

CS-GY 6923: Lecture 11

Convolutional Neural Networks, Intro to Autoencoders

NYU Tandon School of Engineering, Prof. Christopher Musco

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.

Fully connected neural networks learn general linear features.
Other architectures are specialized to more application specific features.

For audio/visual applications **convolutional features** are of primary importance.

1D CONVOLUTION

X	1	4	3	-1	2	-4	1	0	2	-1
----------	---	---	---	----	---	----	---	---	---	----

W	1	2	1
----------	---	---	---



X	1	4	3	-1	2	-4	1	0	2	-1
----------	---	---	---	----	---	----	---	---	---	----

W	1	2	1
----------	---	---	---



X	1	4	3	-1	2	-4	1	0	2	-1
----------	---	---	---	----	---	----	---	---	---	----

W	1	2	1
----------	---	---	---



X	1	4	3	-1	2	-4	1	0	2	-1
----------	---	---	---	----	---	----	---	---	---	----

W	1	2	1
----------	---	---	---



w is often called the convolutional “filter” or “kernel”, not to be confused with the other “kernels” we’ve seen.

2D CONVOLUTION

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

3 ₀	3 ₁	2 ₂	1	0
0 ₂	0 ₂	1 ₀	3	1
3 ₀	1 ₁	2 ₂	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3 ₀	2 ₁	1 ₂	0
0 ₂	1 ₂	3 ₀	1	
3 ₁	2 ₀	2	3	
2 ₀	0 ₁	0 ₂	2	
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2 ₀	1 ₁	0 ₂
0 ₀	1 ₂	3 ₂	1 ₀	
3 ₁	2 ₀	2 ₁	3 ₂	
2 ₀	0 ₁	0 ₂	2 ₂	
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	0 ₁	1 ₂	3	1
3 ₂	1 ₂	2 ₀	2	3
2 ₀	0 ₁	0 ₂	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	1 ₁	3 ₂	1	
3 ₁	2 ₂	2 ₀	3	
2 ₀	0 ₁	0 ₂	2	
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	1 ₀	3 ₁	1 ₂	
3 ₁	2 ₂	2 ₂	3 ₀	
2 ₀	0 ₀	2 ₁	2 ₂	
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	1	3	1	
3 ₀	1 ₁	2 ₂	2	3
2 ₂	0 ₂	0 ₀	2	2
2 ₀	0 ₁	0 ₂	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	1	3	1	
3 ₁	2 ₁	2 ₂	3	
2 ₀	0 ₂	0 ₀	2	
2	0 ₀	0 ₁	0 ₂	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 ₀	1	3	1	
3 ₁	2 ₀	2 ₁	3 ₂	
2 ₀	0 ₂	2 ₂	2 ₀	
2	0 ₀	0 ₁	0 ₂	1 ₂

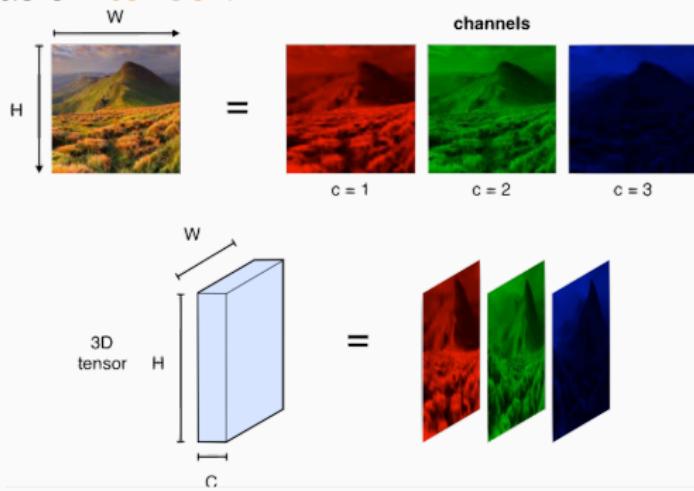
12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3D CONVOLUTION

Recall that color images actually have three color channels for **red, green, blues**. Each pixel is represented by 3 values (e.g. in $0, \dots, 255$) 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**:



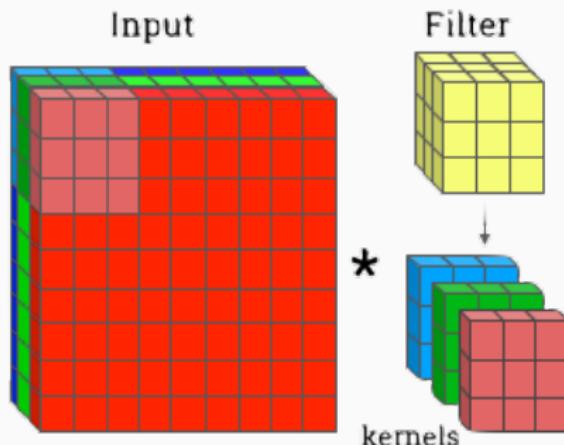
3D CONVOLUTION

Definition (Discrete 2D convolution)

Given tensors $x \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $w \in \mathbb{R}^{k_1 \times k_2 \times k_3}$ the discrete convolution $x \circledast w$ is a

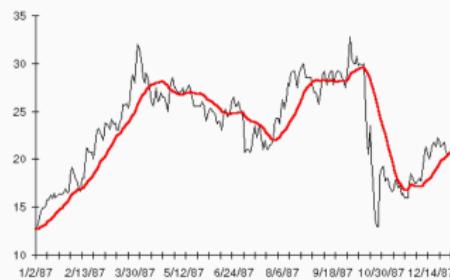
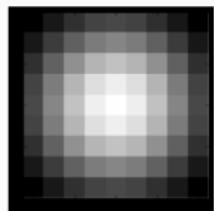
$(d_1 - k_1 + 1) \times (d_2 - k_2 + 1) \times (d_3 - k_3 + 1)$ tensor with:

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



APPLICATION 1: SMOOTHING

A uniform or Gaussian filter can be used to smooth input data:

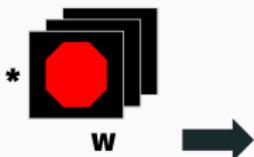


APPLICATION 2: PATTERN MATCHING

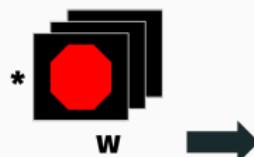
Convolution can be used to find local patterns in images:



x_1



x_2



red channel



blue channel



green channel



$$\text{Yellow} = -1$$

$$\text{Black} = 0$$

$$\text{White} = 1$$

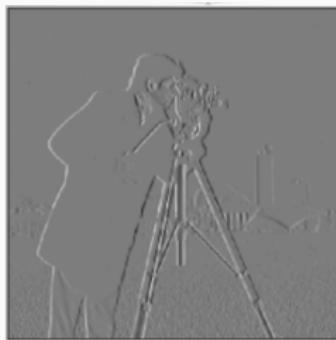
APPLICATIONS OF CONVOLUTION

Application 3: Edge detection.

These are 2D edge detection filter:

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

$$W_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$



$x * ?$



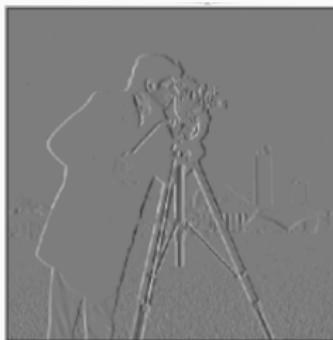
$x * ?$

APPLICATIONS OF CONVOLUTION

Sobel filter is more commonly used:

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

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



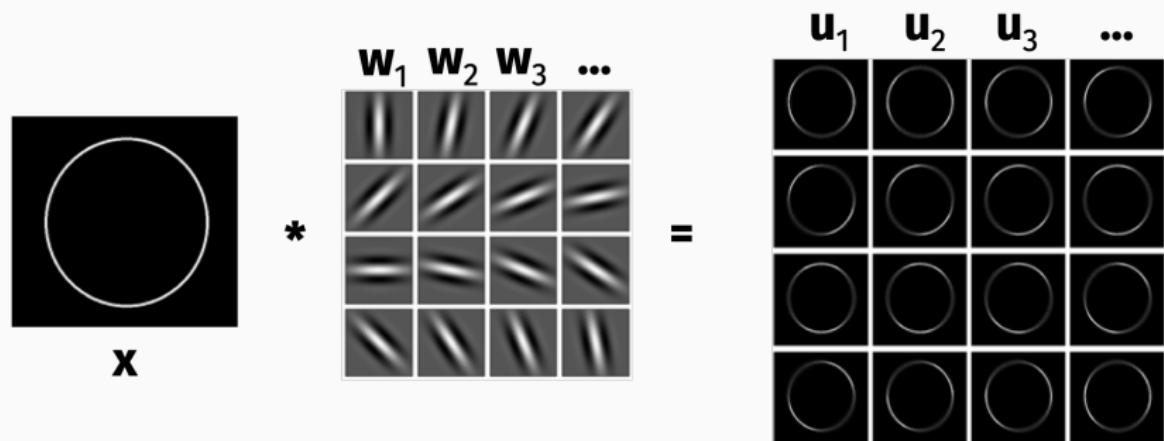
$x * ?$



$x * ?$

DIRECTIONAL EDGE DETECTION

Can define edge detection filters for any orientation.

$$\begin{matrix} \text{X} \\ \text{---} \\ \text{---} \end{matrix} * \begin{matrix} \text{W}_1 & \text{W}_2 & \text{W}_3 & \dots \\ \text{---} \\ \text{---} \end{matrix} = \begin{matrix} \text{U}_1 & \text{U}_2 & \text{U}_3 & \dots \\ \text{---} \\ \text{---} \end{matrix}$$


EDGE DETECTION

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



EDGE DETECTION



I_C

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



E_C



I_L

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



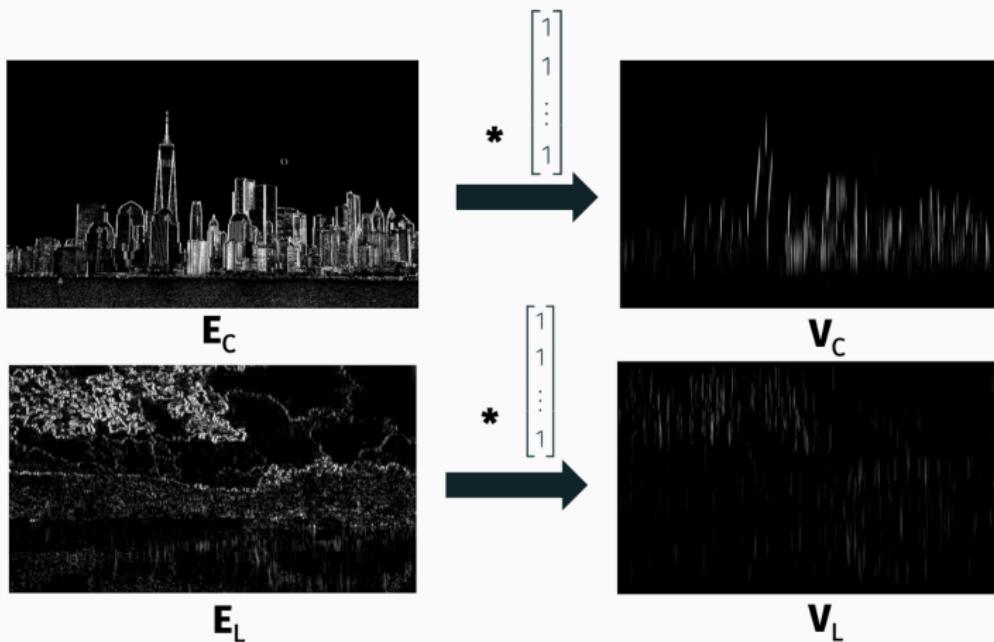
E_L

$$\text{mean}(I_C) = .108 \quad \text{vs.} \quad \text{mean}(I_L) = .123$$

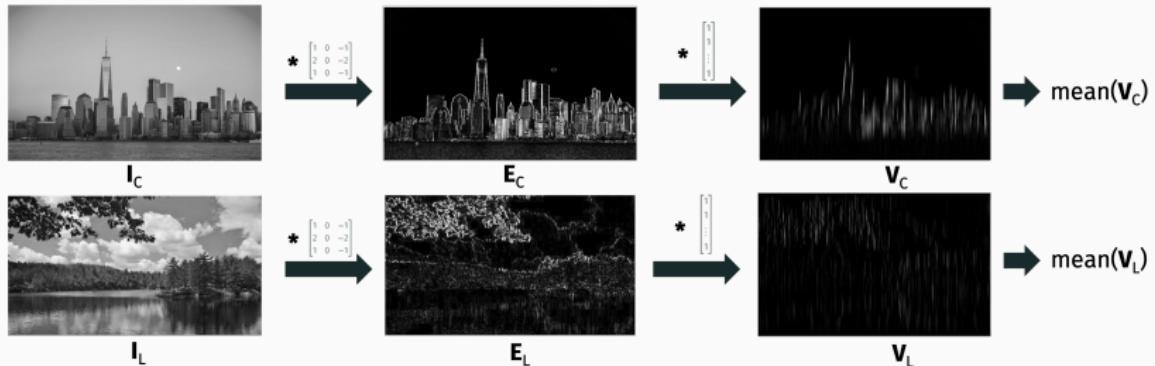
The image with highest vertical edge response isn't the city-scape.

EDGE DETECTION + PATTERN MATCHING

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



HIERARCHICAL CONVOLUTIONAL FEATURES



$$\text{mean}(V_C) = .062 \quad \text{vs.} \quad \text{mean}(V_L) = .054$$

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

$\text{mean}(V) = V^T \beta$ where $\beta = [1/n, \dots, 1/n]$. So the new features in 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**.

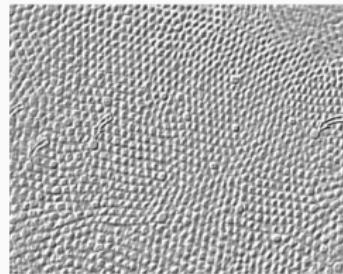
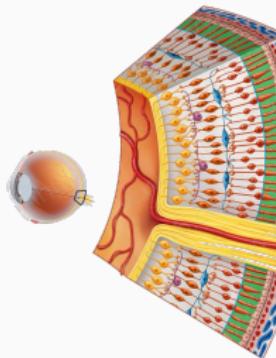
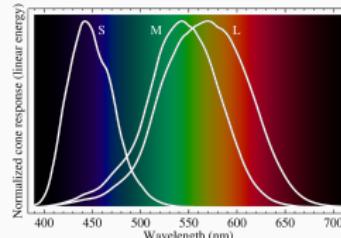


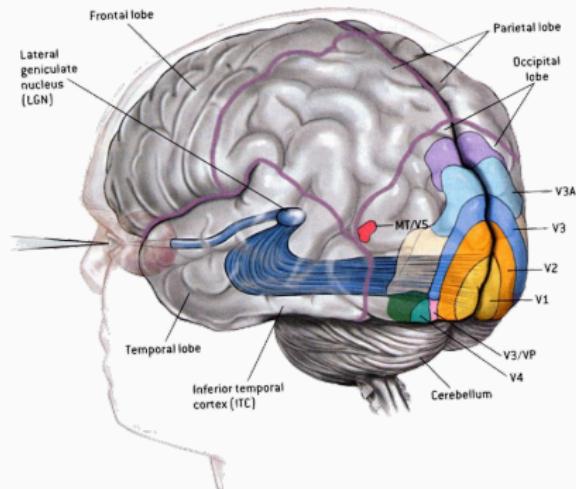
Fig. 13. Tangential section through the human fovea.
Larger cones (arrows) are blue cones. From Ahnelt et al. 1987.

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.

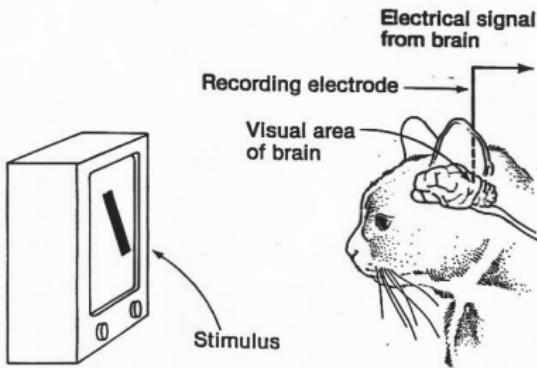


What sort of processing happens in the primary cortex?

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

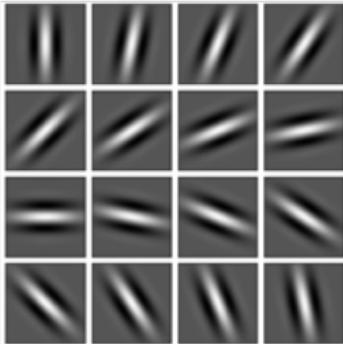
<https://www.youtube.com/watch?v=0GxVfKJqX5E>.

"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 ~ 10 years ago:

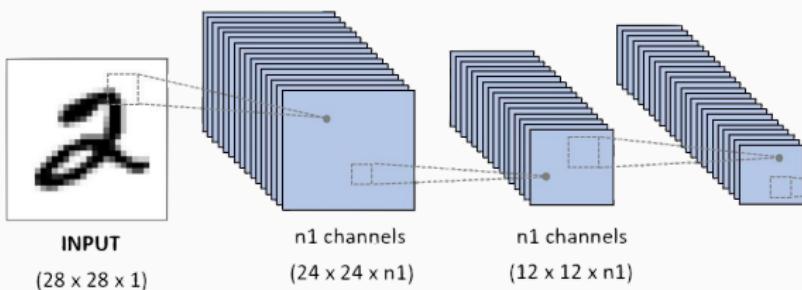
- Convolve image with edge detection filters at many different angles.
- Hand engineer features based on the responses.
- SIFT and HOG features were especially popular.



CONVOLUTIONAL NEURAL NETWORKS

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

Convolutional Layer

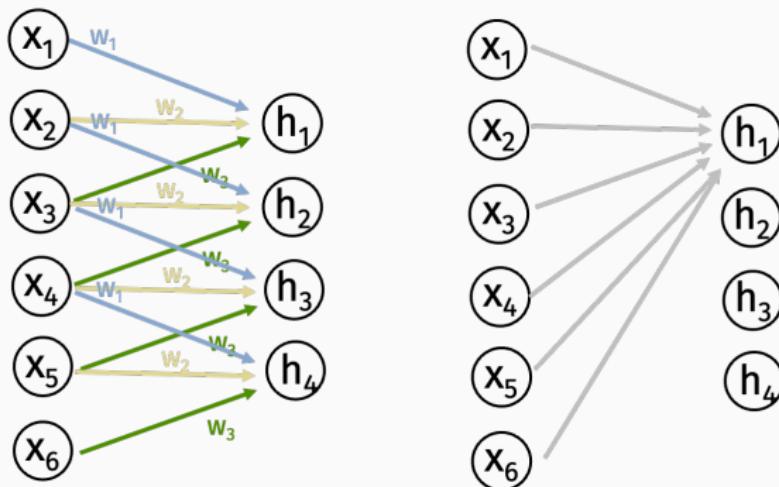


First convolutional layer involves n convolution filters W_1, \dots, W_n . Each is small, e.g. 5×5 . Every entry in W_i is a free parameter: $\sim 25 \cdot n$ parameters to learn.

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.



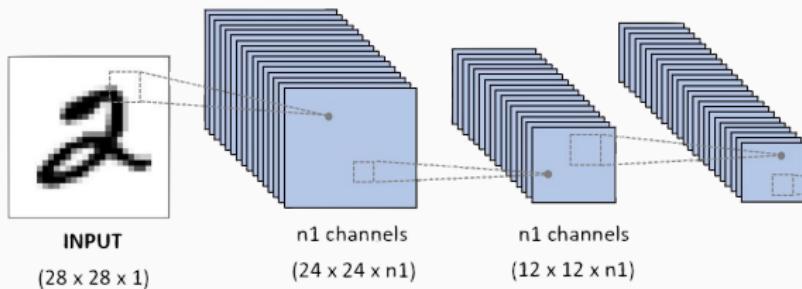
Weight sharing needs to be accounted for when running backprop/gradient descent.

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.

By “baking in” knowledge about what type of features matter, we greatly simplify the network.

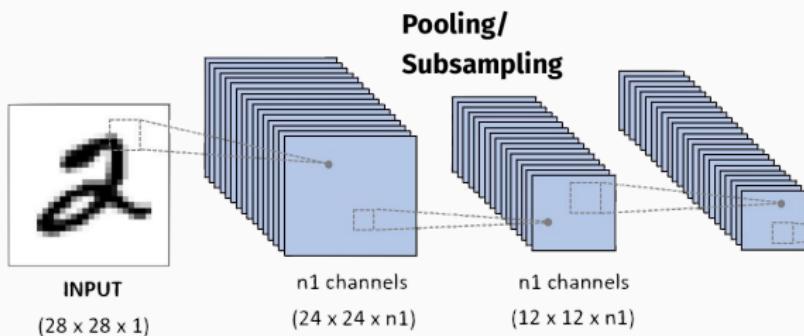
Convolutional Layer



Each of the n outputs 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

Max Pooling			
29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

Average Pooling			
31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

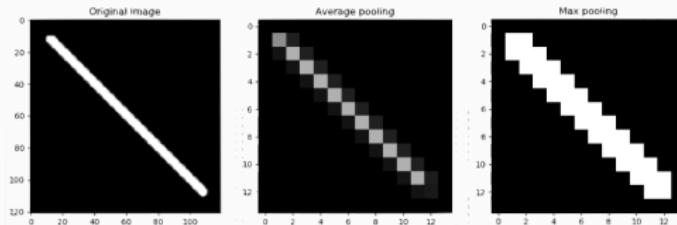
2 x 2
pool size

100	184
12	45

2 x 2
pool size

36	80
12	15

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

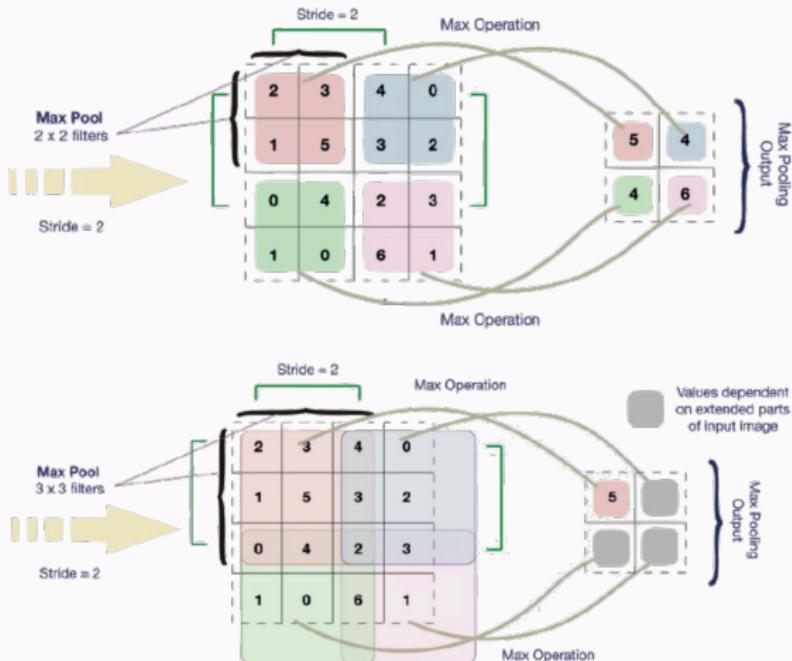


POOLING AND DOWNSAMPLING

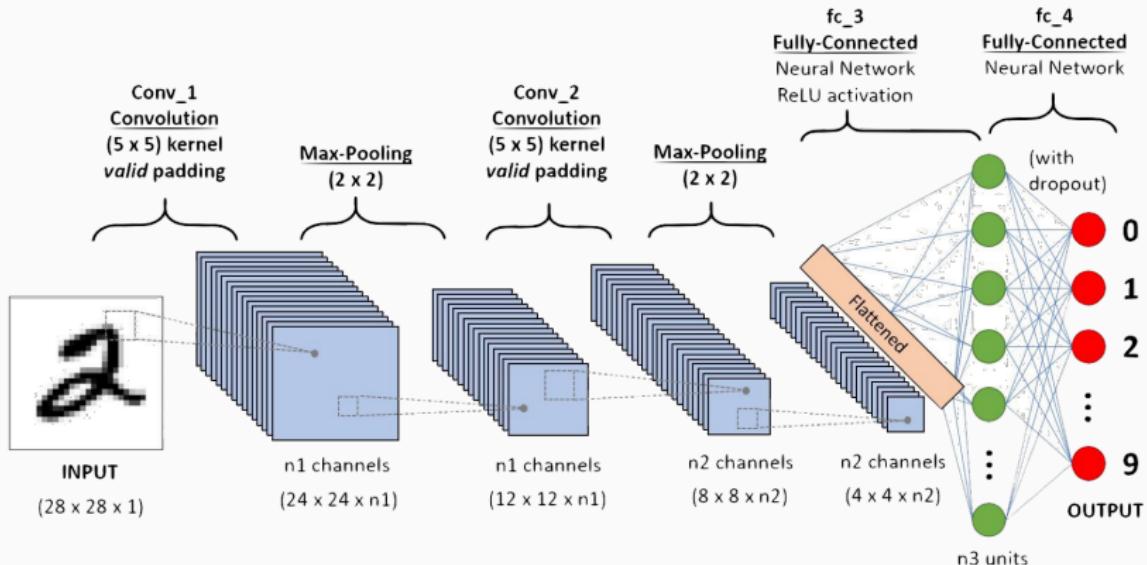
Many possible variations on standard 2x2 max-pooling.

An example Image Portion
for Max Pooling
Numbers represent
the pixel values

2	3	4	0
1	5	3	2
0	4	2	3
1	0	6	1



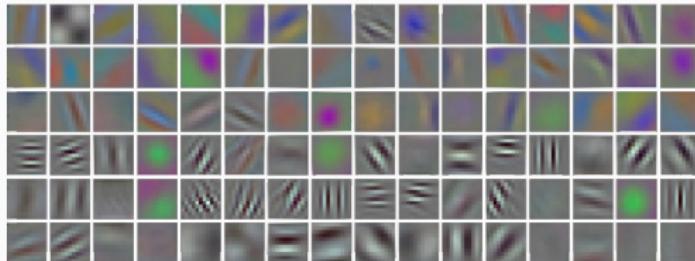
OVERALL NETWORK ARCHITECTURE



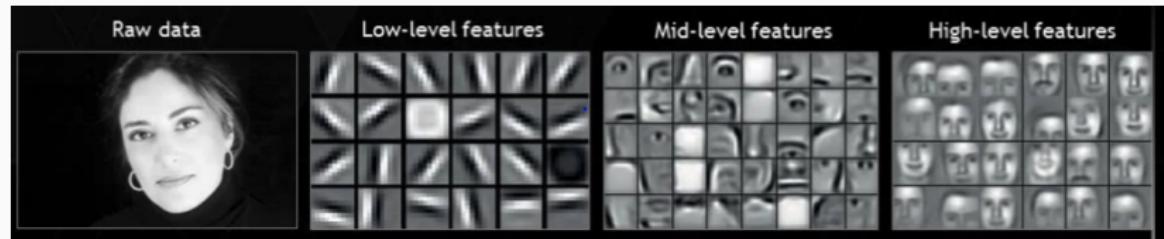
Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

UNDERSTANDING LAYERS

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

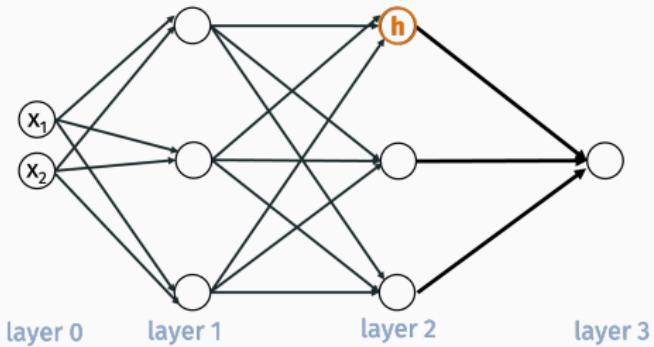


Other layers are harder to understand... but roughly hidden variables later in the network encode for “higher level features”:



UNDERSTANDING LAYERS

How can we know?

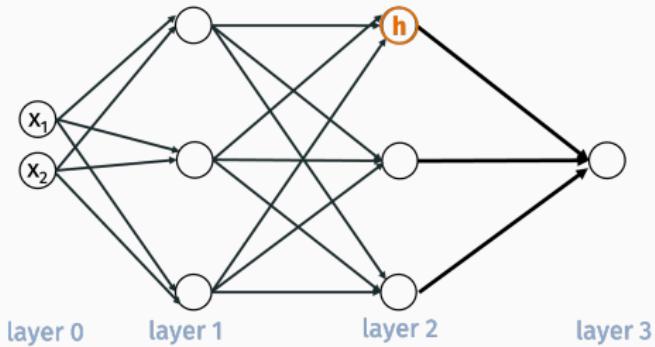


Go through dataset and find the inputs that most “excite” a given neural. I.e. for which $|h(x)|$ is largest.



UNDERSTANDING LAYERS

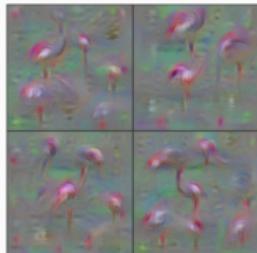
How can we know?



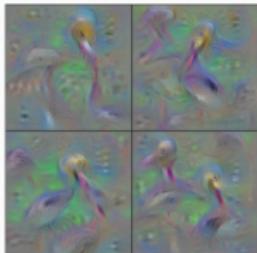
Alternative approach: Solve the optimization problem
 $\max_x |h(x)|$ e.g. using gradient descent.

UNDERSTANDING LAYERS

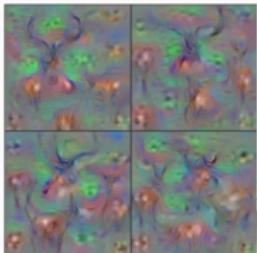
Early work had some interesting results.



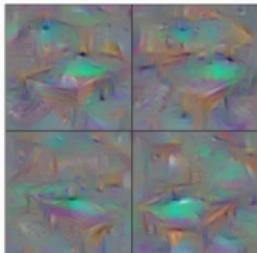
Flamingo



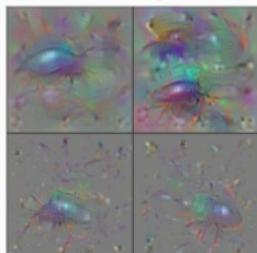
Pelican



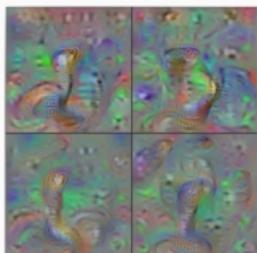
Hartebeest



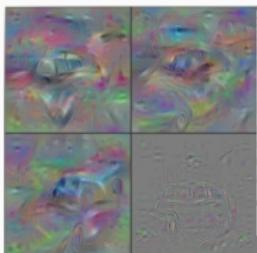
Billiard Table



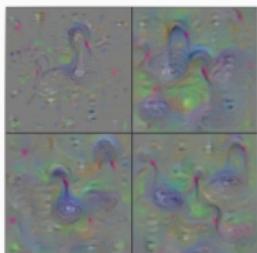
Ground Beetle



Indian Cobra



Station Wagon

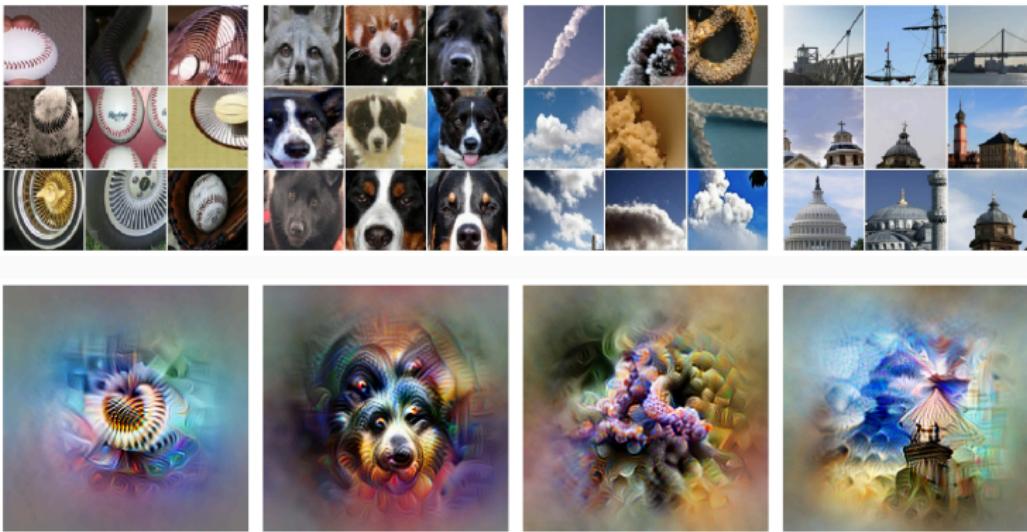


Black Swan

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

UNDERSTANDING LAYERS

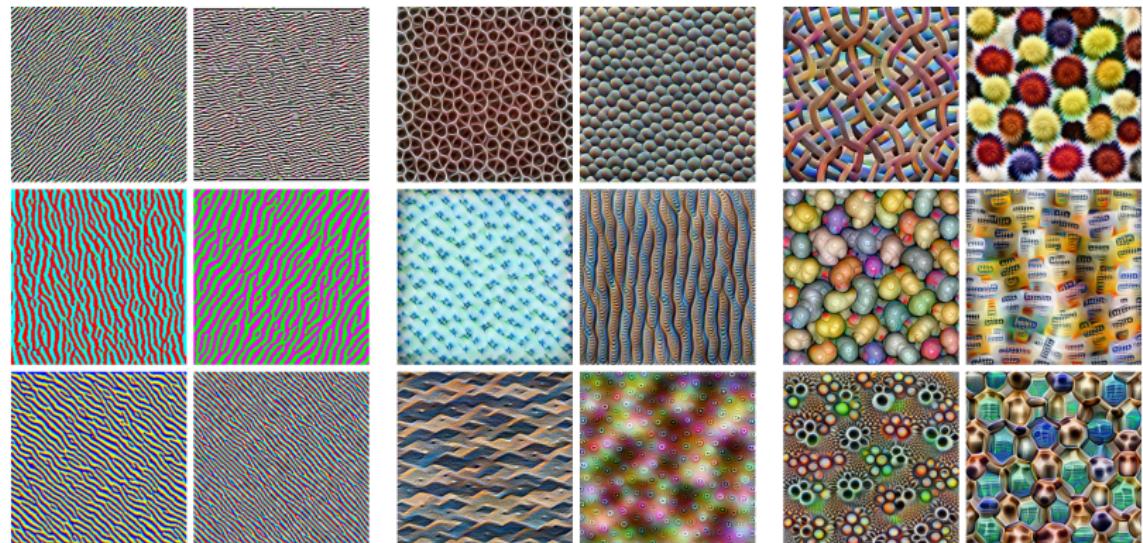
There has been a lot of work on improving these methods by regularization. I.e. solve $\max_x |h(x)| + g(x)$ where g constrains x to look more like a “natural image”.



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.



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

TRICKS OF THE TRADE

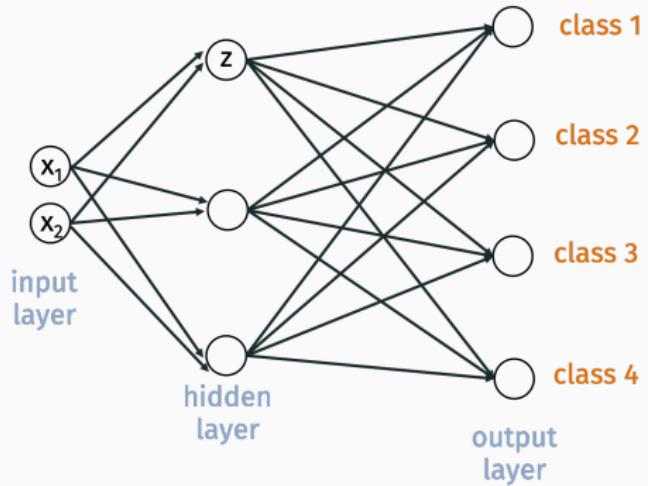
Beyond techniques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of “tricks”.

- 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 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 x . We can write it as $\bar{z}(x)$. Consider the mean and standard deviation of the hidden variable over our entire dataset x_1, \dots, x_n :

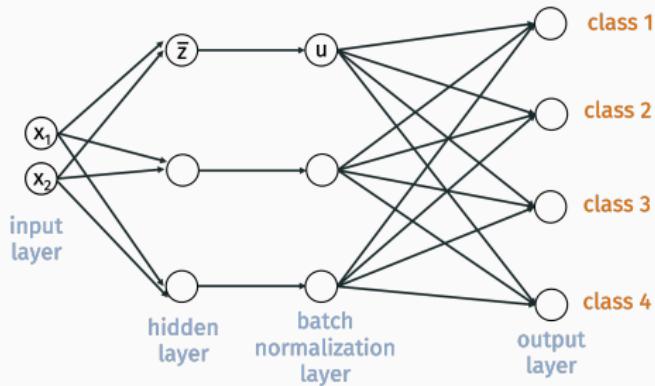
$$\mu = \frac{1}{n} \sum_{j=1}^n \bar{z}(x_j)$$

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

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:



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

$$u = s(\gamma \cdot \bar{u} + c).$$

Where γ and c are **learned parameters**. Has the effect of mean-centering/normalizing \bar{z} , and then mapping back to have a new mean and new standard deviation.

BATCH NORMALIZATION

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

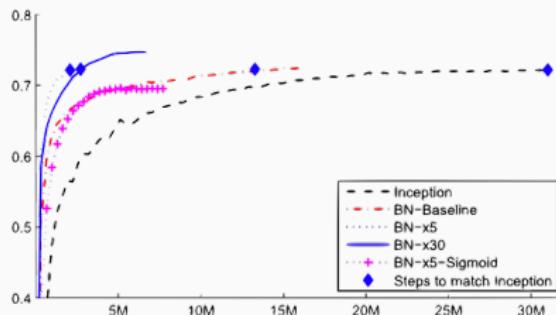
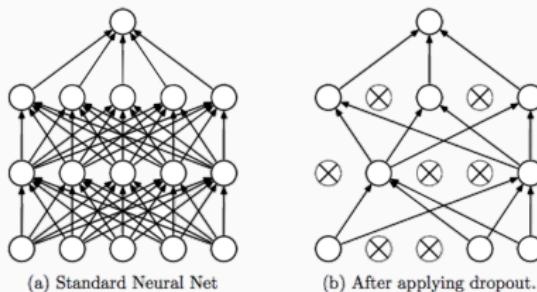


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

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

DROPOUT

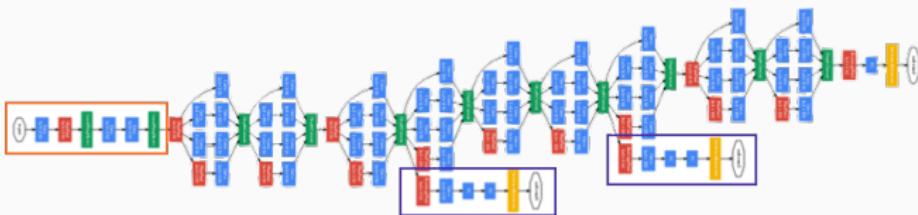
Proposed in 2012: “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, Srivastava et al.



During training, ignore a random subset of neurons during each gradient step. Select each neuron to be included independently with probability p (typically $p \approx .5$). During testing, no dropout is used.

DROPOUT

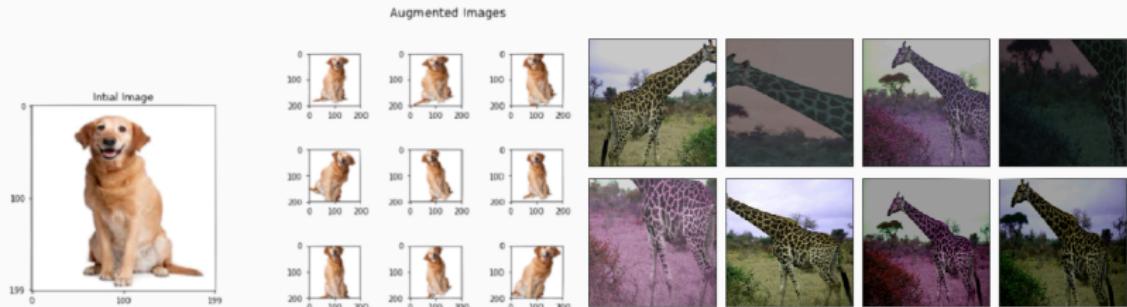
- Only used on fully connected layers.
- Simultaneously performs model regularization (model simplification) and model averaging.
- Has become less important in modern CNNs (convolutional neural nets) as the final fully connected layers become less important. But still a very helpful technique to know about!



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 constant evolving.

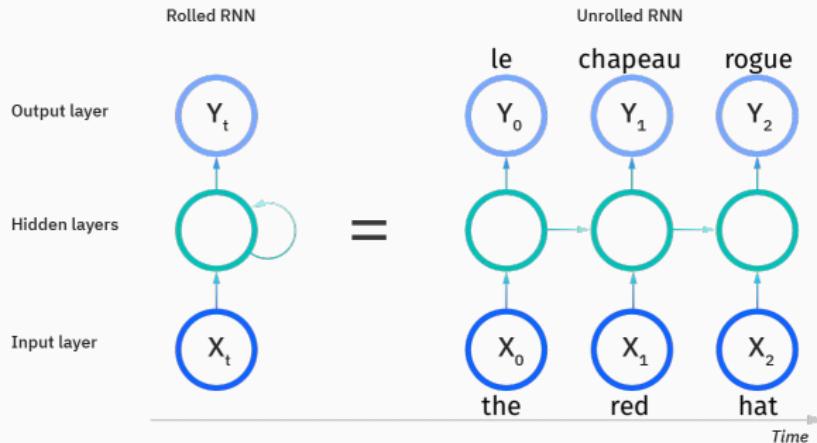
BEYOND CONVOLUTIONAL NETWORKS

Convolution networks make sense for computer vision applications, but not for many other applications. E.g. they aren't useful for machine translation, document summarization, etc.

Typically design alternative neural networks for these applications. Like convolutional networks, these can usually be viewed as appropriately constrained versions of fully-connected neural networks.

RECURRENT NEURAL NETWORKS

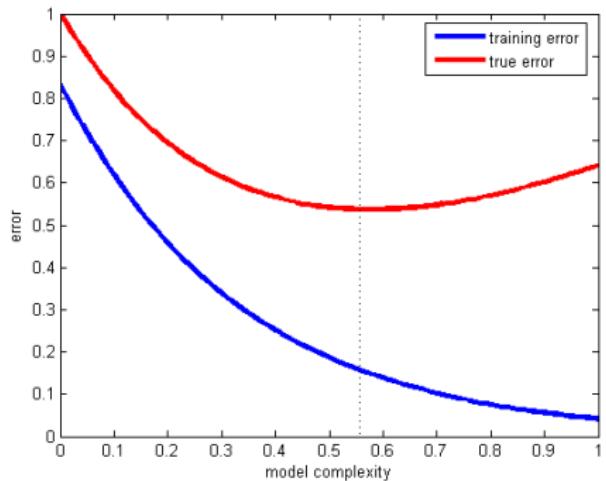
Example: Recurrent neural networks for sequential data (e.g. speech, written sentences).



Weight matrices are shared between time steps. This is a very oversimplified view of recurrent networks.

GENERALIZATION FOR NEURAL NETWORKS

Even with weight sharing, convolution, etc. modern neural networks often have millions of parameters. And we don't run them with explicit 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 RE-THINKING 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

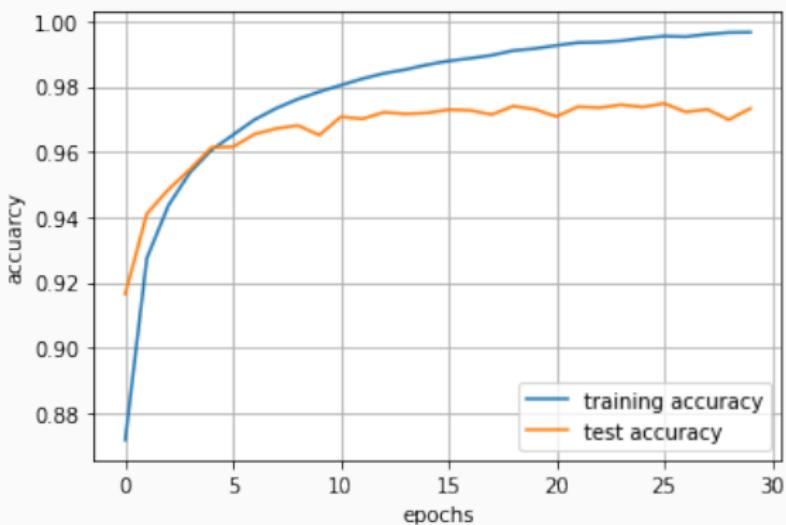
Oriol Vinyals

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.

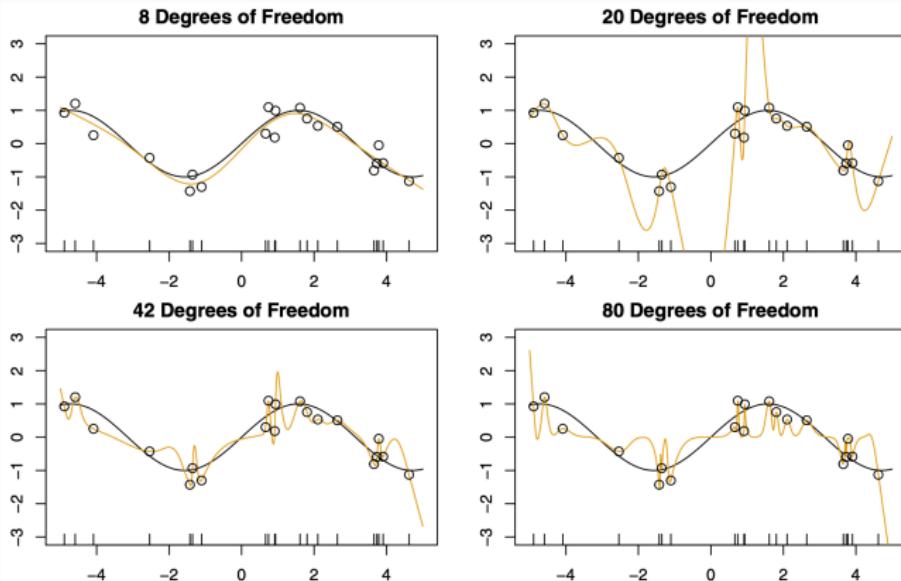
GENERALIZATION FOR NEURAL NETWORKS

We even see this lack of overfitting in the simple MNIST demo `keras_demo_mnist.ipynb` I posted on the website:



GENERALIZATION FOR NEURAL NETWORKS

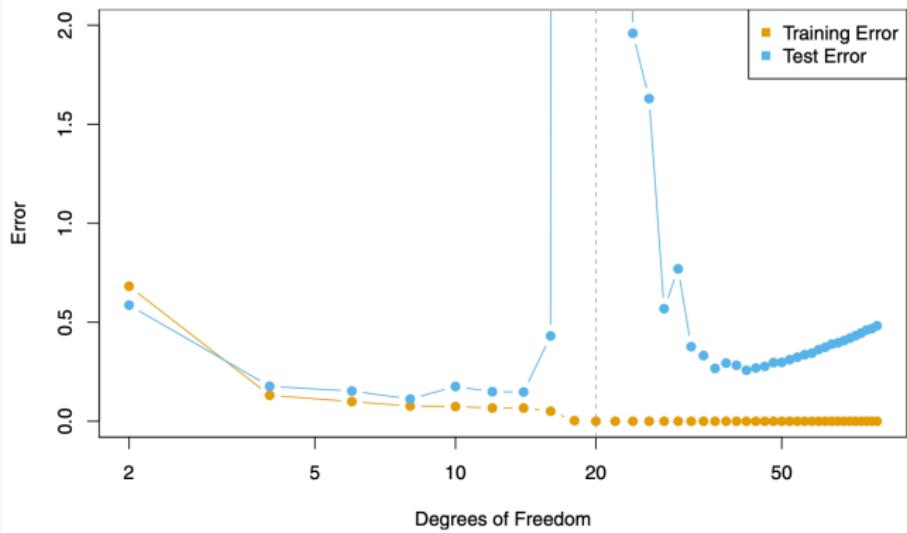
One growing realization is that this phenomena doesn't only apply to neural networks – it can also be true for fitting highly-overparameterized polynomials.



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 get better with lots of overparameterization.



We don't always see this curve for neural networks.

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

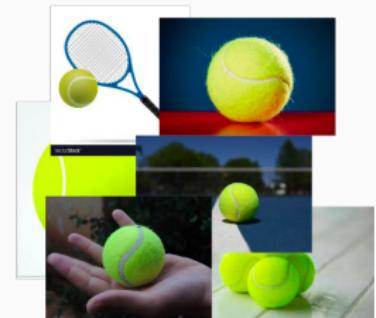
A human could probably achieve near perfect classification accuracy even given access to a **single labeled example** from each class:



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

TRANSFER LEARNING

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.

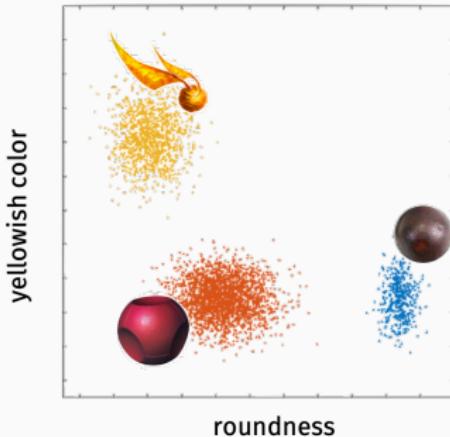
FEATURE LEARNING

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

		Classes					
							
Features	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

FEATURE LEARNING

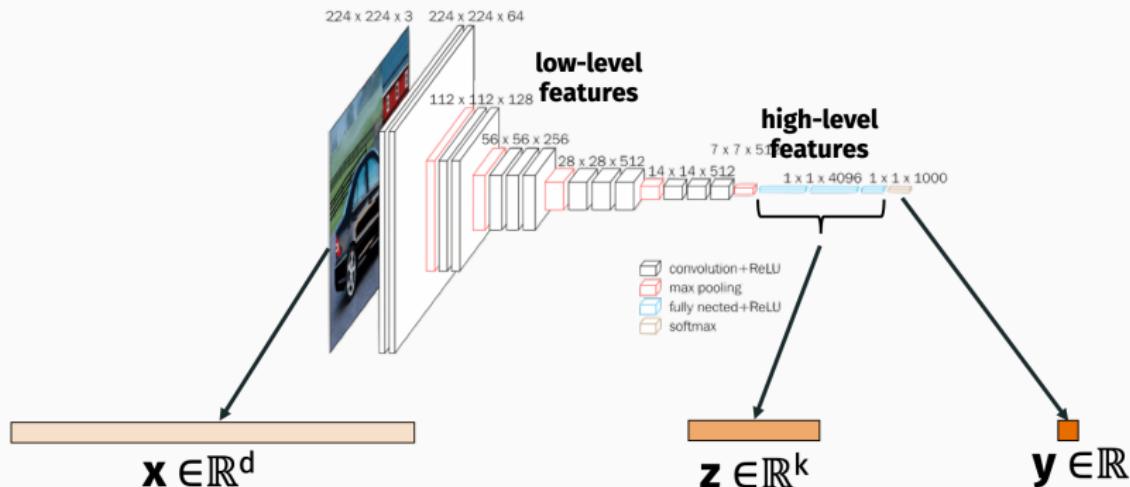
If these features are highly informative (i.e. lead to highly separable data) few training examples are needed to learn.



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

TRANSFER LEARNING

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 z for any new image 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

UNSUPERVISED FEATURE LEARNING

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?

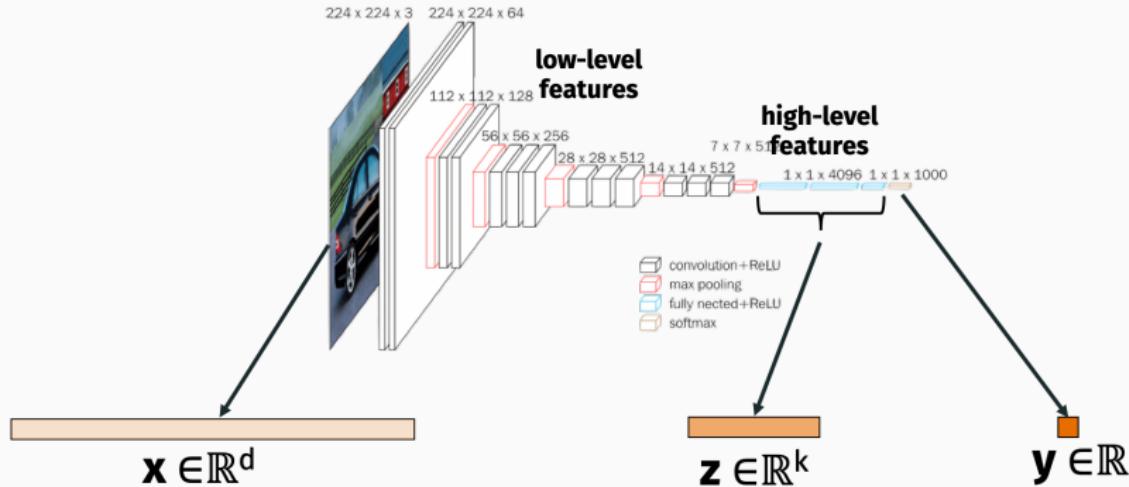
How to extract features in a data-driven way from unlabeled data is one of the central problems in **unsupervised learning**.

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.

TRANSFER LEARNING

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



AUTOENCODER

Simple but clever idea: If we have inputs $x_1, \dots, x_n \in \mathbb{R}^d$ but no targets y_1, \dots, y_n to learn, just make the inputs the targets.

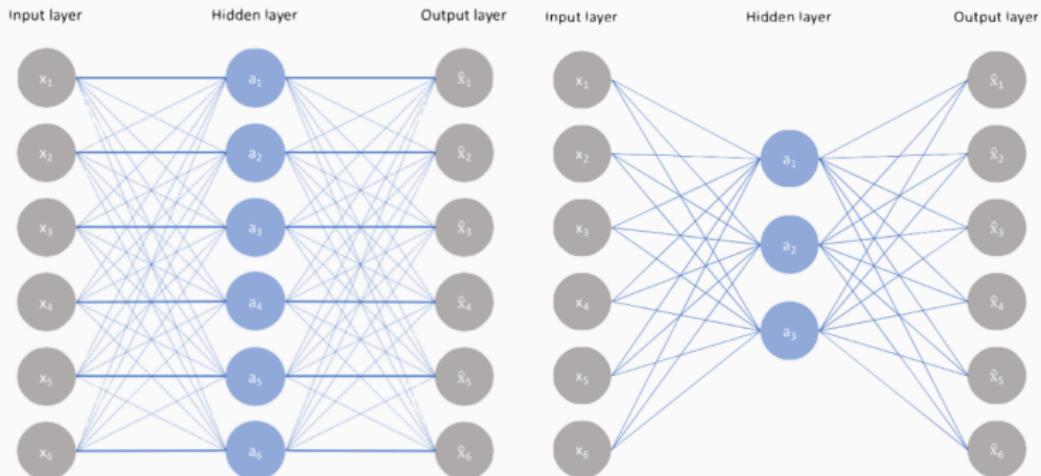
- Let $f_{\theta} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ be our model.
- Let L be a loss function. E.g. squared loss:
$$L_{\theta}(x) = \|x - f_{\theta}(x)\|_2^2.$$
- Train model: $\theta^* = \min_{\theta} \sum_{i=1}^n L_{\theta}(x_i)$.

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

f_{θ} is called an **autoencoder**. It maps inputs space to inputs space.

AUTOENCODER

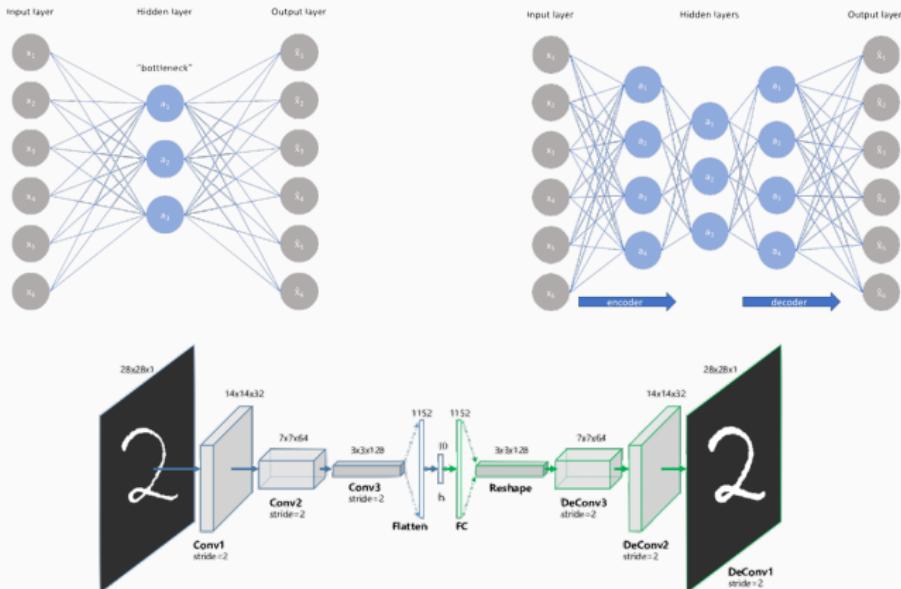
Two examples of autoencoder architectures:



Which would lead to better feature learning?

AUTOENCODER

Important property of autoencoders: no matter what architecture is used, 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.



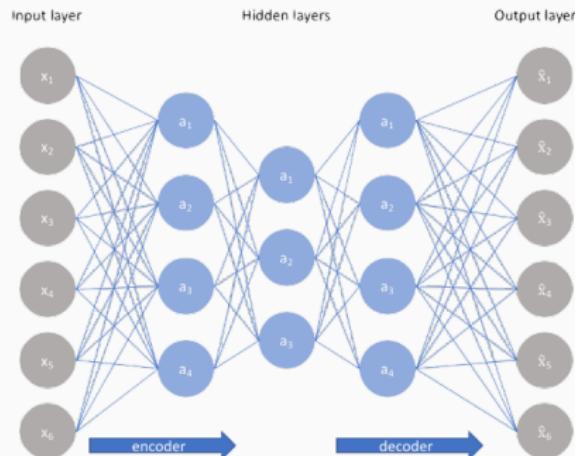
AUTOENCODER

Architecture typically split into two parts:

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

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

$$f(x) =$$

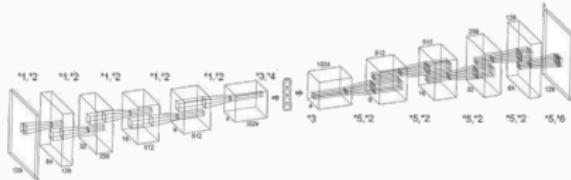


Often symmetric, but does not have to be.

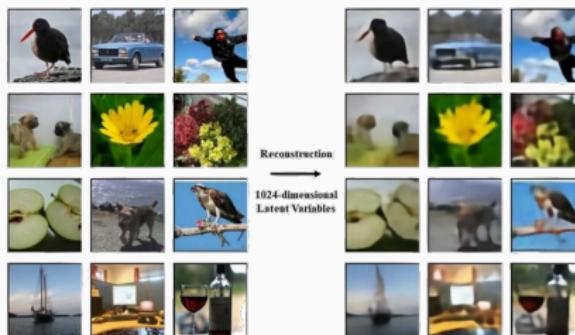
AUTOENCODER RECONSTRUCTION

Example image reconstructions from autoencoder:

(A)



(B)



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

Input parameters: $d = 49152$.

Bottleneck “latent” parameters: $k = 1024$.

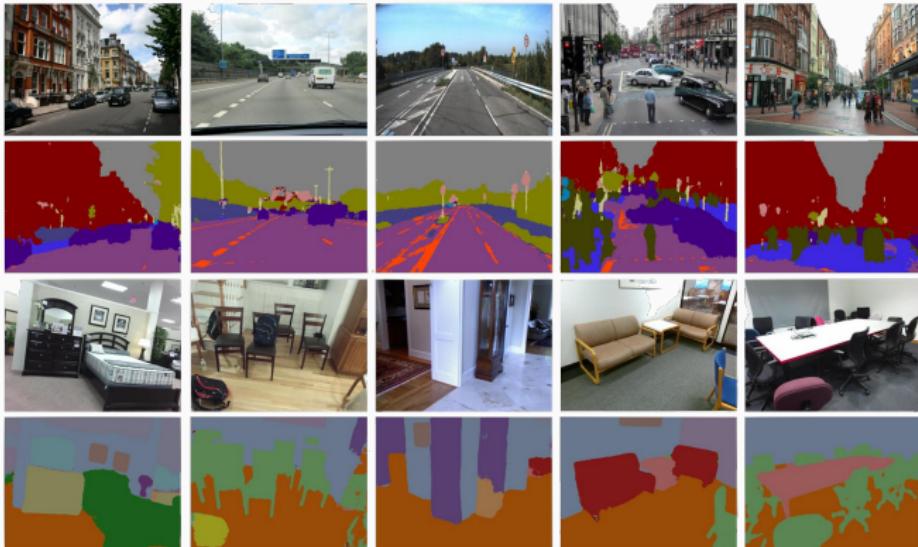
AUTOENCODERS FOR FEATURE EXTRACTION

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.

AUTOENCODERS FOR IMAGE SEGMENTATION

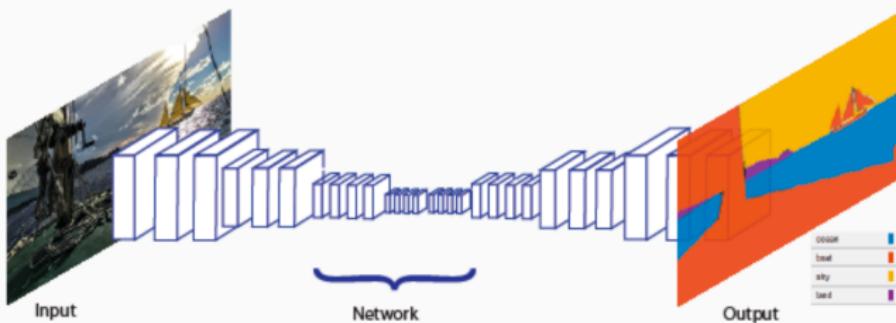
Goal: Learn mask which separates image pixels by what object (foreground or background) that they belong to.



First step in multi-objects classification and scene understanding. Harder than classifying single objects.

AUTOENCODERS FOR IMAGE SEGMENTATION

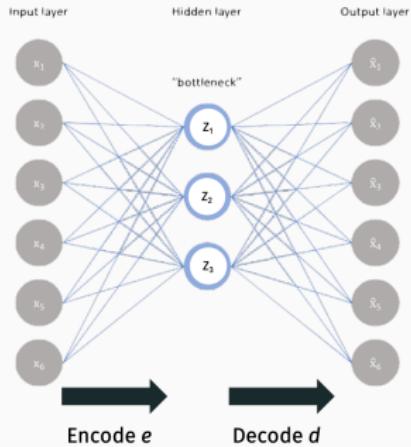
Change in design: Input is image x , output is image m that has the same size as x , but each pixel value is a label for a segmented region.



Now our training process is actually supervised, but uses the same structure as an autoencoder.

AUTOENCODERS FOR IMAGE COMPRESSION

Due to their bottleneck design, autoencoders perform dimensionality reduction and thus data compression.



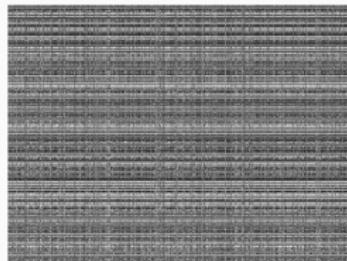
Given input x , we can completely recover $f(x)$ from $z = e(x)$. z typically has many fewer dimensions than x and for a typical image $f(x)$ will closely approximate x .

AUTOENCODERS FOR IMAGE COMPRESSION

The best lossy compression algorithms are tailor made for specific types of data:

- JPEG 2000 for images
- MP3 for digital audio.
- MPEG-4 for video.

All of these algorithms take advantage of specific structure in these data sets. E.g. JPEG assumes images are locally “smooth”.



AUTOENCODERS FOR IMAGE COMPRESSION

With enough input data, autoencoders can be trained to find this structure on their own.



Proposed method, 5908 bytes (0.167 bit/px), PSNR: luma 23.38 dB/chroma 31.86 dB, MS-SSIM: 0.9219



JPEG 2000, 5908 bytes (0.167 bit/px), PSNR: luma 23.24 dB/chroma 31.04 dB, MS-SSIM: 0.8803



Proposed method, 6021 bytes (0.170 bit/px), PSNR: 24.12 dB, MS-SSIM: 0.9292



JPEG 2000, 6037 bytes (0.171 bit/px), PSNR: 23.47 dB, MS-SSIM: 0.9036

"End-to-end optimized image compression", Ballé, Laparra, Simoncelli

Need to be careful about how you choose loss function, design the network, etc. but can lead to much better image compression than "hand-tuned" algorithms like JPEG.

AUTOENCODERS FOR IMAGE RESTORATION

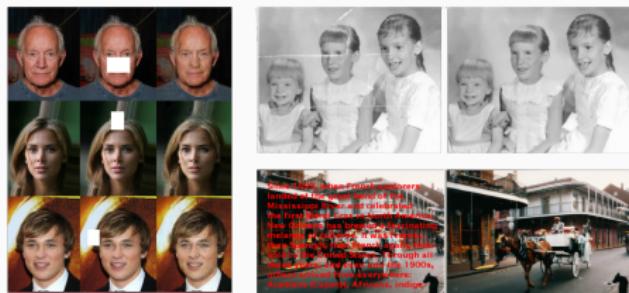
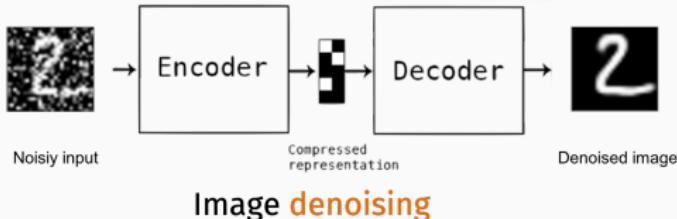
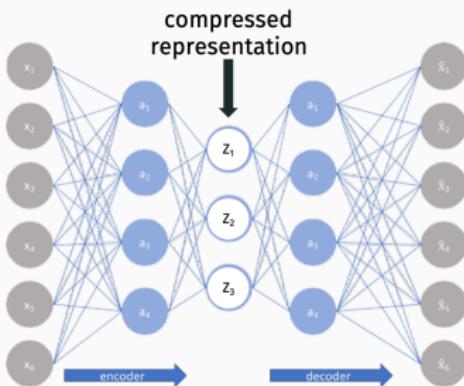


Image inpainting

Train autoencoder on uncorrupted images (unsupervised). Pass corrupted image x through autoencoder and return $f(x)$ as repaired result.

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

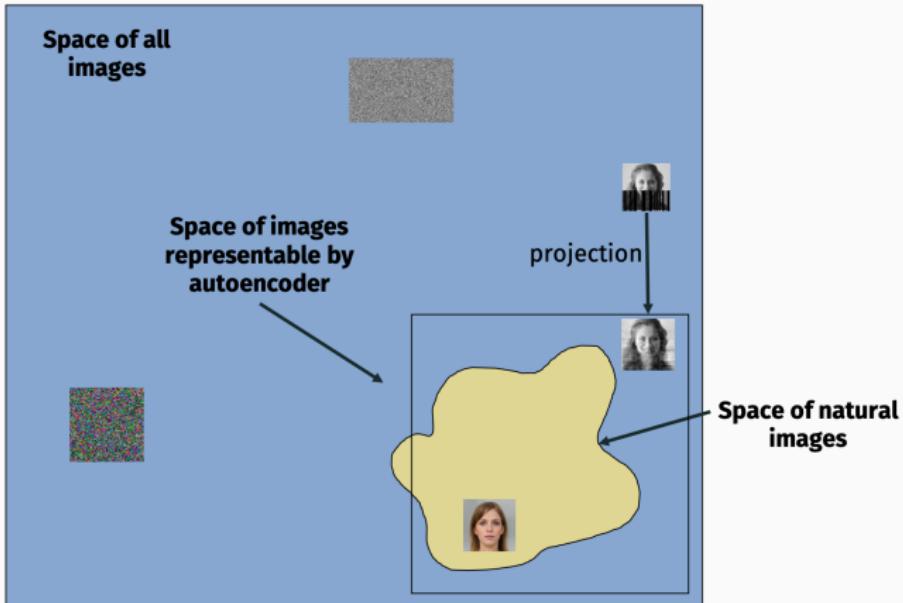
Why does this work?



Consider $128 \times 128 \times 3$ images with pixels values in $0, 1, \dots, 255$.
How many possible images are there?

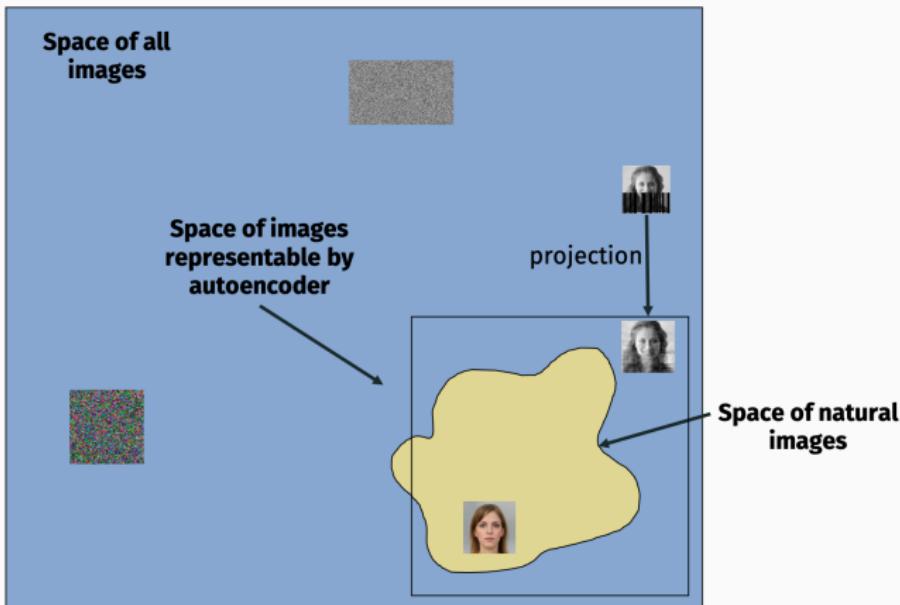
If z holds k values between $0, .1, .2, \dots, 1$, how many unique images w can be output by the autoencoder function f ?

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



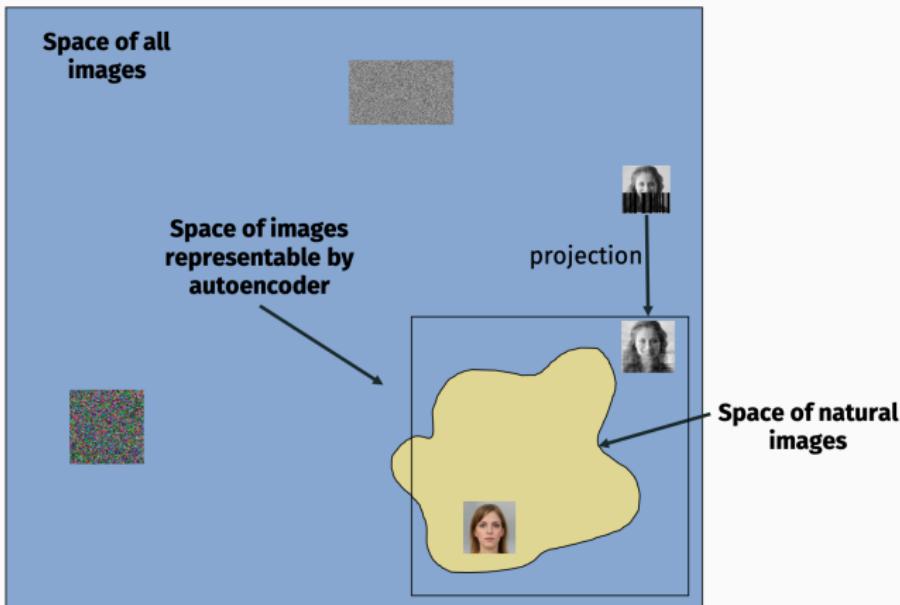
For a good (accurate, small bottlenecked) autoencoder, \mathcal{S} will closely approximate \mathcal{I} . Both will be much smaller than \mathcal{A} .

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



$f(x)$ projects an image x closer to the space of natural images.

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

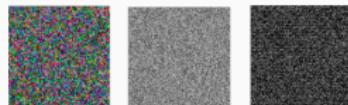


$f(x)$ projects an image x closer to the space of natural images.

AUTOENCODERS FOR DATA GENERATION

Suppose we want to generate a random natural image. How might we do that?

- **Option 1:** Draw each pixel in \mathbf{x} value uniformly at random.
Draws a random image from \mathcal{A} .



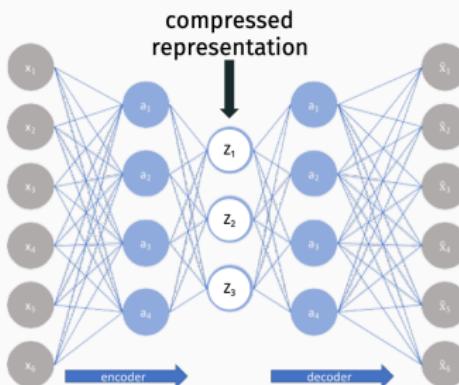
- **Option 2:** Draws \mathbf{x} randomly image from \mathcal{S} , the space of images representable by the autoencoder.



How do we randomly select an image from \mathcal{S} ?

AUTOENCODERS FOR DATA GENERATION

How do we randomly select an image x from \mathcal{S} ?



Randomly select code z , then set $x = e(z)$.¹

¹Lots of details to think about here. In reality, people use “variational autoencoders” (VAEs), which are a natural modification of AEs.

AUTOENCODERS FOR DATA GENERATION



Generative models are a growing area of machine learning, driven by a lot of interesting new ideas. Generative Adversarial Networks in particular are now a major competitor with variational autoencoders.

PRINCIPAL COMPONENT ANALYSIS

Next lecture: Deeper dive into understanding a simple, but powerful autoencoder architecture. Specifically we will learn about **principal component analysis (PCA)** as a type of autoencoder.

PCA is the “linear regression” of unsupervised learning: often the go-to baseline method for feature extraction and dimensionality reduction.

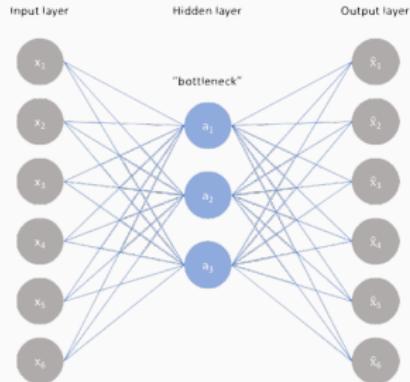
Very important outside machine learning as well.

Like any autoencoder, PCA can be used for:

- Feature extraction
- Denoising and rectification
- Data generation
- Compression
- Visualization

PRINCIPAL COMPONENT ANALYSIS

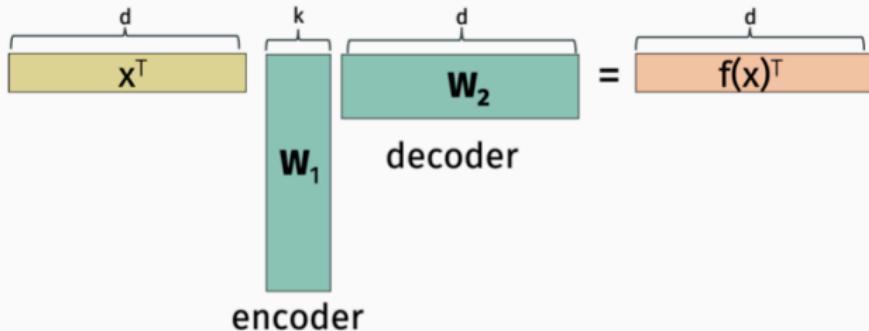
Consider the simplest possible autoencoder:



- One hidden layer. No non-linearity. No biases.
- Latent space of dimension k .
- Weight matrices are $W_1 \in \mathbb{R}^{d \times k}$ and $W_2 \in \mathbb{R}^{k \times d}$.

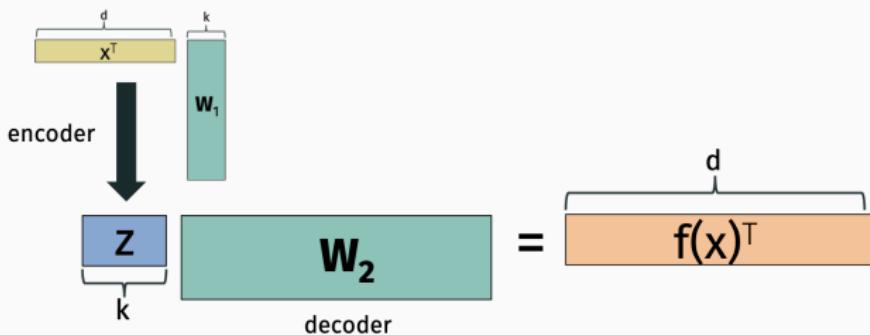
PRINCIPAL COMPONENT ANALYSIS

Given input $\mathbf{x} \in \mathbb{R}^d$, what is $f(\mathbf{x})$ expressed in linear algebraic terms?



$$f(\mathbf{x})^T = \mathbf{x}^T \mathbf{W}_1 \mathbf{W}_2$$

PRINCIPAL COMPONENT ANALYSIS



$$\text{Encoder: } e(x) = x^T w_1.$$

$$\text{Decoder: } d(z) = z w_2$$