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

NYU Tandon School of Engineering, 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.

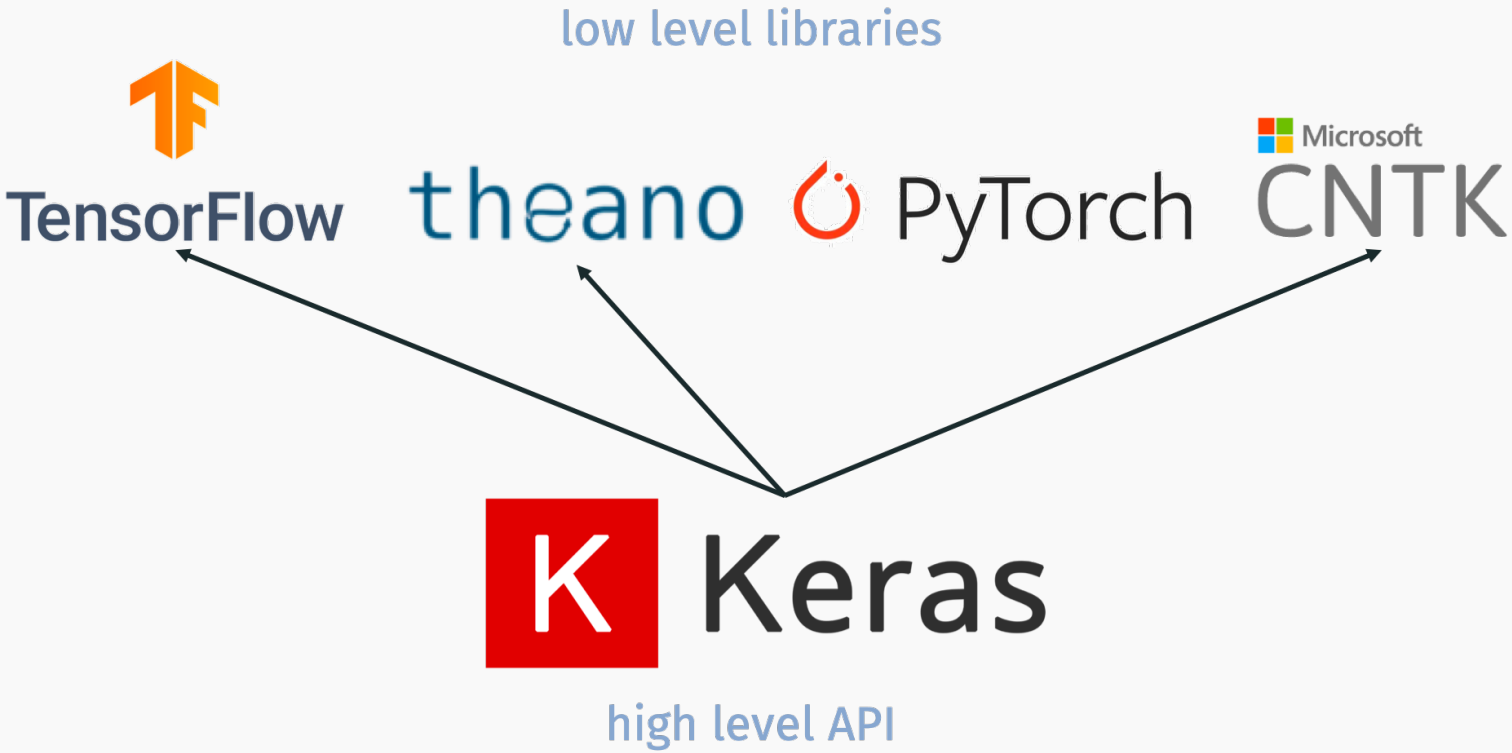
NEURAL NETWORK DEMOS

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 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'])
```

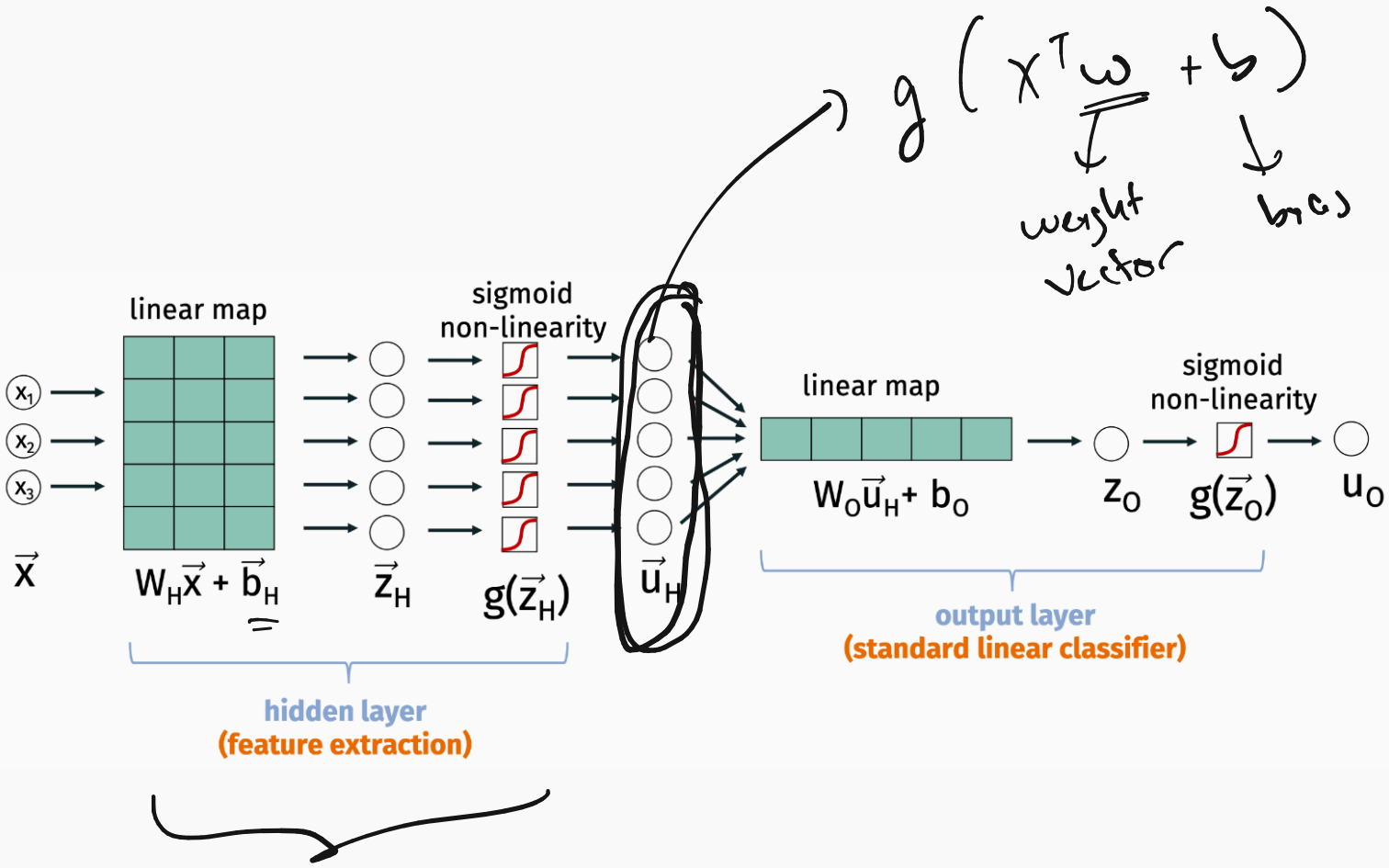
Train model:

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

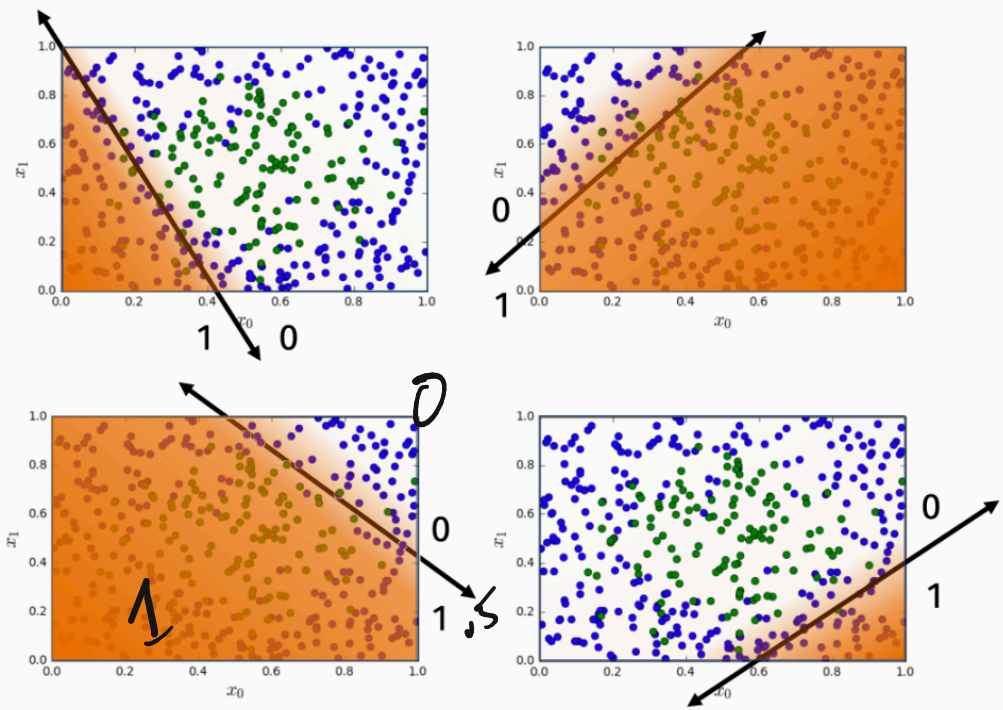
Why do neural networks work so well?

- 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



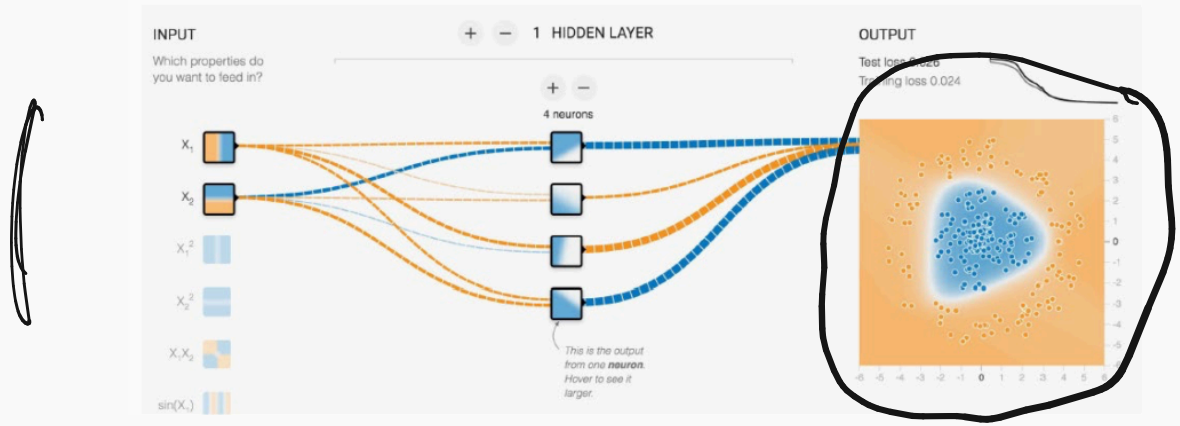
Sigmoid activation: Each hidden variable z_i equals $\frac{1}{1+e^{-z_i}}$ where $z_i = \mathbf{w}^T\mathbf{x} + b$ for input $\mathbf{x}$.



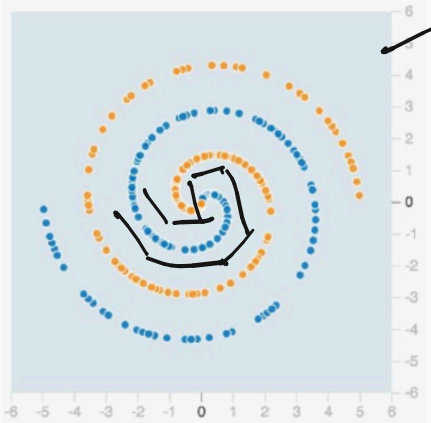
Other non-linearities yield similar features.

BASIC FEATURE EXTRACTION

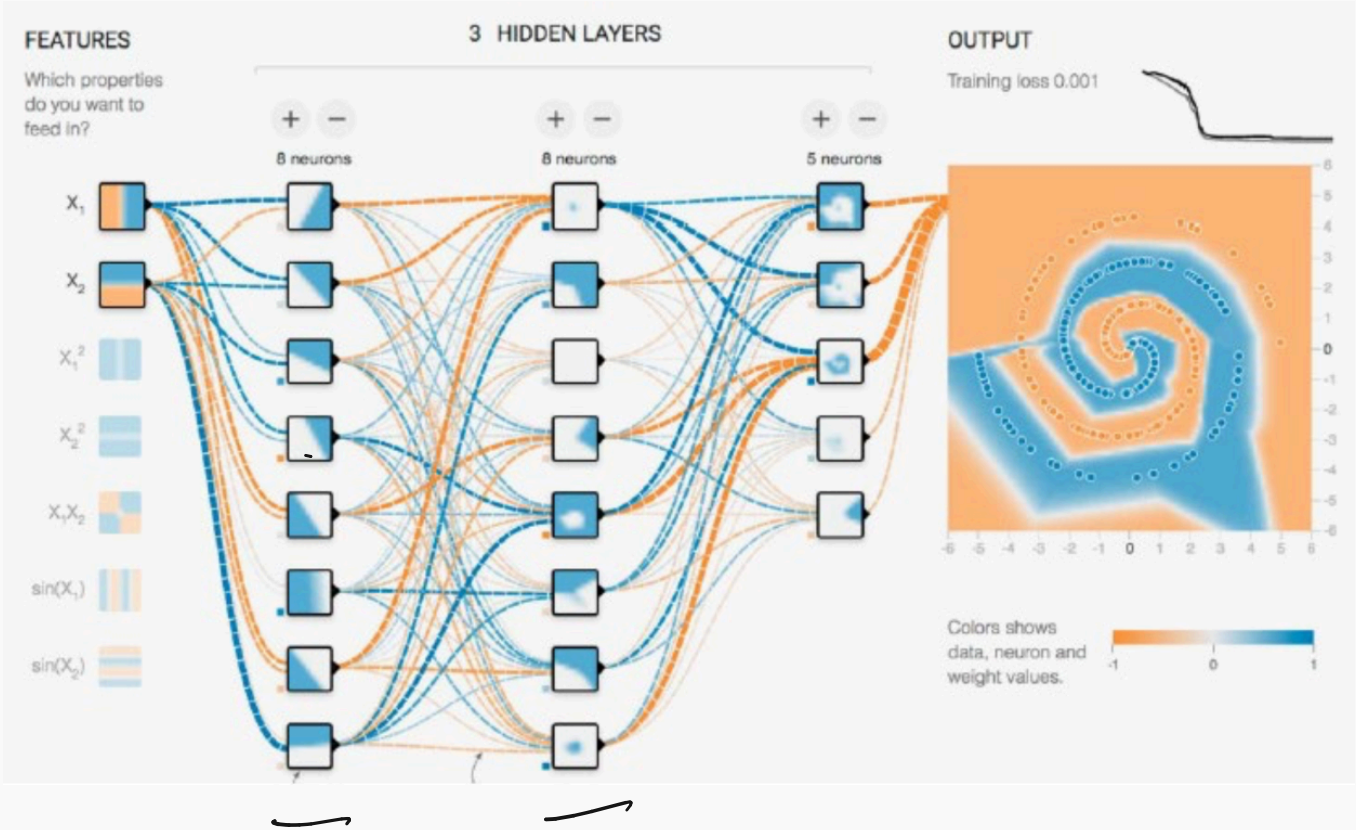
If you combine more hidden variables, you can start building more complex classifiers.



What about even more complex datasets?



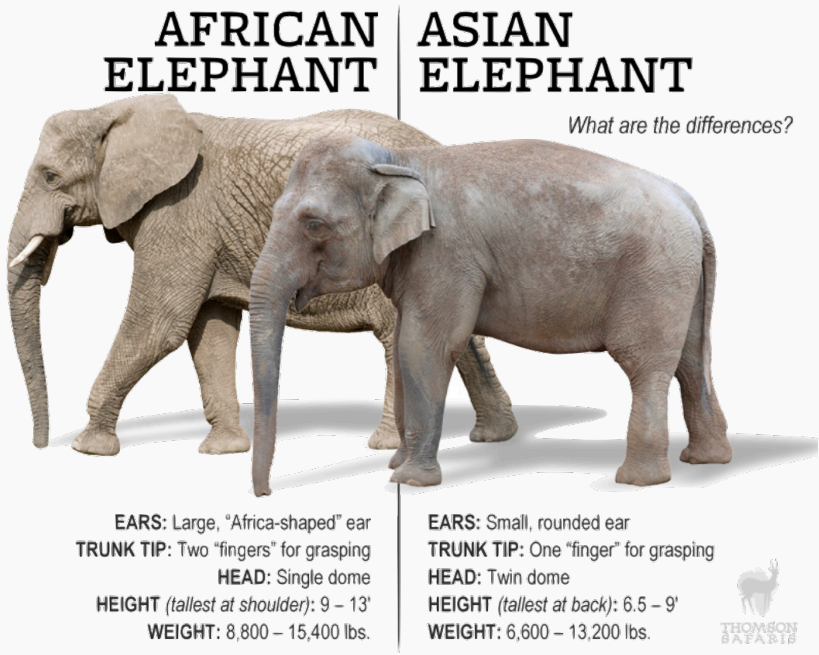
With more layers, complexity starts ramping up:



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

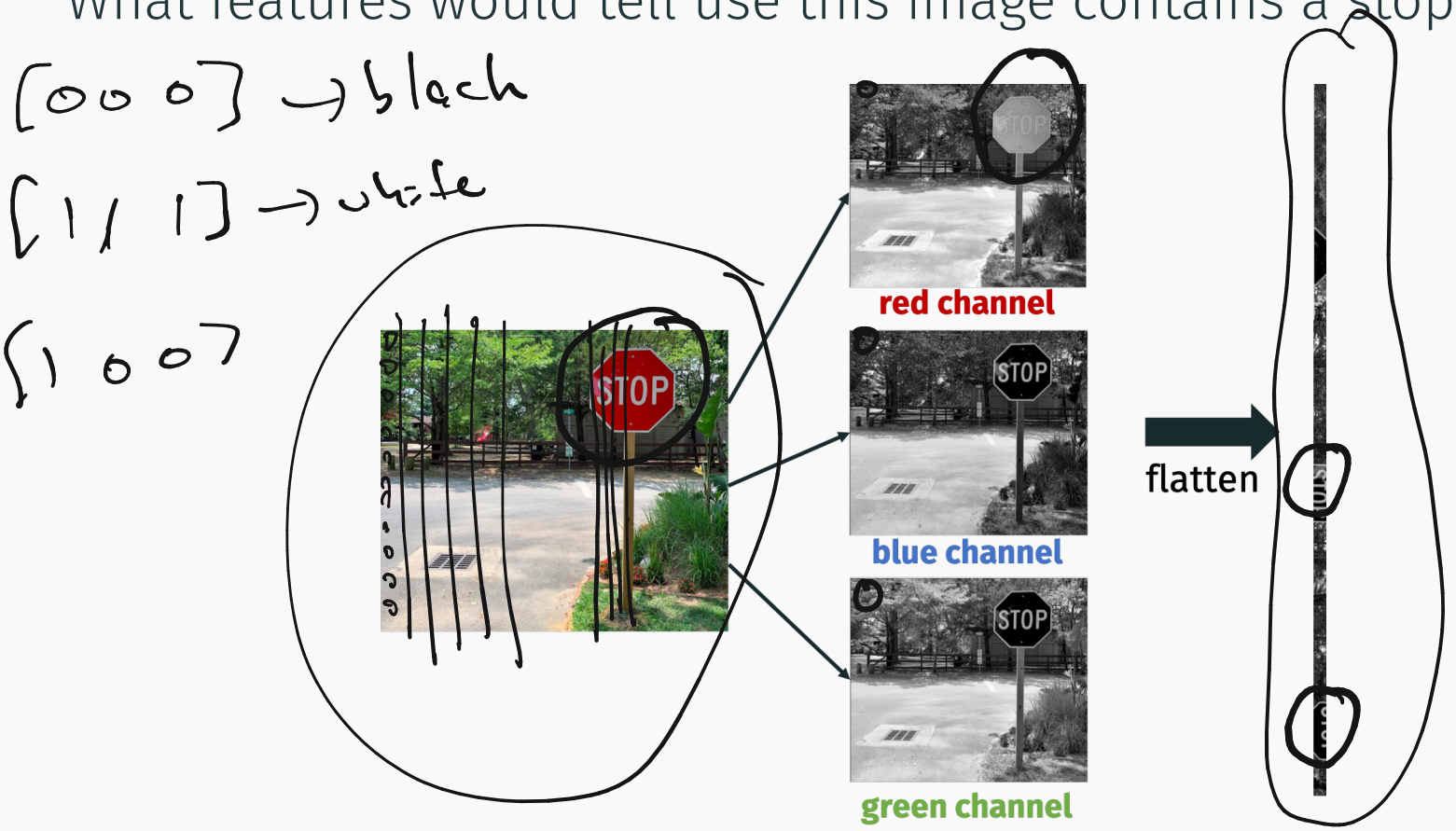
Today's topic: Understand why convolution is a powerful way of extracting features from image data. Also super valuable for

- Audio data.
- Time series data.

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

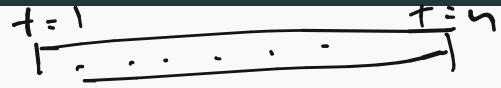
MOTIVATING EXAMPLE

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:

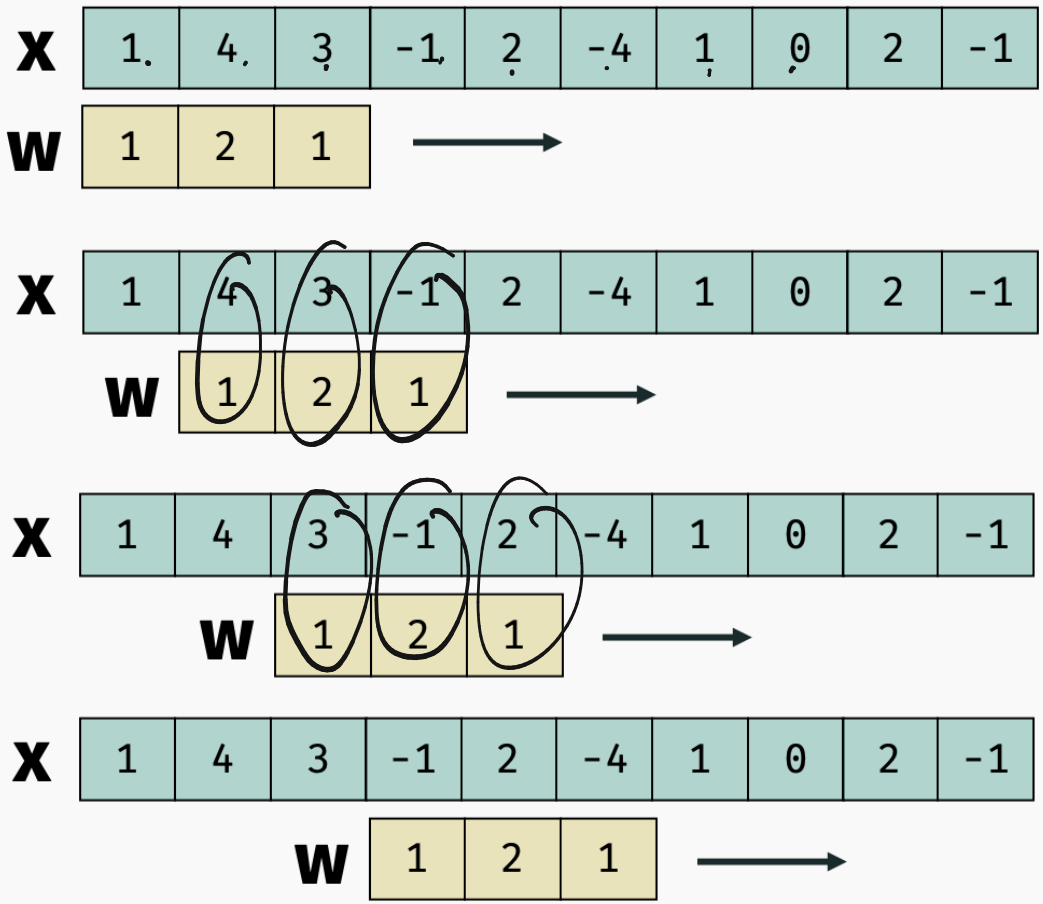
$$[\mathbf{x} \circledast \mathbf{w}]_i = \sum_{j=1}^{k} \mathbf{x}_{(j+i-1)} \mathbf{w}_j$$

$\mathbf{u} \in \mathbb{R}^{d-k+1}$

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



$1 \times 1 + 4 \times 2 + 1 \times 3$

$1 \times 4 + 3 \times 2 + (-1) \times 1$

$3 \times 1 + -1 \cdot 2 + 2 \cdot 1$

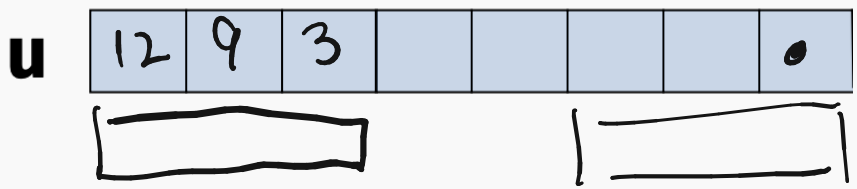
$w = (3\ 10\ 4)$

$w' \rightarrow (4\ 10\ 3)$

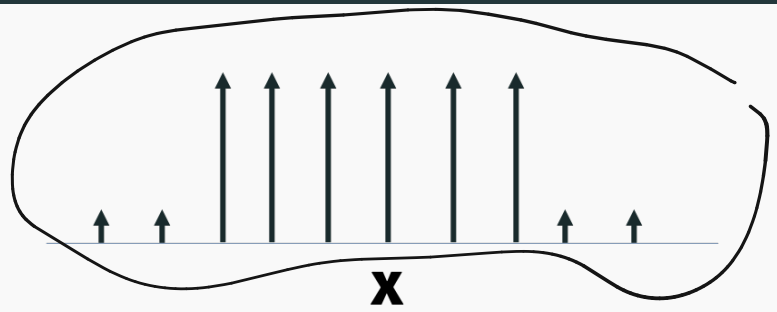
$d = 10$

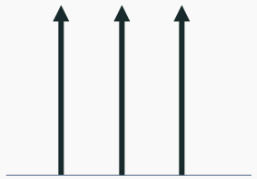
$k = 3$

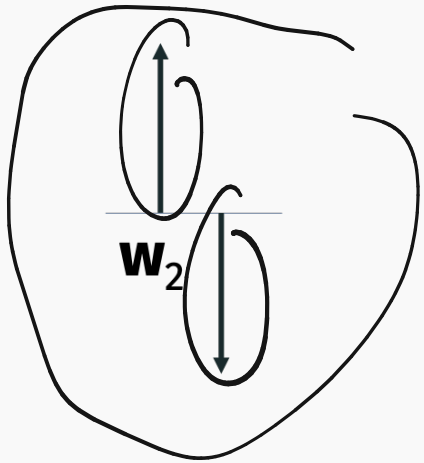
length $u = 8 = d - k + 1$



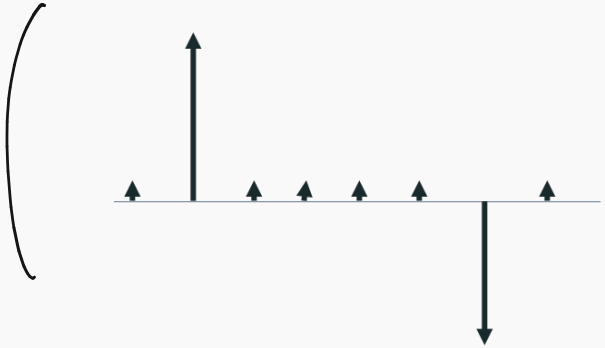
MATCH THE CONVOLUTION



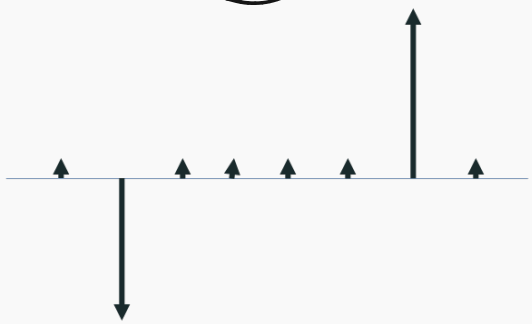




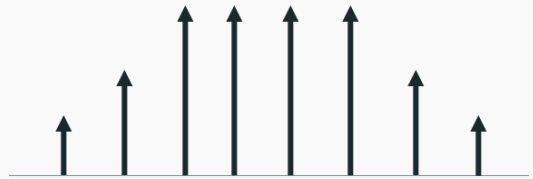
outputs



$x \circledast w_3$



$x \circledast w_2$



$x \circledast w_1$

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:

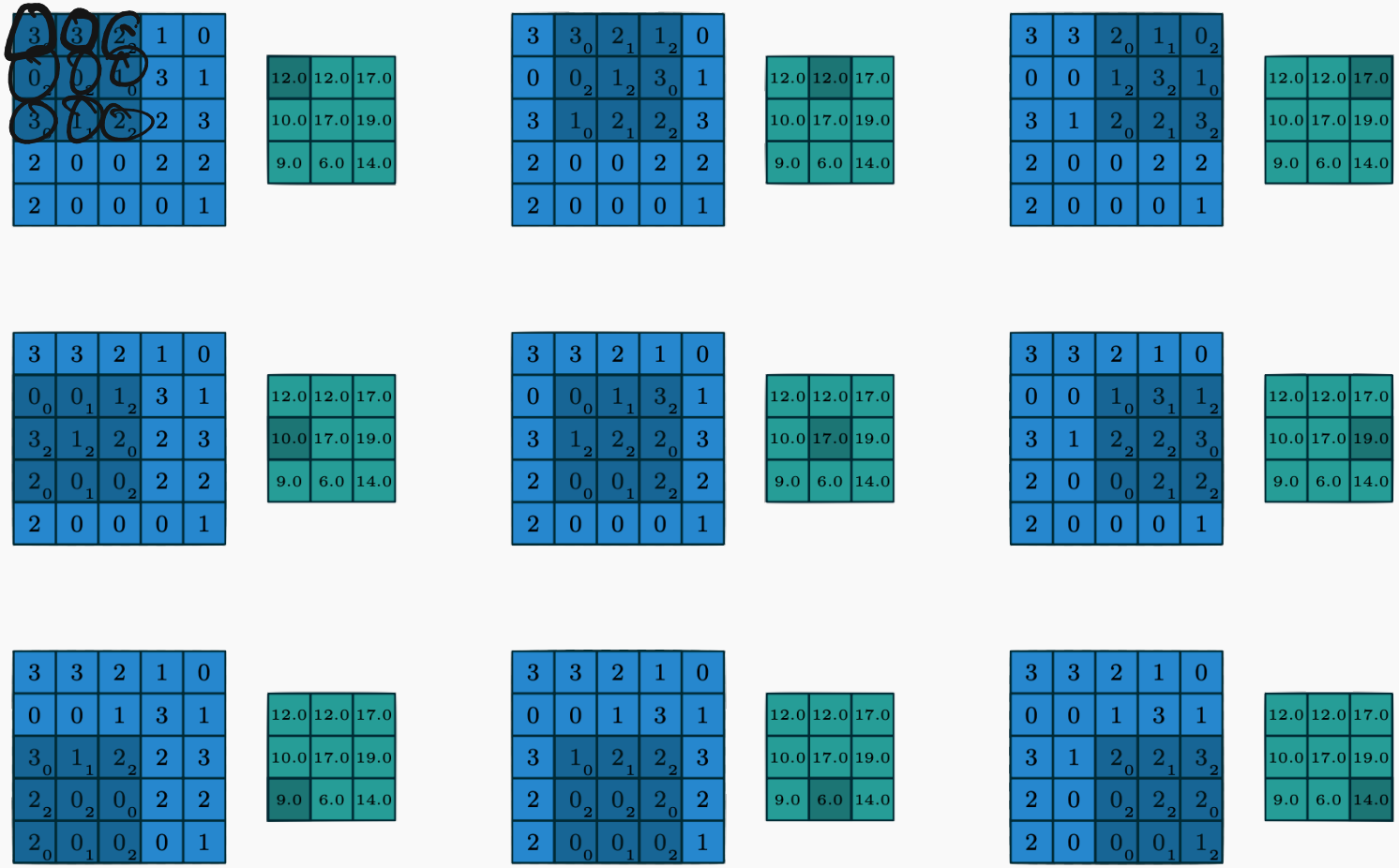
$$[\mathbf{x} \circledast \mathbf{w}]_{i,j} = \sum_{\ell=1}^{k_1} \sum_{h=1}^{k_2} \mathbf{x}_{(i+\ell-1),(j+h-1)} \cdot \mathbf{w}_{\ell,h}$$

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

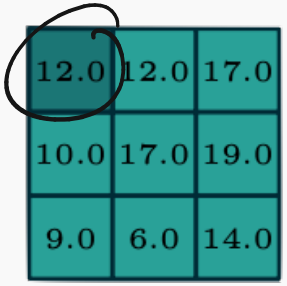
$$\mathbf{w} = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$



2D convolution

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

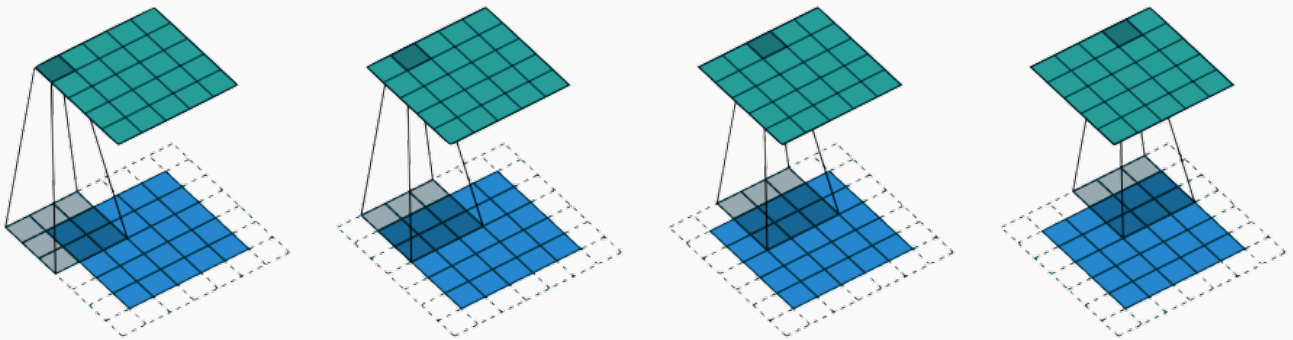
$3.0 + 3.1 + 2.2$
$+ 0.2 + 0.2 + 1.0$
$+ 3.0 + 1.1 + 2.2 = 12$



u

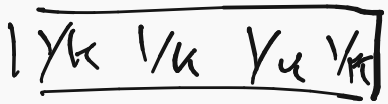
ZERO PADDING

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

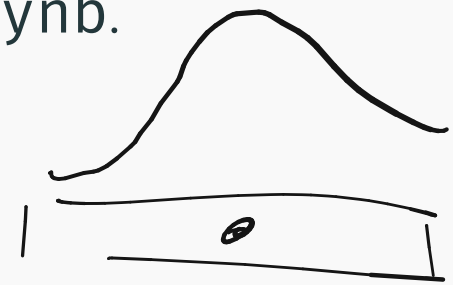


Need to pad on left and right by $(k_1 - 1)/2$ and on top and bottom by $(k_2 - 1)/2$.

APPLICATIONS OF CONVOLUTION



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

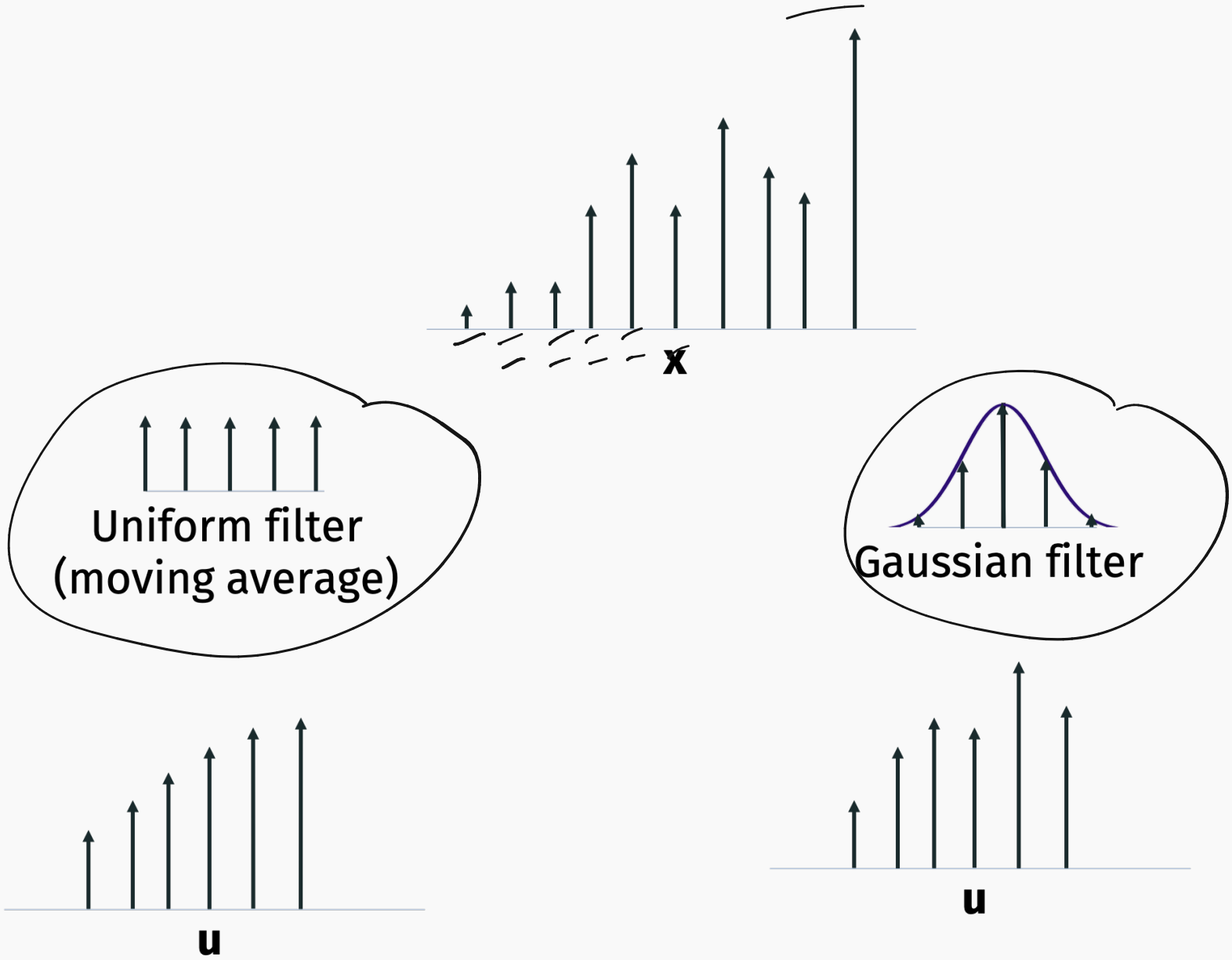


Application 1: Blurring/smooth.

In one dimension:

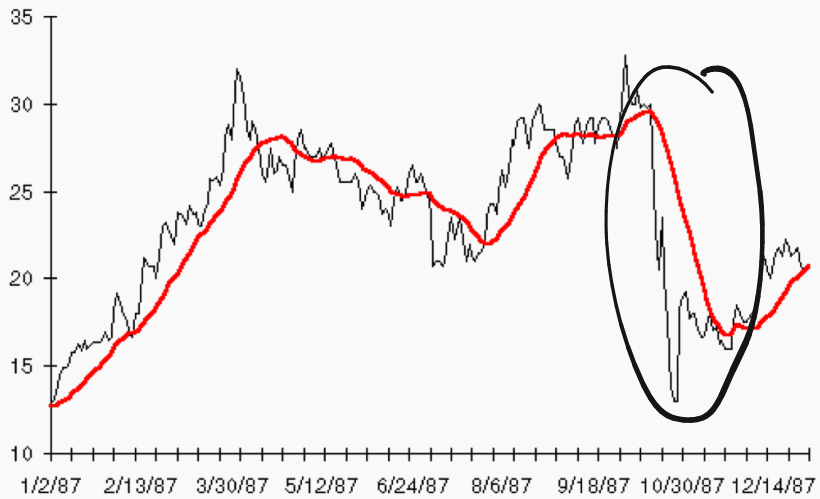
- Uniform (moving average) filter: $\mathbf{w}_i = \frac{1}{k}$ for $i = 1, \ldots, k$.
- Gaussian filter: $\mathbf{w}_i \sim \exp^{(i-k/2)^2/\sigma^2}$ for $i = 1, \ldots, k$.

SMOOTHING FILTERS



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

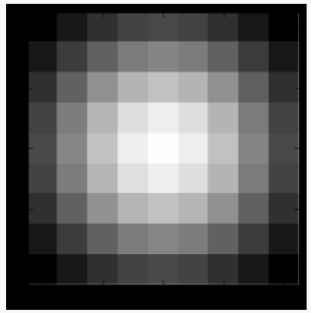
In two dimensions:

- 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

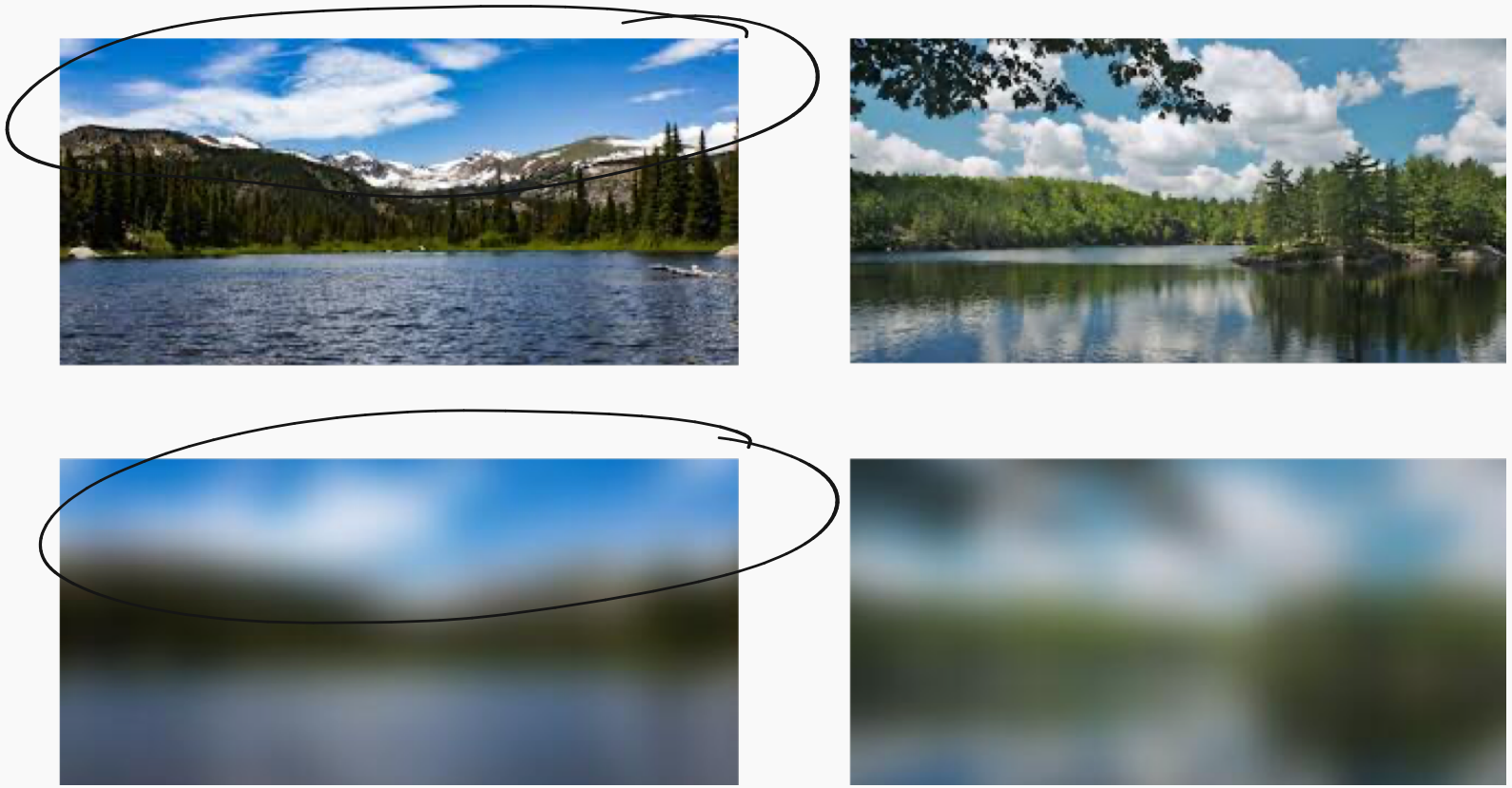
For Gaussian filter, you typically choose $k \gtrsim 2\sigma$ to capture the fall-off of the Gaussian.



Both approaches effectively denoise and smooth images.

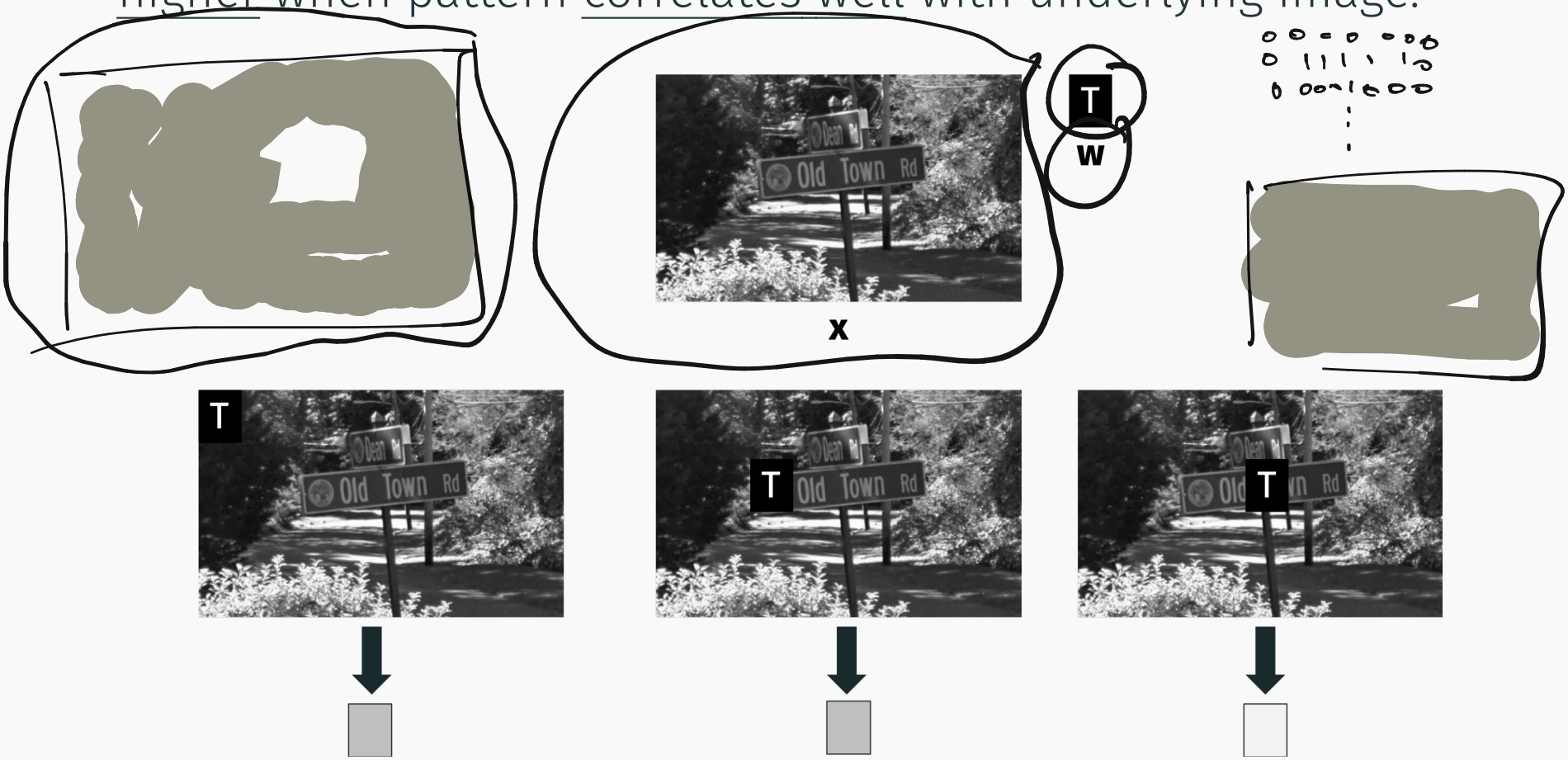
SMOOTHING FOR FEATURE EXTRACTION

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



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:

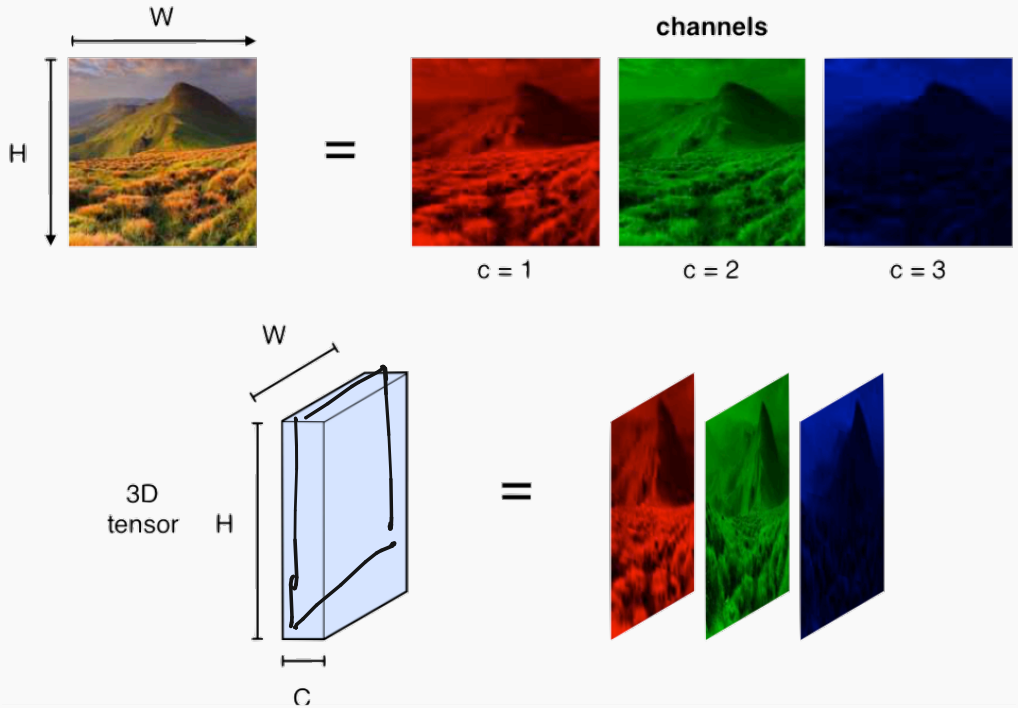
- Check if an image contains text.
- Look for specific sound in audio recording.
- Check for other well-structured objects

3D convolution

Recall that color images actually have three color channels for **red**, **green**, **blue**s. 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:

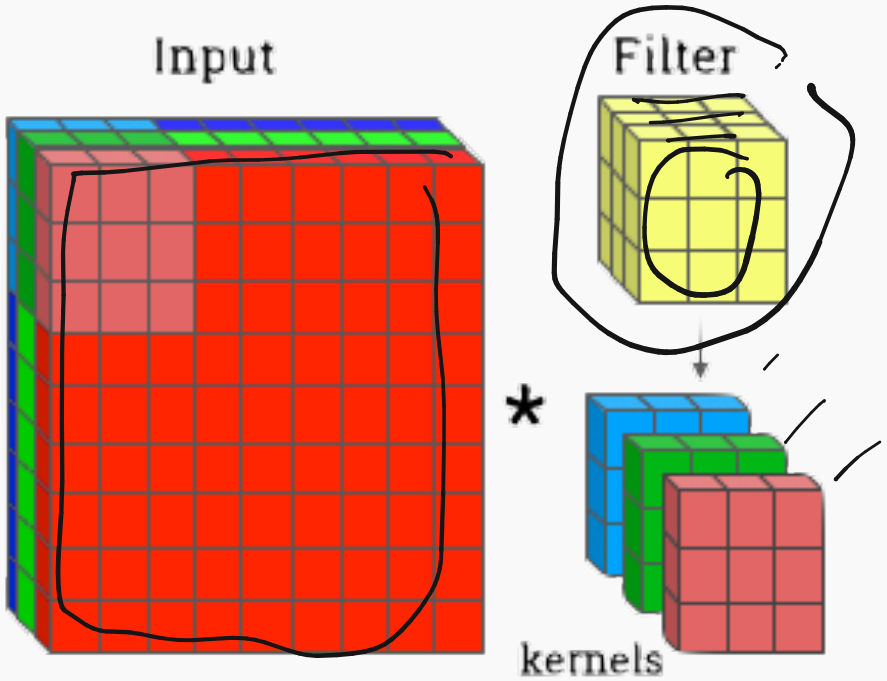


3D CONVOLUTION

Definition (Discrete 3D convolution)

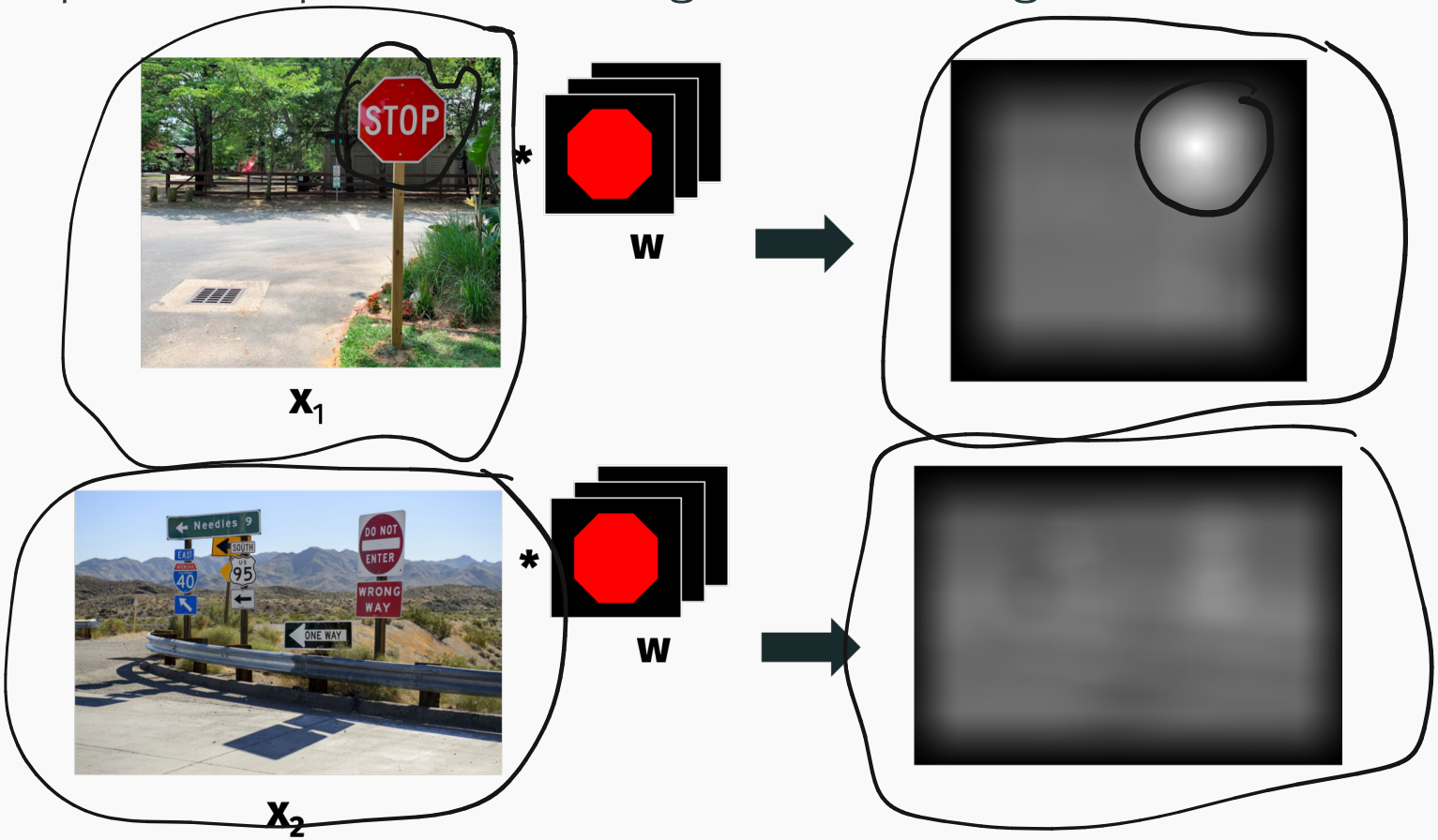
Given tensors $\mathbf{x} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $\mathbf{w} \in \mathbb{R}^{k_1 \times k_2 \times k_3}$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1) \times (d_3 - k_3 + 1)$ tensor with:

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

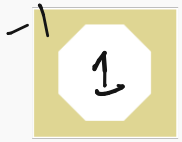


APPLICATION 2: PATTERN MATCHING

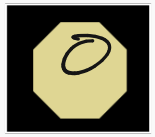
More powerful patter matching in color images:



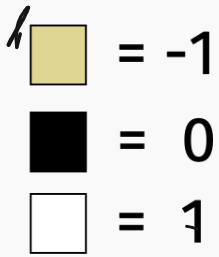
red channel



blue channel



green channel

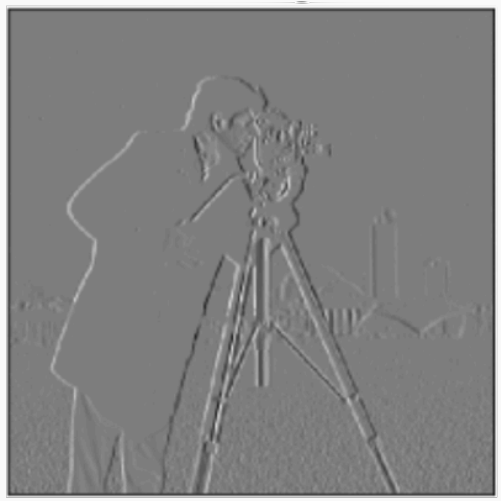


Application 3: Edge detection.

white = -1
black = 1

These are 2D edge detection filter:

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



x * ?

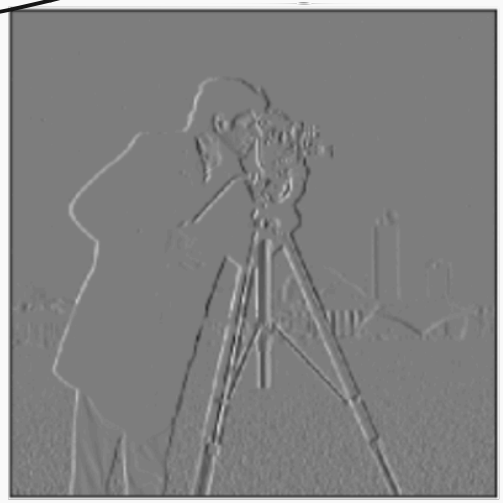
w_1

x * ?

w_2

Sobel filter is more commonly used:

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

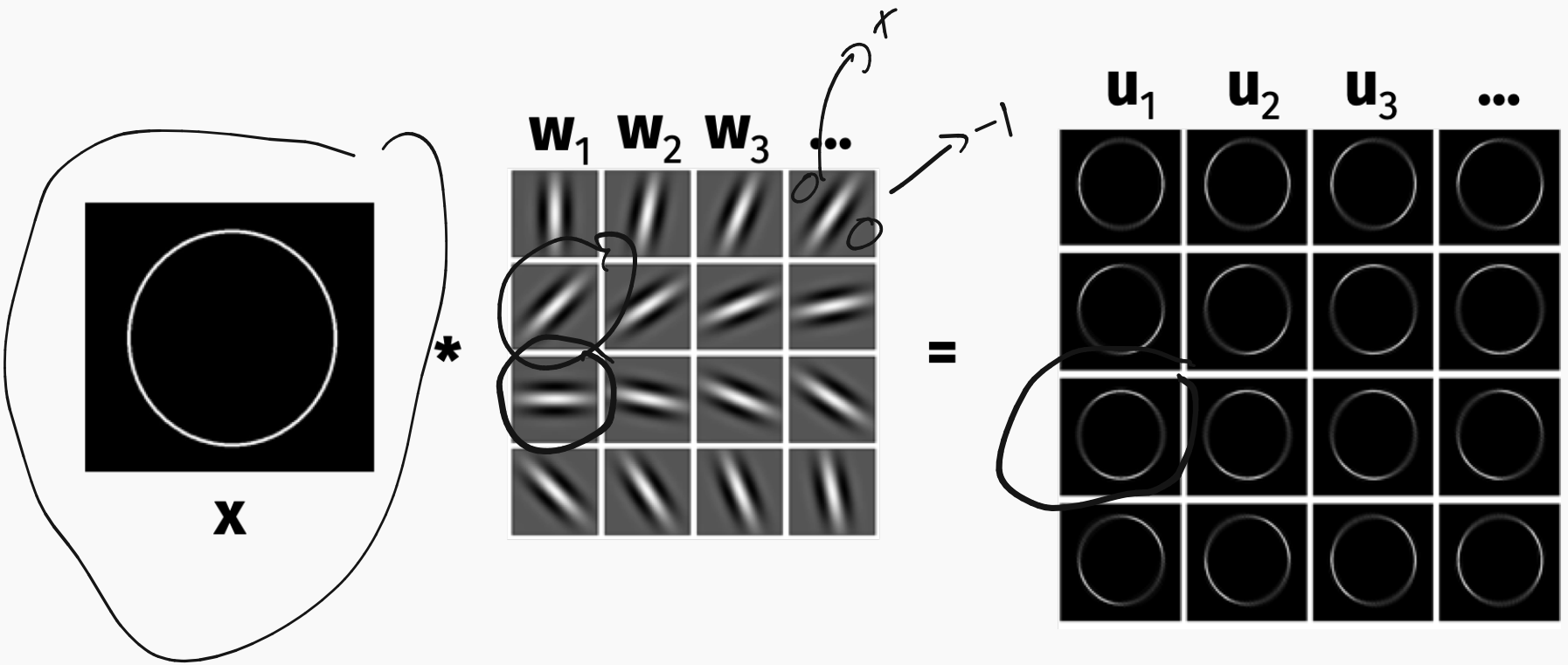


$x * ?$

$x * ?$

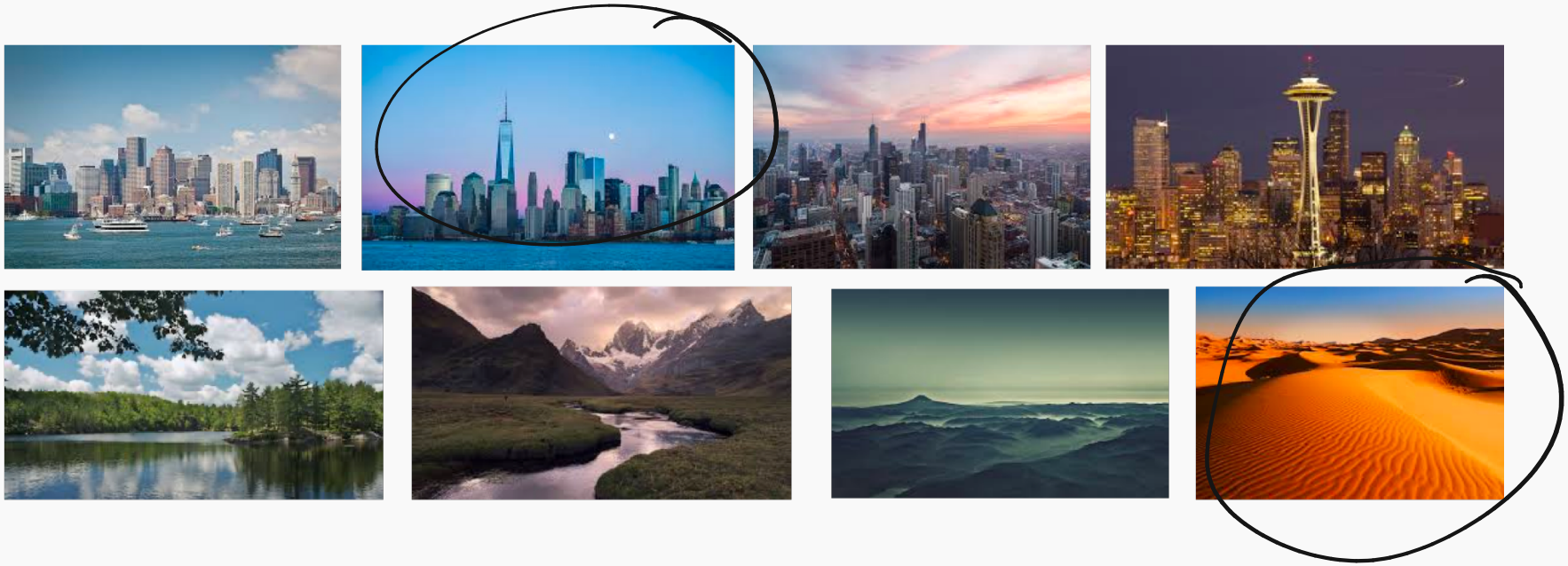
DIRECTIONAL EDGE DETECTION

Can define edge detection filters for any orientation.

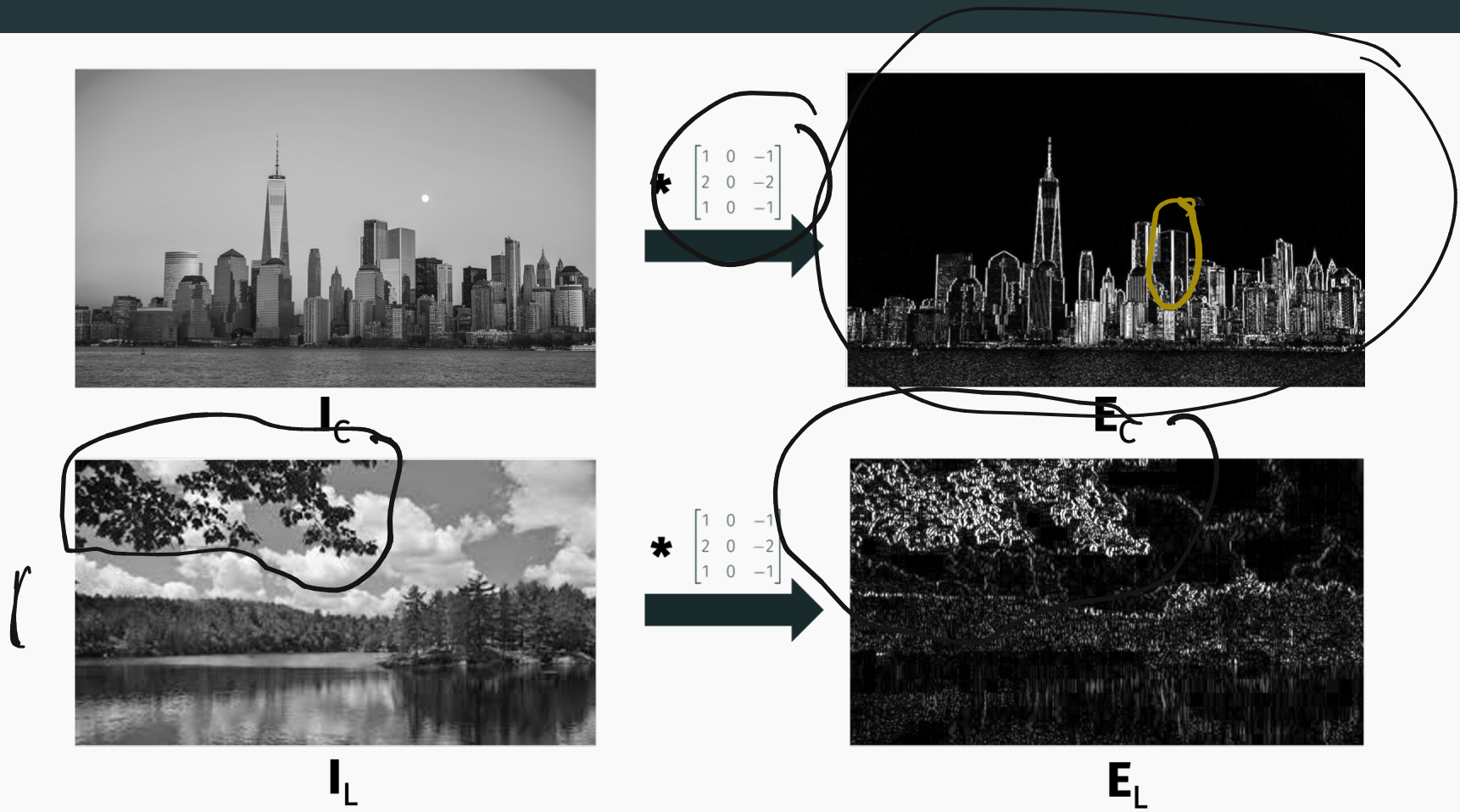


EDGE DETECTION

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



EDGE DETECTION

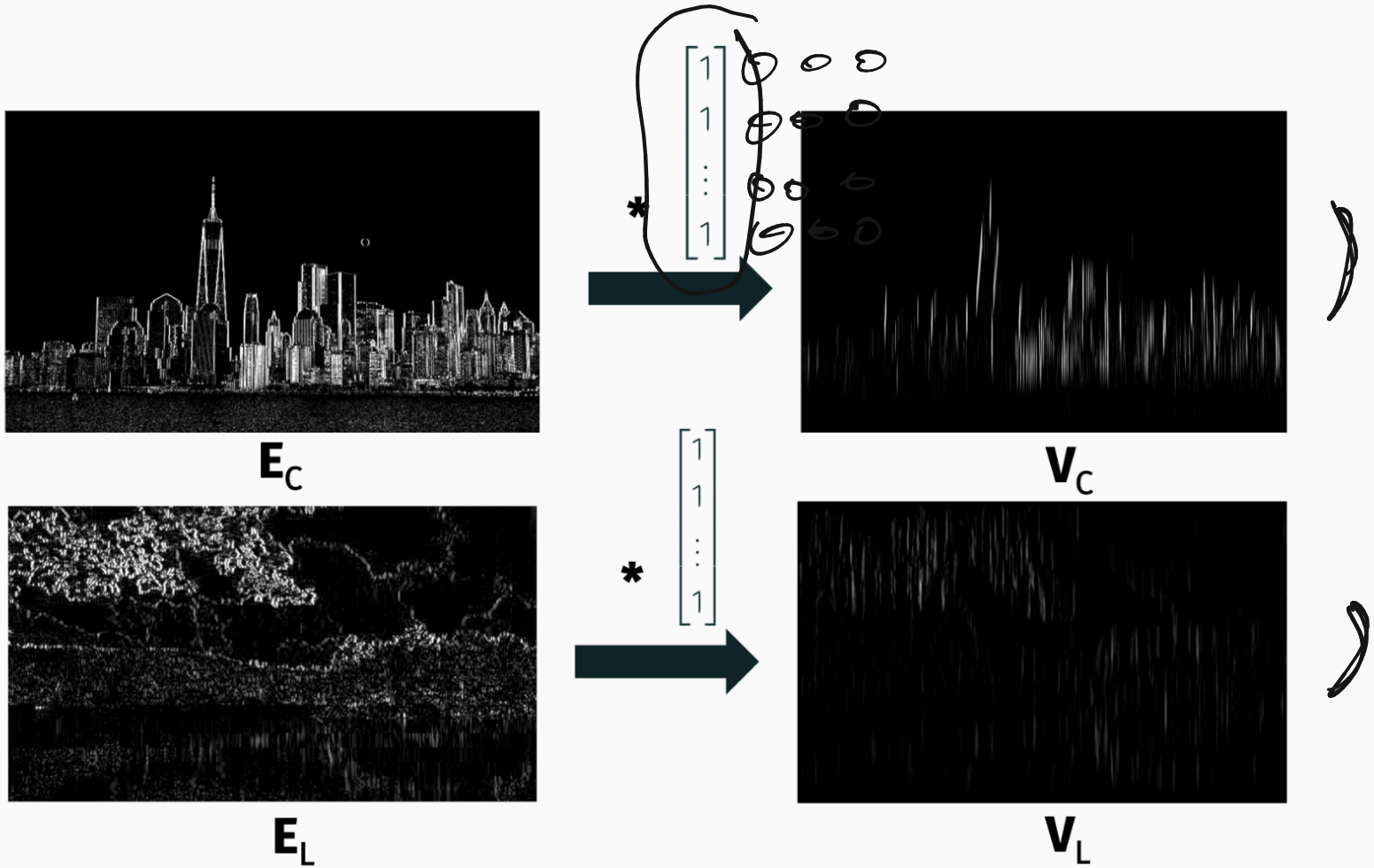


$$\texttt{mean}(\mathbf{E}_C) = .108 \quad \text{vs.} \quad \texttt{mean}(\mathbf{E}_L) = .123$$

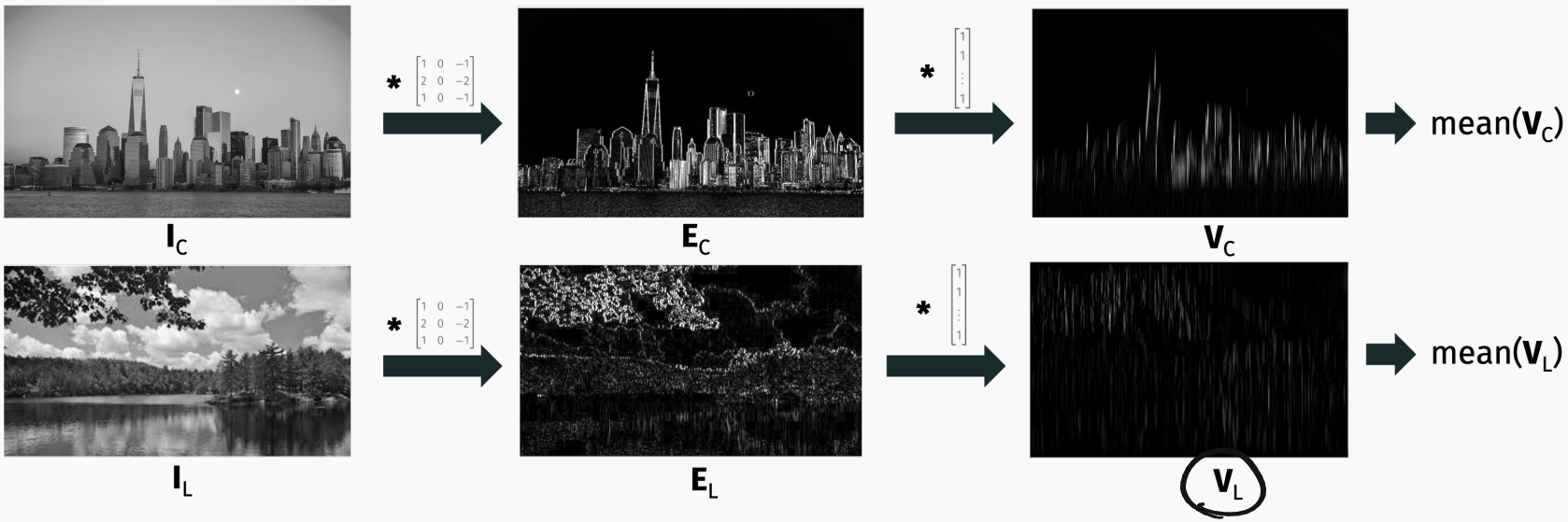
The image with highest vertical edge response isn't the city-scape.

EDGE DETECTION + PATTERN MATCHING

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



HIERARCHICAL CONVOLUTIONAL FEATURES



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

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

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

HIERARCHICAL CONVOLUTIONAL FEATURES

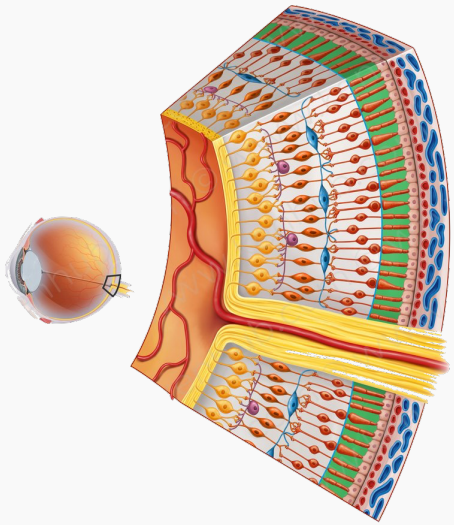
Hierarchical combinations of simple convolution filters are very powerful for understanding images.

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**.



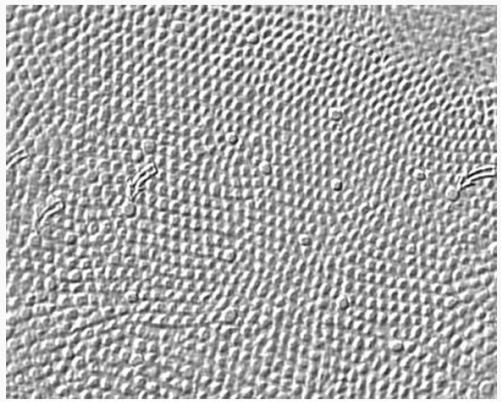
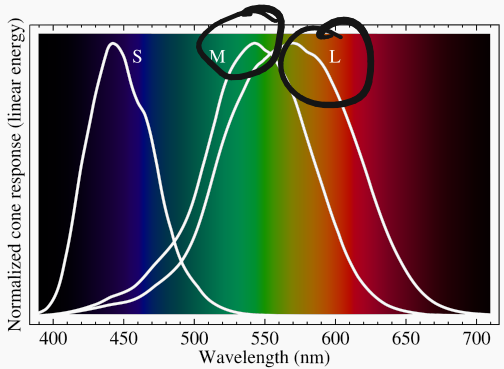


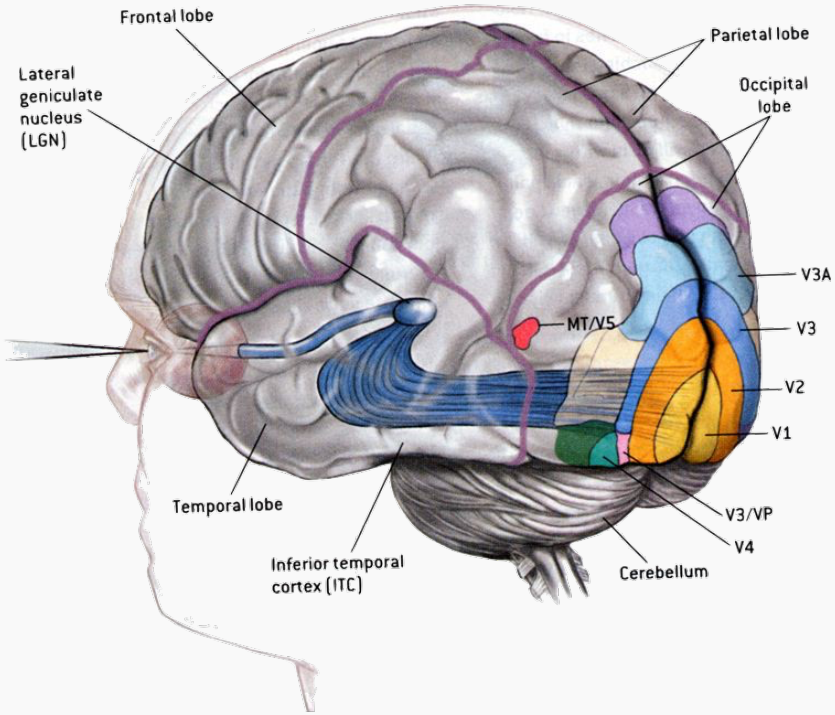
Fig. 13. Tangential section through the human fovea. ***Larger cones (arrows) are blue cones.*** *From Ahnelt et al. 1987.*

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.

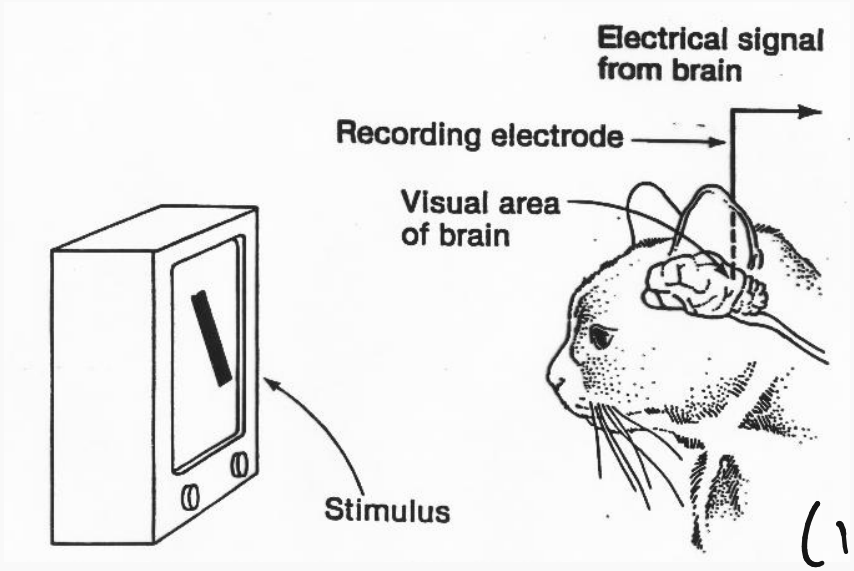


What sort of processing happens in the primary cortex?

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.



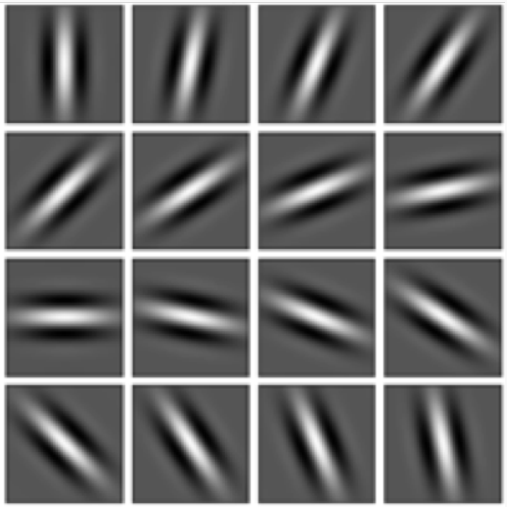
Different neurons fire when the cat is presented with stimuli at different angles. Cool video at `https://www.youtube.com/watch?v=OGxVfKJqX5E`.

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

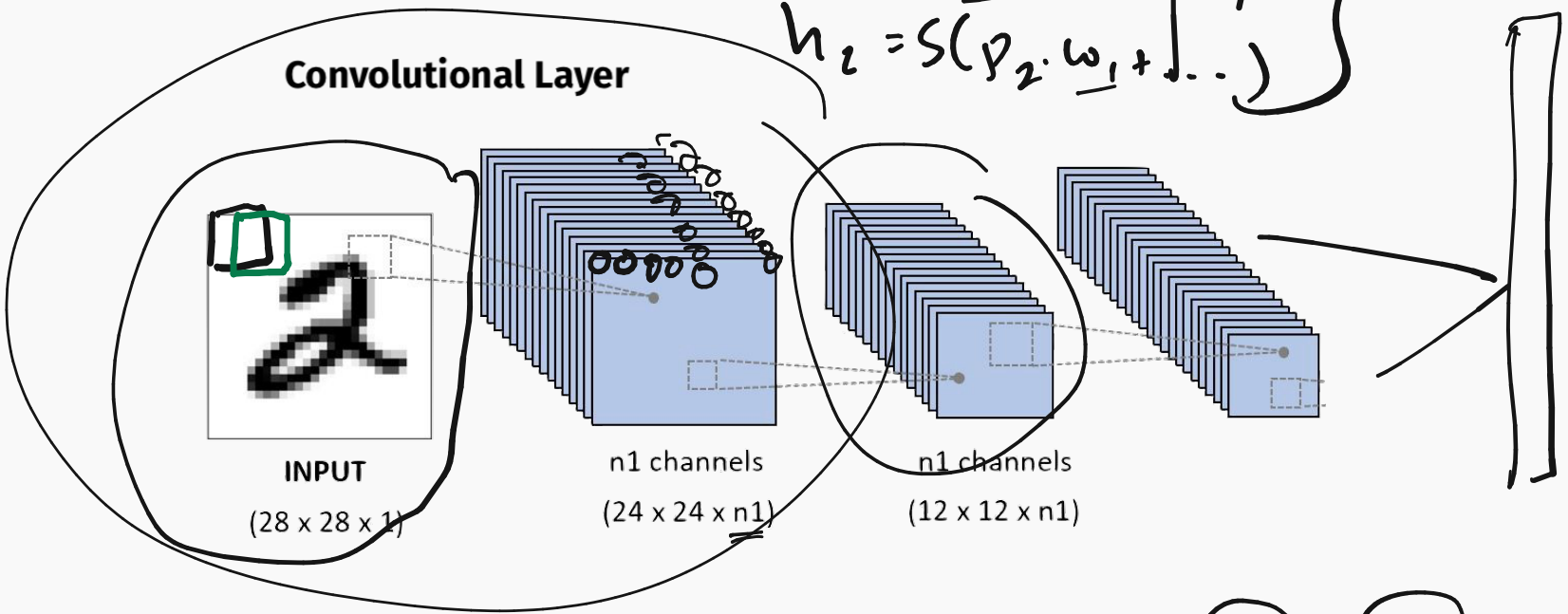
EXPLICIT FEATURE ENGINEERING

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.



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

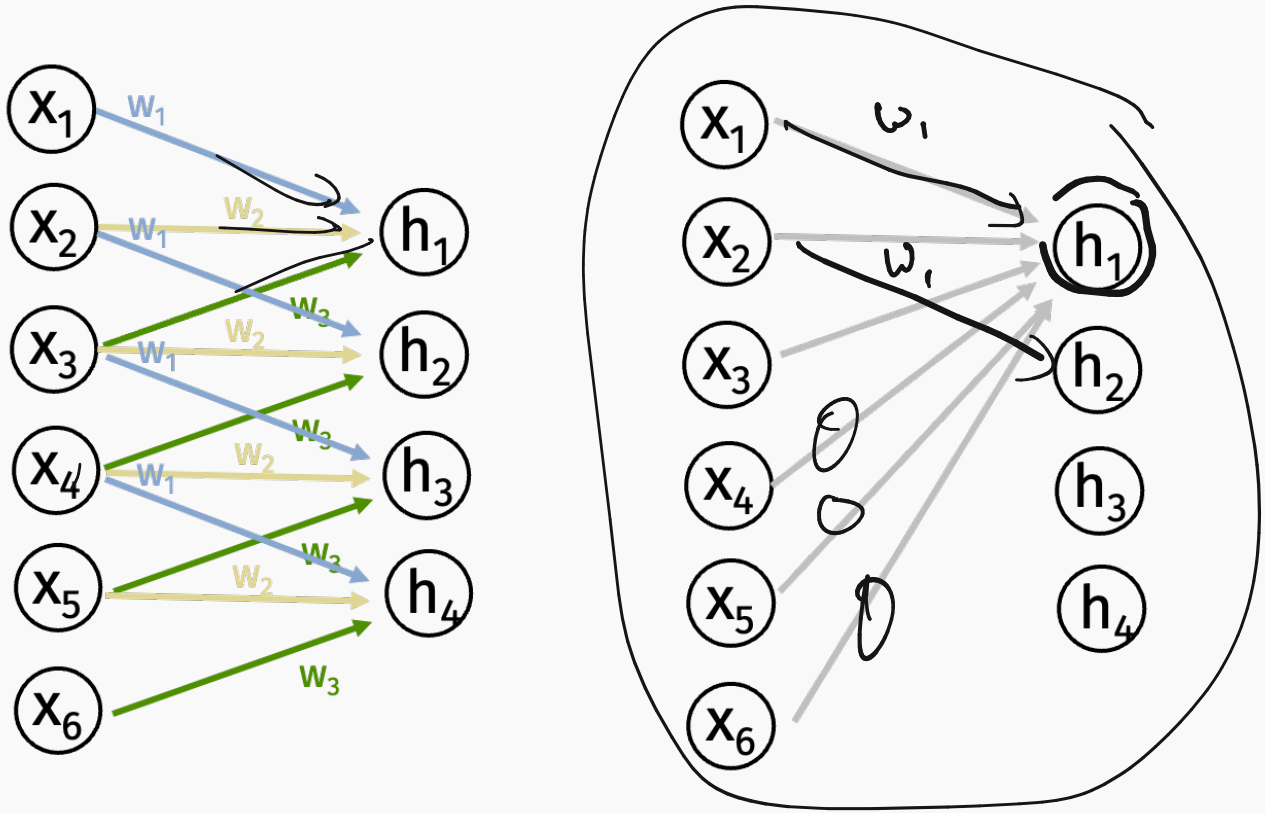


First convolutional layer involves n convolution filters $\mathbf{W}_1, \ldots, \mathbf{W}_n$. Each is small, e.g. 5×5. Every entry in $\mathbf{W}_i$ is a free parameter: $\sim 25 \cdot n$ parameters to learn.

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

WEIGHT SHARING

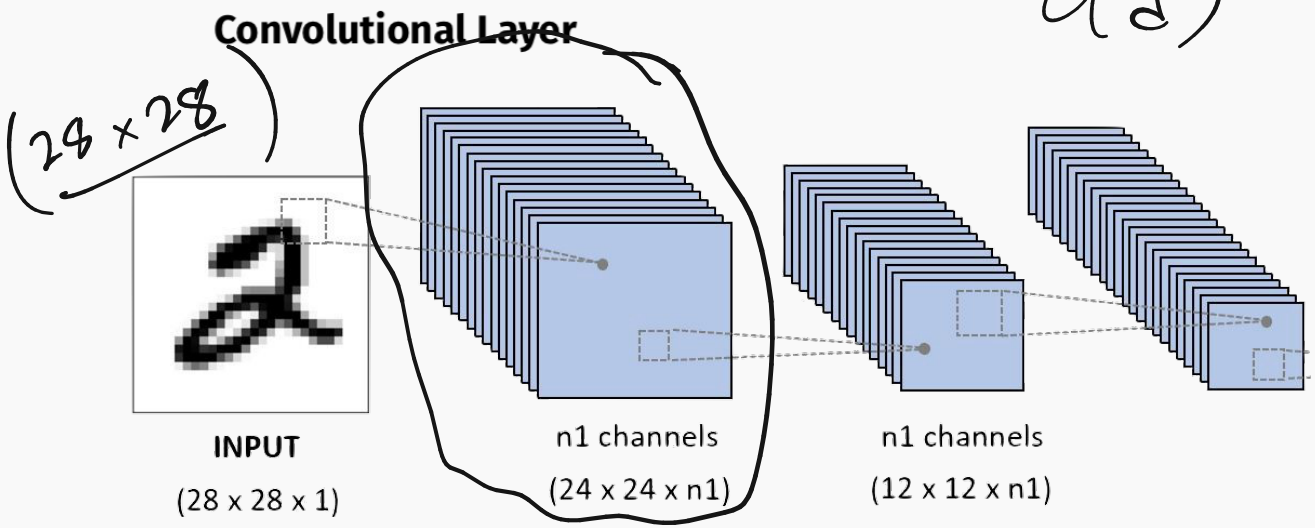
Convolutional layers can be viewed as fully connected layers with added constraints. Many of the weights are forced to 0 and we have weight sharing constraints.



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

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 $25n$.

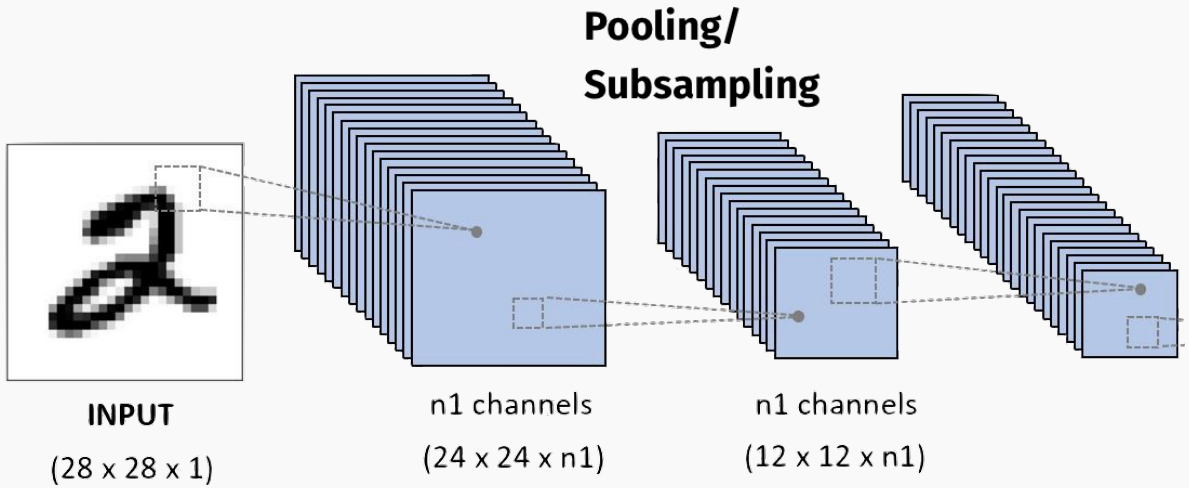
By "baking in" knowledge about what type of features matter, we greatly simply the network.



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

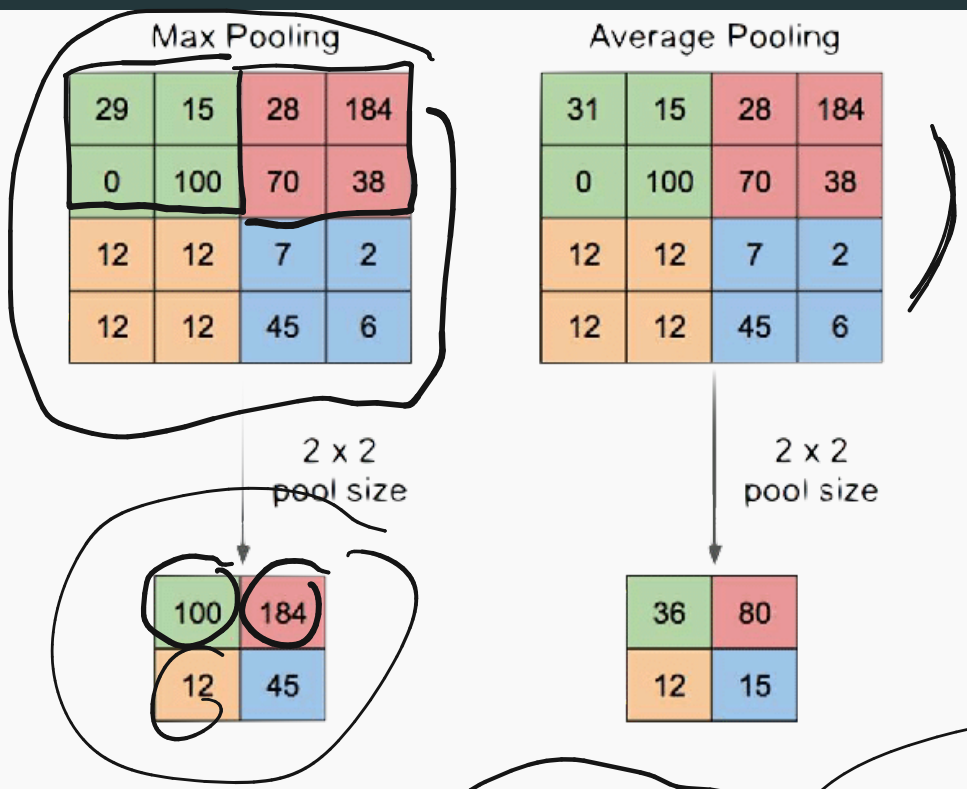
POOLING AND DOWNSAMPLING

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



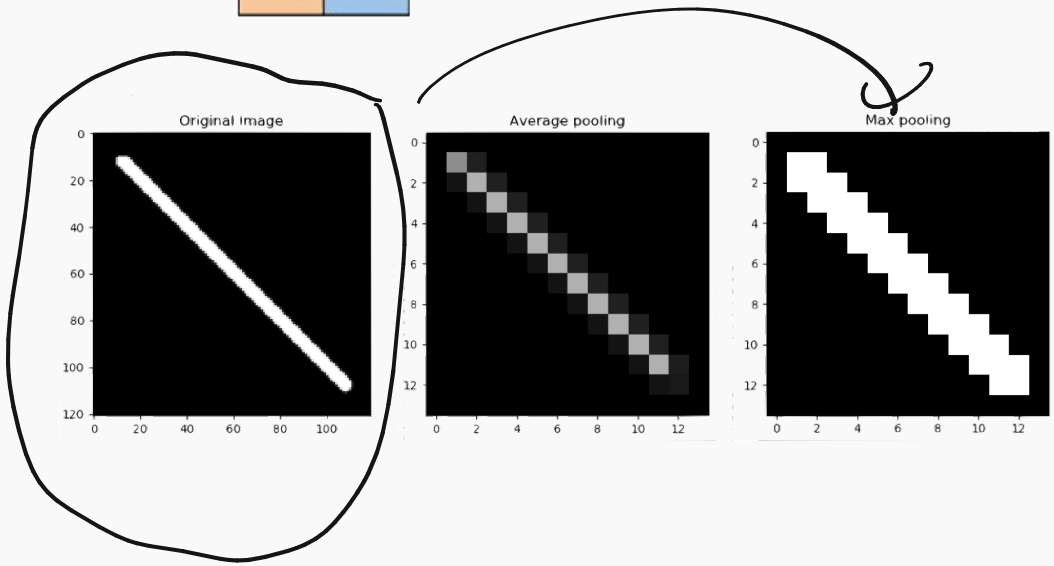
Most common approach is max-pooling.

POOLING AND DOWNSAMPLING

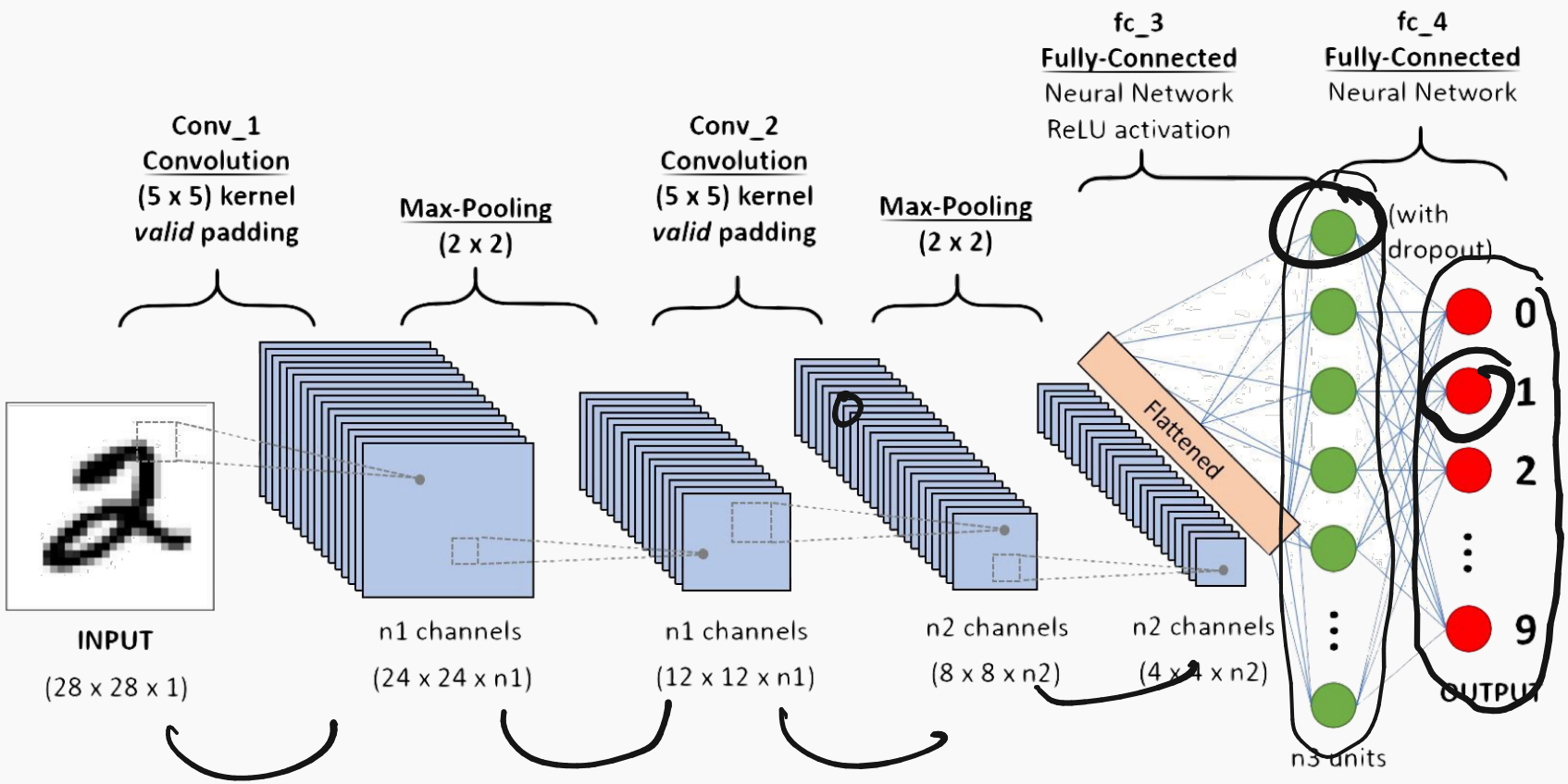


(0, 255)

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



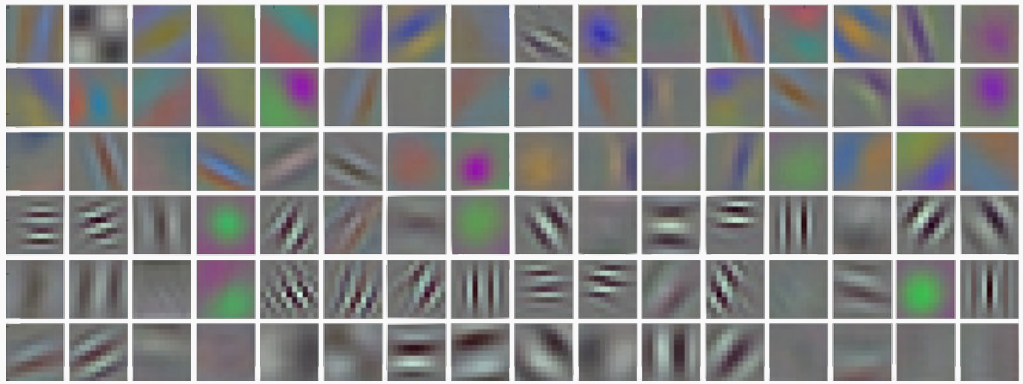
OVERALL NETWORK ARCHITECTURE



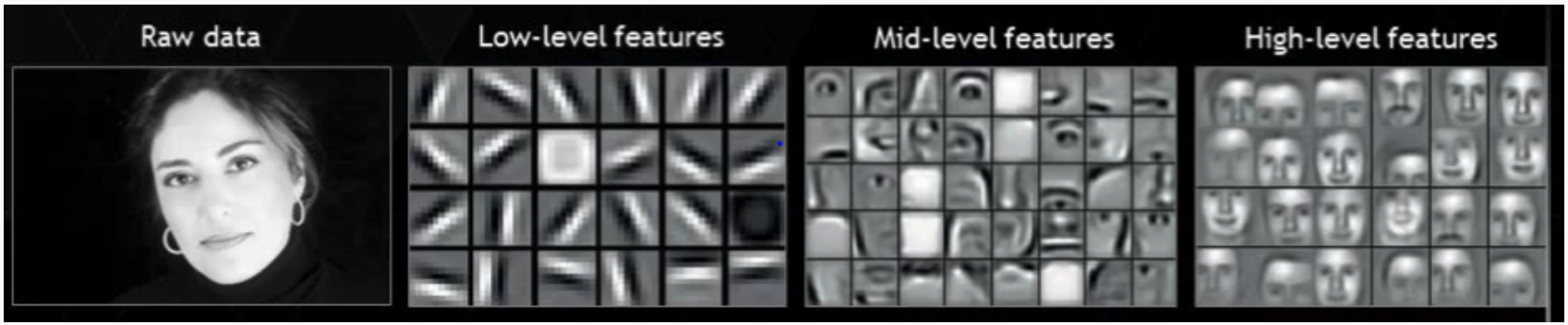
Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

UNDERSTANDING LAYERS

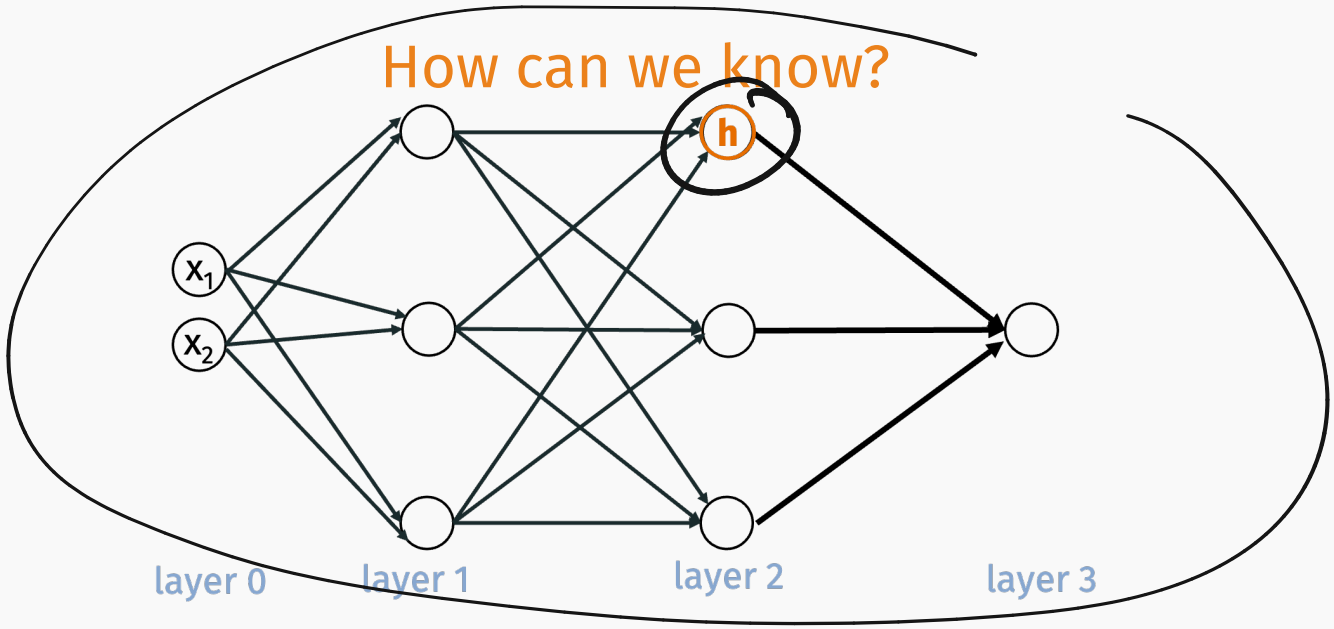
What type of convolutional filters do we learn from gradient descent?
Lots of edge detectors in the first layer!



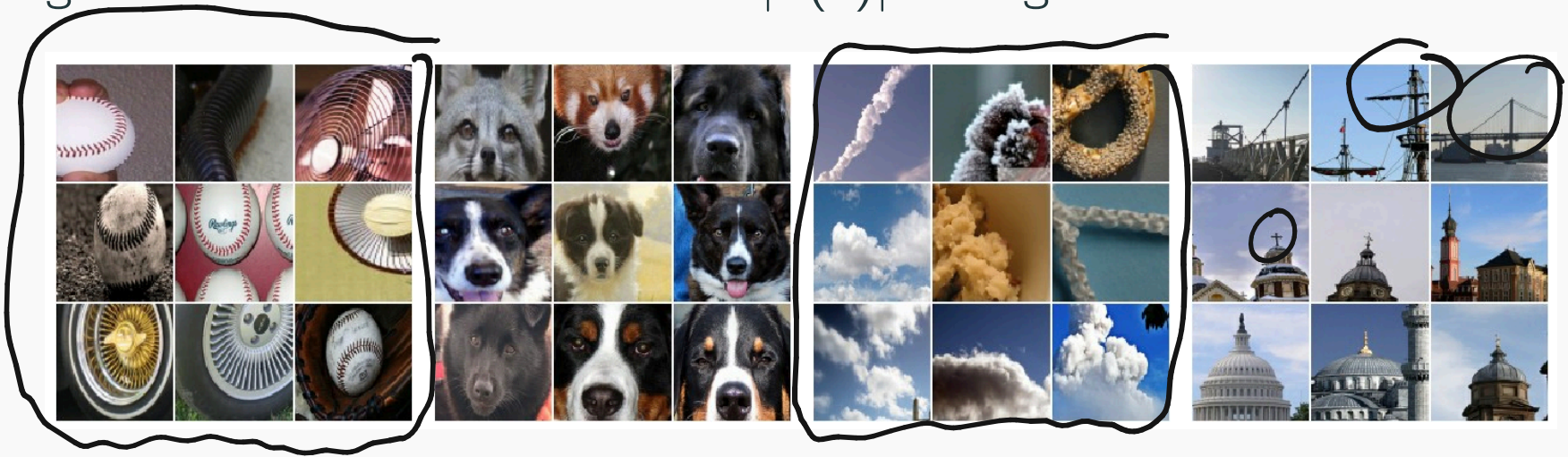
Other layers are harder to understand… but roughly hidden variables later in the network encode for “higher level features”:



UNDERSTANDING LAYERS



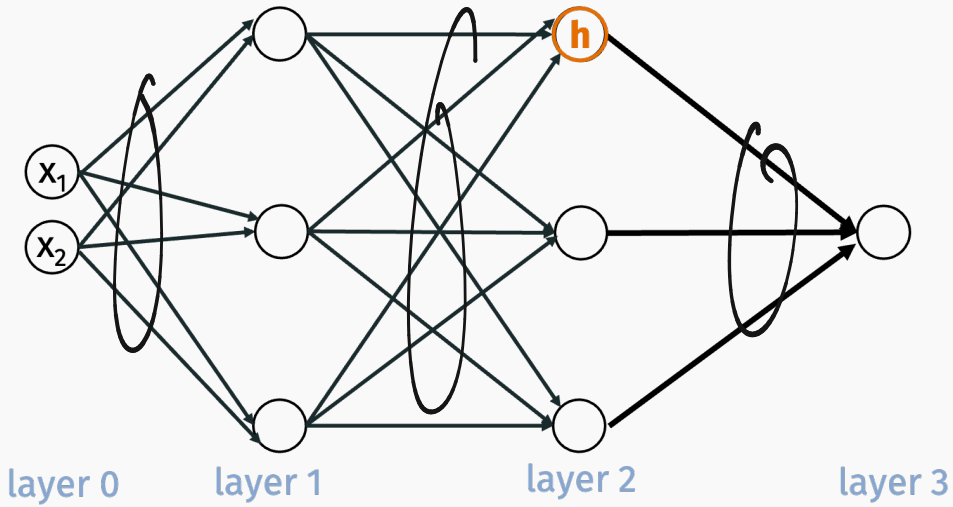
Go through dataset and find the inputs that most "excite" a given neuron h. I.e. for which $|h(\mathbf{x})|$ is largest.



UNDERSTANDING LAYERS

$h(x)$ $\max_x h(x) = \min_x -h(x)$ $\nabla h(x)$

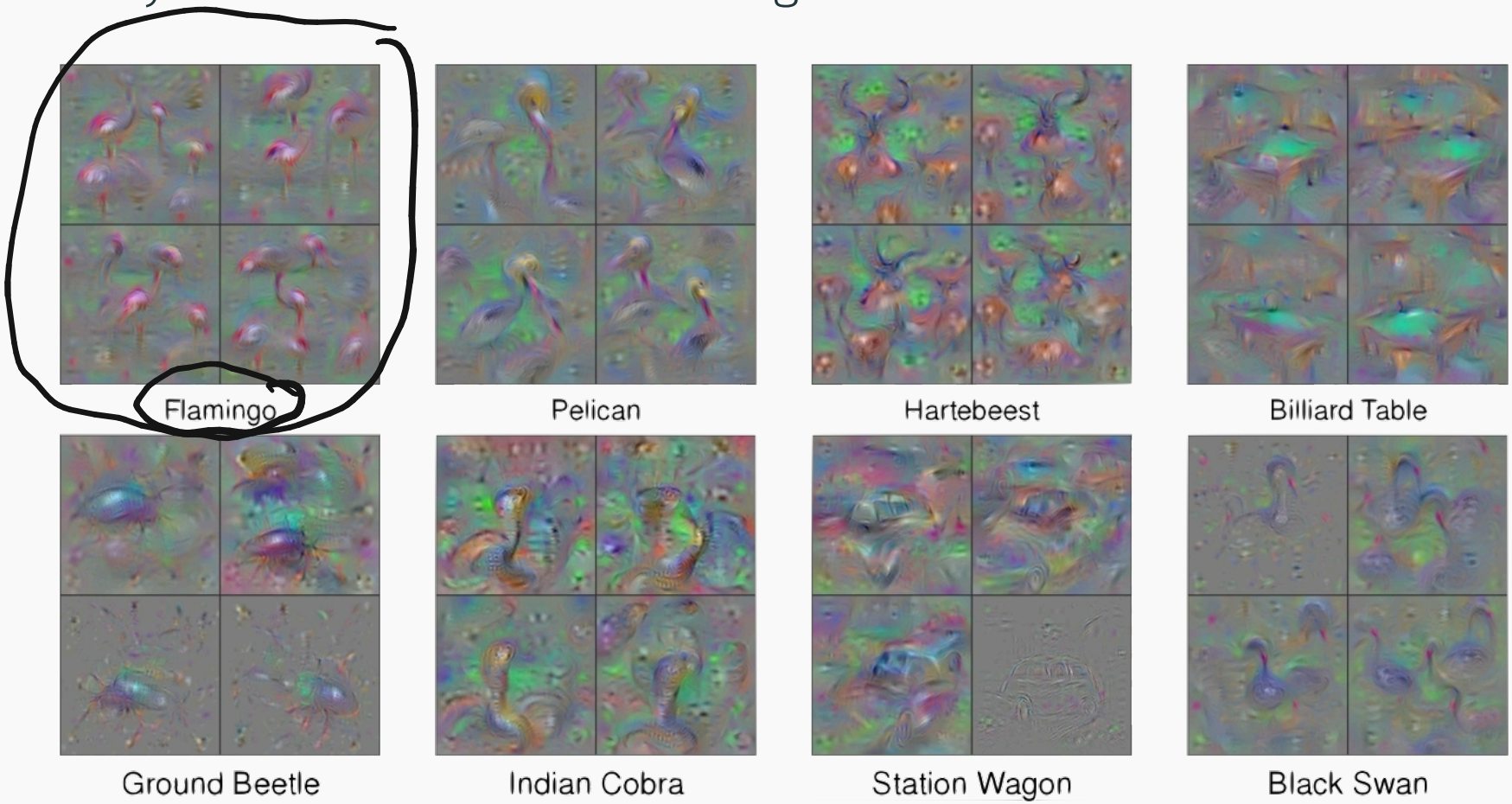
How can we know?



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

UNDERSTANDING LAYERS

Early work had some interesting results.



“Understanding Neural Networks Through Deep Visualization”, Yosinski et al.

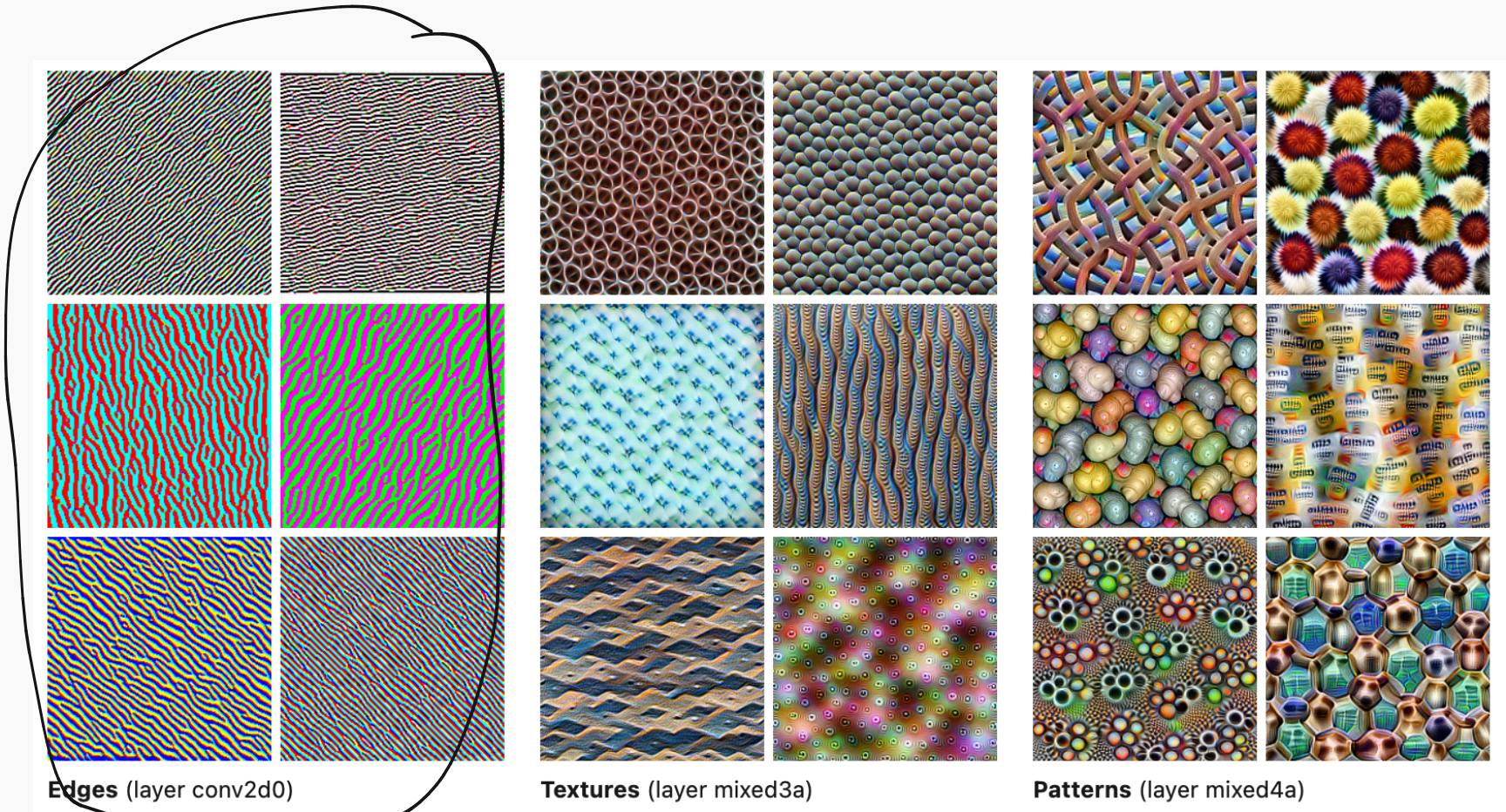
UNDERSTANDING LAYERS

There has been a lot of work on improving these methods by regularization. I.e. solve $\max_x |h(x)| + g(x)$ where g constrains 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.



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

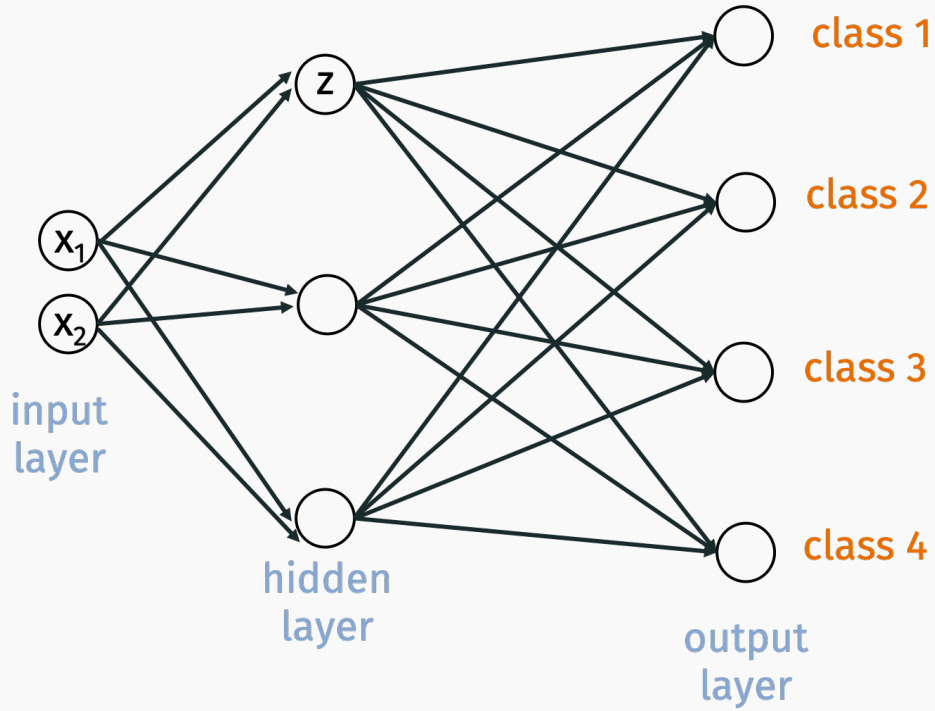
Beyond techinques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of “tricks”.

- 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.

$\bar{z}$ 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 $\mathbf{x}_1 \ldots, \mathbf{x}_n$:

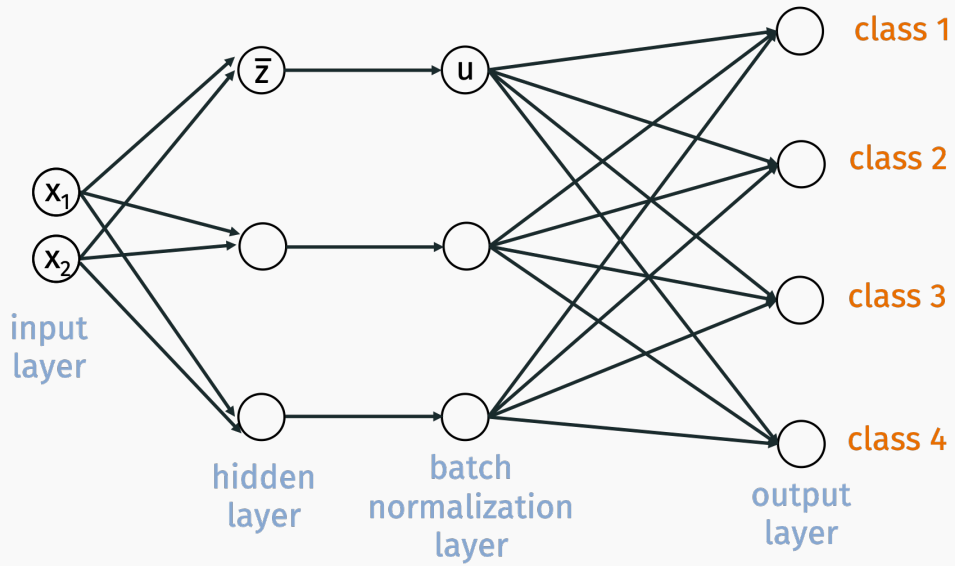
$$\mu = \frac{1}{n} \sum_{j=1}^{n} \bar{z}(\mathbf{x}_j)$$

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

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.

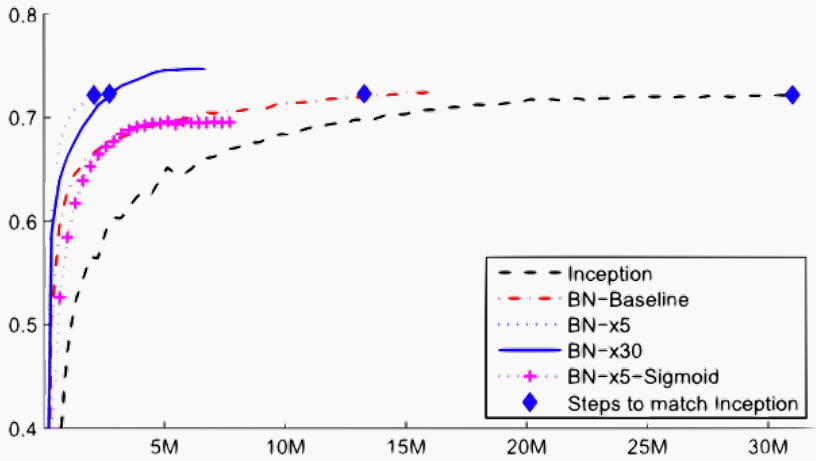


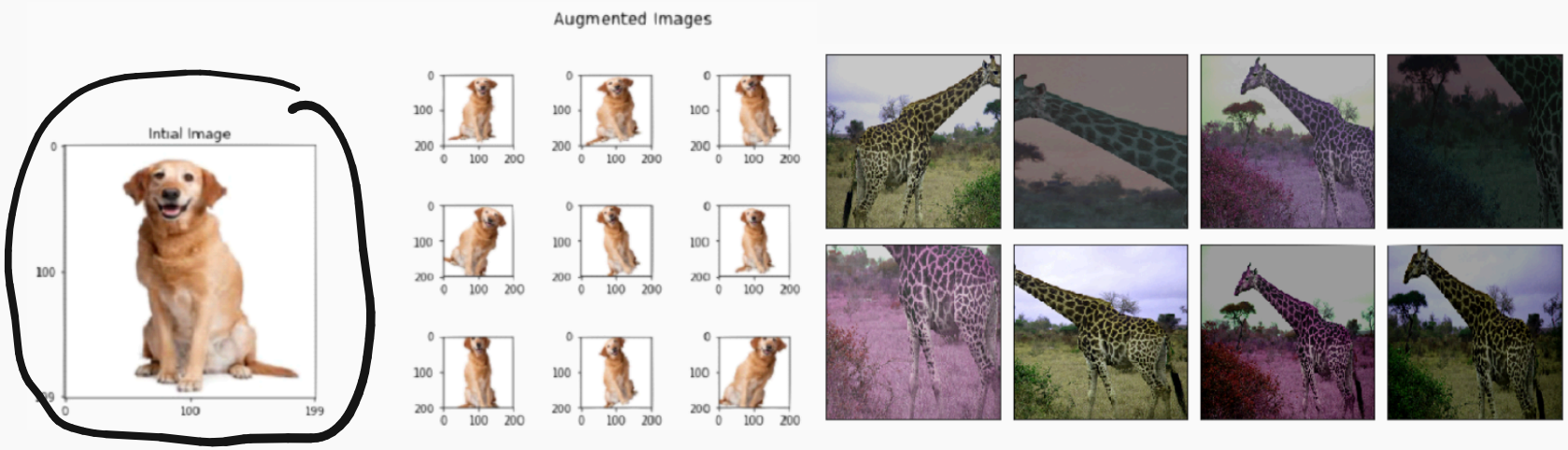
Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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.



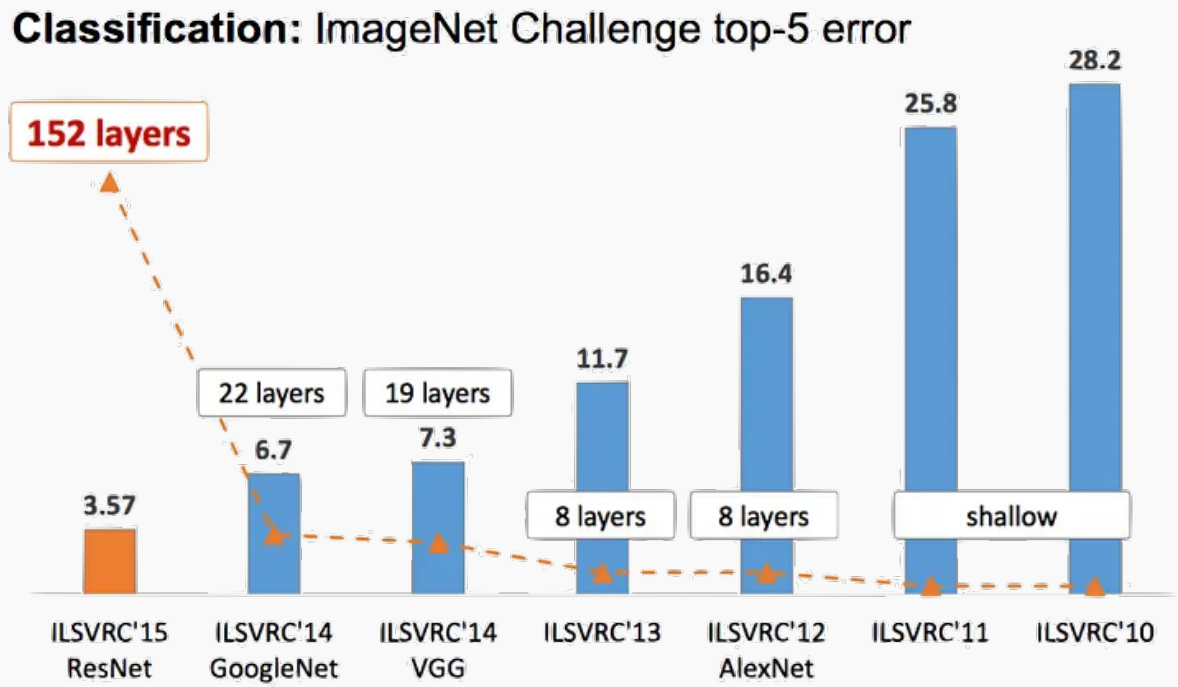
Take training images and randomly shift, flip, rotate, skew, darken, lighten, shift colors, etc. to create new training images. **Final classifier will be more robust to these transformations.**

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.

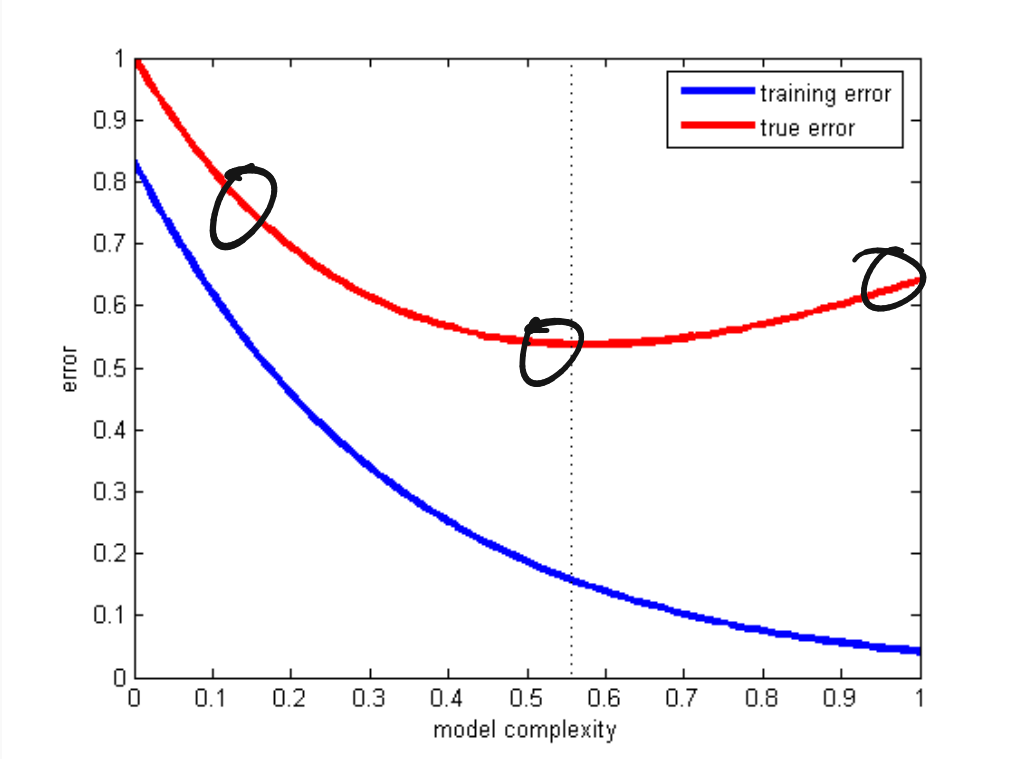
DEEPER AND DEEPER, BIGGER AND BIGGER

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



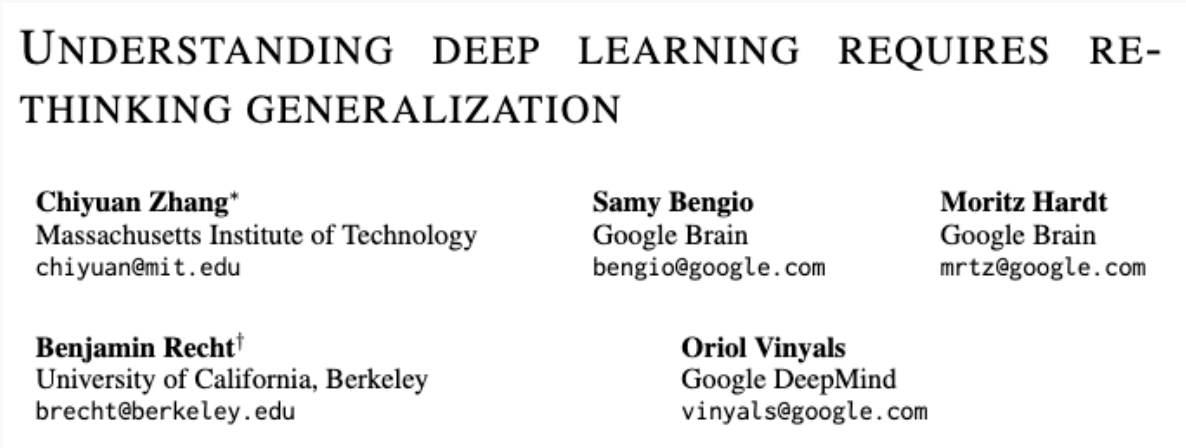
GENERALIZATION FOR NEURAL NETWORKS

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

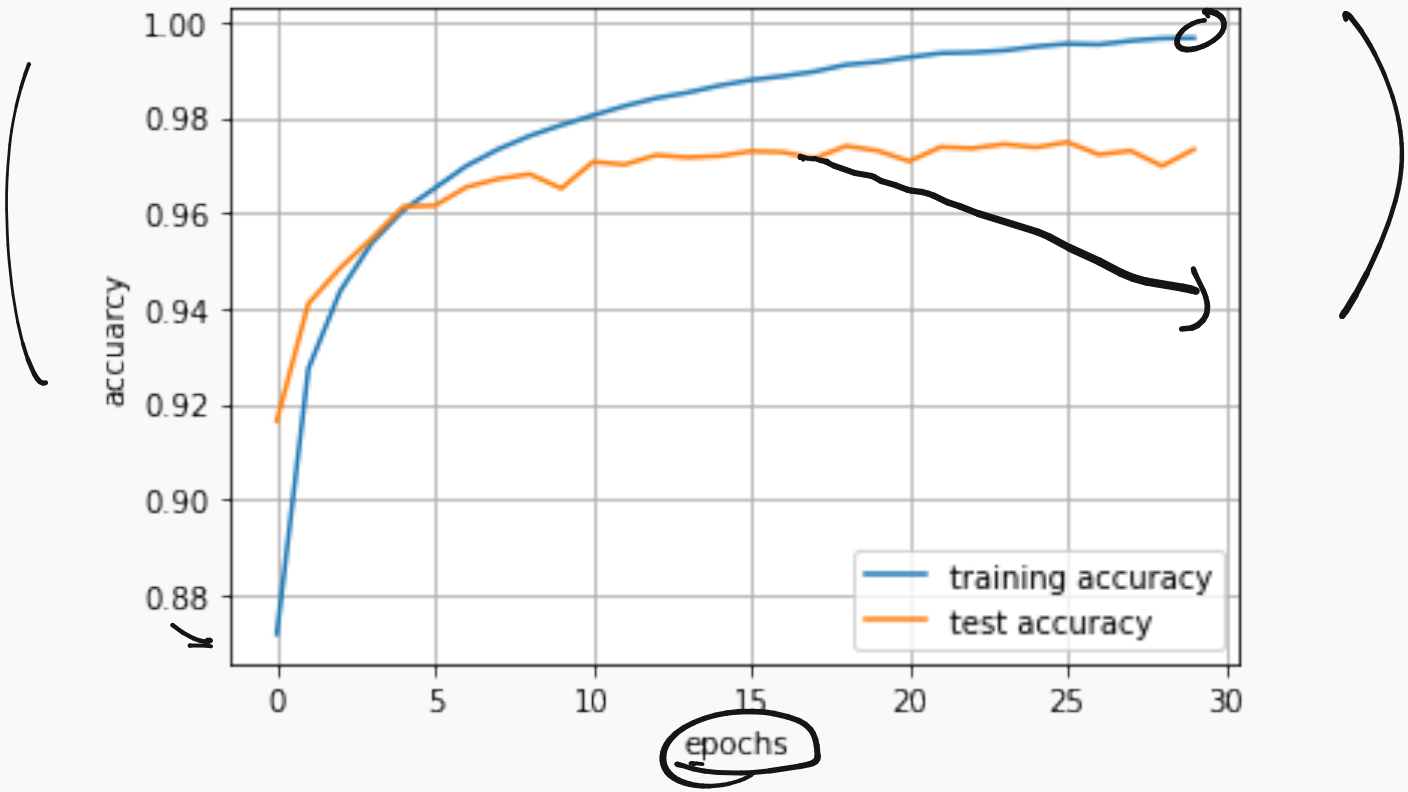
Moritz Hardt
Google Brain
mrtz@google.com

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

Oriol Vinyals
Google DeepMind
vinyals@google.com

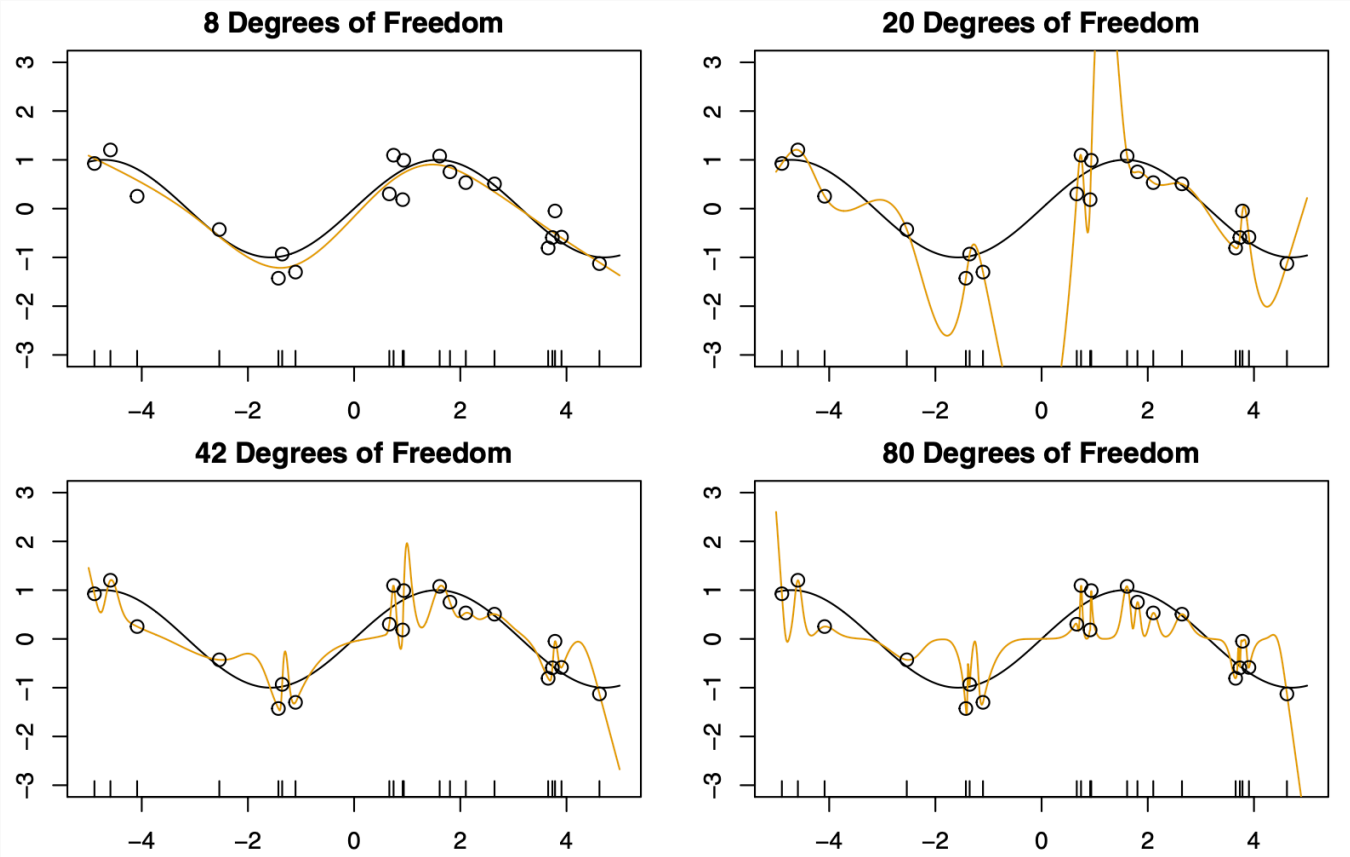
But we don't always see a large gap between training and test error. **Don't take this to mean overfitting isn't a problem when using neural nets!** It's just not always a problem.

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



GENERALIZATION FOR NEURAL NETWORKS

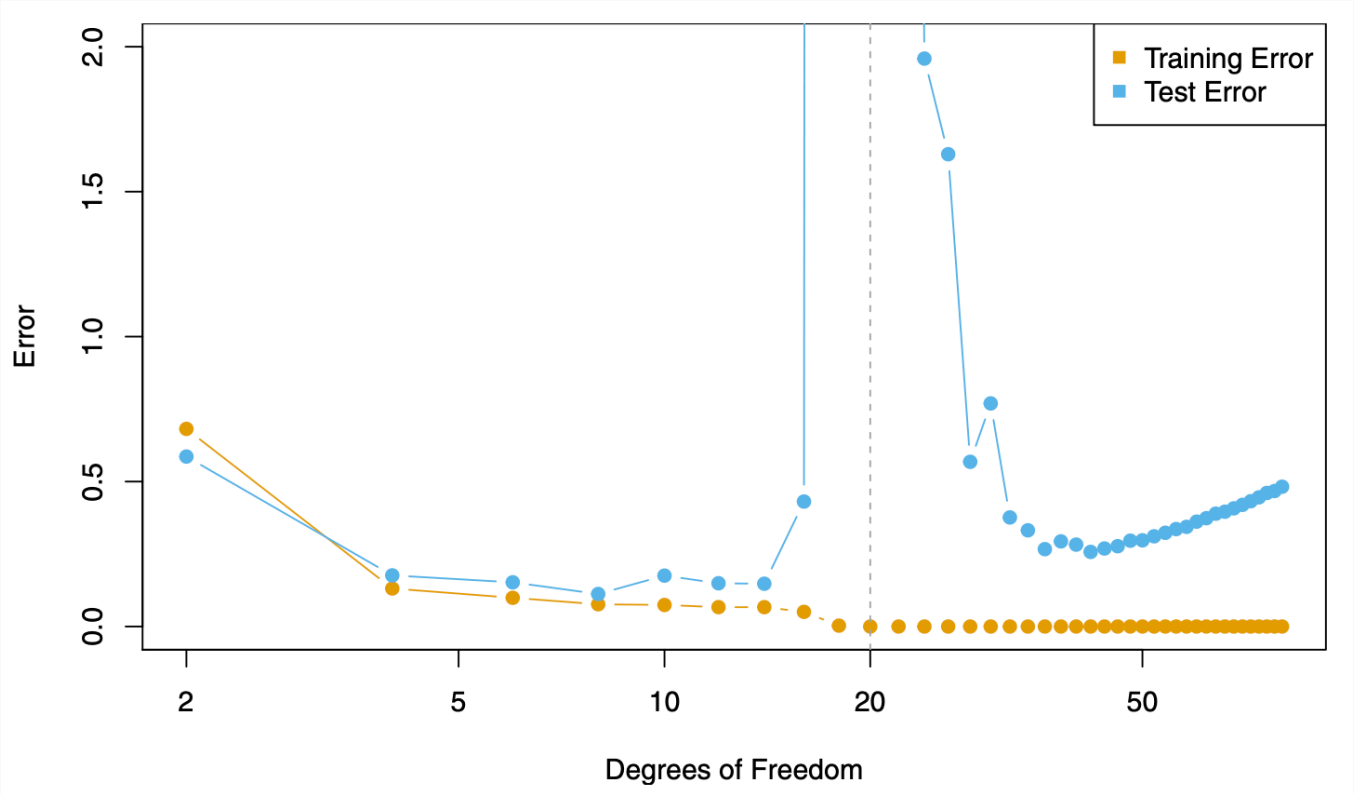
One growing realization is that this phenomena doesn't only apply to neural networks – it can also be true for fitting highly-overparameterized polynomials.



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.

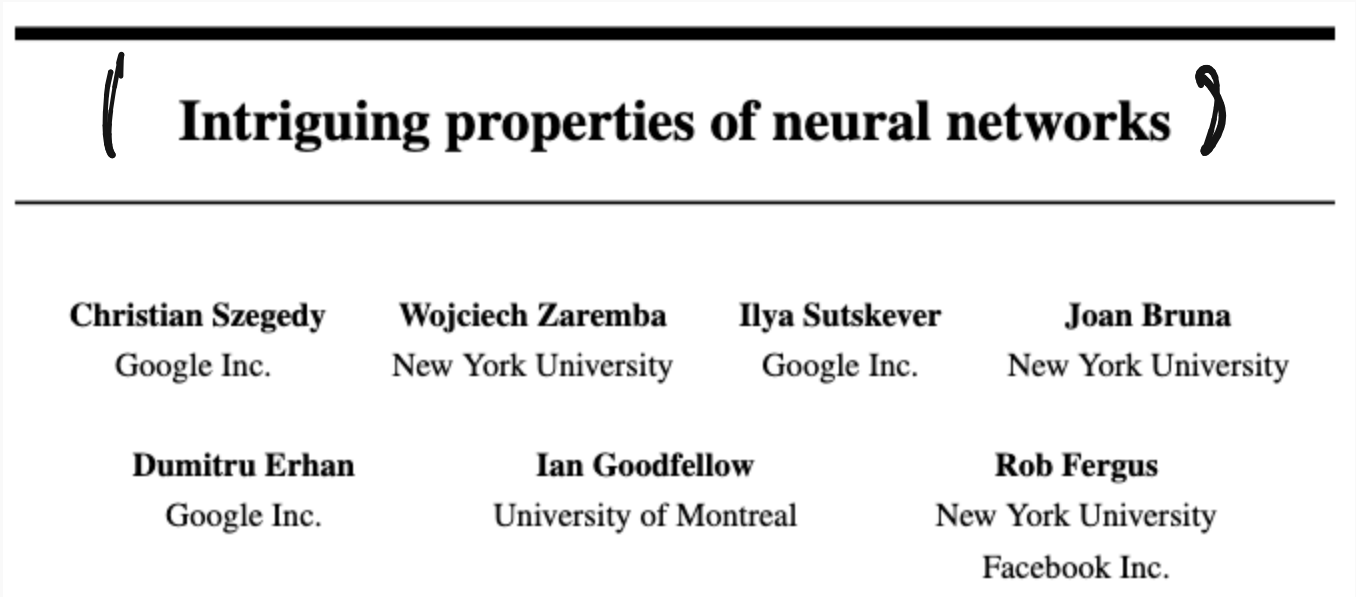


We don’t usually see this same curve for neural networks.

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

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!

How to find "good" perturbations:

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

Solve the optimization problem:

$$\max_{\delta, \|\delta\| \leq \epsilon} \ell(\theta, \mathbf{x} + \delta, y) - \ell(\theta, \mathbf{x} + \delta, y_{\text{true}})$$

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).

TRANSFER LEARNING

ONE-SHOT LEARNING

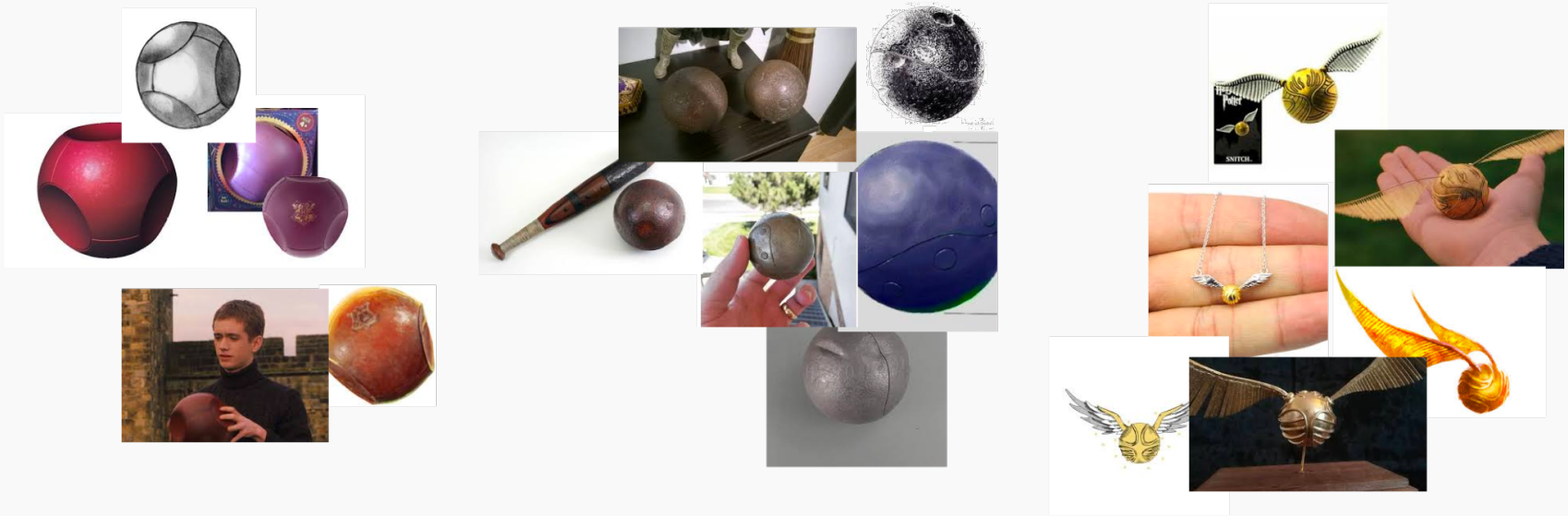
What if you want to apply deep convolutional networks to a problem where you do not have a lot of **labeled data** in the first place?

quaffle bludger snitch

Example: Classify images of different Quidditch balls.

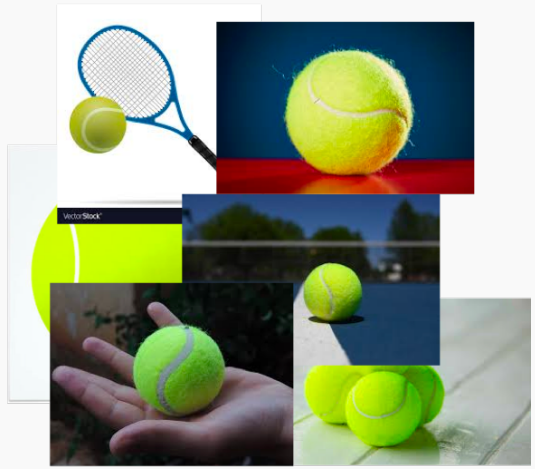
ONE-SHOT LEARNING

A human could probably achieve near perfect classification accuracy even given access to a **single labeled example** from each class:



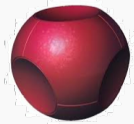
Major question in ML: How? Can we design ML algorithms which can do the same?

Transfer knowledge from one task we already know how to solve to another.

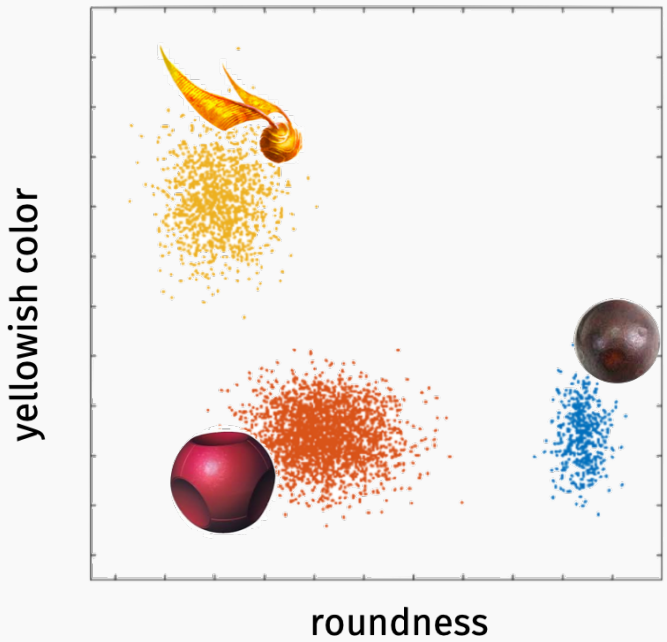


For example, we have learned from past experience that balls used in sports have consistent shapes, colors, and sizes. These features can be used to distinguish balls of different type.

Examples of possible high-level features a human would learn:

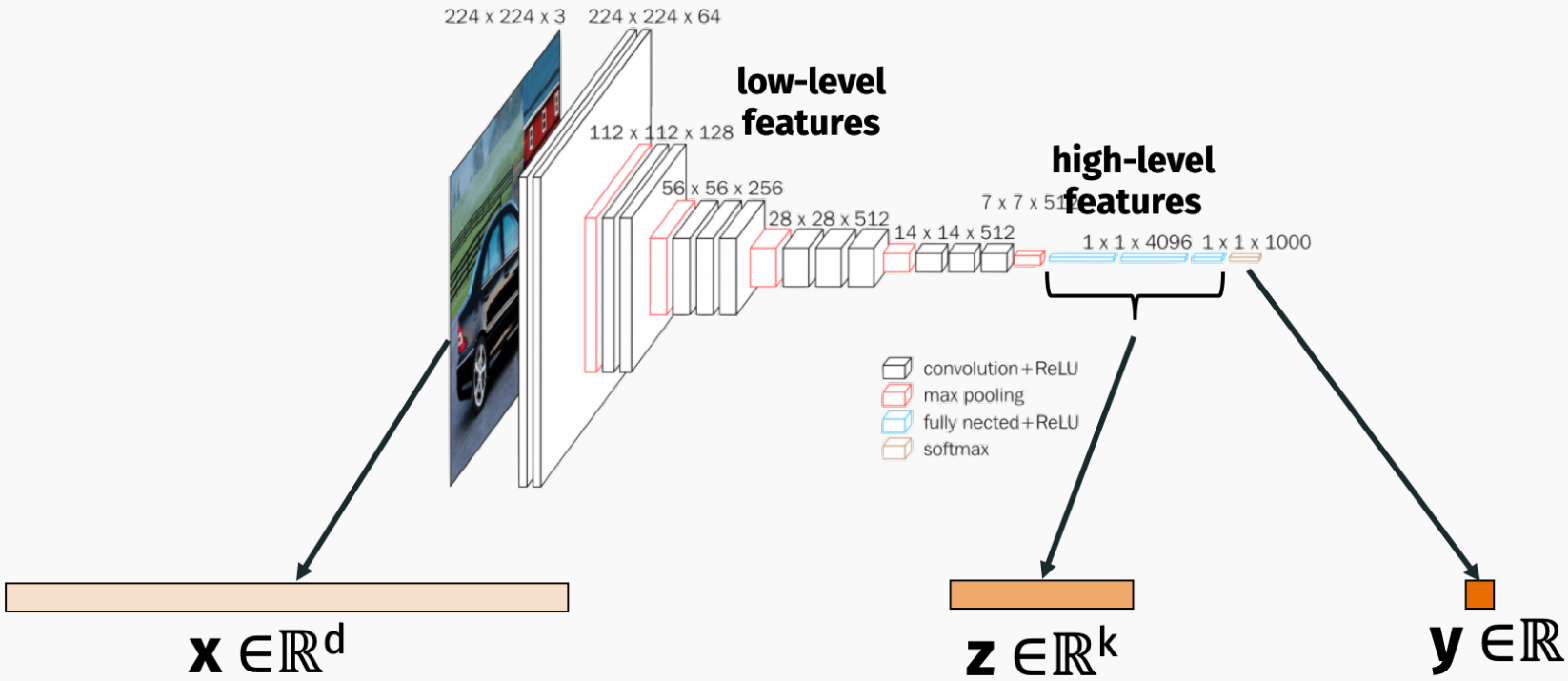
Features	Classes					
						
roundness	1	.1	1	.6	1	.4
size relative to human hand	10	7	2	7	5	1
yellowish color	.2	.1	1	.1	0	.9

If these features are highly informative (i.e. lead to highly separable data) few training examples are needed to learn.



Might suffice to classify ball using nearest training example in feature space, even if just a handful of training examples.

Empirical observation: Features learned when training models like deep neural nets seem to capture exactly these sorts of high-level properties.



Even if we can't put into words what each feature in **z** means...

TRANSFER LEARNING

This is now a common technique in computer vision:

1. Download network trained on large image classification dataset (e.g. Imagenet).
2. Extract features $\mathbf{z}$ for amy new image $\mathbf{x}$ by running it through the network up until layer before last.
3. Use these features in a simpler machine learning algorithm that requires less data (nearest neighbor, logistic regression, etc.).

This approach has even been used on the quidditch problem:
`github.com/thatbrguy/Object-Detection-Quidditch`

Transfer learning: Lots of labeled data for one problem makes up for little labeled data for another.

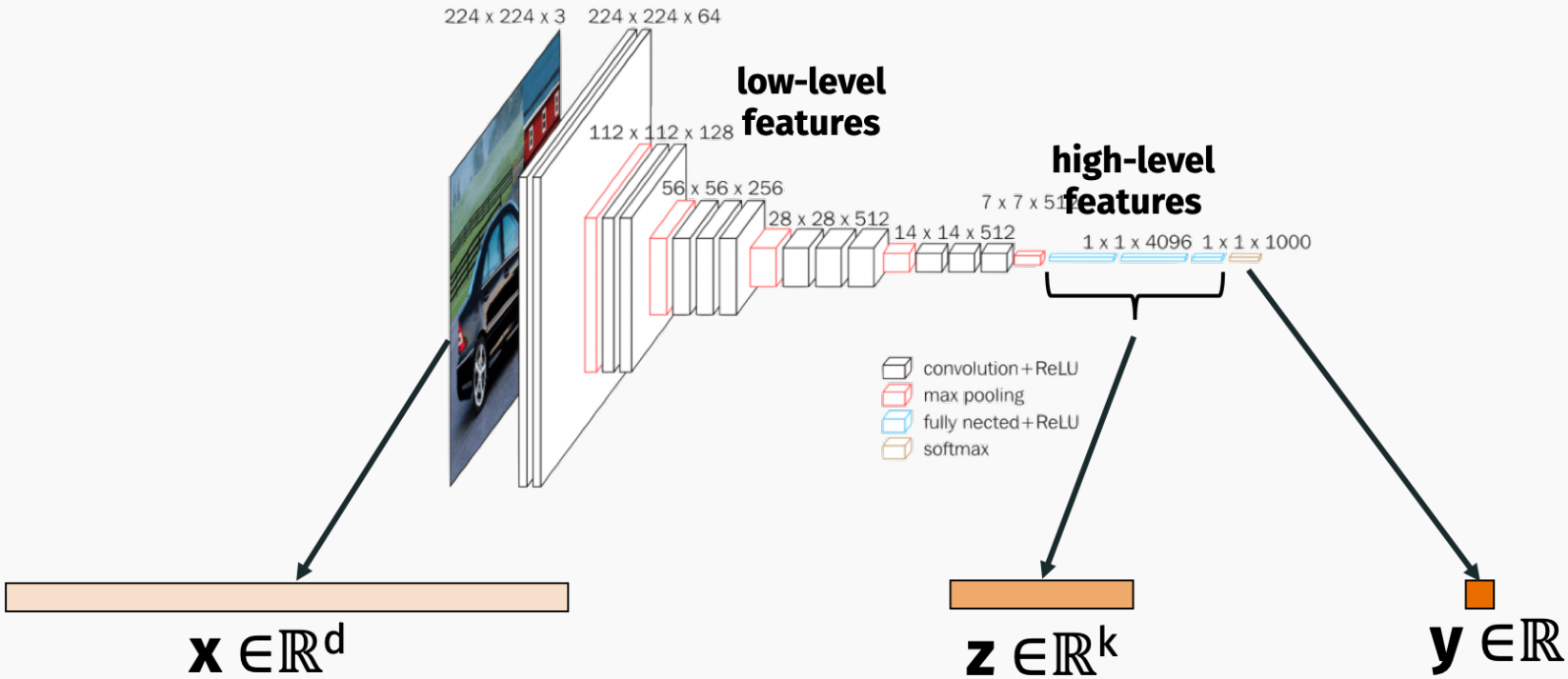
What if we don't even have much labeled data for irrelevant classes?

How to extract features in a data-driven way from unlabeled data is one of the central problems in unsupervised learning.

SUPERVISED VS. UNSUPERVISED LEARNING

- **Supervised learning:** All input data examples come with targets/labels. What machines are good at now.
- **Unsupervised learning:** No input data examples come with targets/labels. Interesting problems to solve include clustering, anomaly detection, semantic embedding, etc.
- **Semi-supervised learning:** Some (typically very few) input data examples come with targets/labels. What human babies are really good at, and we are just starting to make machines better at.

Back to the problem at hand: Want to extract meaningful features from an already trained neural network.



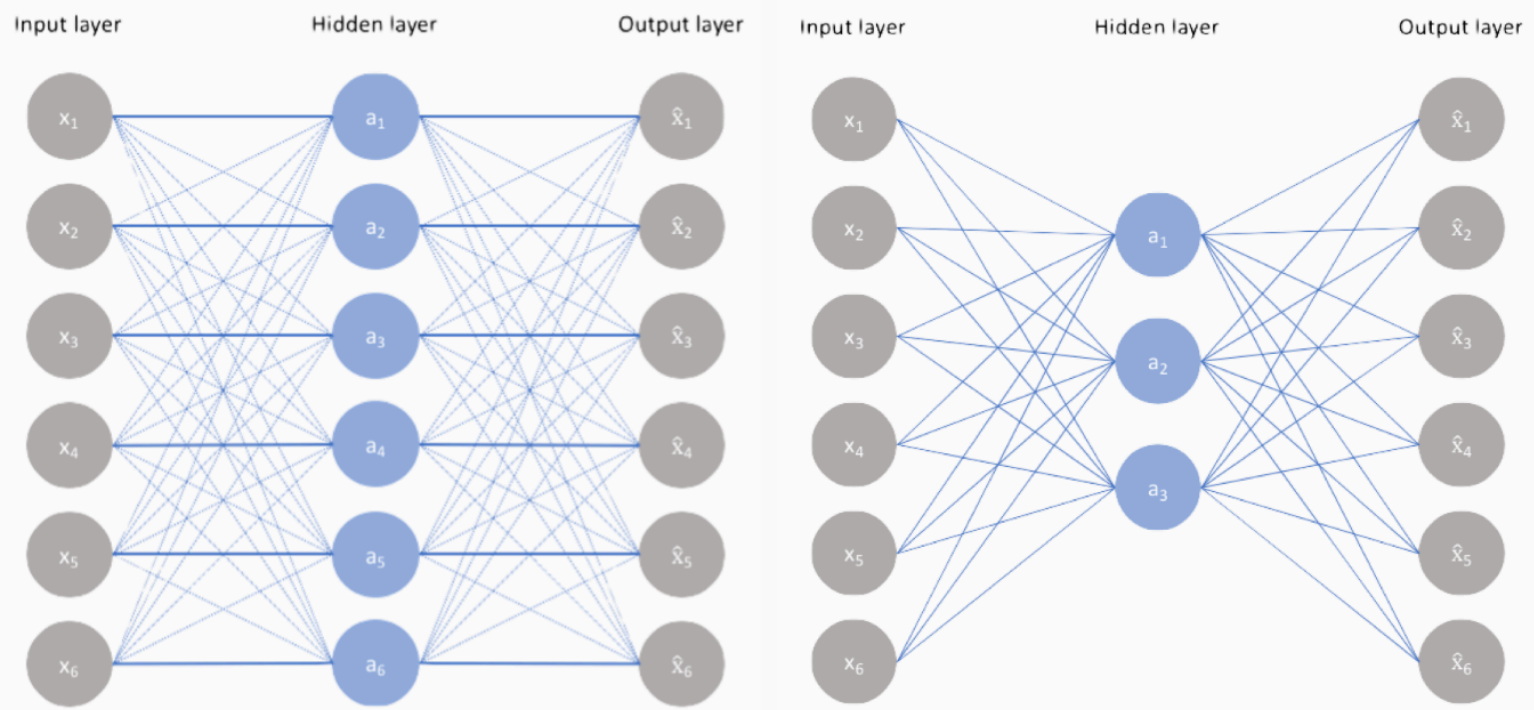
Simple but clever idea: If we have inputs $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ but no targets $y_1, \ldots, y_n$ to learn, just make the inputs the targets.

- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$ be our model.
- Let L be a loss function. E.g. squared loss: $L_{\boldsymbol{\theta}}(\mathbf{x}) = \|\mathbf{x} - f_{\boldsymbol{\theta}}(\mathbf{x})\|_2^2$.
- Train model: $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \sum_{i=1}^n L_{\boldsymbol{\theta}}(\mathbf{x})$.

If $f_{\boldsymbol{\theta}}$ is a model that incorporates feature learning, hopefully these features will capture high-level meaning.

$f_{\boldsymbol{\theta}}$ is called an autoencoder. It maps inputs space to inputs space.

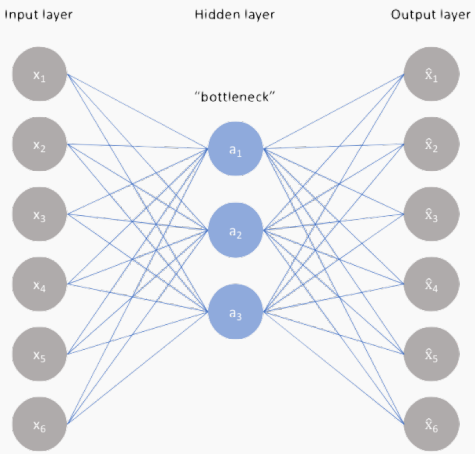
Two examples of autoencoder architectures:

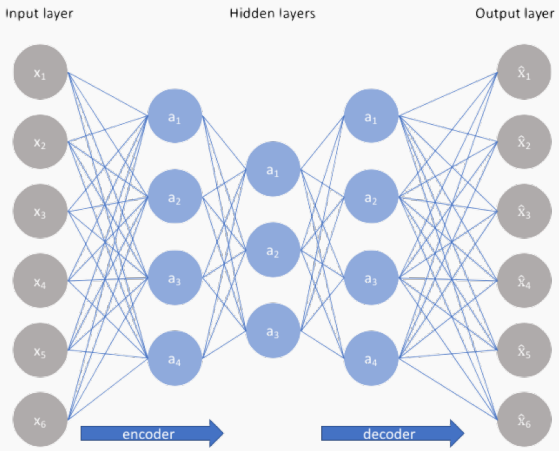


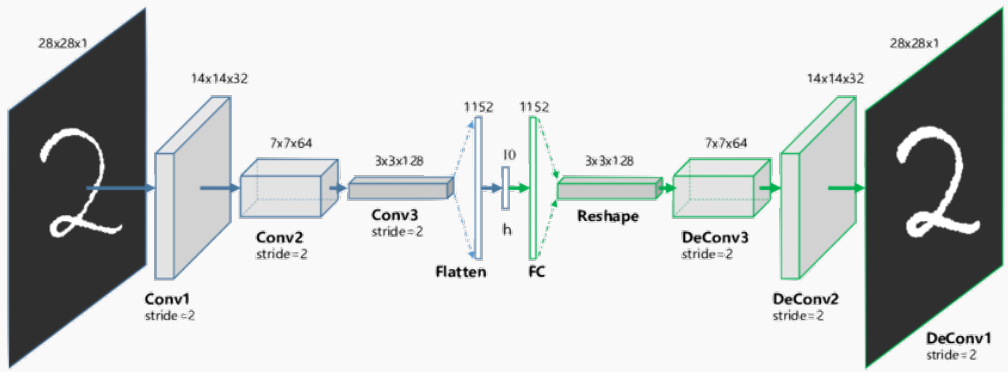
Which would lead to better feature learning?

AUTOENCODER

Important property of autoencoders: no matter what architecture is use, there must always be a **bottleneck** with fewer parameters than the input. The bottleneck ensures information is “distilled” from low-level features to high-level features.







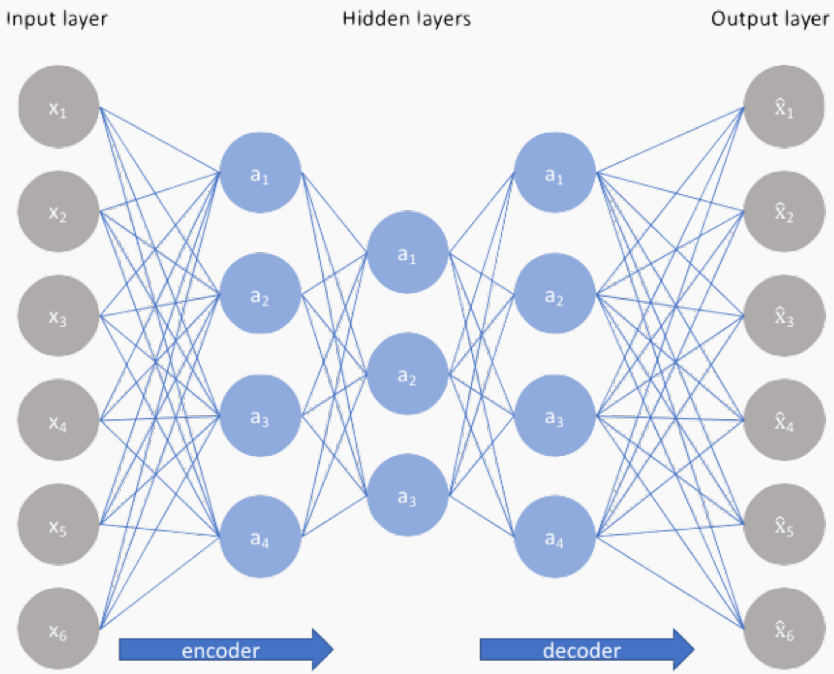
AUTOENCODER

Architecture typically split into two parts:

Encoder: $e : \mathbb{R}^d \to \mathbb{R}^k$

Decoder: $d : \mathbb{R}^d \to \mathbb{R}^k$

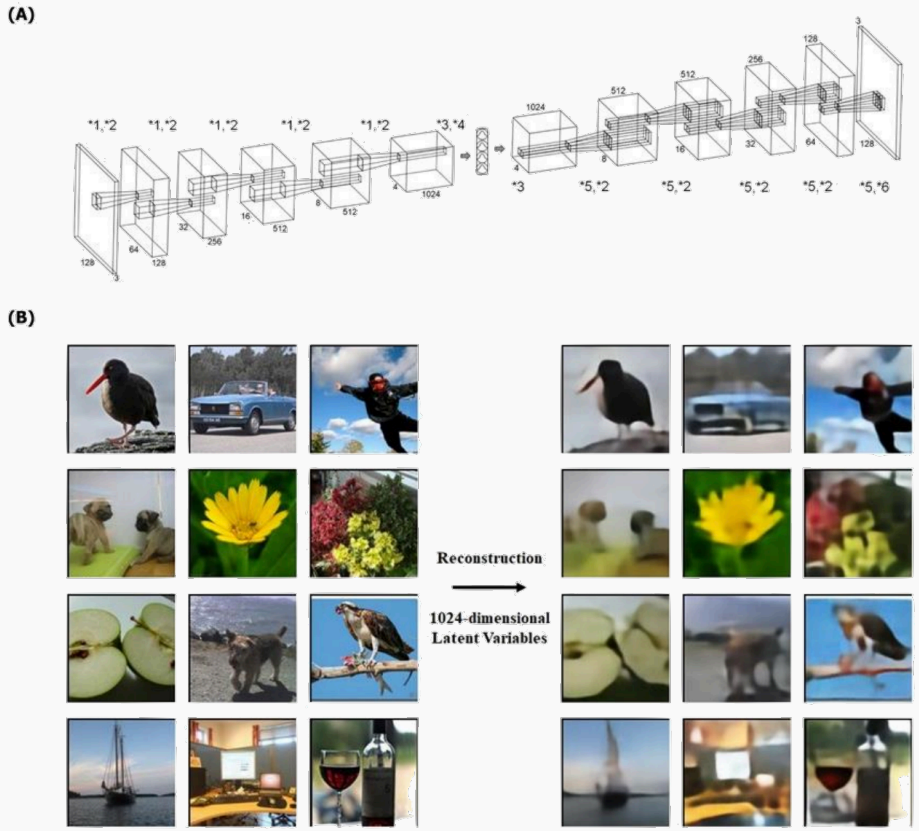
$$f(\mathbf{x}) =$$



Often symmetric, but does not have to be.

AUTOENCODER RECONSTRUCTION

Example image reconstructions from autoencoder:



https://www.biorxiv.org/content/10.1101/214247v1.full.pdf

Input parameters: $d = 49152$.

Bottleneck "latent" parameters: $k = 1024$.

AUTOENCODERS FOR FEATURE EXTRACTION

The best autoencoders do not work as well as for feature extraction as supervised methods. But, they have many other applications.

- Image segmentation.
- Learned image compression.
- Denoising and in-painting.
- Image synthesis.