CS-GY 6923: Lecture 10
Convolutional Neural Networks, Adversarial Examples

NYU Tandon School of Engineering, Akbar Rafiey
Slides by Prof. Christopher Musco

Recap from last lecture

Backprop

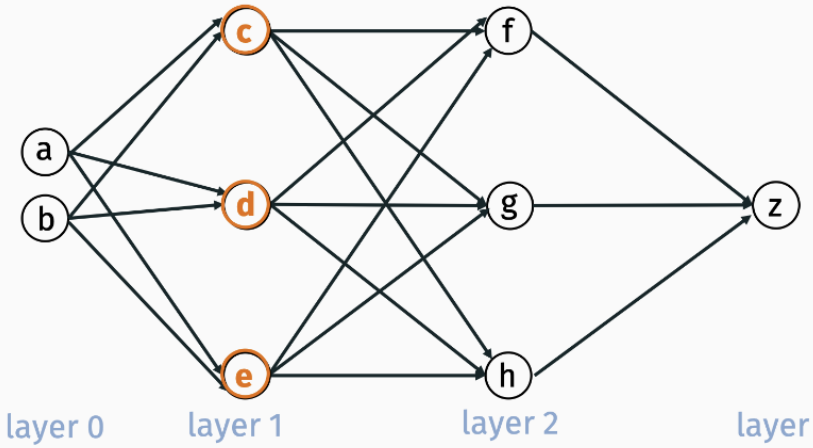
For any feed-forward neural network with d parameters:

- Backpropagation can be used to compute derivatives with respect to one particular input in $O(d)$ time.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be performed quickly on a GPU.

Allows for very fast implementation of Stochastic Gradient Descent for training neural networks.

Backprop example

Step 2: Backward pass. Next layer.



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

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

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

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

Backprop linear algebra

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.

Backprop linear algebra

Linear algebraic view.

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

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

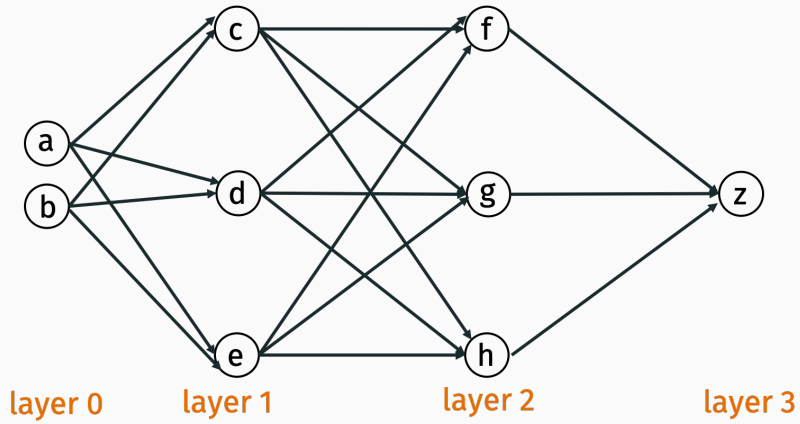
Let $\bar{\boldsymbol{\delta}}_i$ be a vector containing $\partial z / \partial \bar{j}$ for all nodes j in layer i.

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

Note: $\bar{\boldsymbol{\delta}}_i = s'(\bar{\mathbf{v}}_i) \times \boldsymbol{\delta}_i$ where $\times$ denotes entrywise multiplication.

Backprop linear algebra

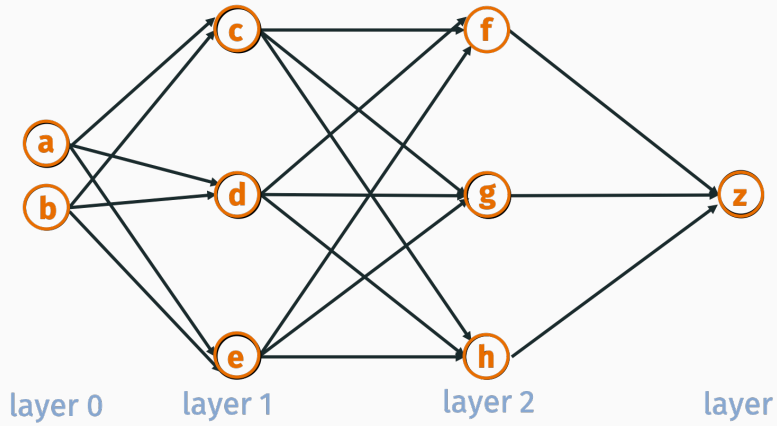
Let $\mathbf{W}_i$ be a matrix containing all the weights for edges between layer i and layer $i+1$.



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

$$\mathbf{W}_0 = \begin{bmatrix} W_{a,c} & W_{b,c} \\ W_{a,d} & W_{b,d} \\ W_{a,e} & W_{b,e} \end{bmatrix}$$

Backprop linear algebra

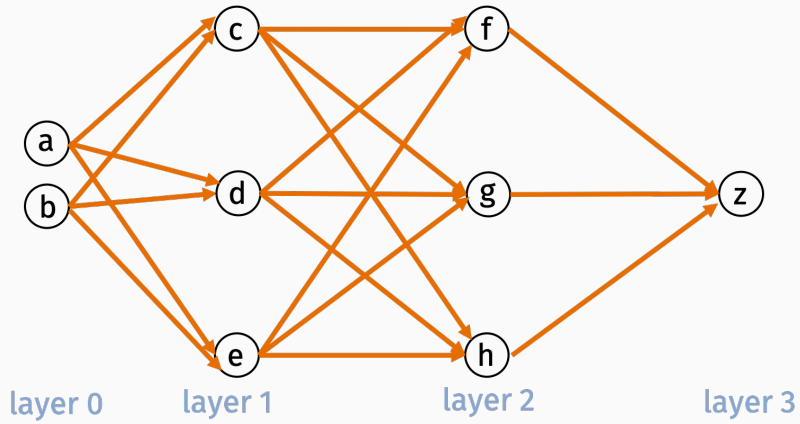


Claim 1: Node derivative computation is matrix multiplication.

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

Backprop linear algebra

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

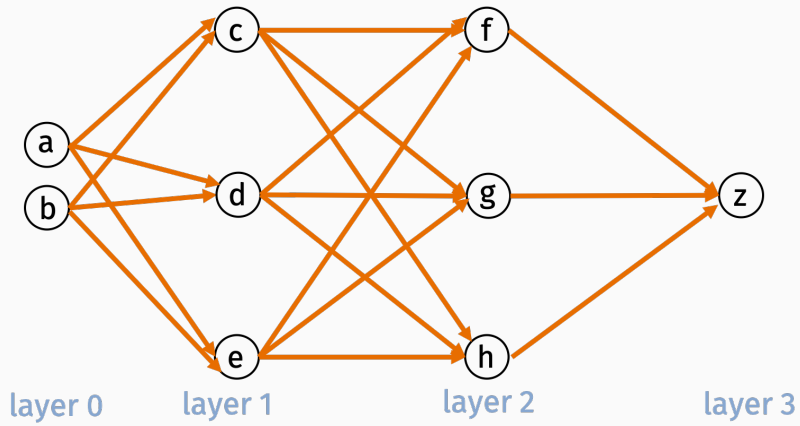


$$\mathbf{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

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

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

Backprop linear algebra



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

$$\mathbf{\Delta}_i = \mathbf{v}_i \delta_{i+1}^T.$$

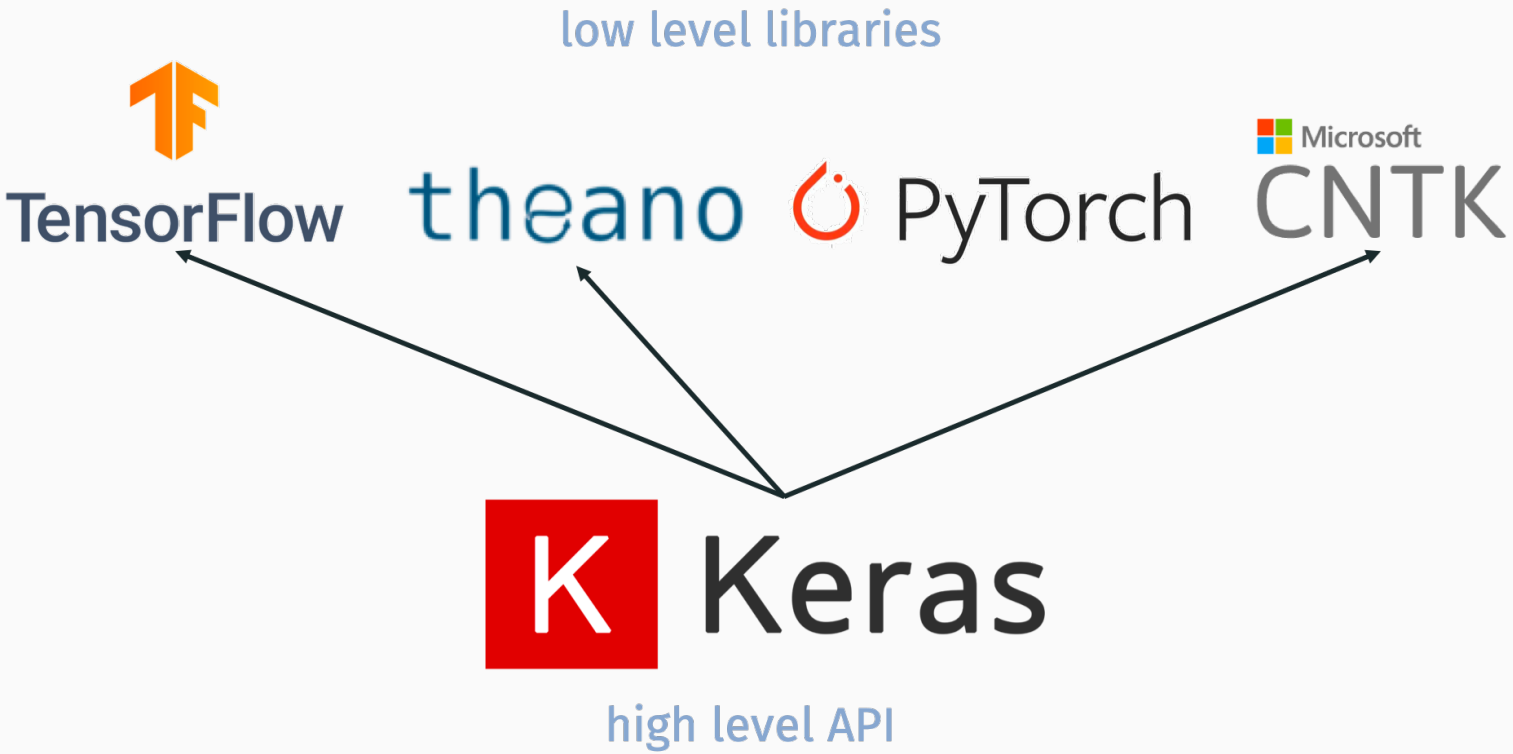
Neural network demos

Two demos will be uploaded on neural networks:

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

Please spend some time working through these.

Neural network software



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

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

Neural network software

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))
```

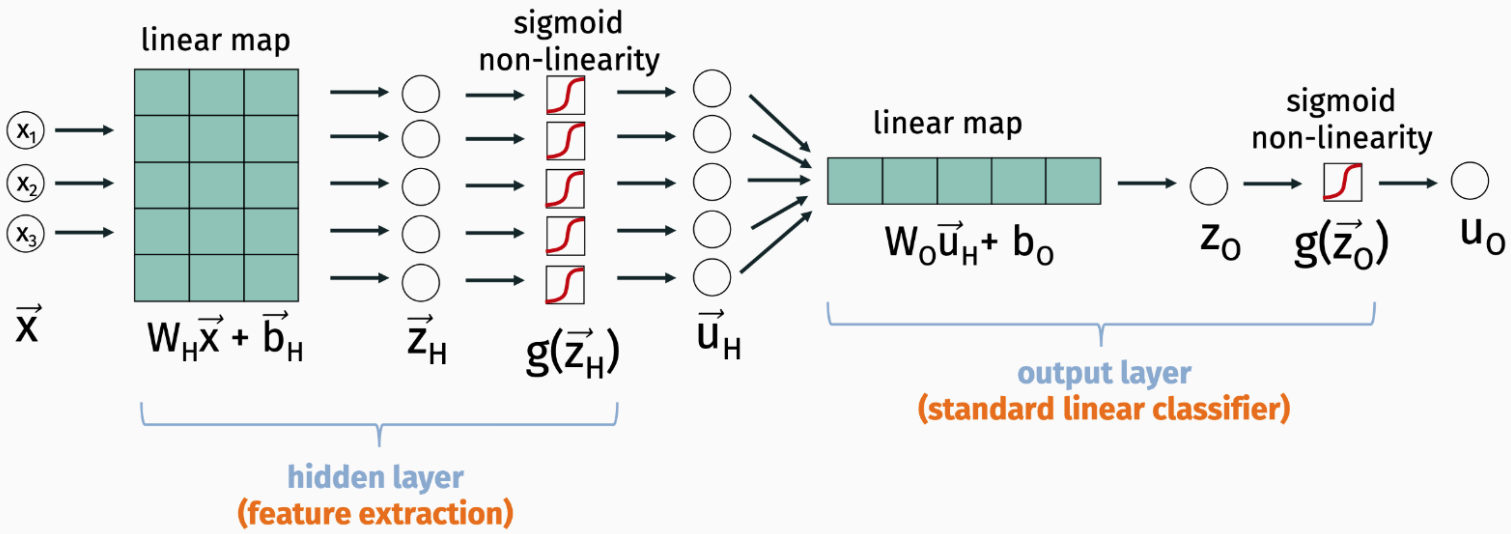
Feature extraction

Why do neural networks work so well?

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

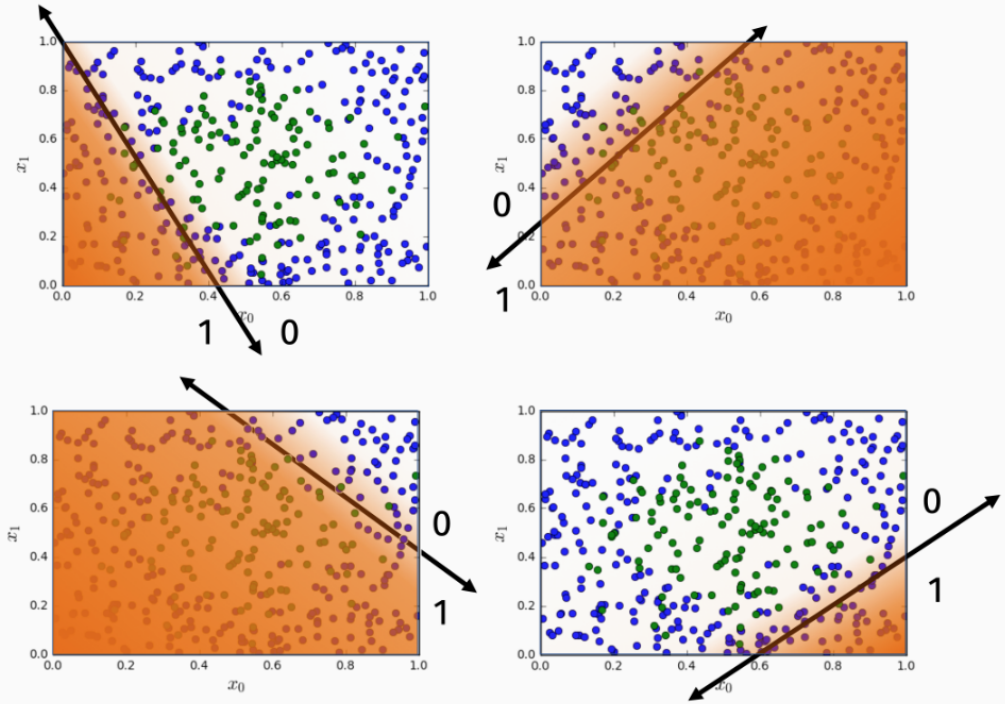
But sometimes they still need a nudge in the right direction...

Basic feature extraction



Basic feature extraction

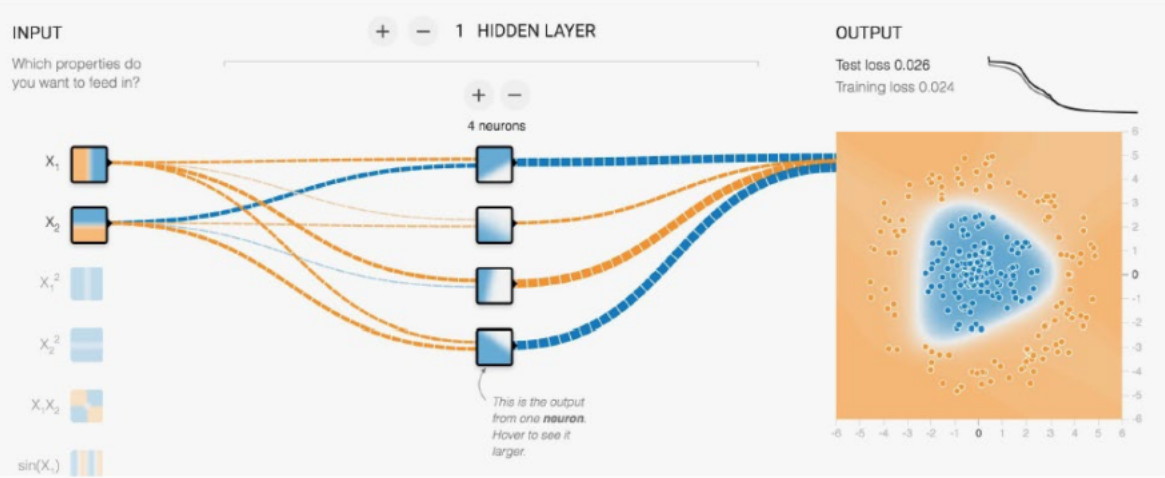
Sigmoid activation: Each hidden variable z_i equals $\frac{1}{1+e^{-z_i}}$ where $z_i = \mathbf{w}^T\mathbf{x} + b$ for input $\mathbf{x}$.



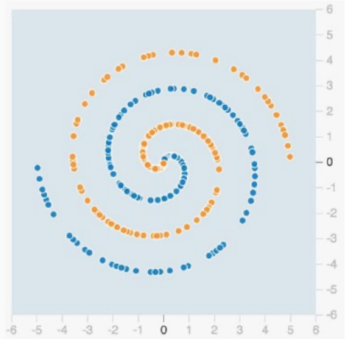
Other non-linearities yield similar features.

Basic feature extraction

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

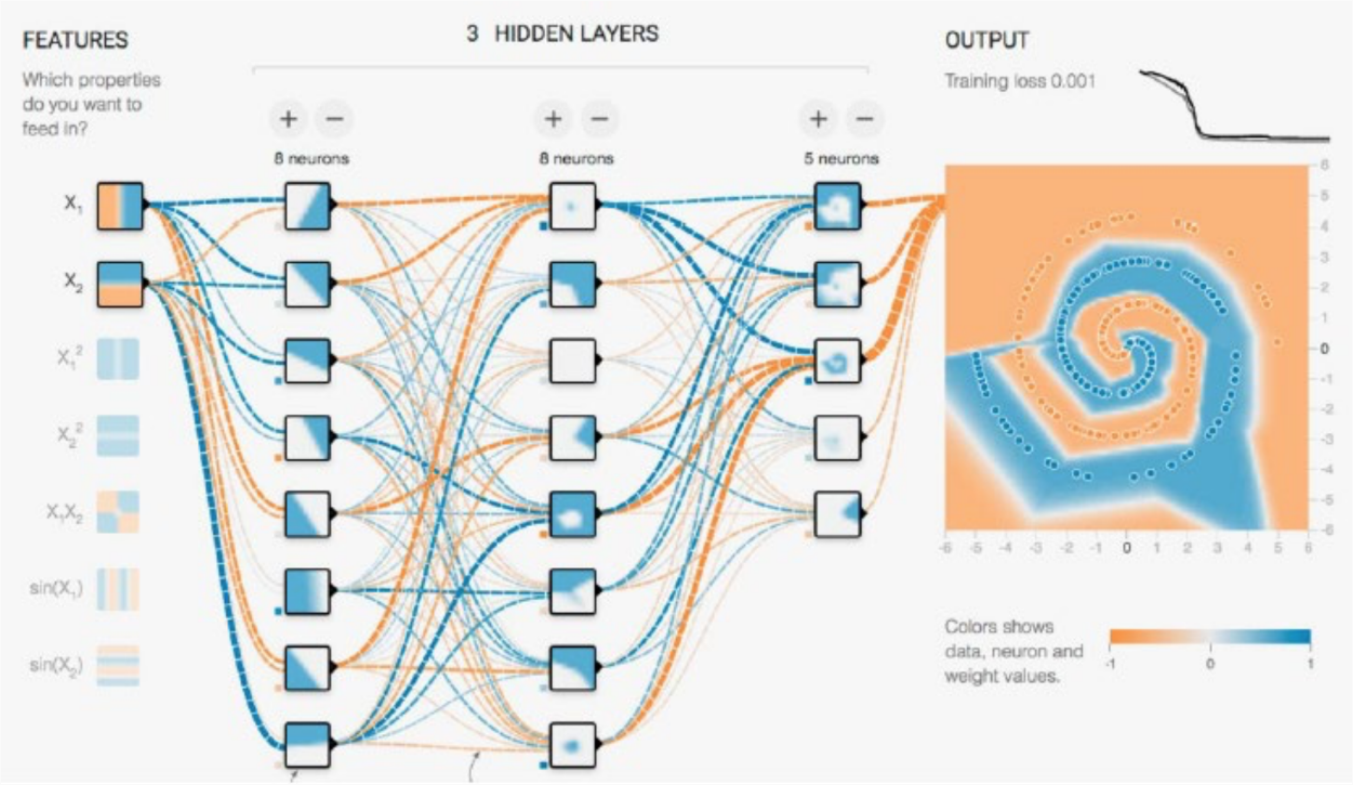


What about even more complex datasets?



Basic feature extraction

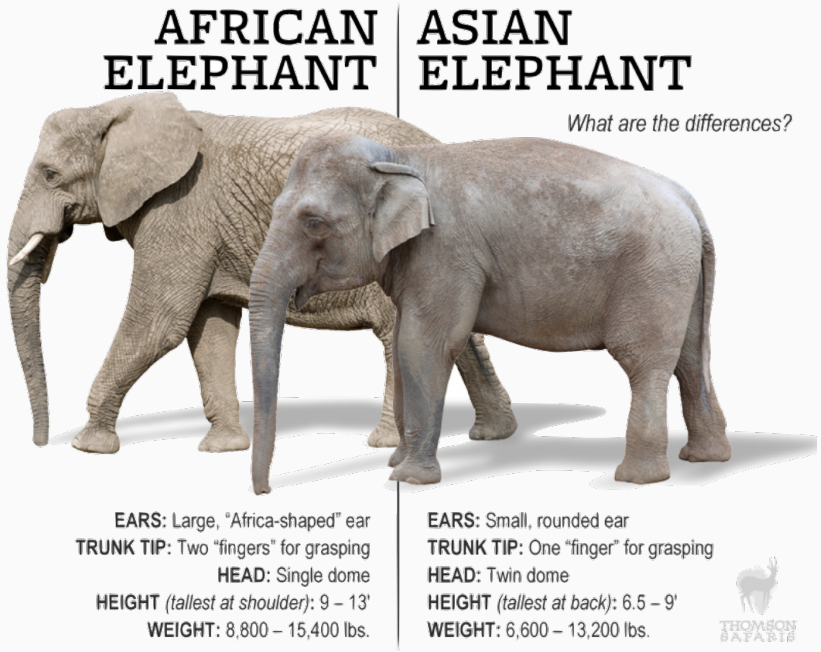
With more layers, complexity starts ramping up:



But there is a limit...

Basic feature extraction

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



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

Convolutional feature extraction

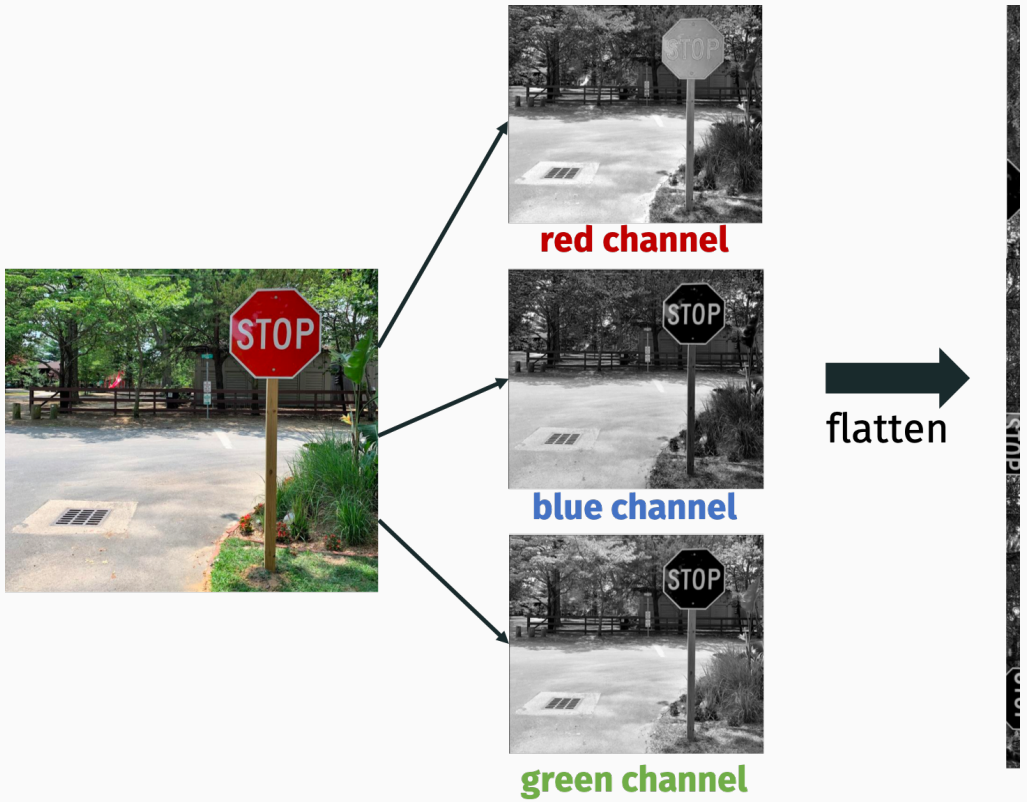
Today's topic: Understand why convolution is a powerful way of extracting features from image data. Also super valuable for

- 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])

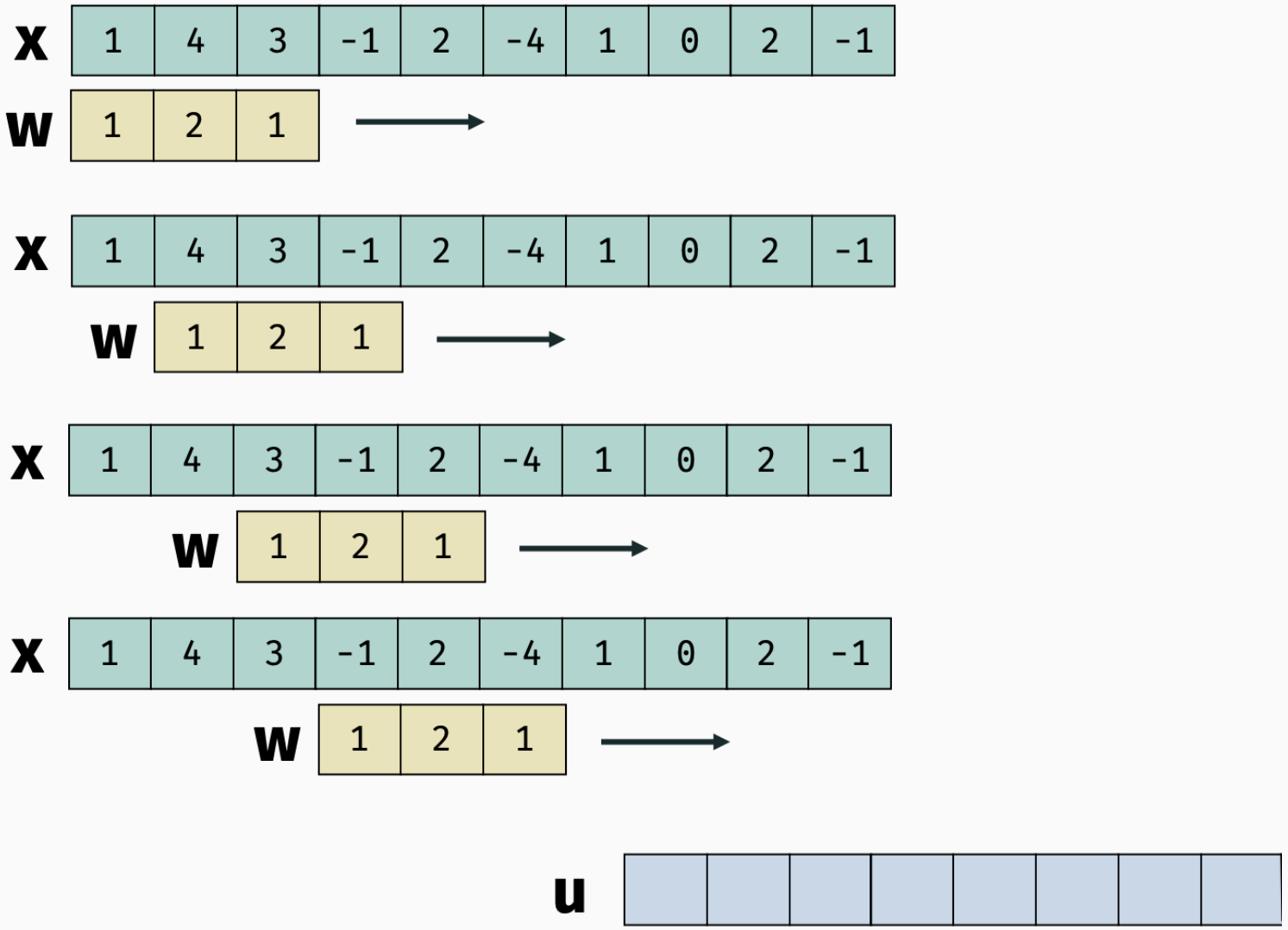
Given $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{w} \in \mathbb{R}^k$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $d - k + 1$ vector with:

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

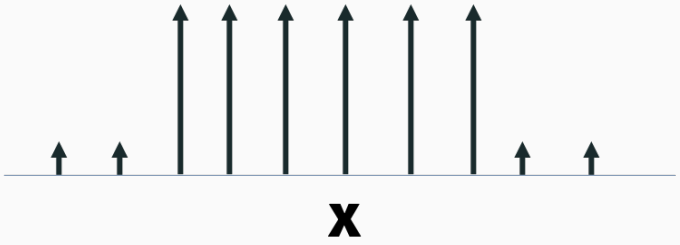
Think of $\mathbf{x} \in \mathbb{R}^d$ as long **data vector** (e.g. d = 512) and $\mathbf{w} \in \mathbb{R}^k$ as short **filter vector** (e.g. k = 8). $\mathbf{u} = [\mathbf{x} \circledast \mathbf{w}]$ is a feature transformation.

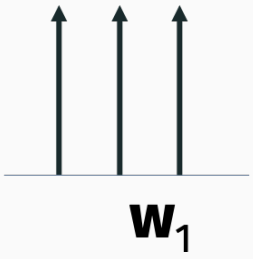
[1] This is slightly different from the definition of convolution you might have seen in a Digital Signal Processing class because $\mathbf{w}$ does not get "flipped". In signal processing our operation would be called correlation.

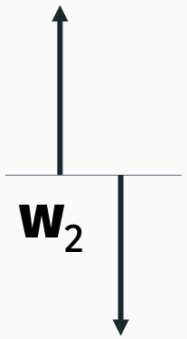
1D convolution

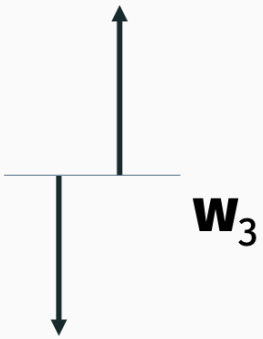


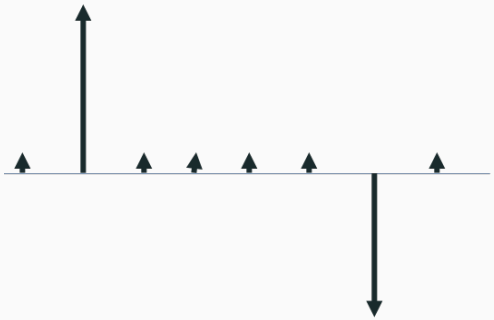
Match the convolution

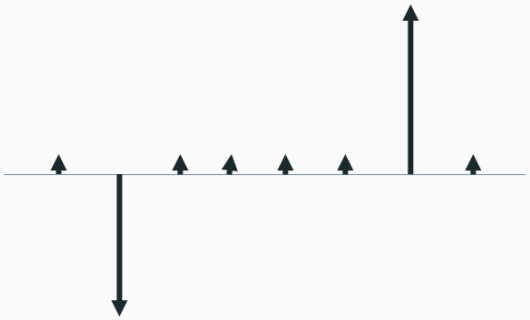


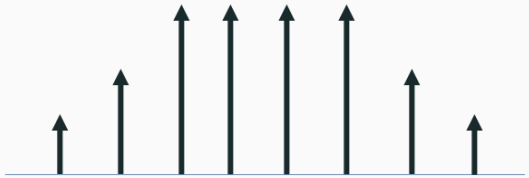












2D convolution

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:

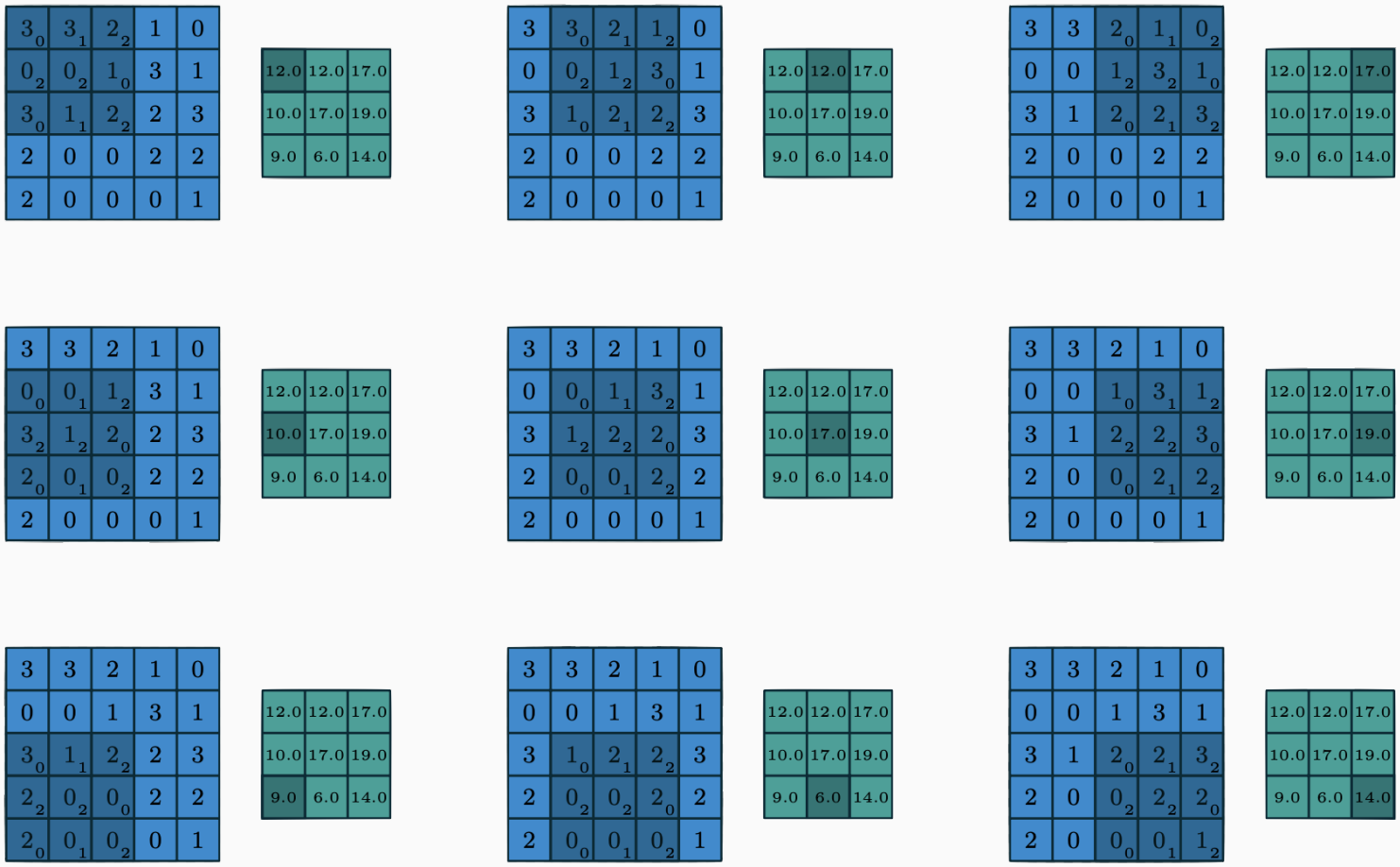
$$[\mathbf{x} \circledast \mathbf{w}]_{i,j} = \sum_{\ell=1}^{k_1} \sum_{h=1}^{k_2} \mathbf{x}_{(i+\ell-1),(j+h-1)} \cdot \mathbf{w}_{\ell,h}$$

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 $\mathbf{x}$.

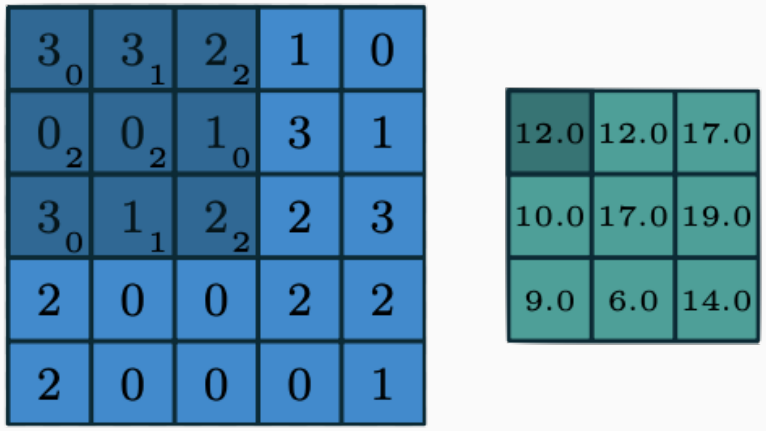
2D convolution

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



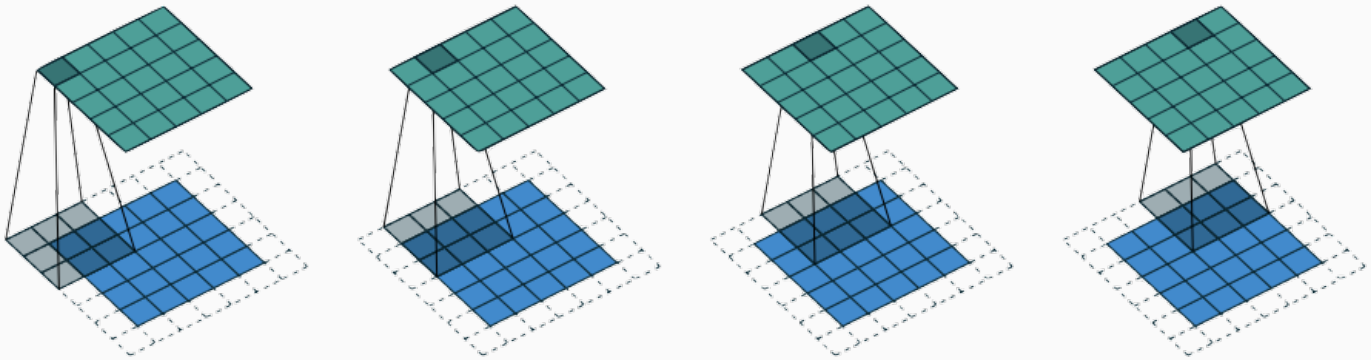
2D convolution

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



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 $(k_2 - 1)/2$.

Applications of convolution

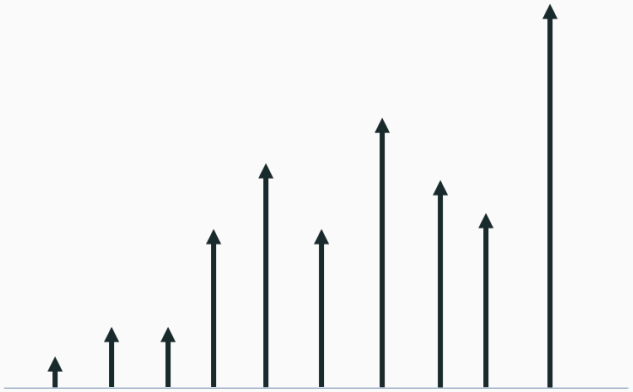
Examples code will be available in `demo1_convolutions.ipynb`.

Application 1: Blurring/smooth.

In one dimension:

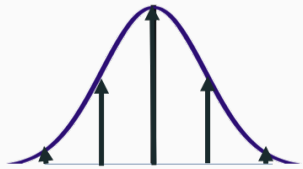
- Uniform (moving average) filter: $\mathbf{w}_i = \frac{1}{k}$ for $i = 1, \ldots, k$.
- Gaussian filter: $\mathbf{w}_i \sim \exp^{(i-k/2)^2/\sigma^2}$ for $i = 1, \ldots, k$.

Smoothing filters

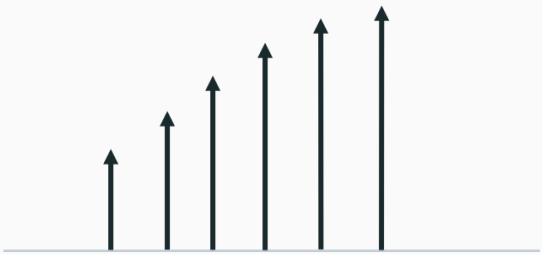


x

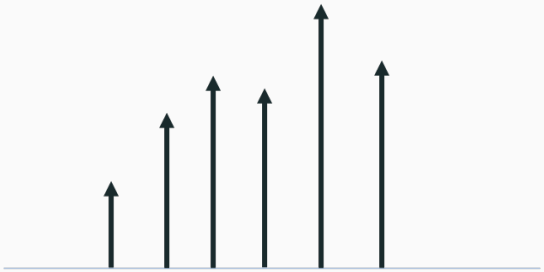
Uniform filter
(moving average)



Gaussian filter



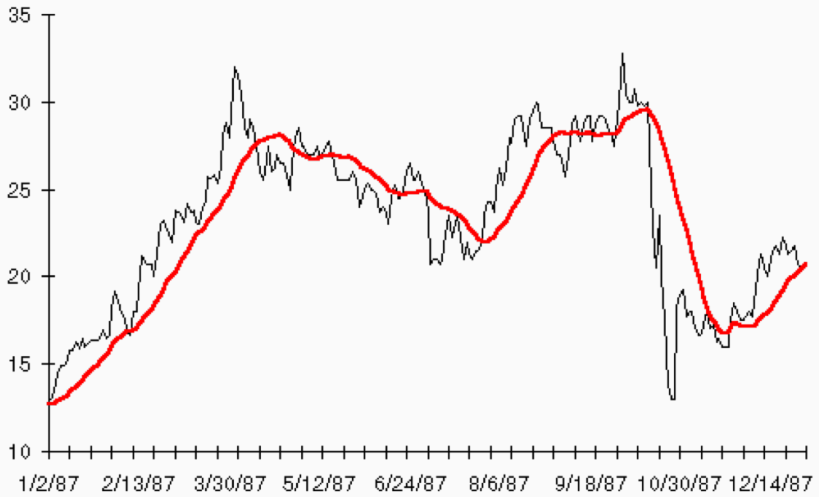
u



u

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

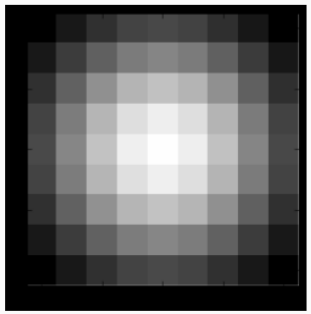
In two dimensions:

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

Larger filter equates to more smoothing.

Smoothing in two dimensions

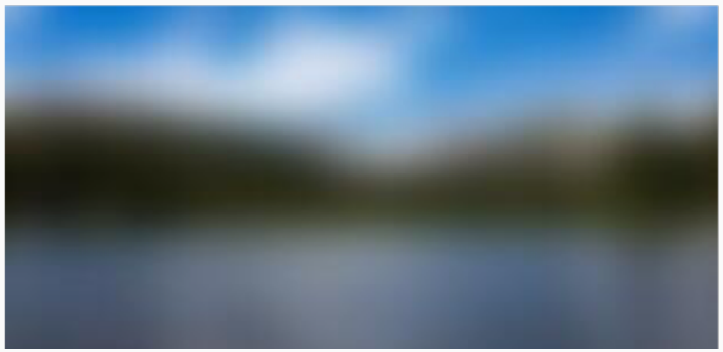
For Gaussian filter, you typically choose $k \gtrsim 2\sigma$ to capture the fall-off of the Gaussian.



Both approaches effectively denoise and smooth images.

Smoothing for feature extraction

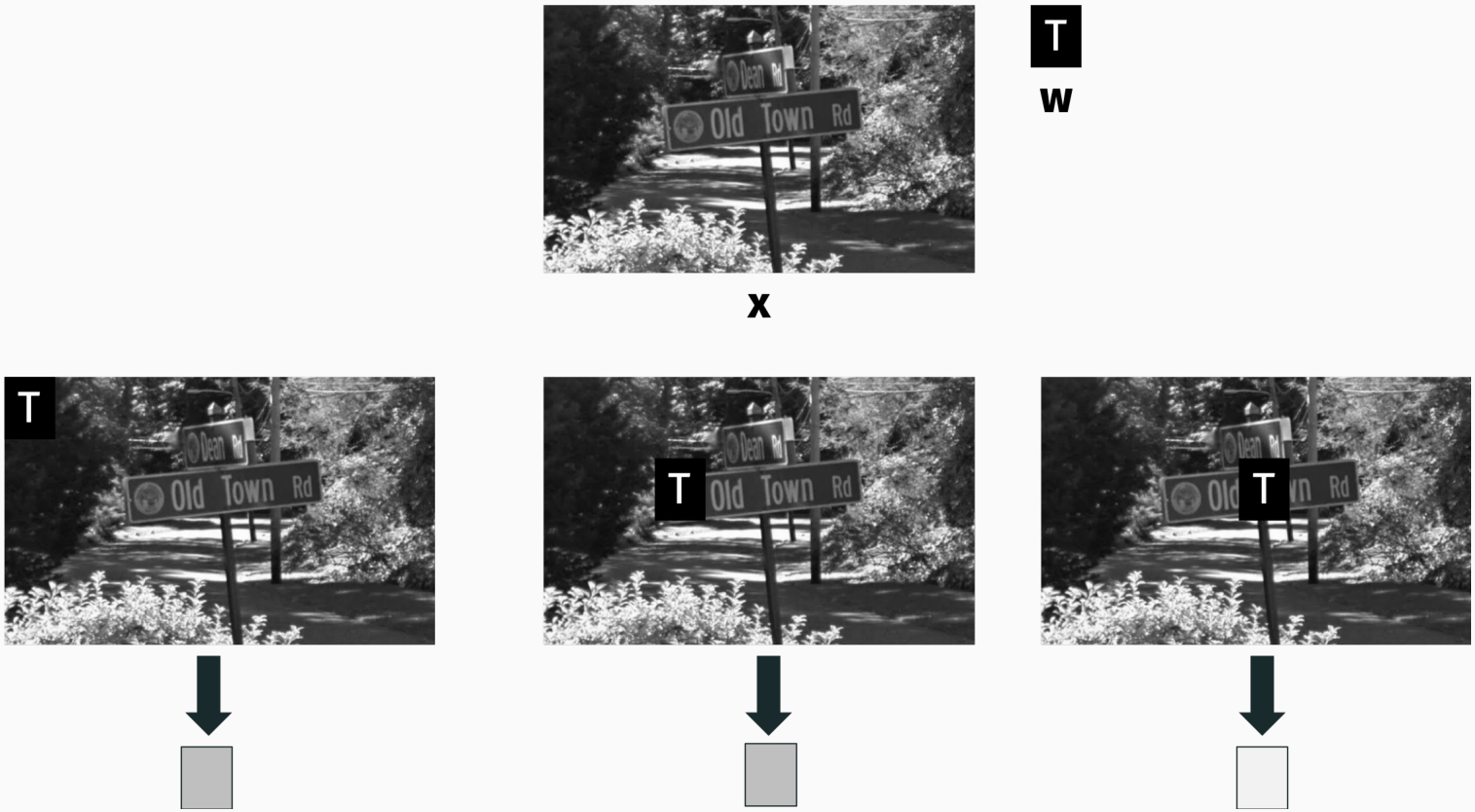
When combined with other feature extractors, smoothing at various levels allows the algorithm to focus on high-level features over low-level features.



Applications of convolution

Application 2: Pattern matching.

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



Local pattern matching

Applications of local pattern matching:

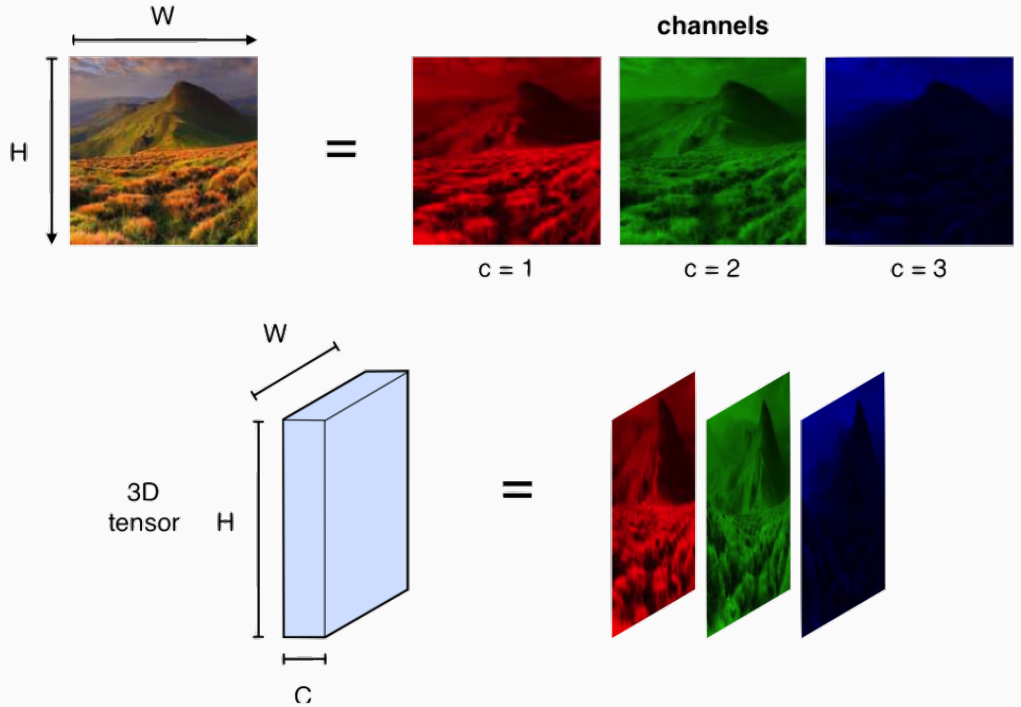
- Check if an image contains text.
- Look for specific sound in audio recording.
- Check for other well-structured objects

3D convolution

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

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

View image as 3D tensor:

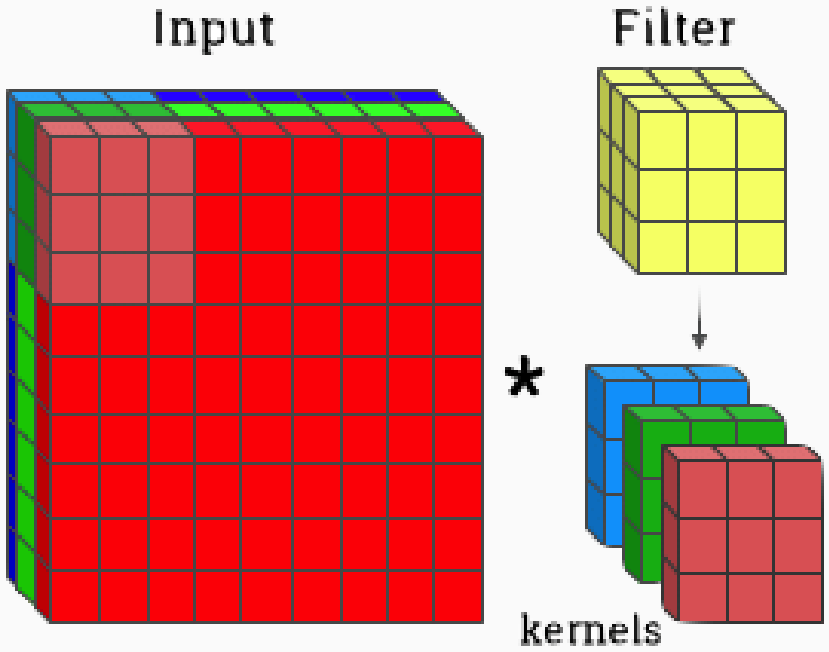


3D convolution

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:

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

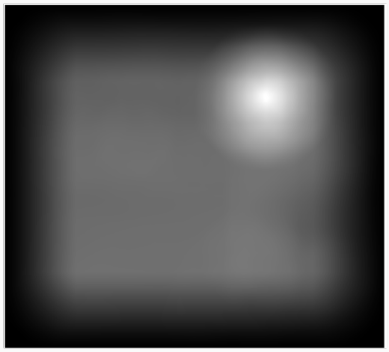


Application 2: pattern matching

More powerful patter matching in color images:

x_1

* w

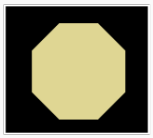


x_2

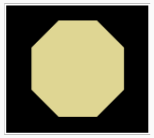
* w

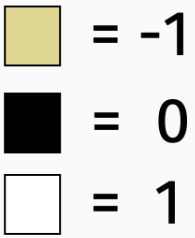
red channel

blue channel



green channel



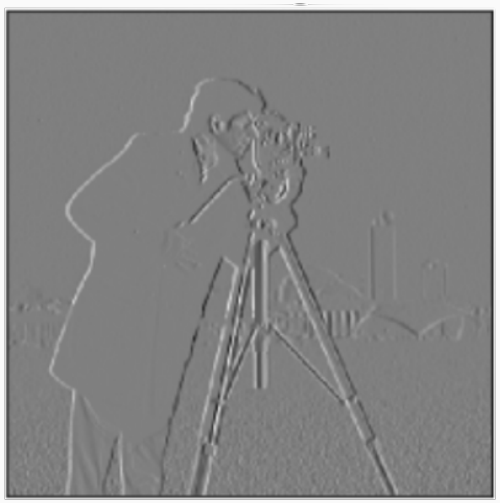


Applications of convolution

Application 3: Edge detection.

These are 2D edge detection filter:

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



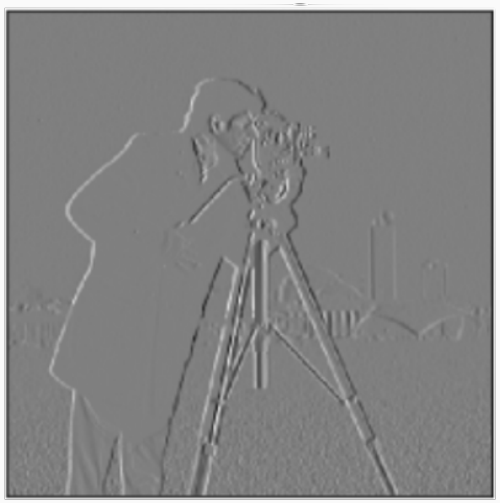
x * ?

x * ?

Applications of convolution

Sobel filter is more commonly used:

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

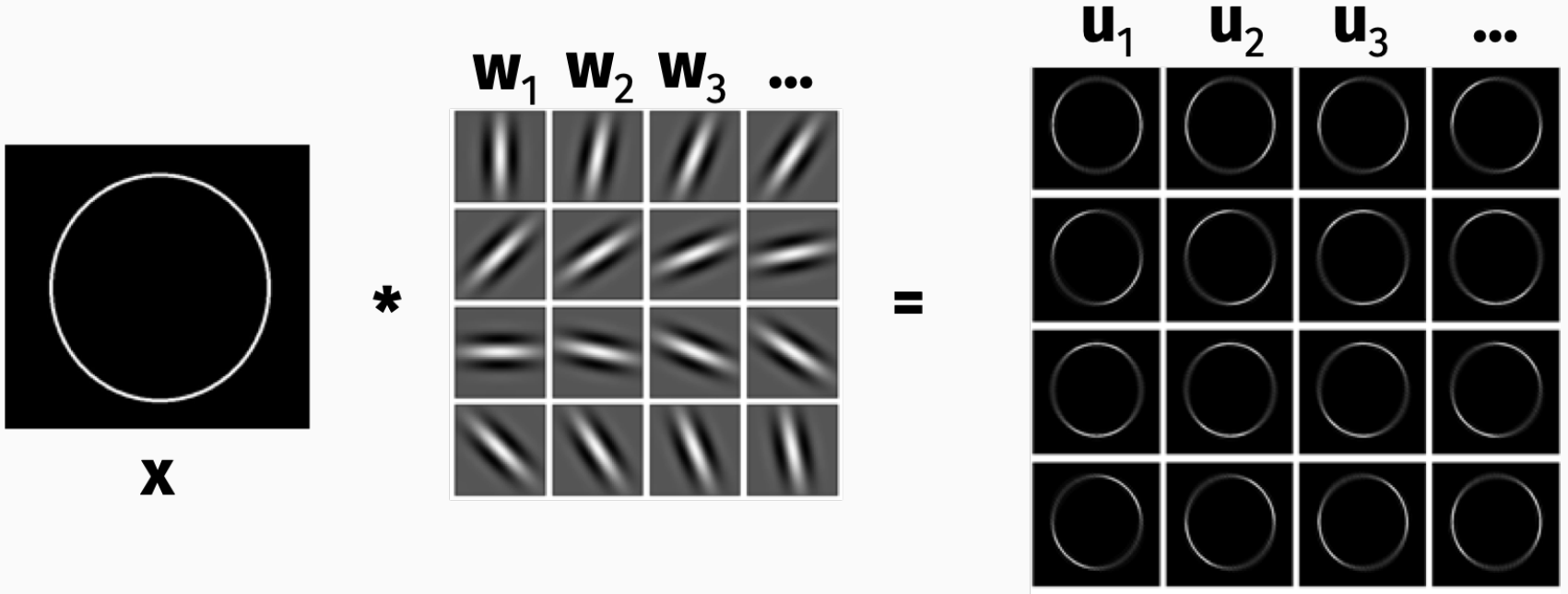


x * ?

x * ?

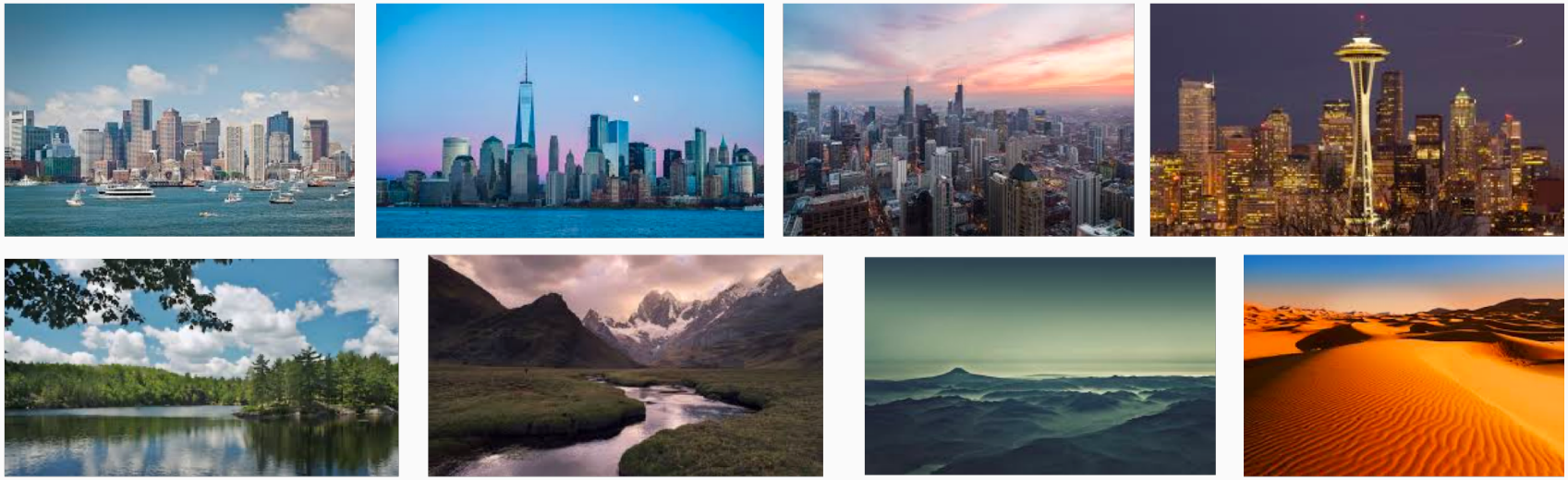
Directional edge detection

Can define edge detection filters for any orientation.

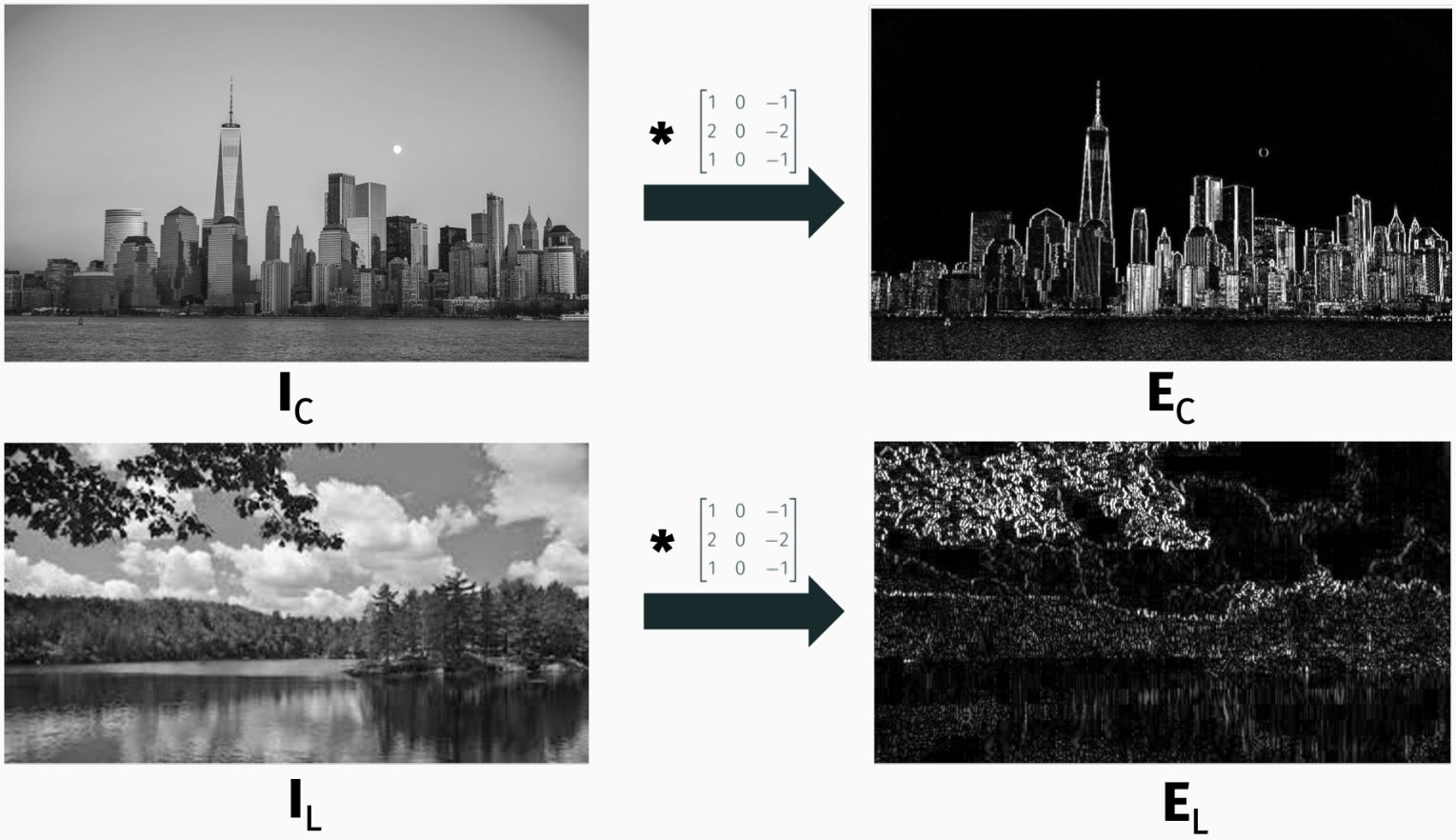


Edge detection

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



Edge detection

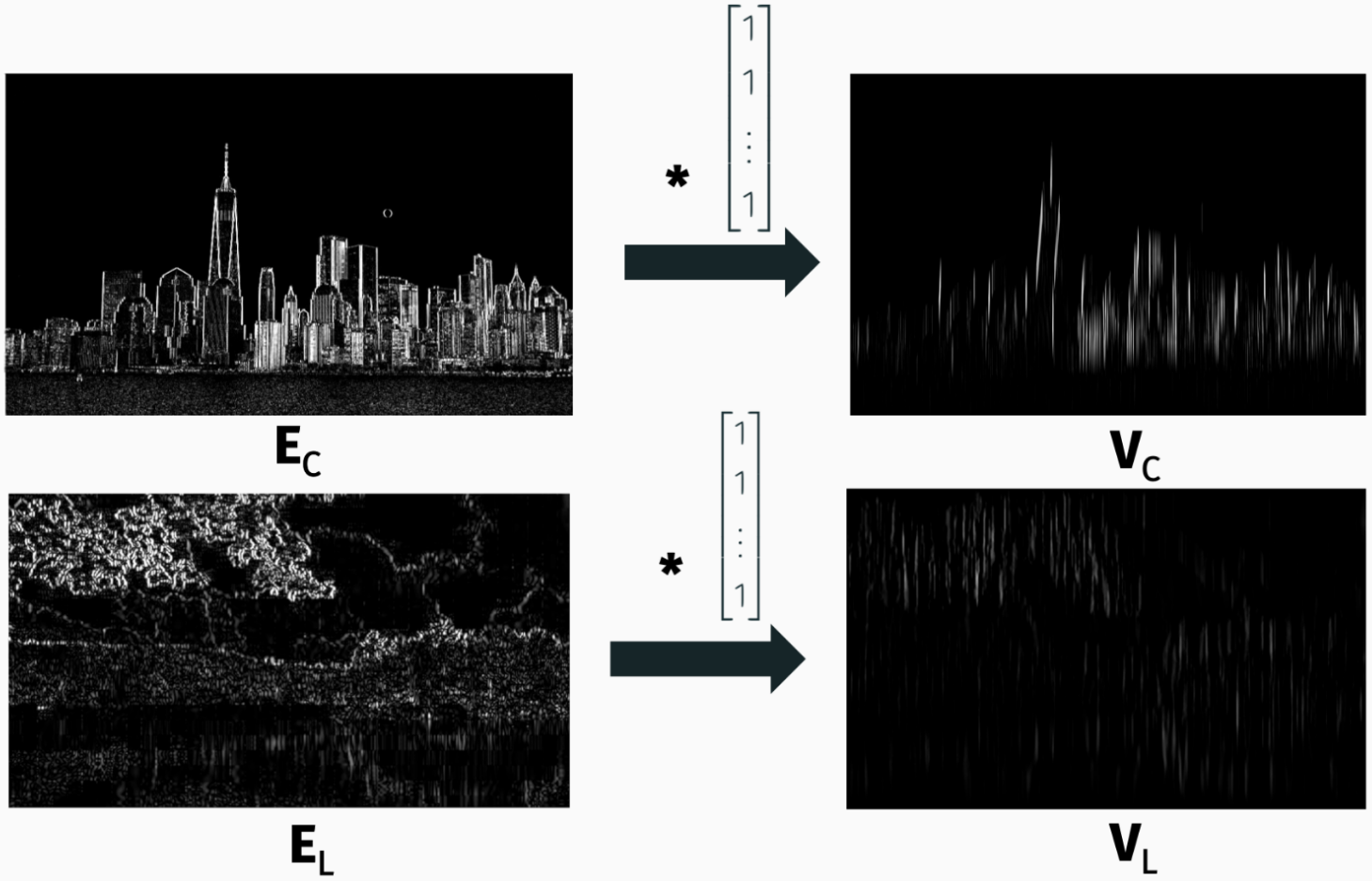


$$\texttt{mean}(|\mathbf{E}_C|) = .108 \qquad \text{vs.} \qquad \texttt{mean}(|\mathbf{E}_L|) = .123$$

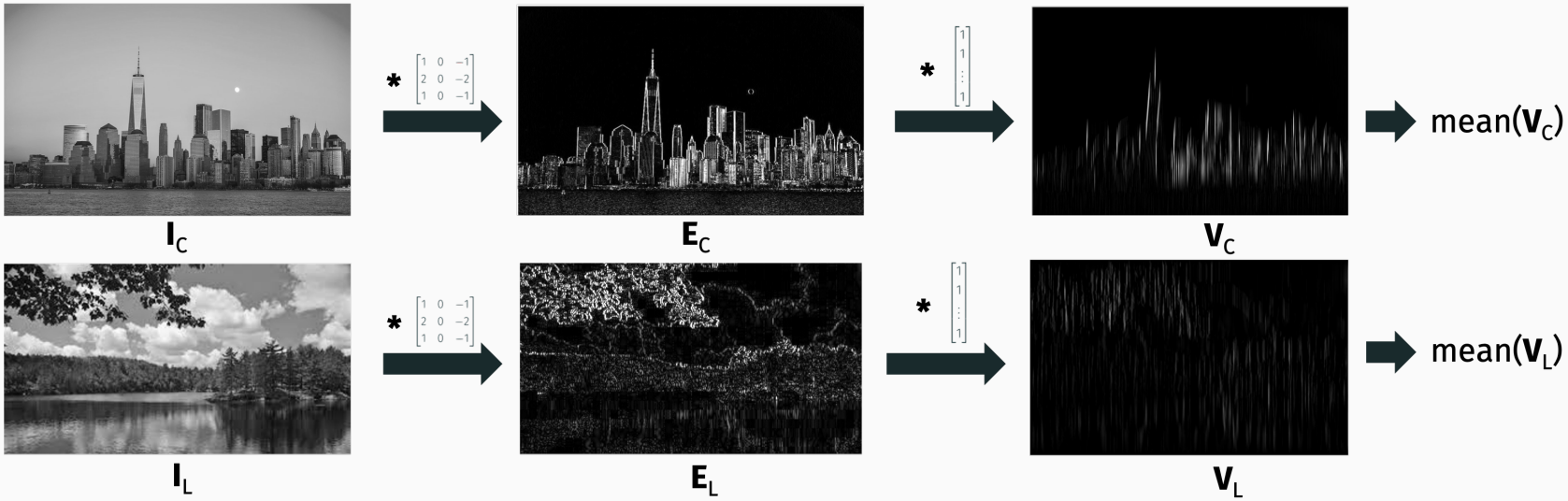
The image with highest vertical edge response isn't the city-scape.

Edge detection + pattern matching

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



Hierarchical convolutional features



$$\texttt{mean}(\mathbf{V}_C) = .062 \qquad \text{vs. } \texttt{mean}(\mathbf{V}_L) = .054$$

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

mean($\mathbf{V}$) $= \mathbf{V}^T\beta$ where $\beta = [1/n, \ldots, 1/n]$. So the new features in $\mathbf{V}$ could be combined with a simple linear classifier to separate cityscapes from landscapes

Hierarchical convolutional features

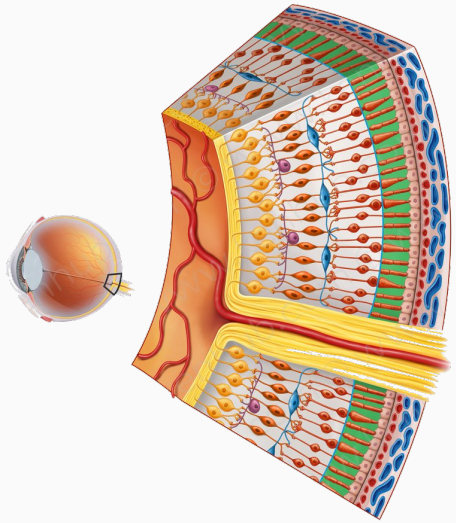
Hierarchical combinations of simple convolution filters are very powerful for understanding images.

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**.



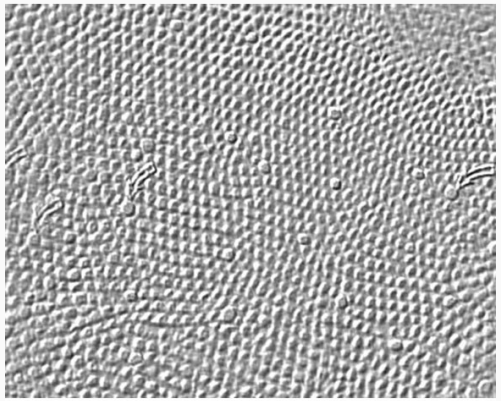
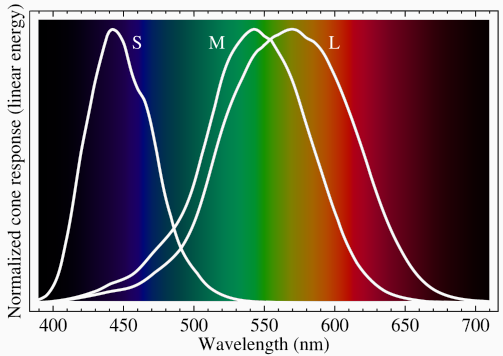


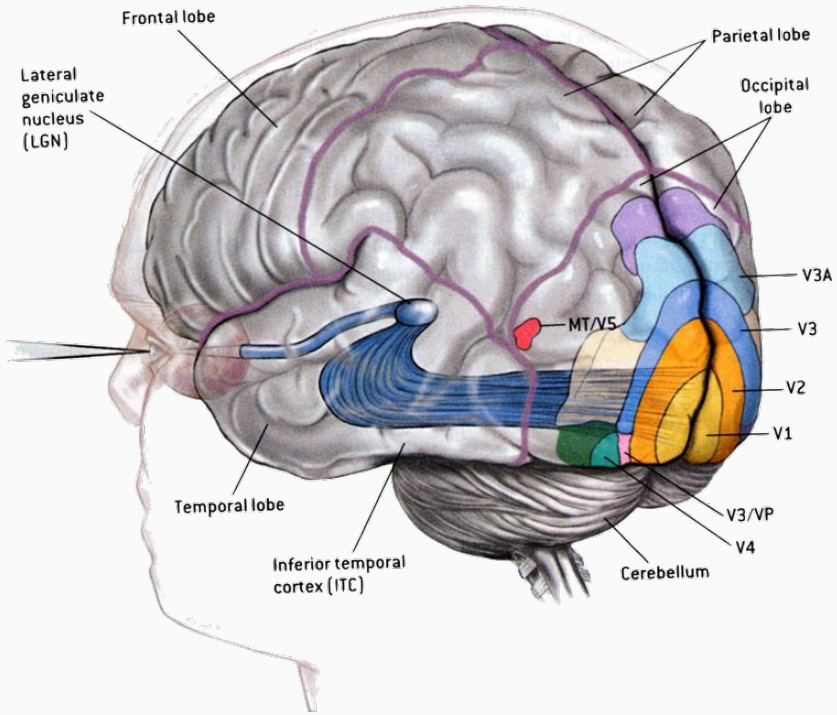
Fig. 13. Tangential section through the human fovea. Larger cones (arrows) are blue cones. From Ahnelt et al. 1987.

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.

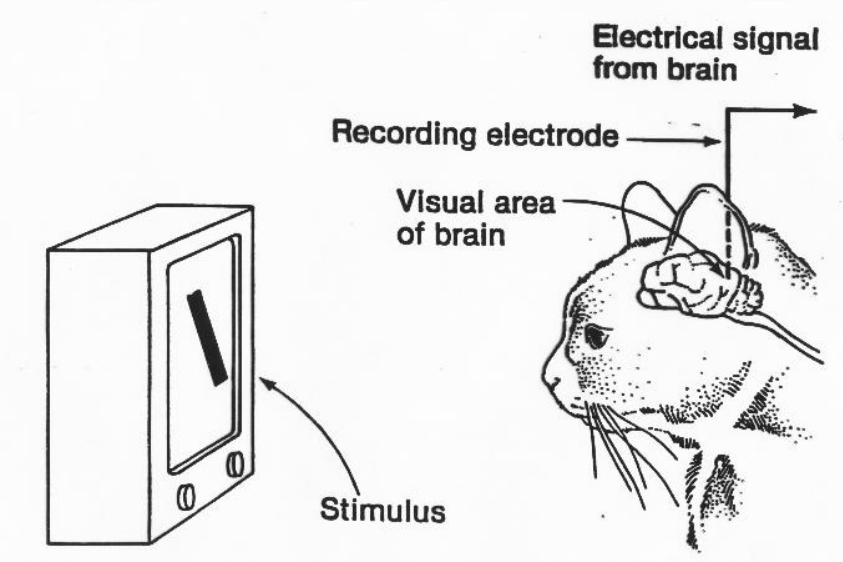


What sort of processing happens in the primary cortex?

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.



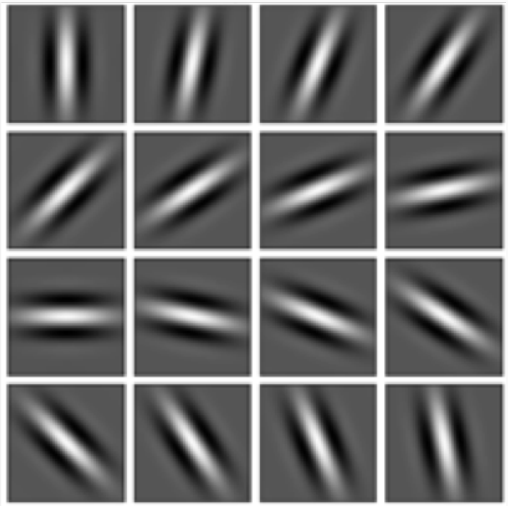
Different neurons fire when the cat is presented with stimuli at different angles. Cool video at
`https://www.youtube.com/watch?v=OGxVfKJqX5E`.

"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

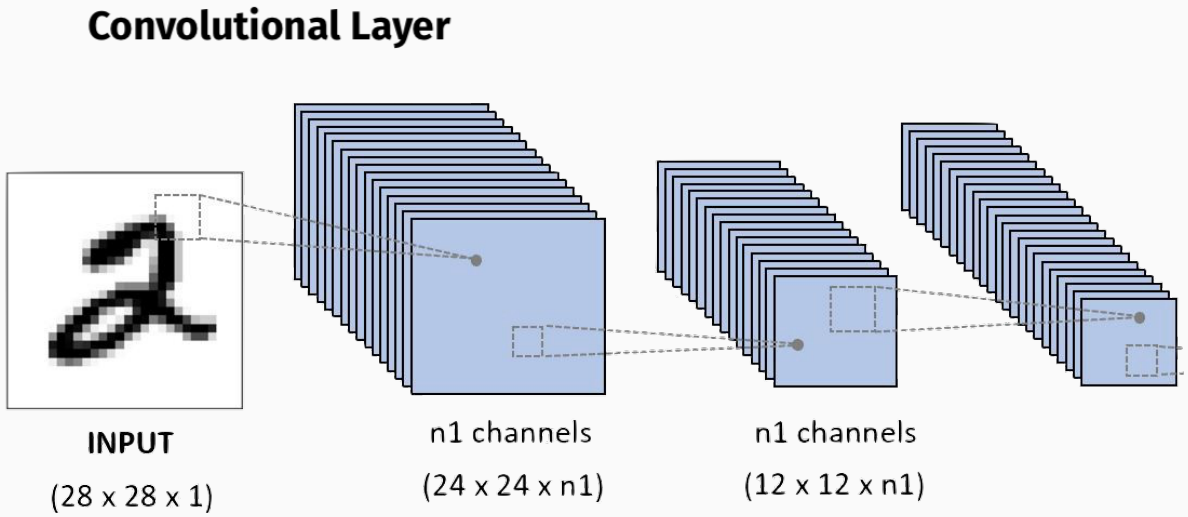
State of the art until 12 years ago:

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



Convolutional neural networks

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

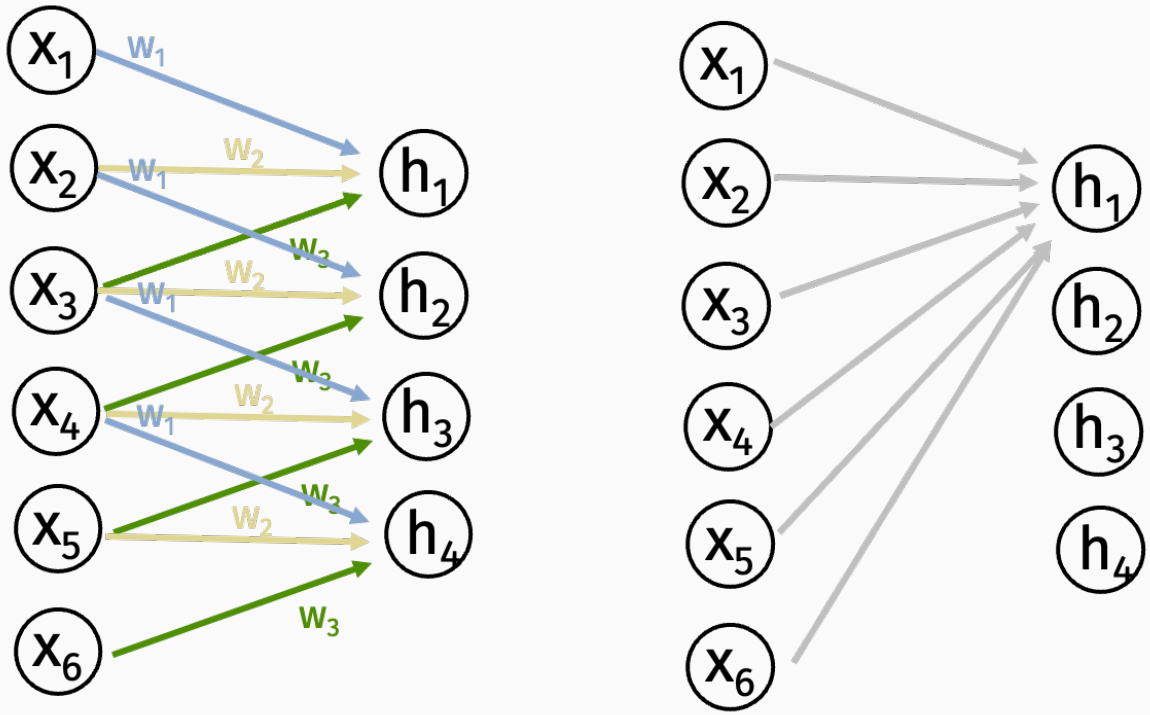


First convolutional layer involves n convolution filters $\mathbf{W}_1, \ldots, \mathbf{W}_n$. Each is small, e.g. 5×5. Every entry in $\mathbf{W}_i$ is a free parameter: $\sim 25 \cdot n$ parameters to learn.

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

Weight sharing

Convolutional layers can be viewed as fully connected layers with added constraints. Many of the weights are forced to 0 and we have weight sharing constraints.

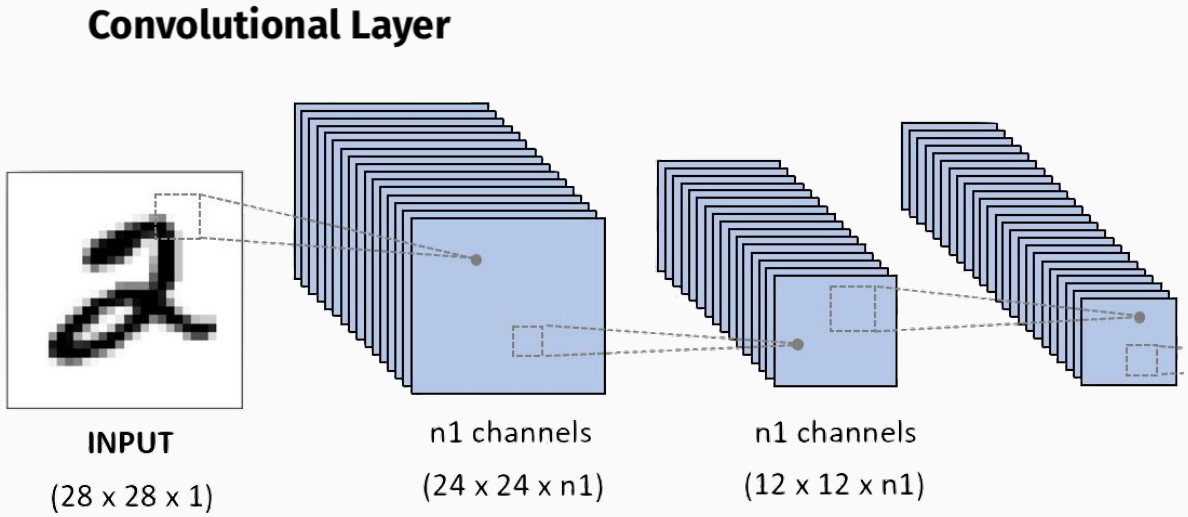


Weight sharing needs to be accounted for when running backprop/gradient descent.

Convolutional neural networks

A fully connected layer that extracts the same feature would require $(28 \cdot 28 \cdot 24 \cdot 24) \cdot n = 451,584 \cdot n$ parameters. Difference of over $200,000x$ from $25n$.

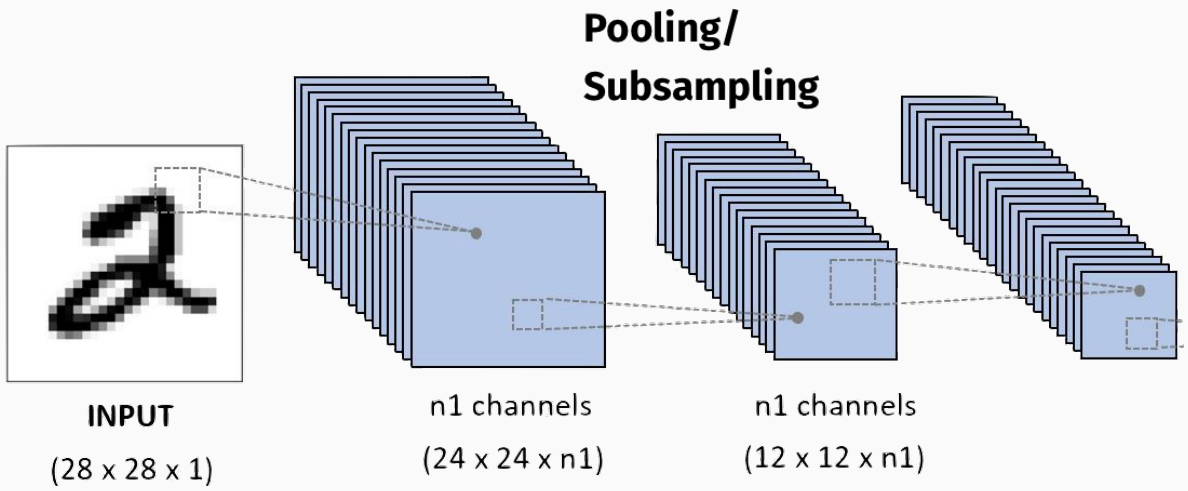
By "baking in" knowledge about what type of features matter, we greatly simply the network.



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

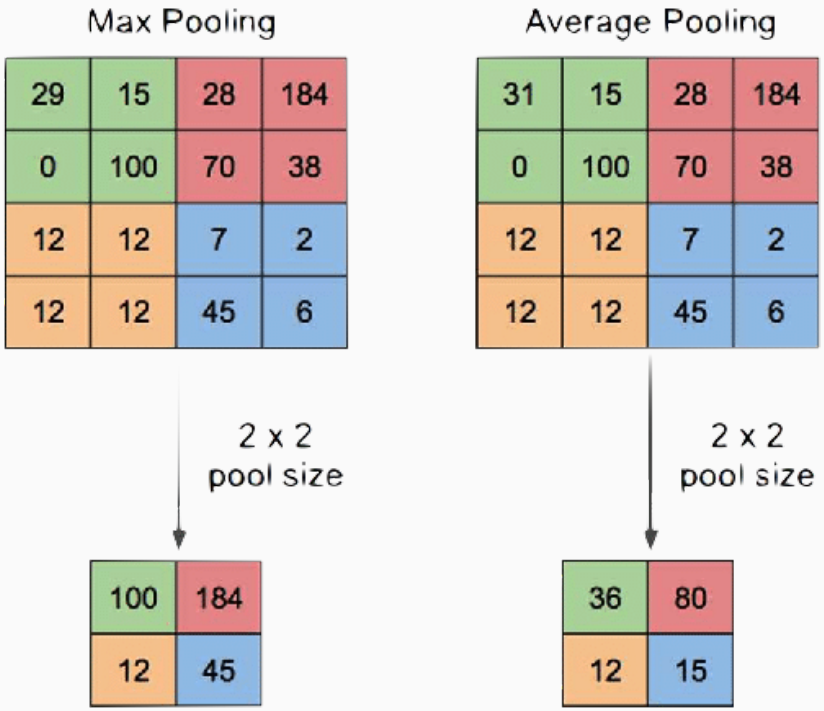
Pooling and downsampling

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

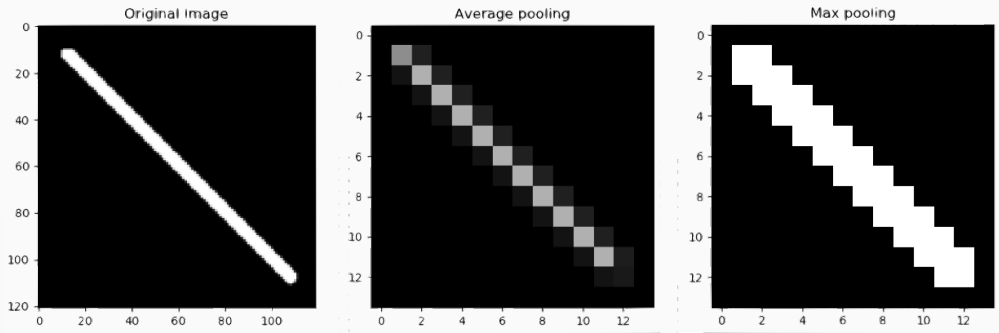


Most common approach is max-pooling.

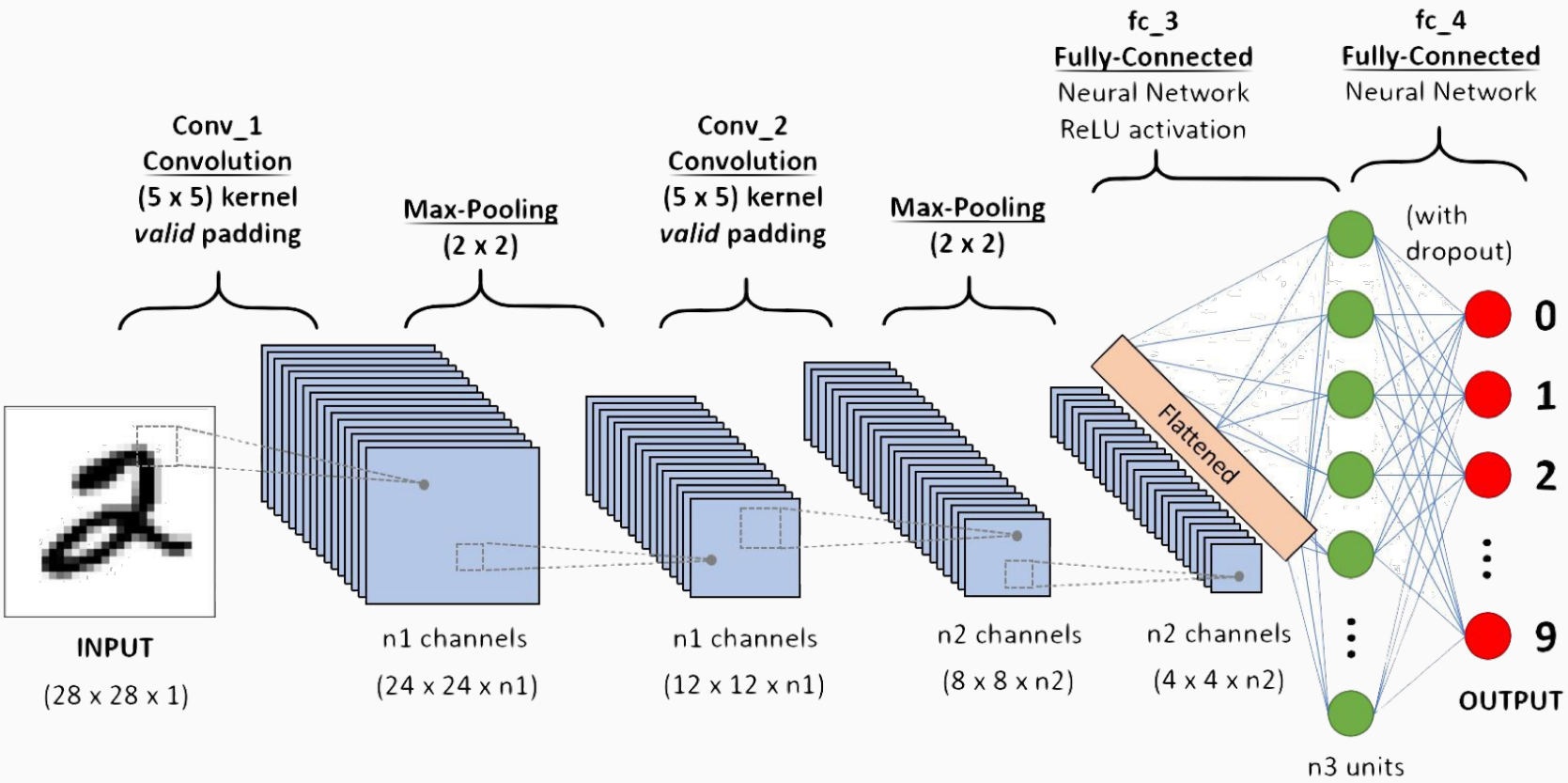
Pooling and downsampling



- Reduces number of variables.
- Helps "smooth" result of convolutional filters.
- Improves shift-invariance.



Overall network architecture

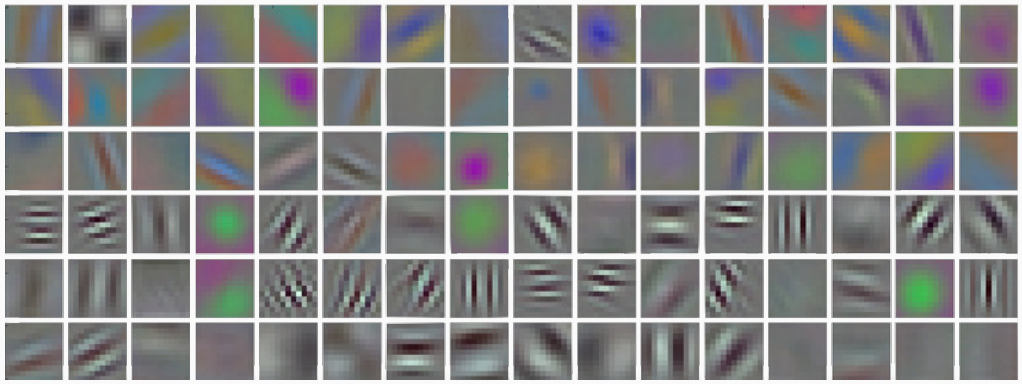


Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

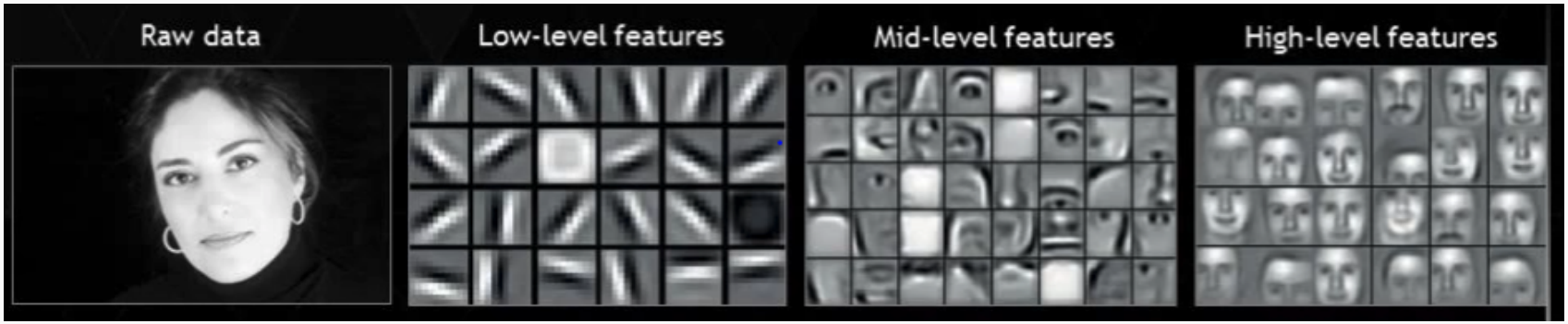
Understanding layers

What type of convolutional filters do we learn from gradient descent?

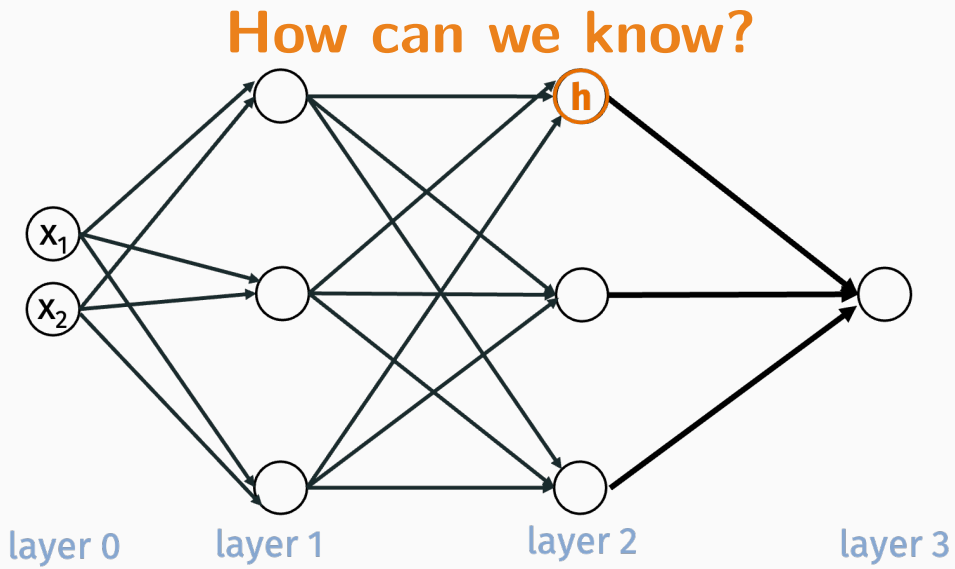
Lots of edge detectors in the first layer!



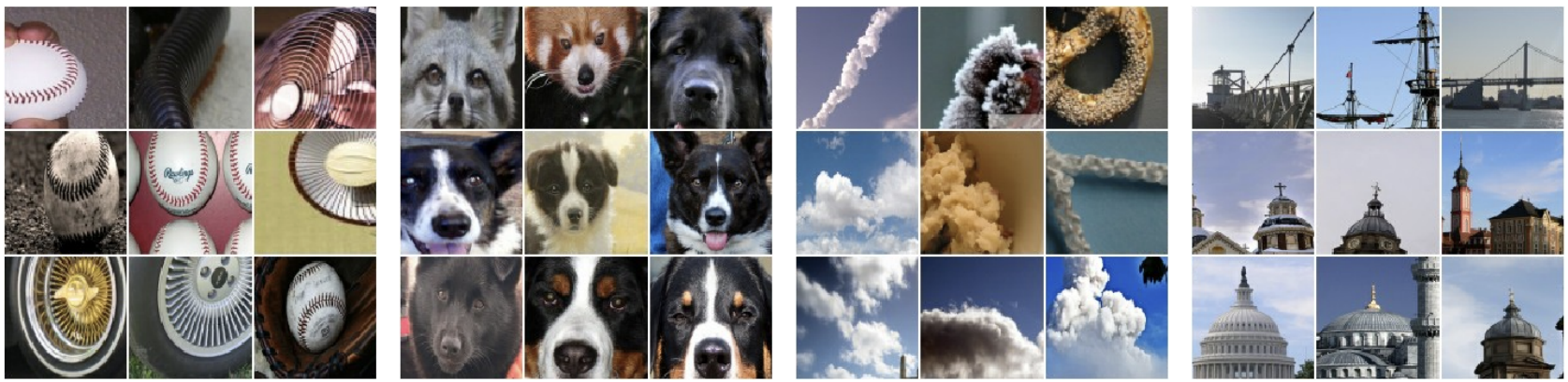
Other layers are harder to understand... but roughly hidden variables later in the network encode for "higher level features":



Understanding layers

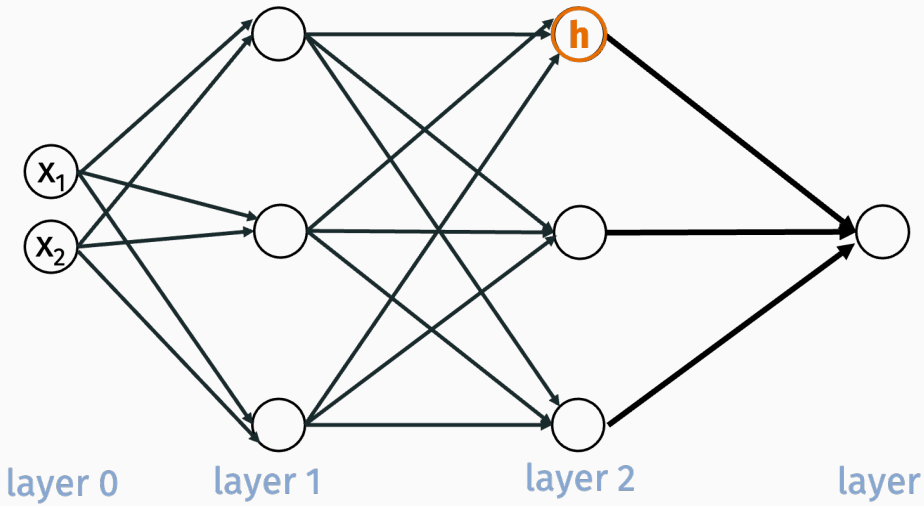


Go through dataset and find the inputs that most "excite" a given neuron h. I.e. for which $|h(\mathbf{x})|$ is largest.



Understanding layers

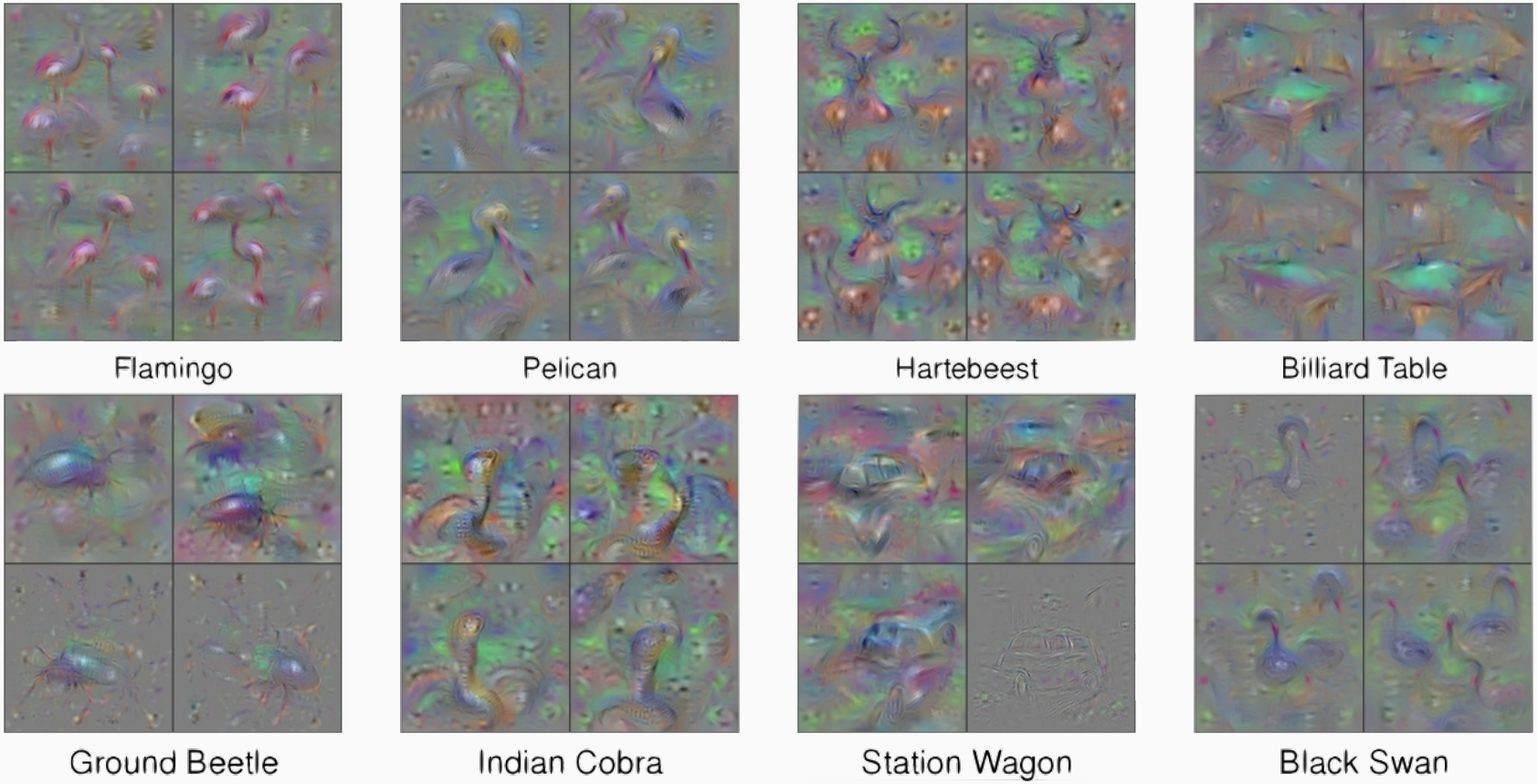
How can we know?



Alternative approach: Solve the optimization problem $\max_{\mathbf{x}} |h(\mathbf{x})|$ e.g. using gradient descent.

Understanding layers

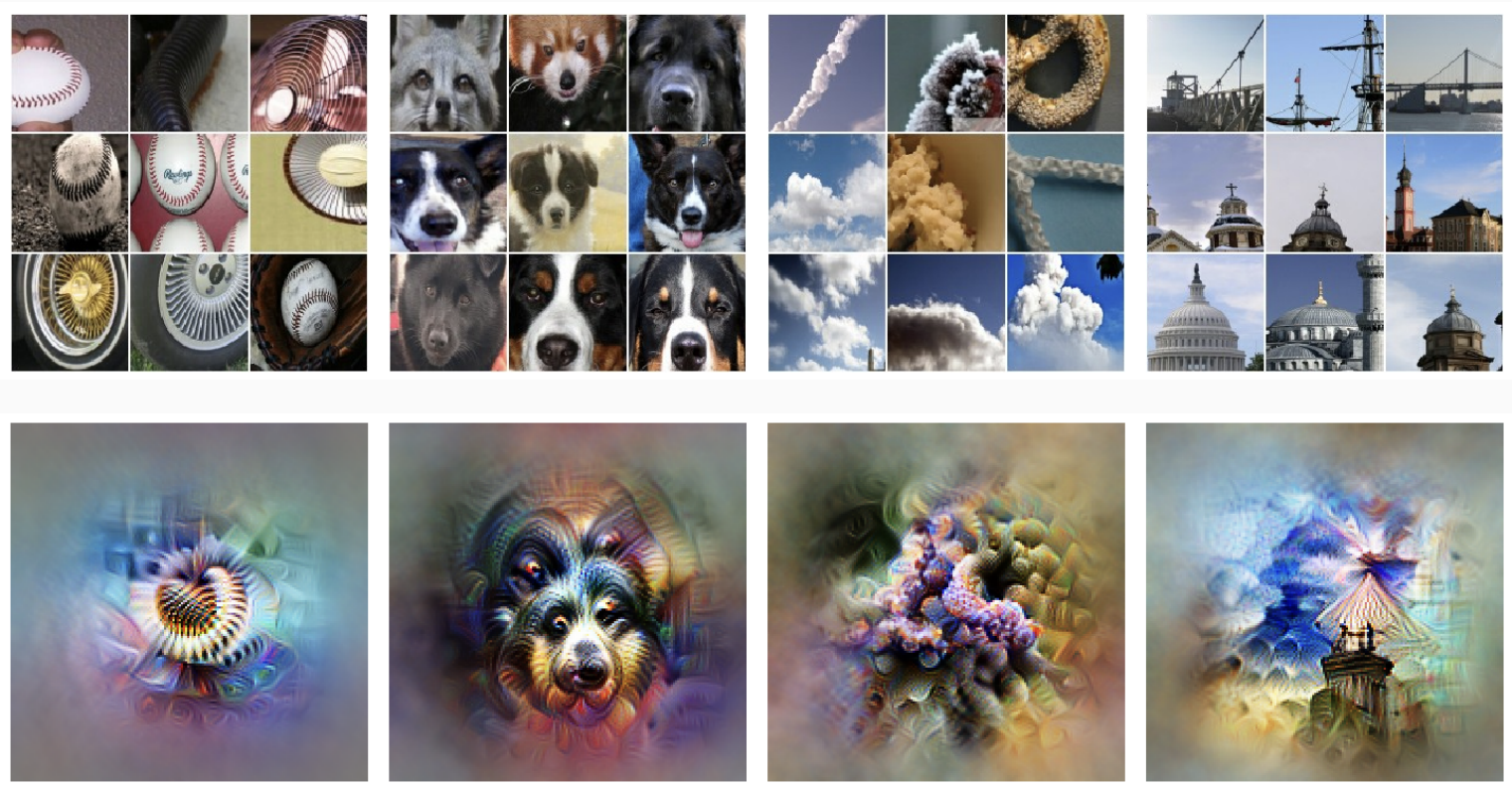
Early work had some interesting results.



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

Understanding layers

There has been a lot of work on improving these methods by regularization. I.e. solve $\max_{\mathbf{x}} |h(\mathbf{x})| + g(\mathbf{x})$ where g constrains $\mathbf{x}$ to look more like a "natural image".



If you are interested in learning more on these techniques, there is a great Distill article at: `https://distill.pub/2017/feature-visualization/`.

Understanding layers

Nodes at different layers have different layers capture increasingly more abstract concepts.

Edges (layer conv2d0)

Textures (layer mixed3a)

Patterns (layer mixed4a)

Understanding layers

Nodes at different layers have different layers capture increasingly more abstract concepts.

Parts (layers mixed4b & mixed4c) **Objects** (layers mixed4d & mixed4e)

General obervation: Depth more important than width. Alexnet 2012 had 8 layers, modern convolutional nets can have 100s.

Tricks of the trade

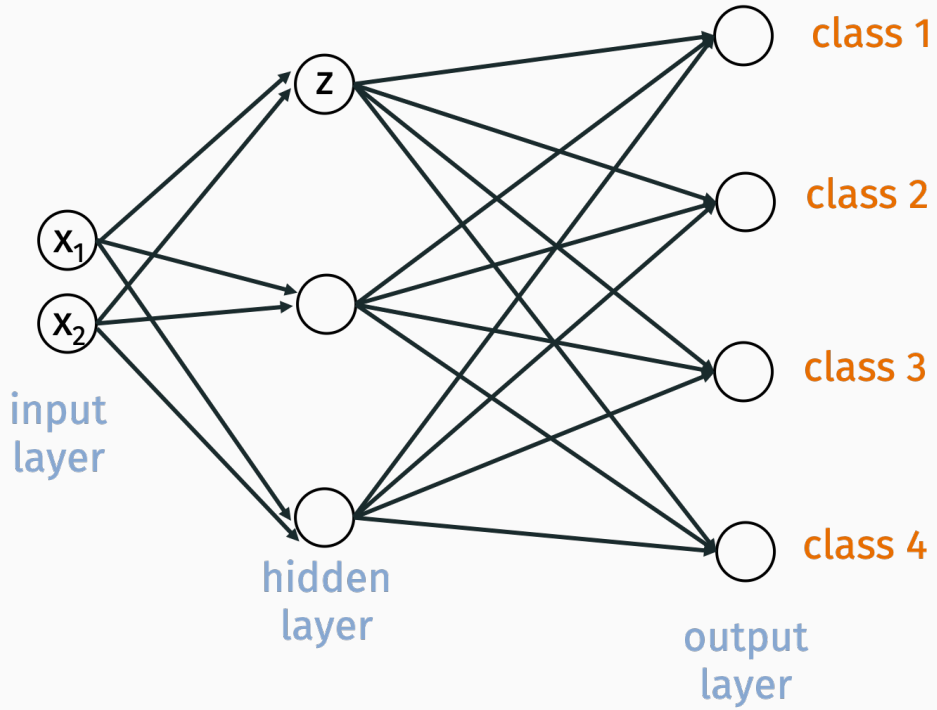
Beyond techniques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of "tricks".

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

And deep networks require lots of training data and lots of time.

Batch normalization

Start with any neural network architecture:



For input $\mathbf{x}$,

$$\bar{z} = \mathbf{w}^T \mathbf{x} + b$$

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

where $\mathbf{w}$, b, and s are weights, bias, and non-linearity.

Batch normalization

$\bar{z}$ is a function of the input $\mathbf{x}$. We can write it as $\bar{z}(\mathbf{x})$. Consider the mean and standard deviation of the hidden variable over our entire dataset $\mathbf{x}_1 \ldots, \mathbf{x}_n$:

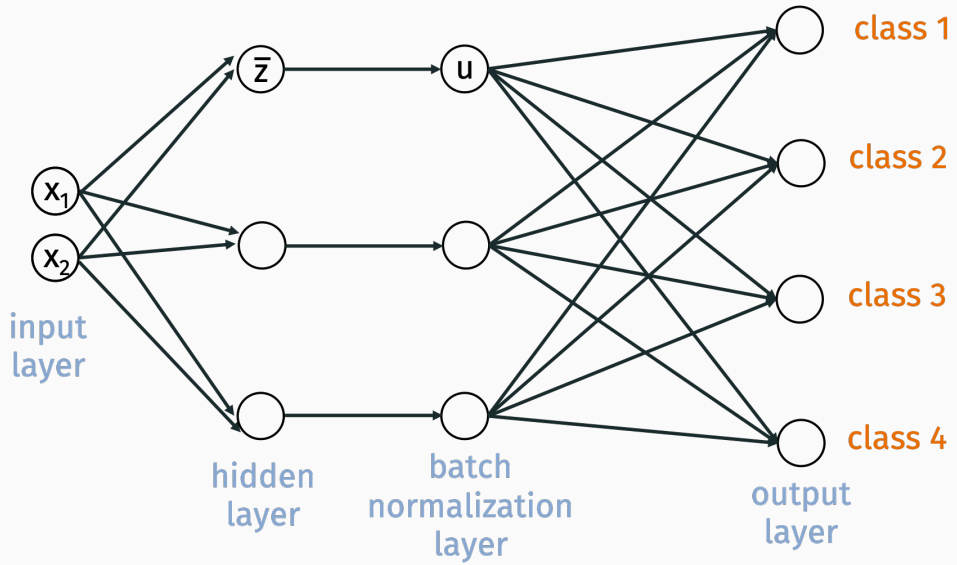
$$\mu = \frac{1}{n} \sum_{j=1}^{n} \bar{z}(\mathbf{x}_j)$$

$$\sigma^2 = \frac{1}{n} \sum_{j=1}^{n} (\bar{z}(\mathbf{x}_j) - \mu)^2$$

Just as normalization (mean centering, scaling to unit variance) is sometimes used for input features, batch-norm applies normalization to learned features.

Batch normalization

Can add a batch normalization layer after any layer:



$$\bar{u} = \frac{\bar{z} - \mu}{\sigma}$$

$$u = s(\bar{u}).$$

Has the effect of mean-centering/normalizing $\bar{z}$. Typically we actualy allow $u = s(\gamma \cdot \bar{u} + c)$ for learned parameters γ and c.

Batch normalization

Proposed in 2015: "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", Ioffe, Szegedy.

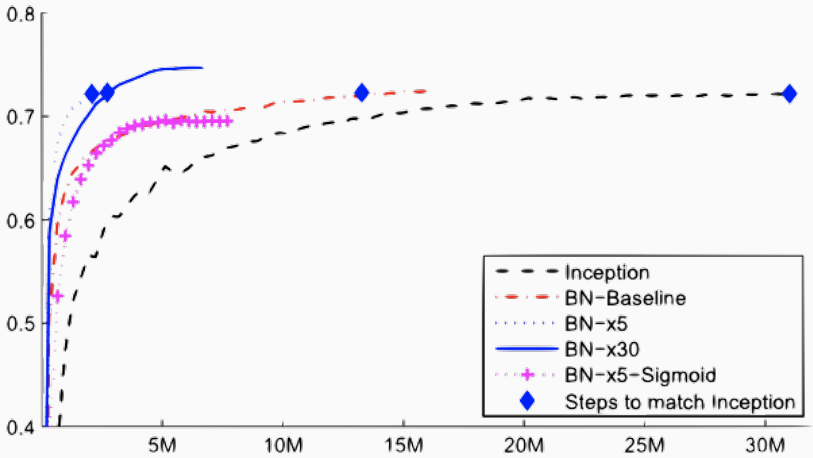


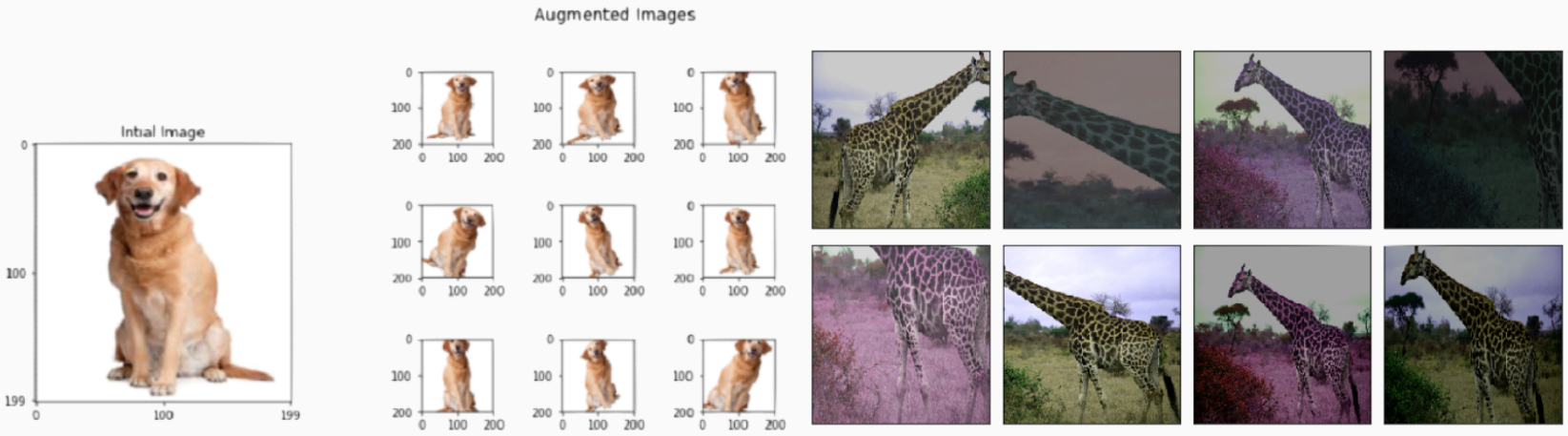
Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Doesn't change the expressive power of the network, but allows for significant convergence acceleration. It is not yet well understood why batch normalizition speeds up training.

Data augmentation

Great general tool to know about. **Main idea:**

- More training data typically leads to a more accurate model.
- Artificially enlarge training data with simple transformations.



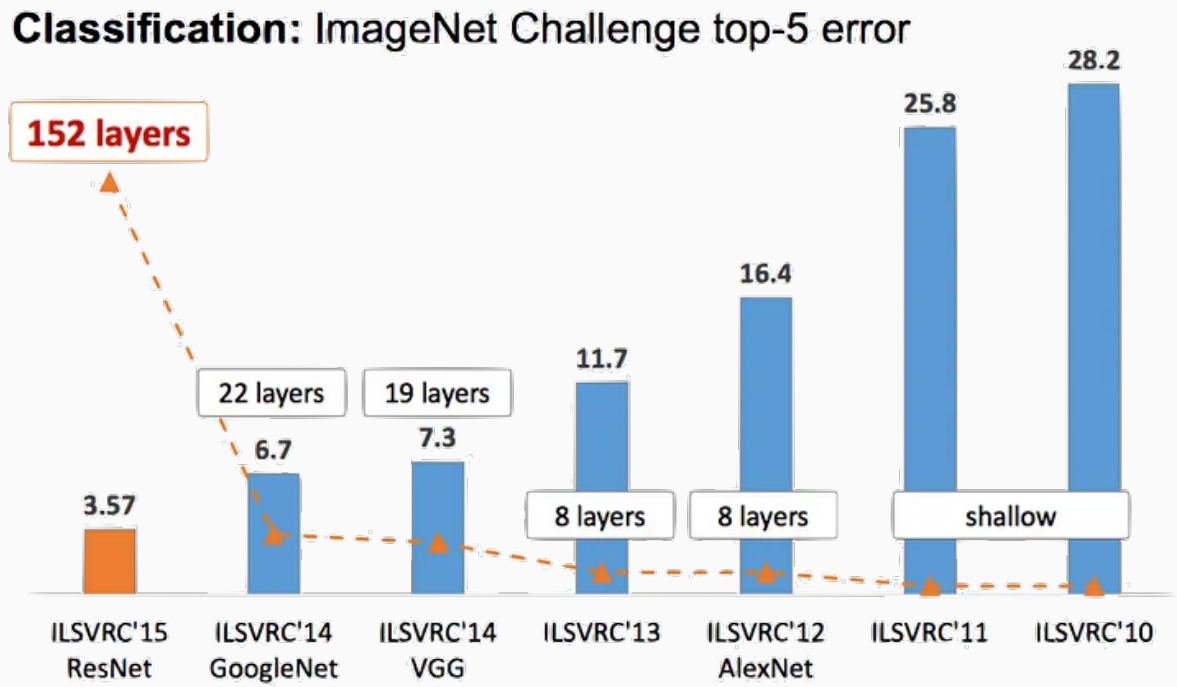
Take training images and randomly shift, flip, rotate, skew, darken, lighten, shift colors, etc. to create new training images. **Final classifier will be more robust to these transformations.**

Deep learning tricks

Need to take a full course on neural networks/deep learning to learn more! State-of-the-art techniques are constantly evolving.

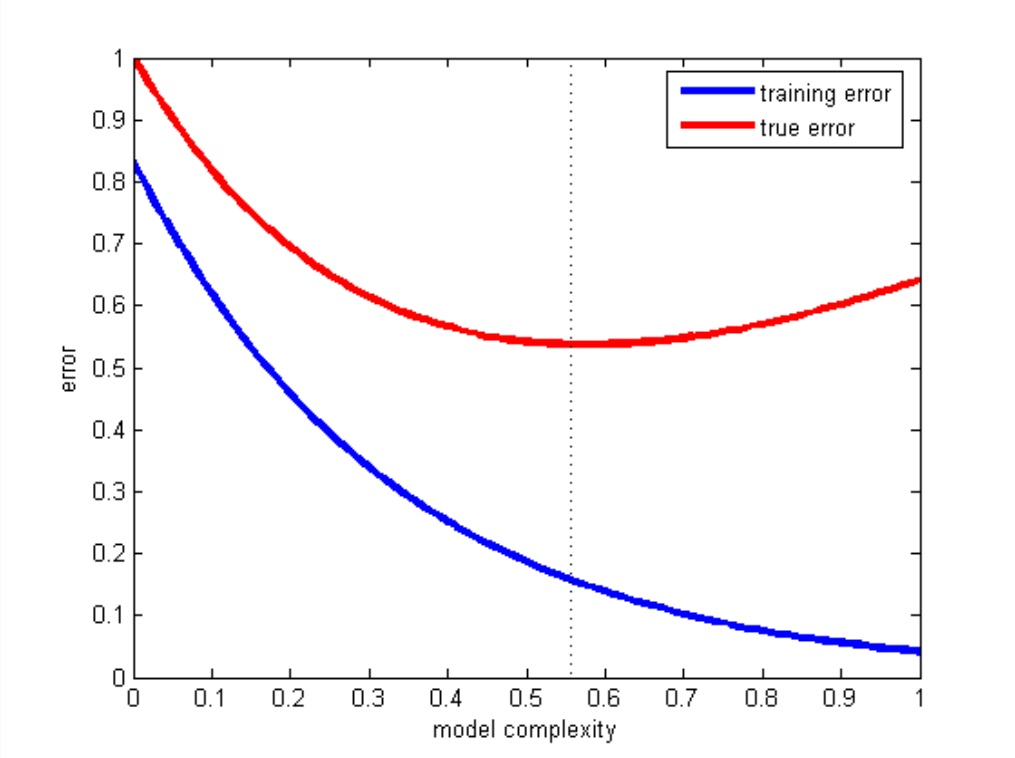
Deeper and deeper, bigger and bigger

After AlexNet (8 layers, 60 million parameters) achieved state of the art performance on ImageNet, progress proceeded rapidly:



Generalization for neural networks

Even with weight sharing, convolution, etc. modern neural networks typically have 100s of millions of parameters. And we don't train them with regularization. Intuitively we might expect them to overfit to training data.



Generalization for neural networks

In fact, we now know that modern neural nets can easily overfit to training data. This work showed that we can fit large vision data sets with random class labels to essentially perfect accuracy.

UNDERSTANDING DEEP LEARNING REQUIRES RETHINKING GENERALIZATION

Chiyuan Zhang*
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

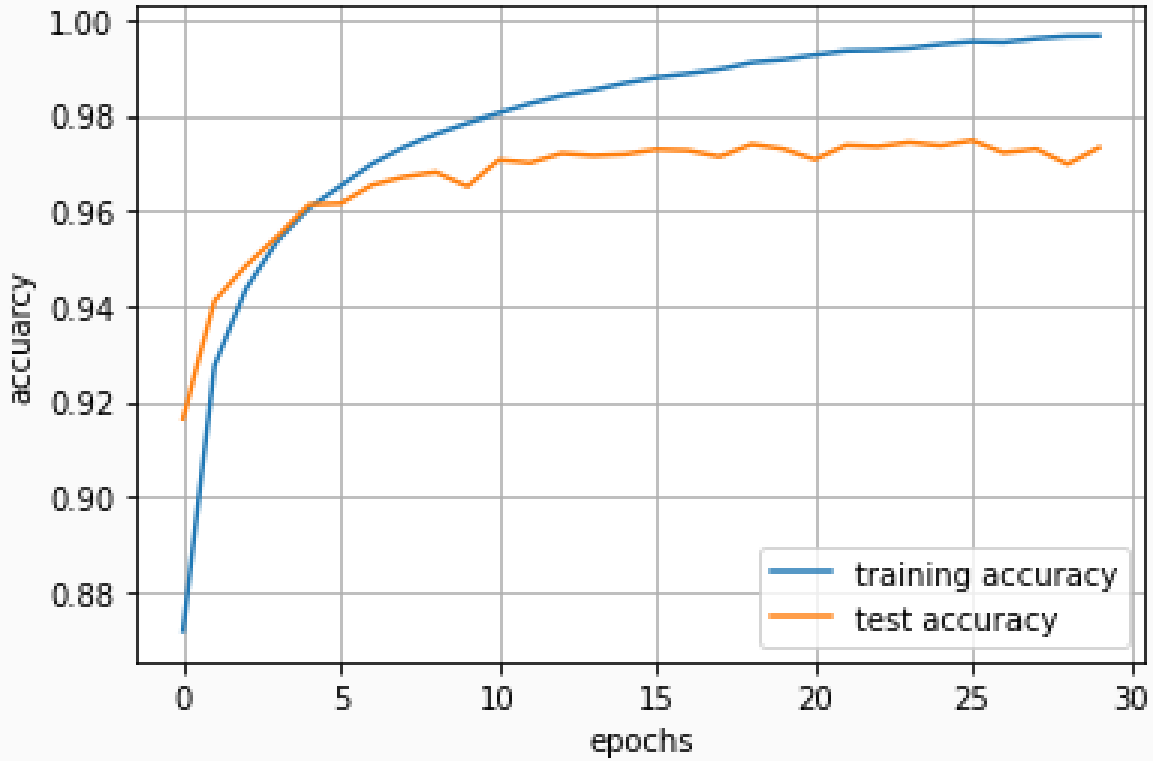
Benjamin Recht†
University of California, Berkeley
brecht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

But we don't always see a large gap between training and test error. **Don't take this to mean overfitting isn't a problem when using neural nets!** It's just not always a problem.

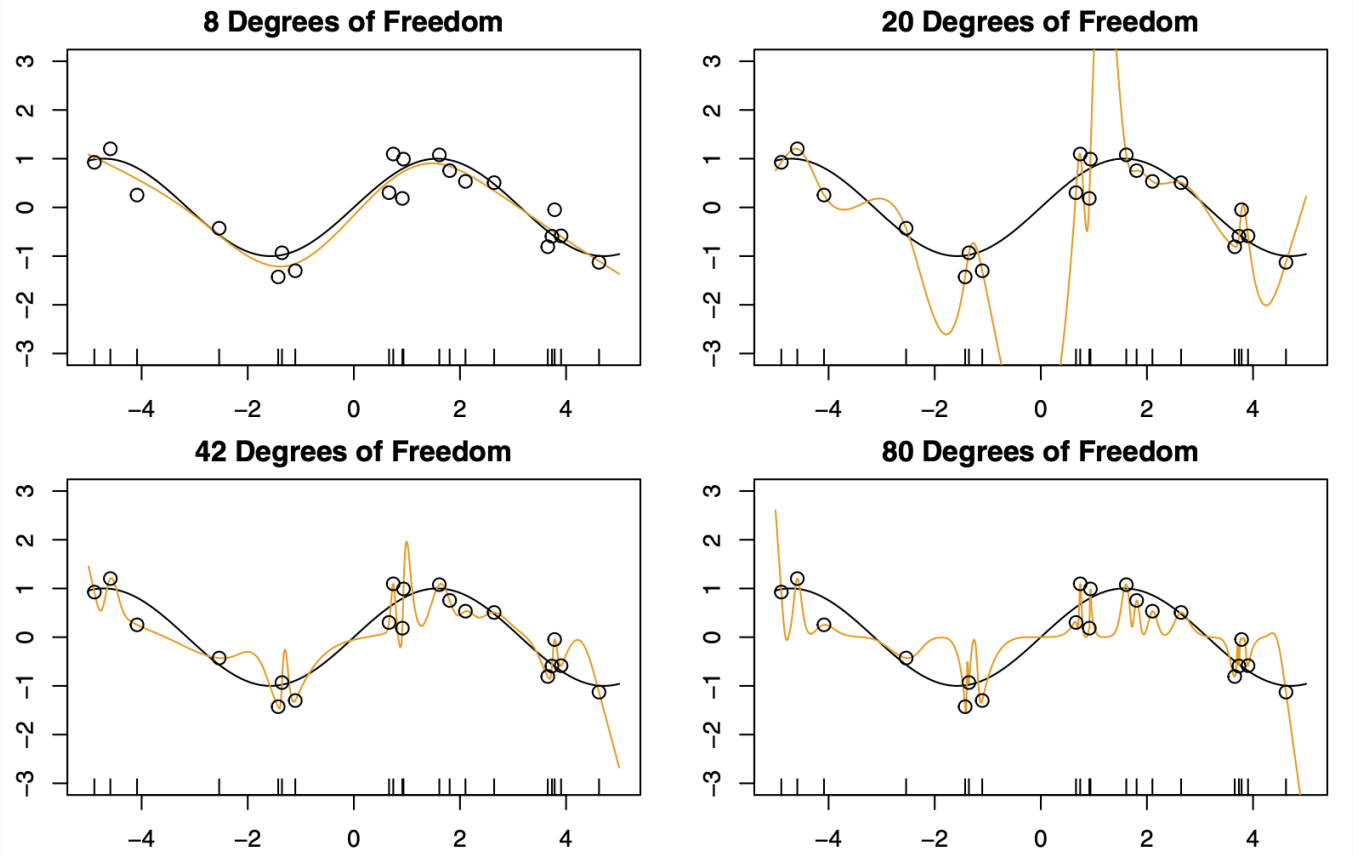
Generalization for neural networks

We even see this lack of overfitting for MNIST data. Check `keras_demo_mnist.ipynb` I posted on the website:



Generalization for neural networks

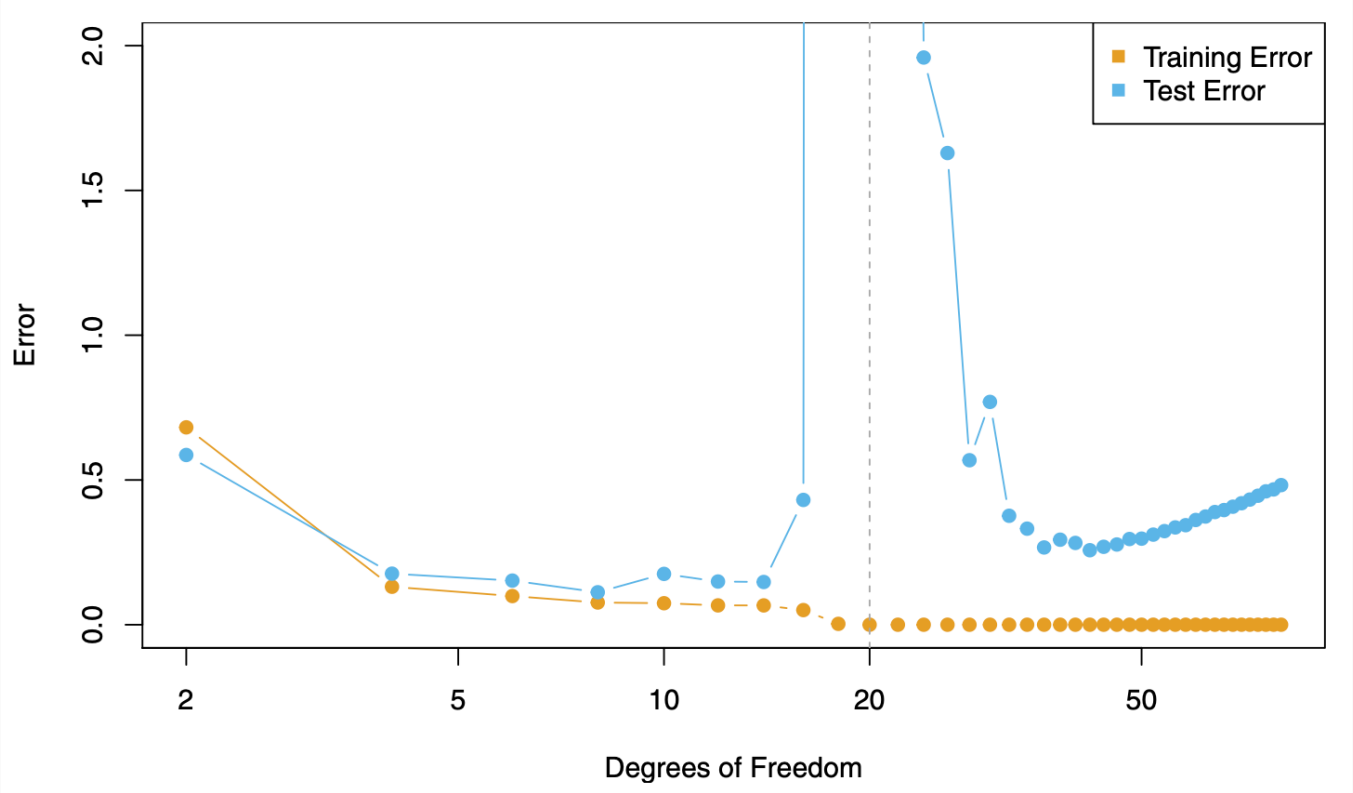
One growing realization is that this phenomena doesn't only apply to neural networks – it can also be true for fitting highly-overparameterized polynomials.



The choice of training algo (e.g. gradient descent) seems important.

Double descent

We sometimes see a "double descent curve" for these models. Test error is worst for "just barely" overparameterized models, but gets better with lots of overparameterization.

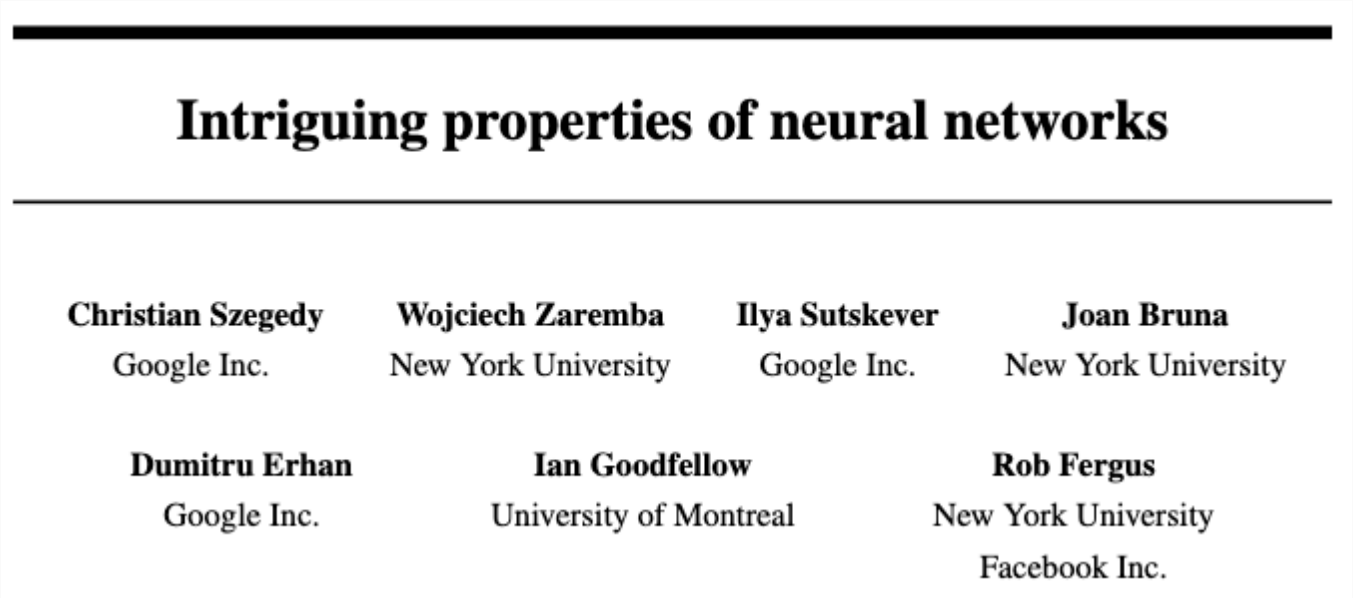


We don't usually see this same curve for neural networks.

Overfitting in neural nets

Take away: Modern neural network overfit, but still seem fairly robust. Perform well on any new test data we throw that them.

Or do they?



Intriguing properties of neural networks

Christian Szegedy
Google Inc.

Wojciech Zaremba
New York University

Ilya Sutskever
Google Inc.

Joan Bruna
New York University

Dumitru Erhan
Google Inc.

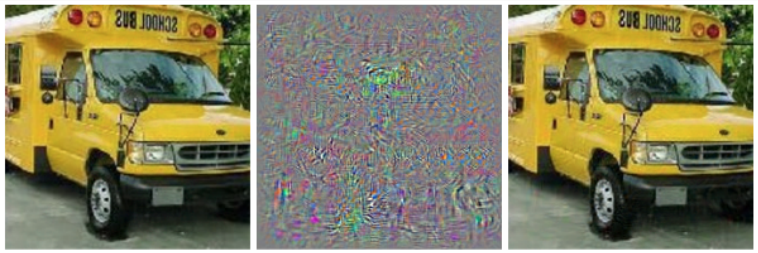
Ian Goodfellow
University of Montreal

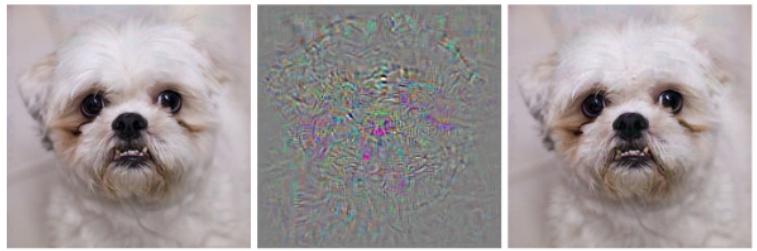
Rob Fergus
New York University
Facebook Inc.

Adversarial examples

Adversarial examples

Main discovery: It is possible to find imperceptibly small perturbations of input images that will fool deep neural networks. This seems to be a universal phenomenon.





Important: Random perturbations do not work!

Adversarial examples

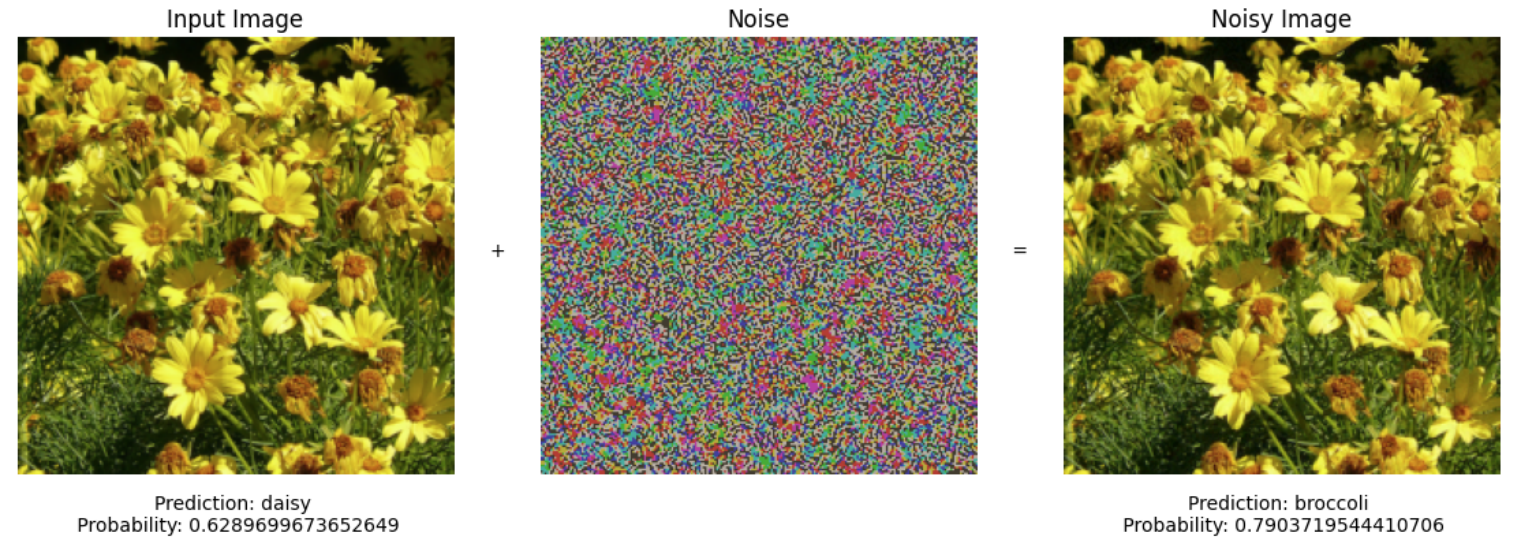
How to find "good" perturbations:

Fix model $f_{\boldsymbol{\theta}}$, input $\mathbf{x}$, correct label y. Consider the loss $\ell(\theta, \mathbf{x}, y)$.

Solve the optimization problem:

$$\max_{\boldsymbol{\delta}, \|\boldsymbol{\delta}\| \leq \epsilon} \ell(\theta, \mathbf{x} + \boldsymbol{\delta}, y)$$

Can be solved using gradient descent! We just need to compute the derivative of the loss with respect to the image pixels. Backprop can do this easily.

Adversarial examples

Teal put together a really cool lab where you can find your own adversarial examples for a model called Resnet18. The entire model + weights are available through PyTorch, so we do not need to train it ourselves (i.e. this is a pre-trained model).