch normalization (accelerate training).
- Dropout (prevent over-fitting)
- Residual connections (accelerate training, allow for more depth – 100s of layers).

And convolutional networks require lots of training data.

TRANSFER LEARNING

What if you want to apply deep convolutional networks to a problem where you don't have a lot of data?

Idea behind transfer learning: features transformations learned when training a classifier on e.g. Imagenet are often useful in other problems, even with different inputs, classes, etc.

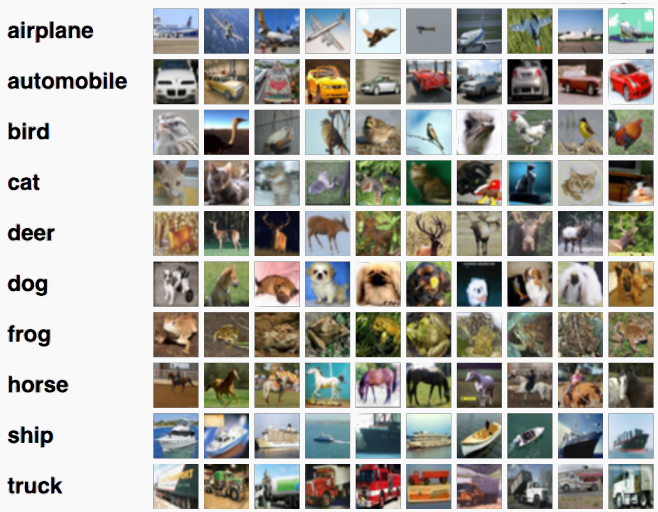


- Download state of the art pre-trained network (Alexnet, VGG, Inception, etc.)
- Chop off classification layer.
- Use first part of network as feature extractor.
- Solve classification problem using more scalable methods: kernel SVM, logistic regression, shallow fully connected net, etc.

Very easy to do in Tensorflow/Keras. Many pre-trained networks are made available.

Two demos to be released shortly:

- Classification of CIFAR-10 dataset using 2 layer neural nets in Keras. You will likely want to use Google Collab to access a GPU.



- Transfer learning in Keras.