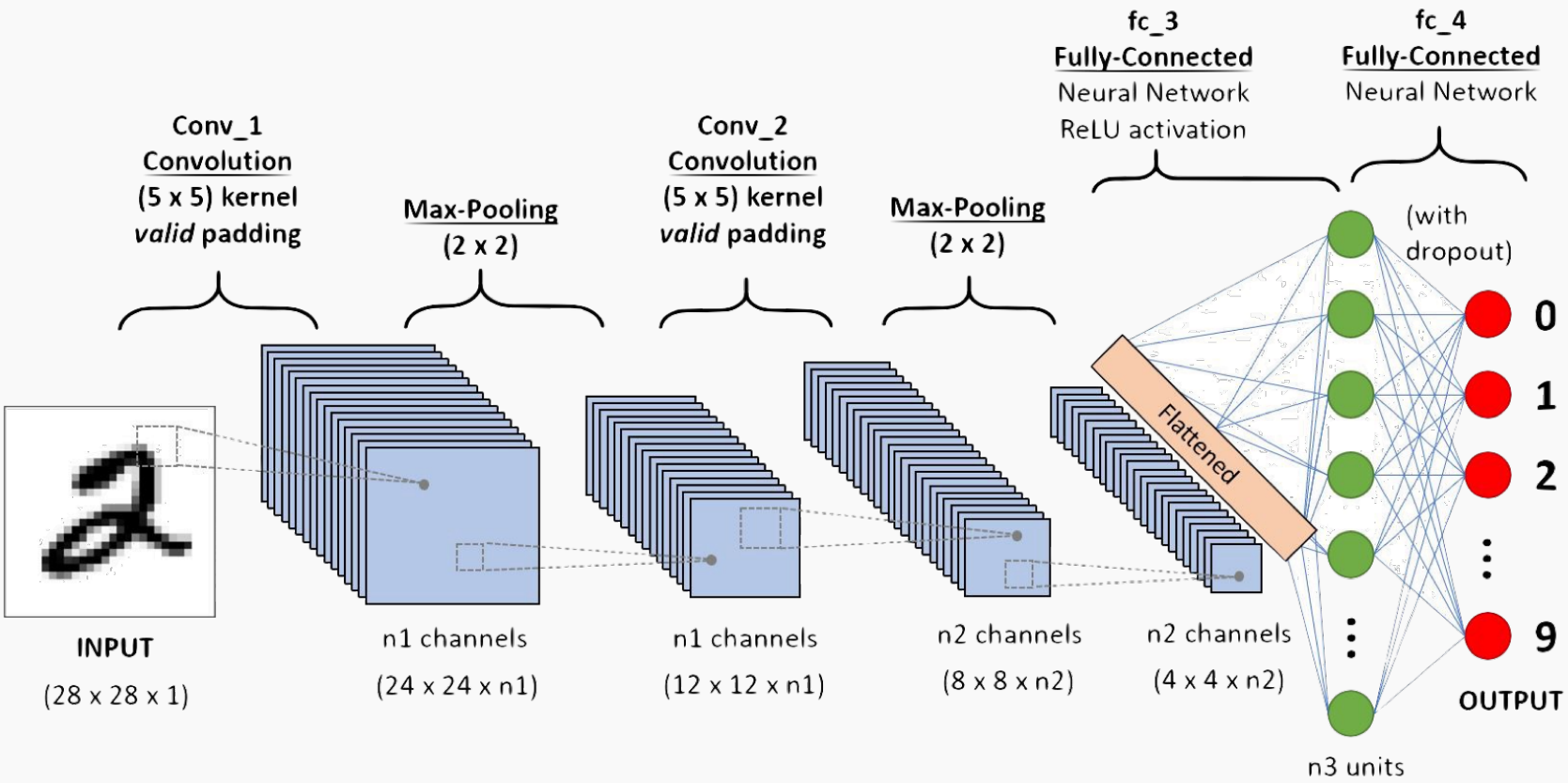
CS-GY 6923: Lecture 11
Finishing CNN and Adversarial examples, Autoencoders, PCA

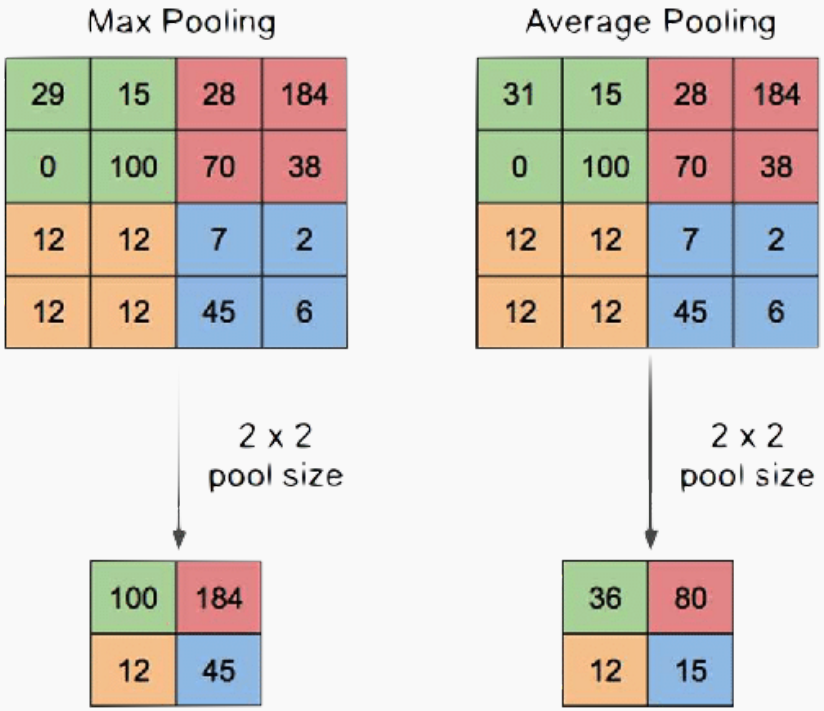
NYU Tandon School of Engineering, Akbar Rafiey
Slides by Prof. Christopher Musco

Overall network architecture

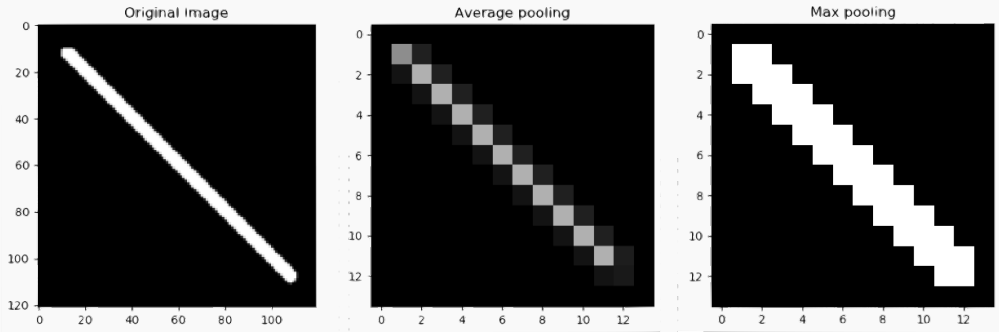


Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

Pooling and downsampling



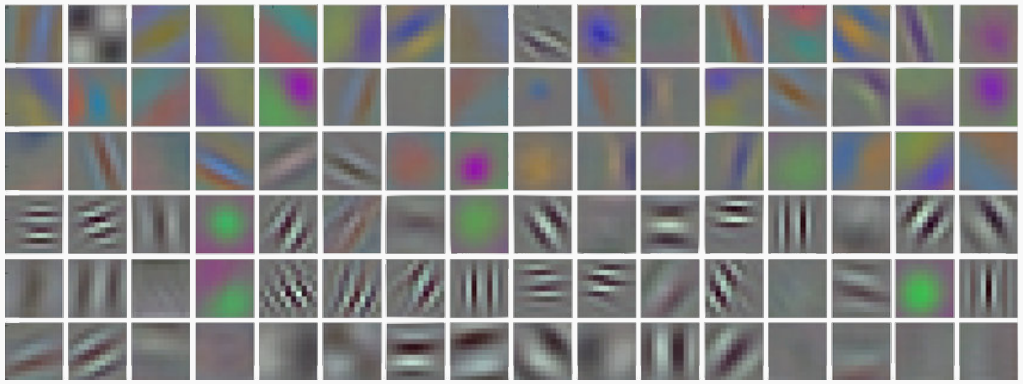
- Reduces number of variables.
- Helps "smooth" result of convolutional filters.
- Improves shift-invariance.



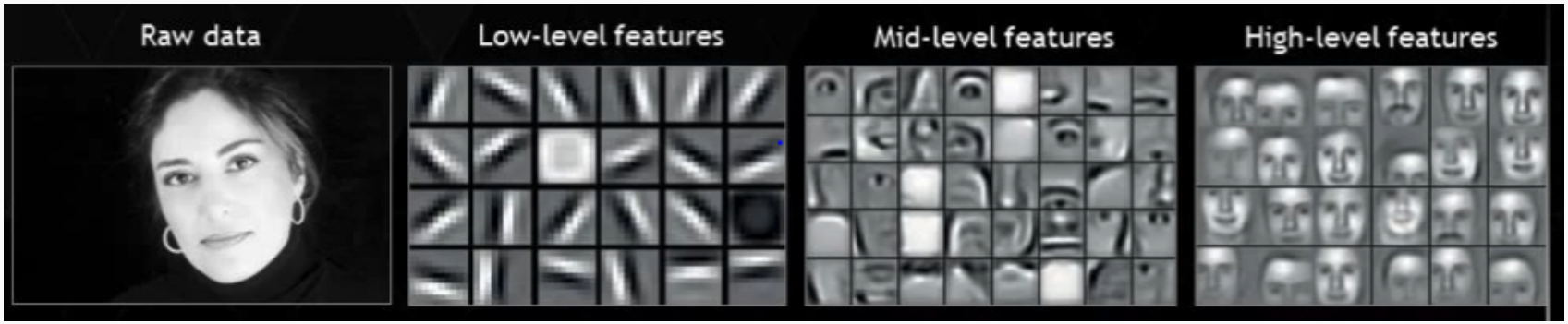
Understanding layers

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

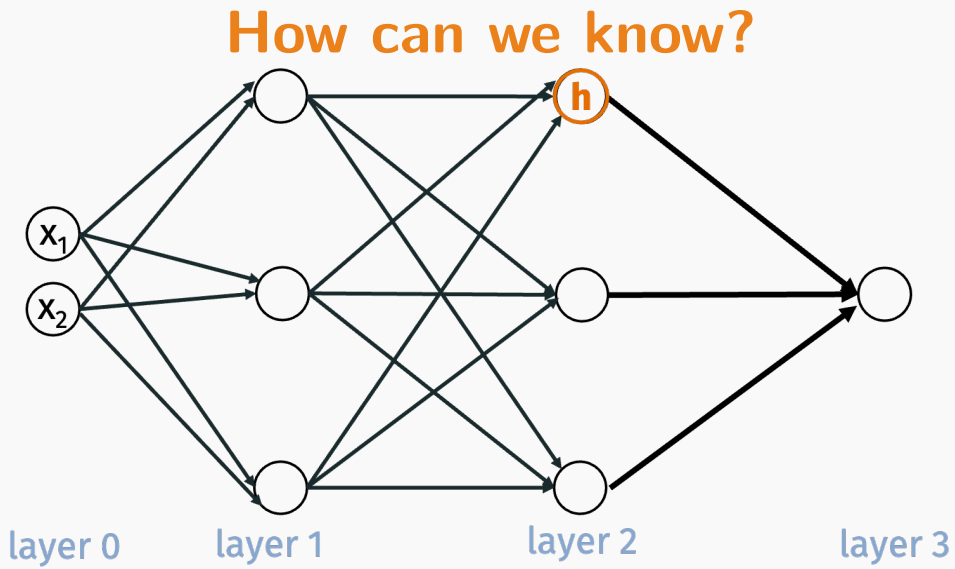
Lots of edge detectors in the first layer!



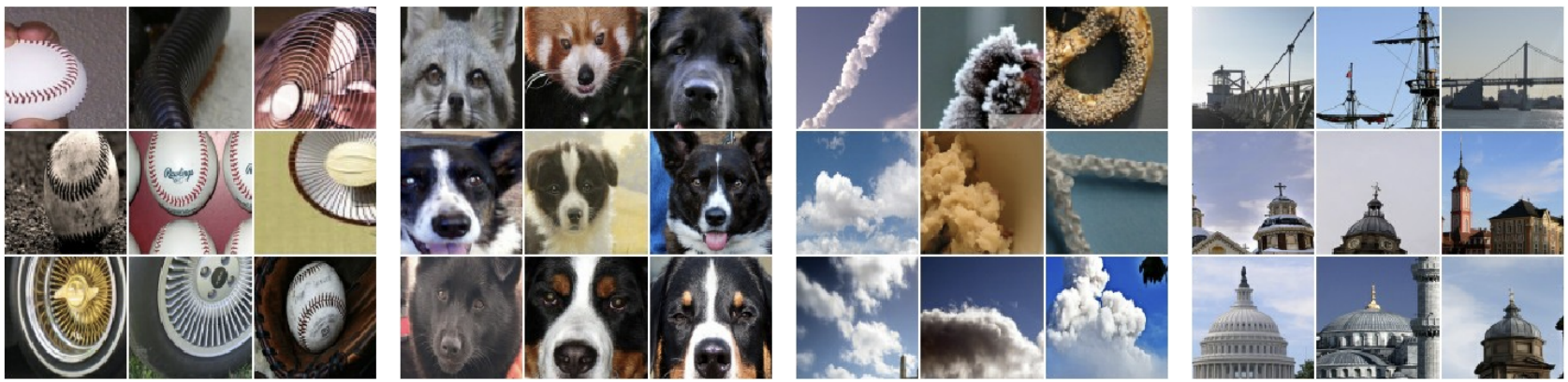
Other layers are harder to understand... but roughly hidden variables later in the network encode for "higher level features":



Understanding layers

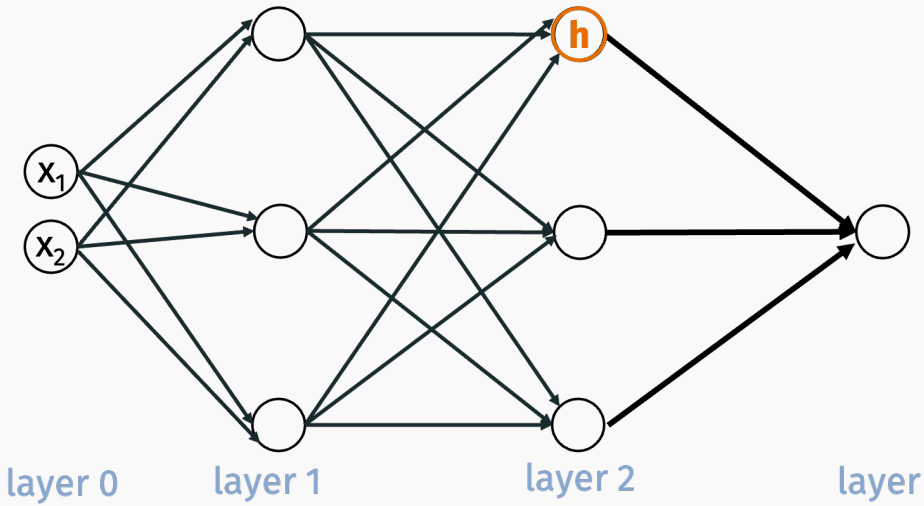


Go through dataset and find the inputs that most "excite" a given neuron h. I.e. for which $|h(\mathbf{x})|$ is largest.



Understanding layers

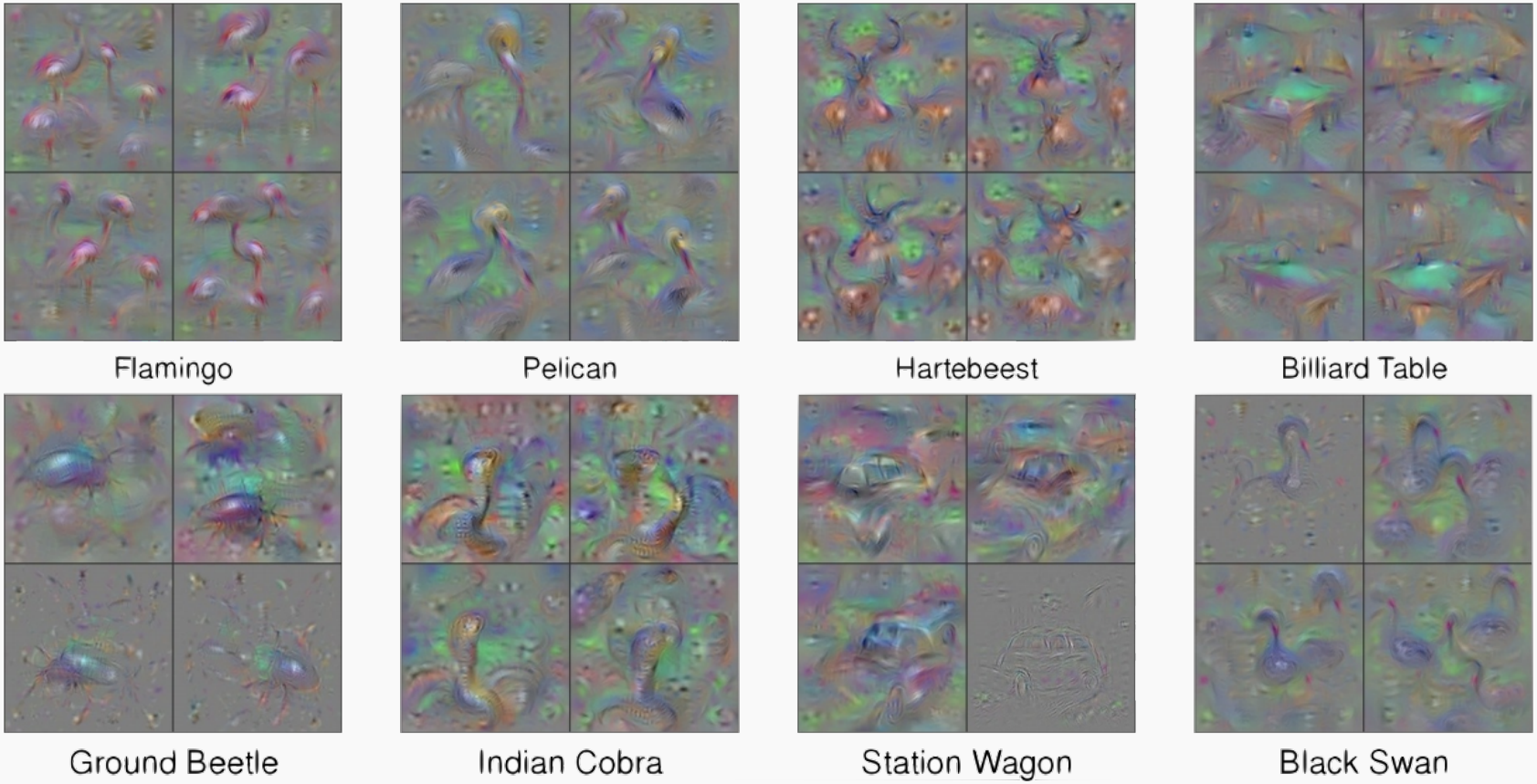
How can we know?



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

Understanding layers

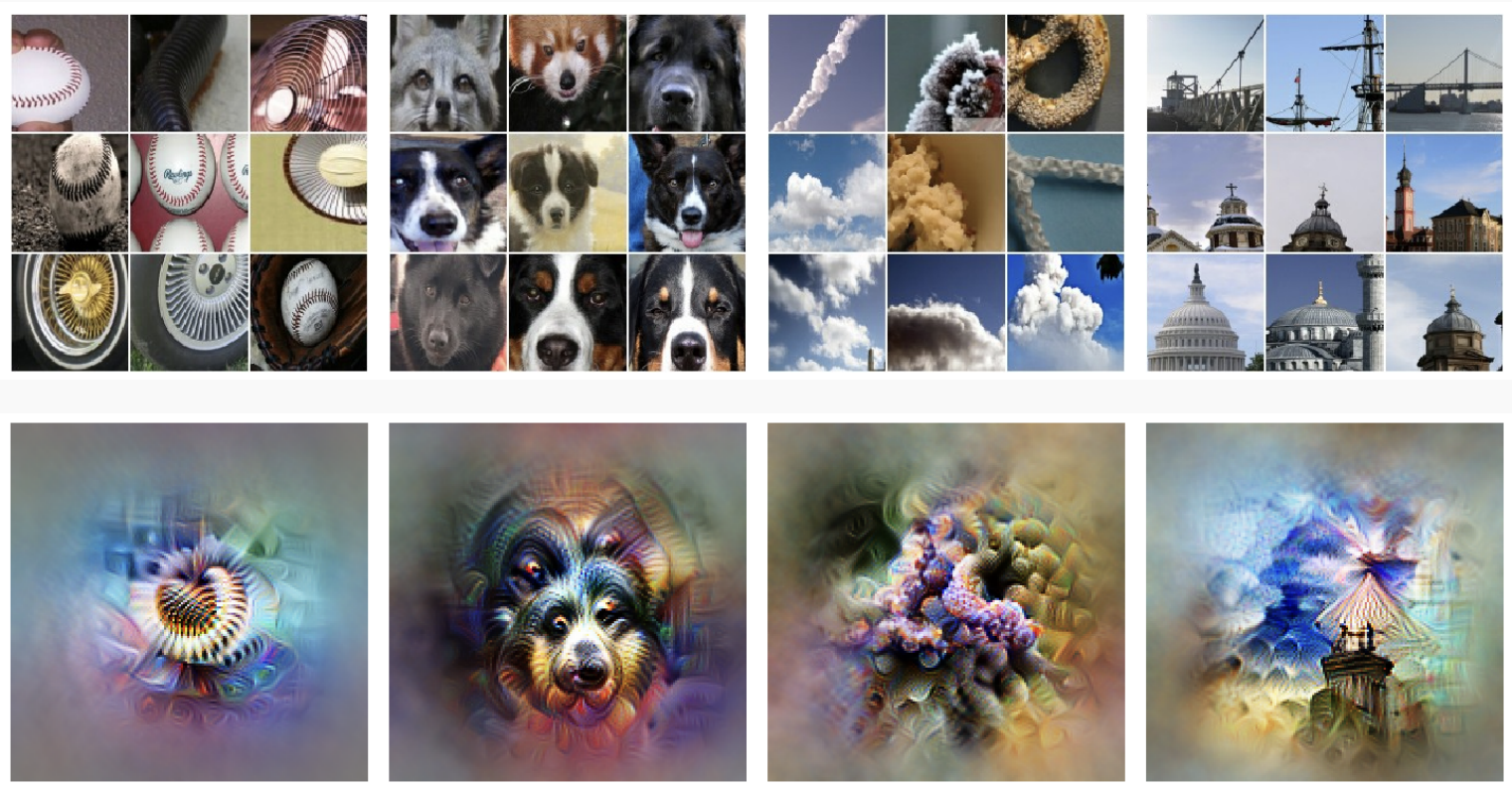
Early work had some interesting results.



"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.

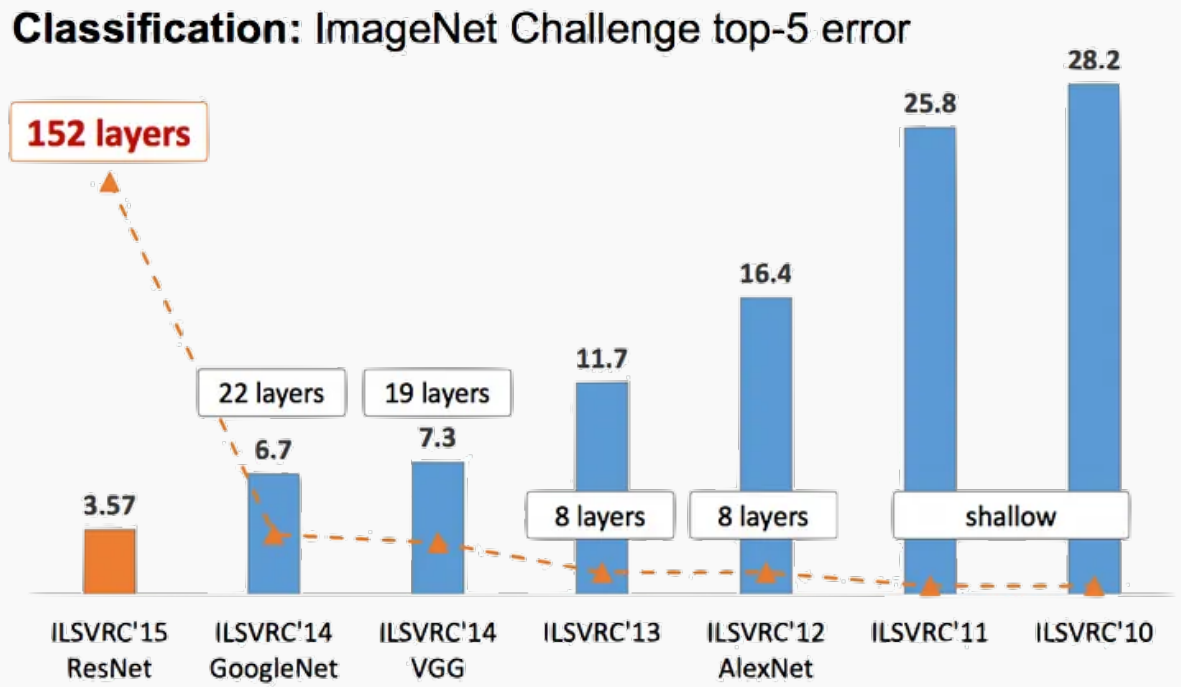
Parts (layers mixed4b & mixed4c)

Objects (layers mixed4d & mixed4e)

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

Deeper and deeper, bigger and bigger

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



Tricks of the trade

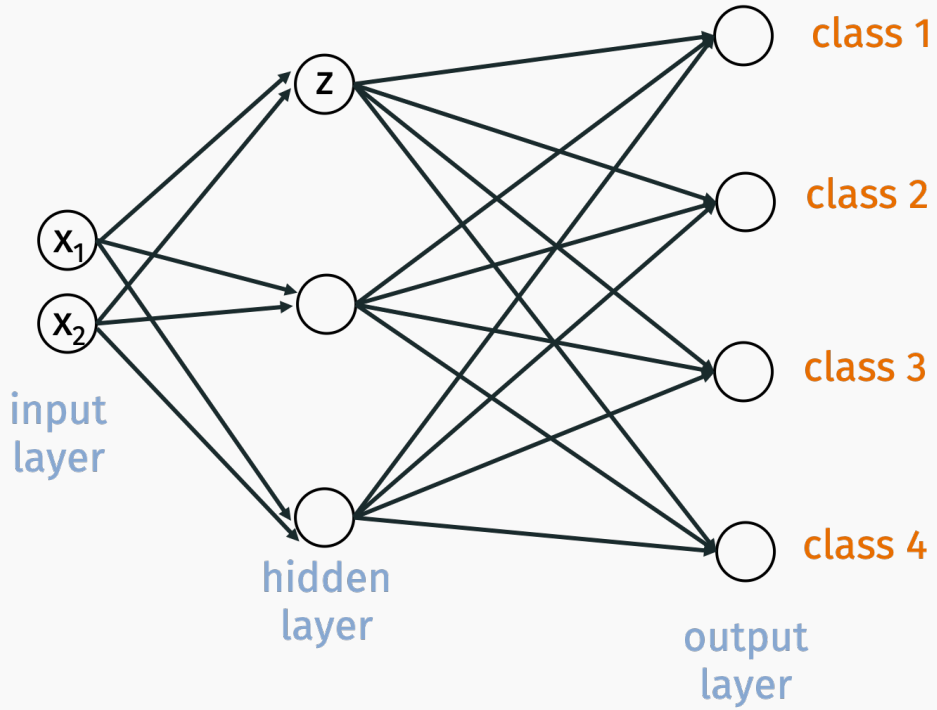
Beyond techniques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of "tricks".

- 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 $\mathbf{x}_1 \ldots, \mathbf{x}_n$:

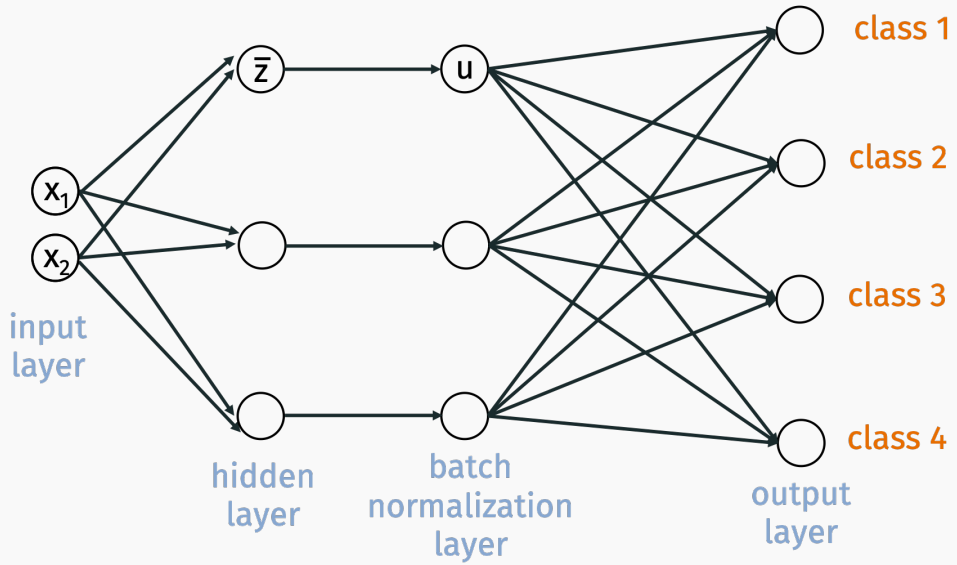
$$\mu = \frac{1}{n} \sum_{j=1}^{n} \bar{z}(\mathbf{x}_j)$$

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

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.

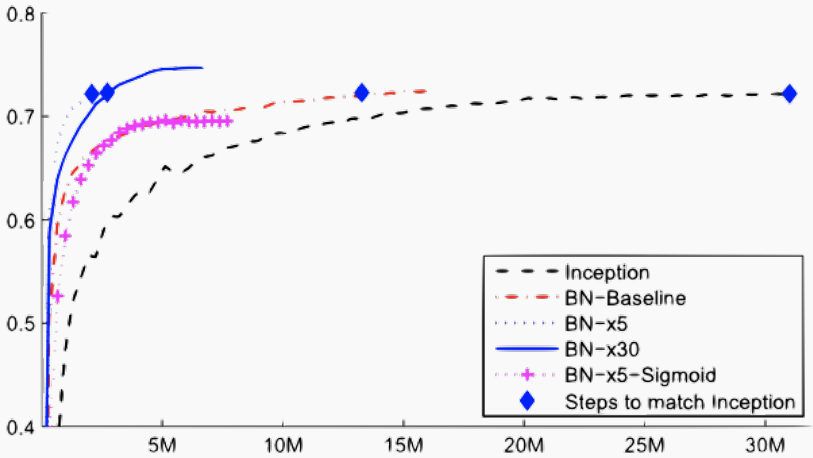


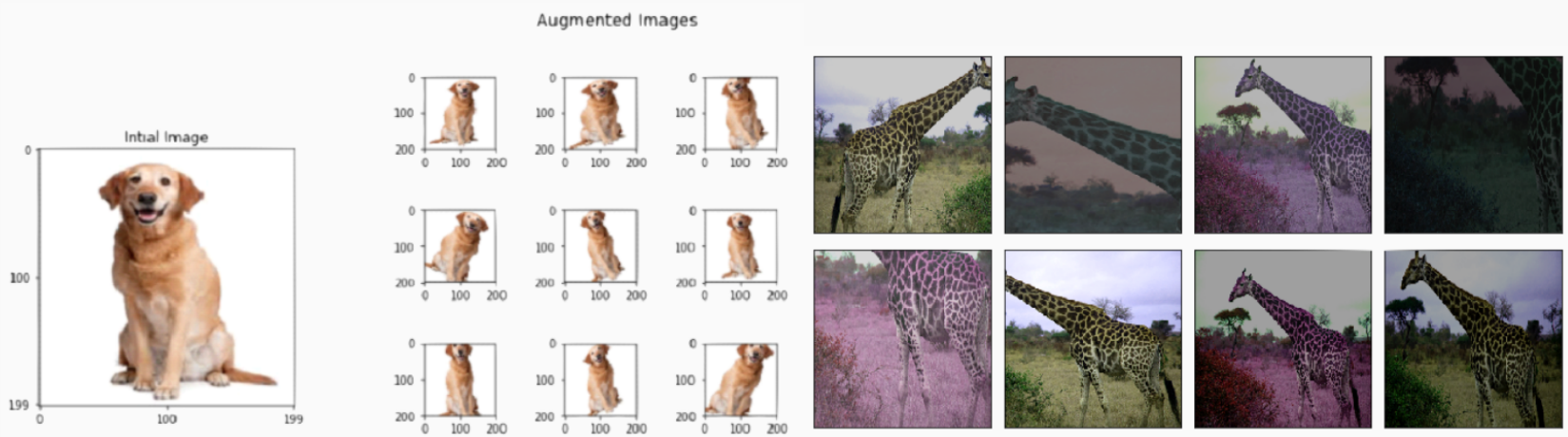
Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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.



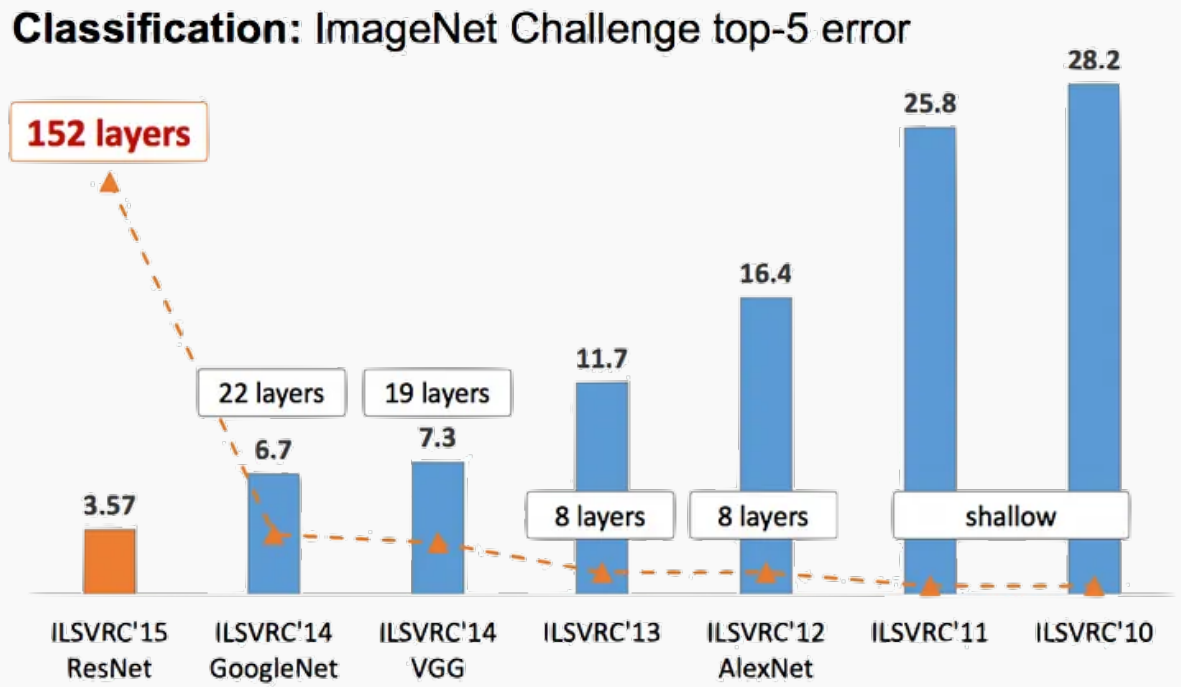
Take training images and randomly shift, flip, rotate, skew, darken, lighten, shift colors, etc. to create new training images. **Final classifier will be more robust to these transformations.**

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.

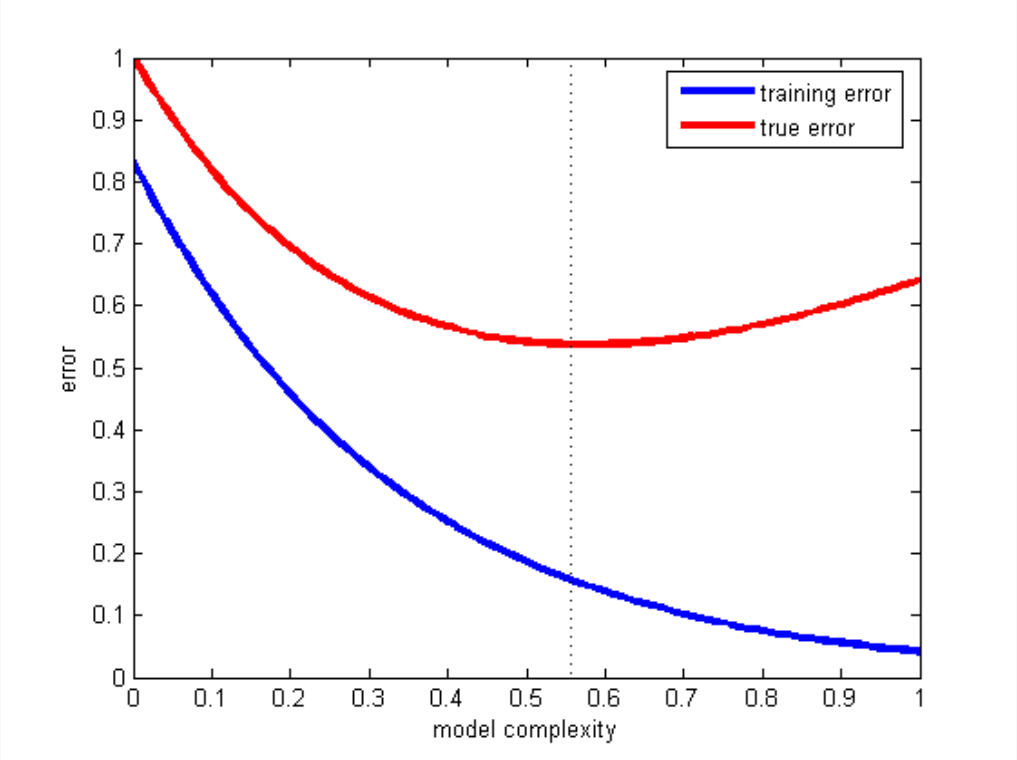
Deeper and deeper, bigger and bigger

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

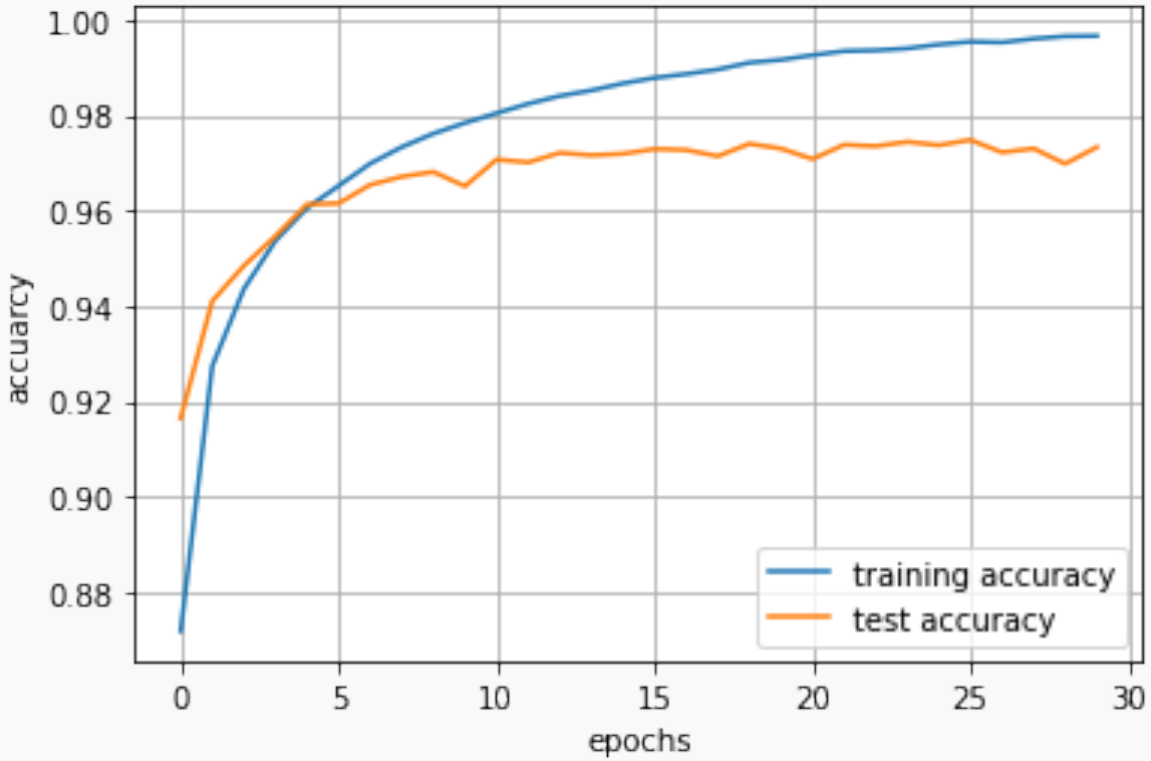
Benjamin Recht†
University of California, Berkeley
brecht@berkeley.edu

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.

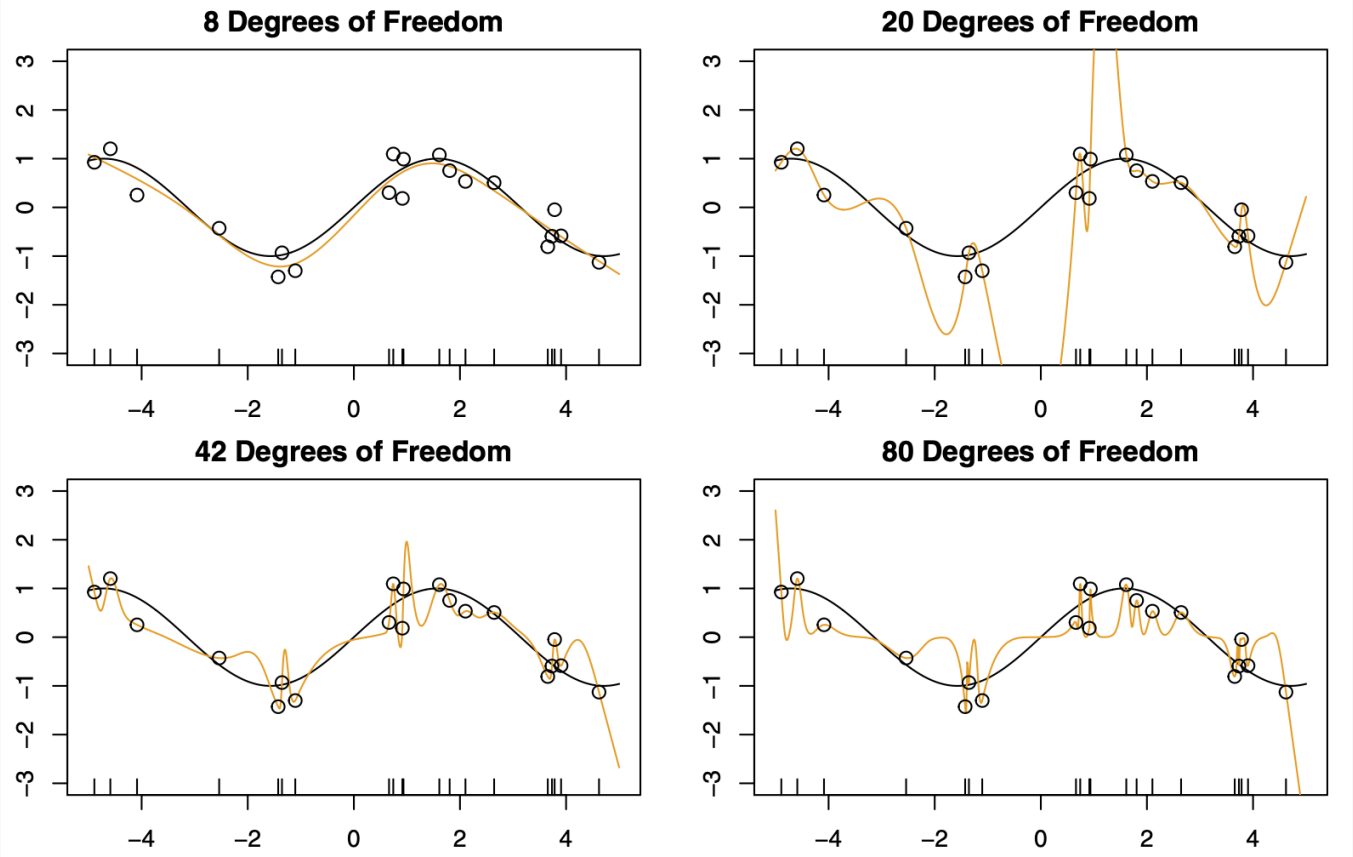
Generalization for neural networks

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



Generalization for neural networks

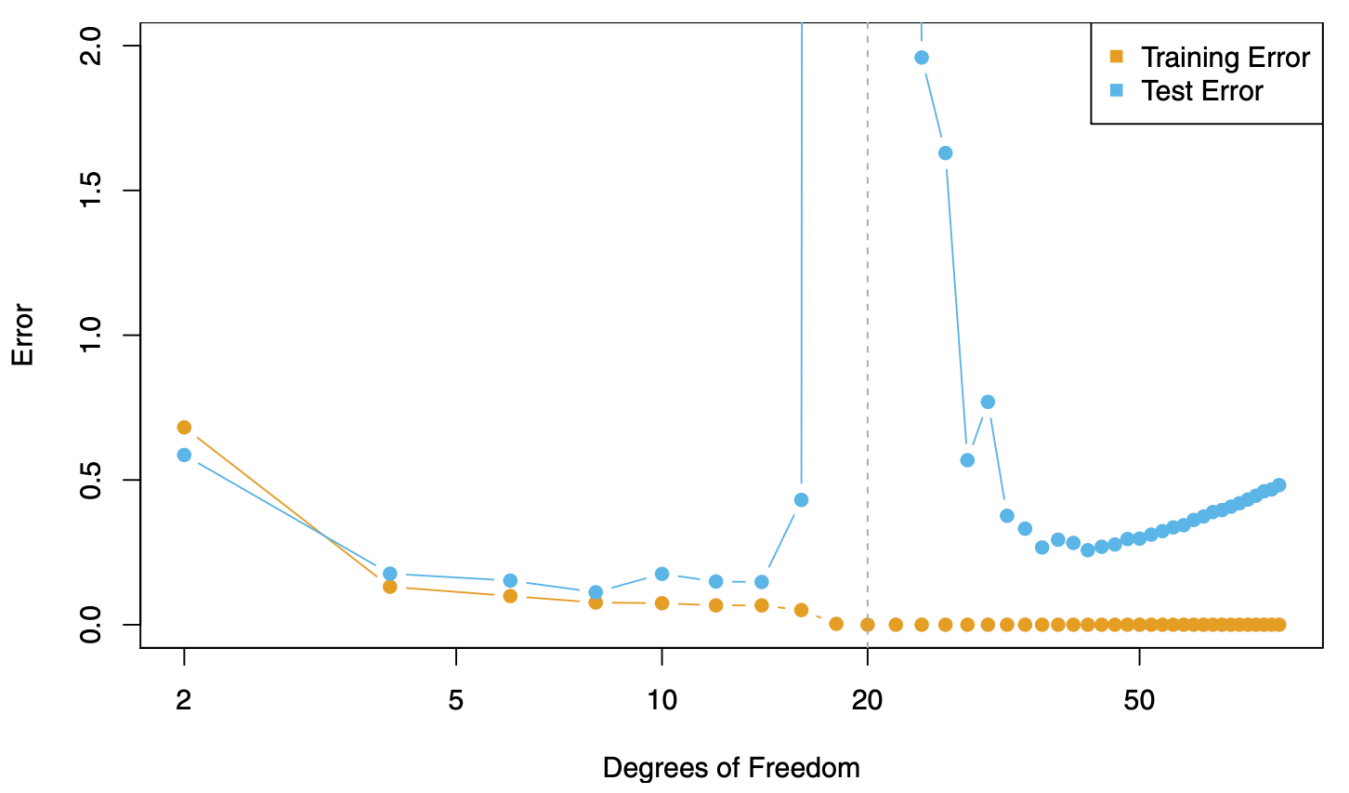
One growing realization is that this phenomena doesn't only apply to neural networks – it can also be true for fitting highly-overparameterized polynomials.



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.

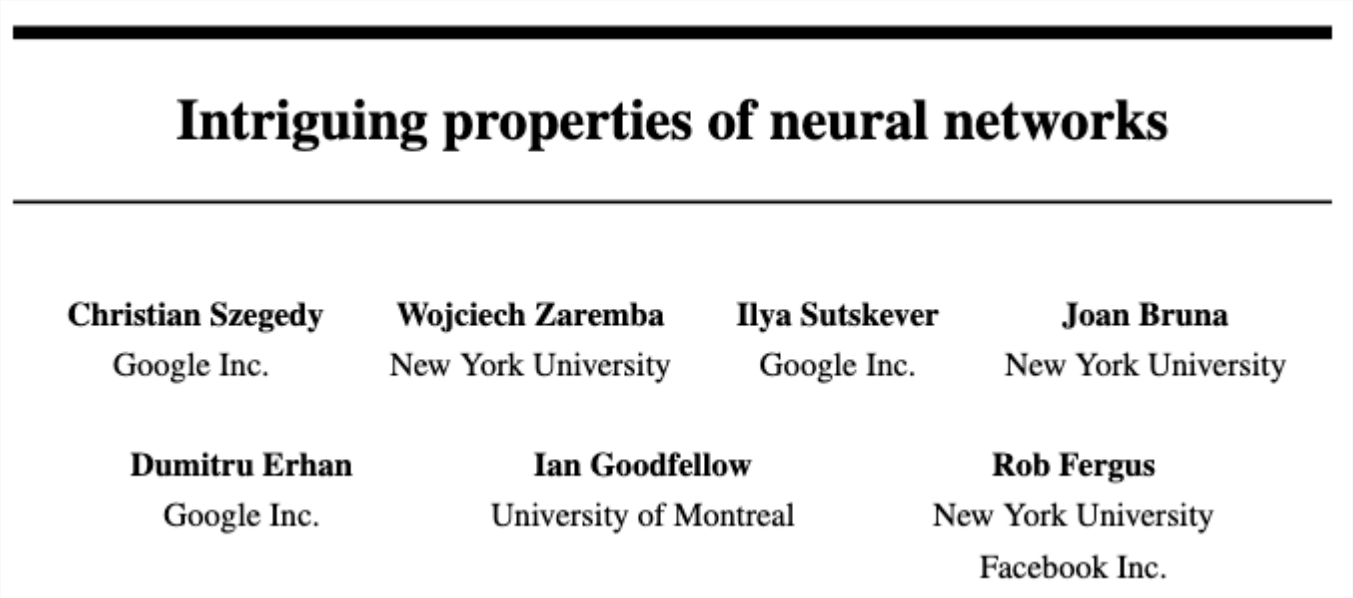


We don't usually see this same curve for neural networks.

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.

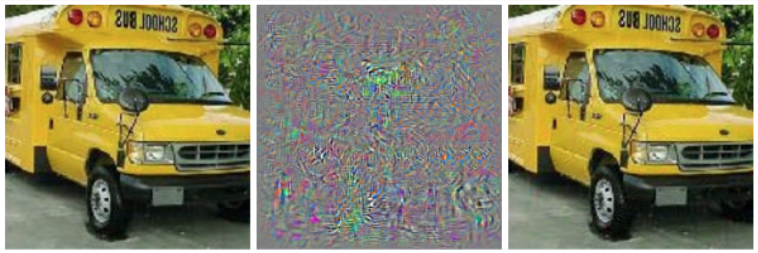
Ian Goodfellow
University of Montreal

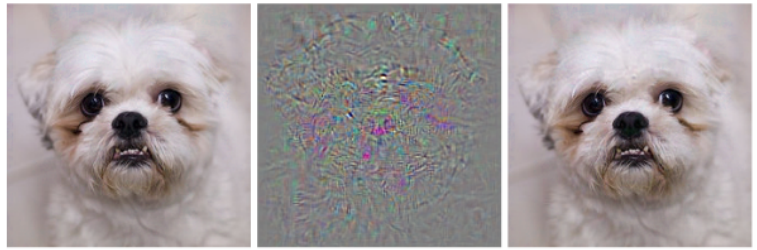
Rob Fergus
New York University
Facebook Inc.

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 $\ell(\theta, \mathbf{x}, y)$.

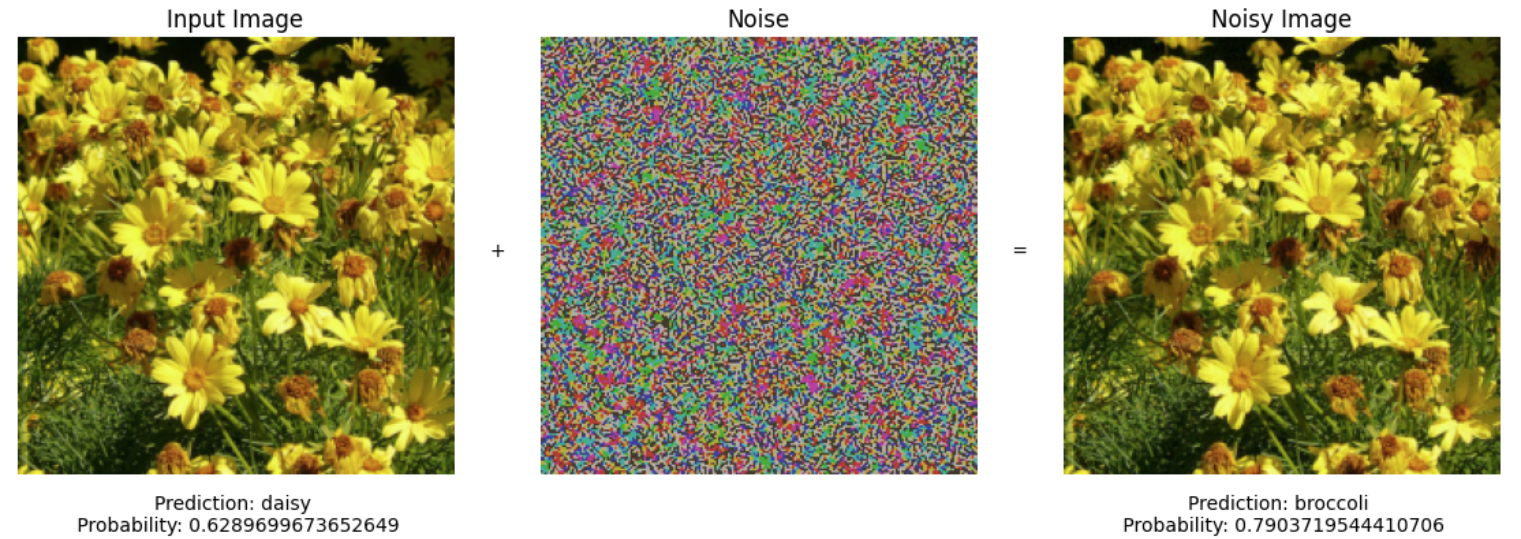
Solve the optimization problem:

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

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).



Transfer Learning and Autoencoders

Transfer learning

State-of-the-art supervised learning models like neural networks learn **very good features**.

But they require lots and lots of data. ImageNet has 14 million labeled images. Mostly of everyday objects.

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

Real example: Classify images of insects for use in agricultural applications in new localities.

Zero-Shot Insect Detection via Weak Language Supervision

Benjamin Feuer,[1] **Ameya Joshi,**[1] **Minsu Cho,**[1] **Kewal Jani,**[1] **Shivani Chiranjeevi,** [2] **Zi Kang Deng,** [3] **Aditya Balu,** [2] **Asheesh K. Singh,** [2] **Soumik Sarkar,** [2] **Nirav Merchant,** [3] **Arti Singh,** [2] **Baskar Ganapathysubramanian,** [2] **Chinmay Hegde** [1]

[1] New York University
[2] Iowa State University
[3] University of Arizona

Aedes Vexans

Creatonotos Gangis

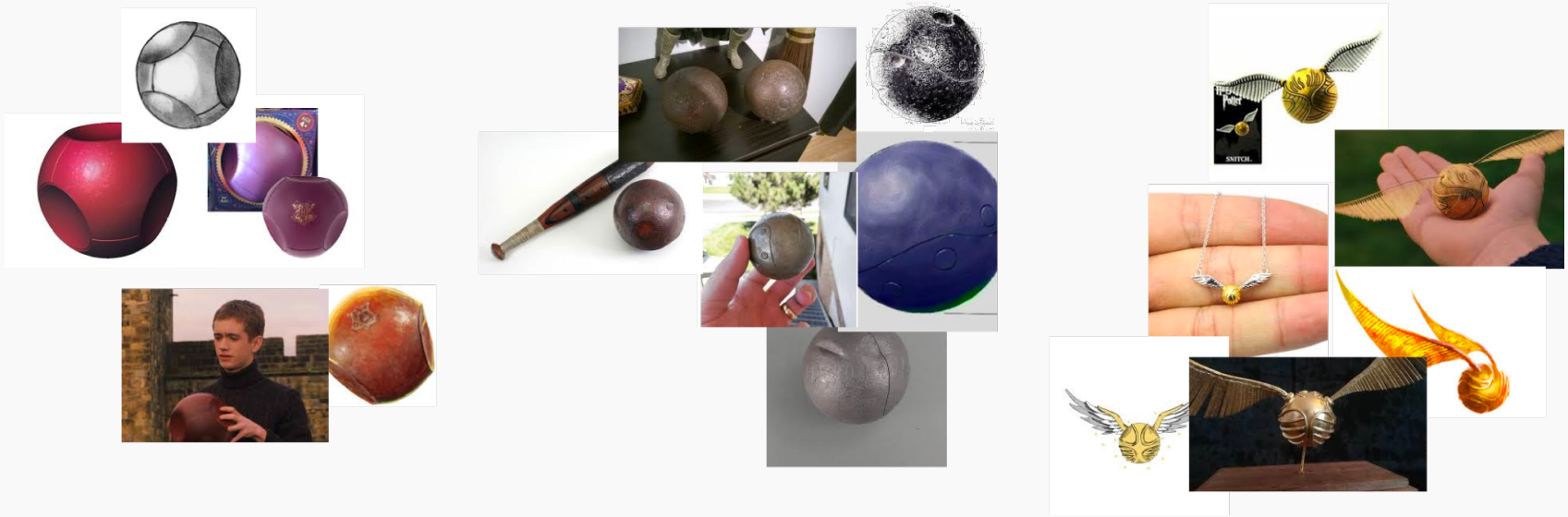
Daphnis Neril

Hypena Deceptalis

Pyralis Farinalis

One-shot learning

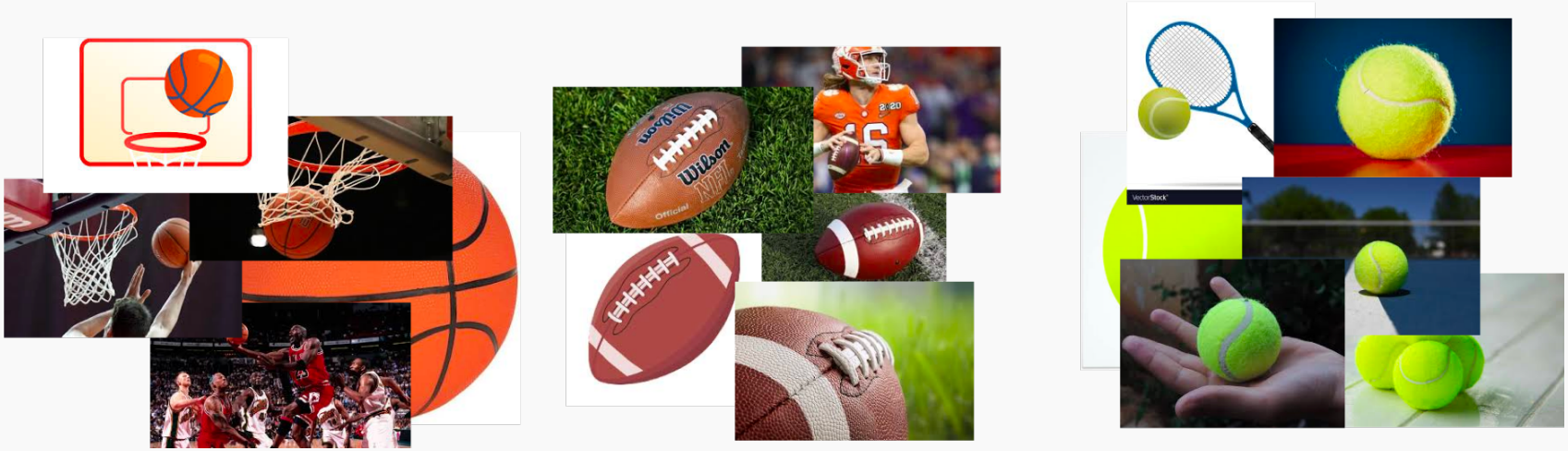
A human could probably achieve near perfect classification accuracy even given access to a **single labeled example** from each class:



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

Transfer learning

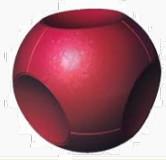
Transfer knowledge from one task we already know how to solve to another.



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.

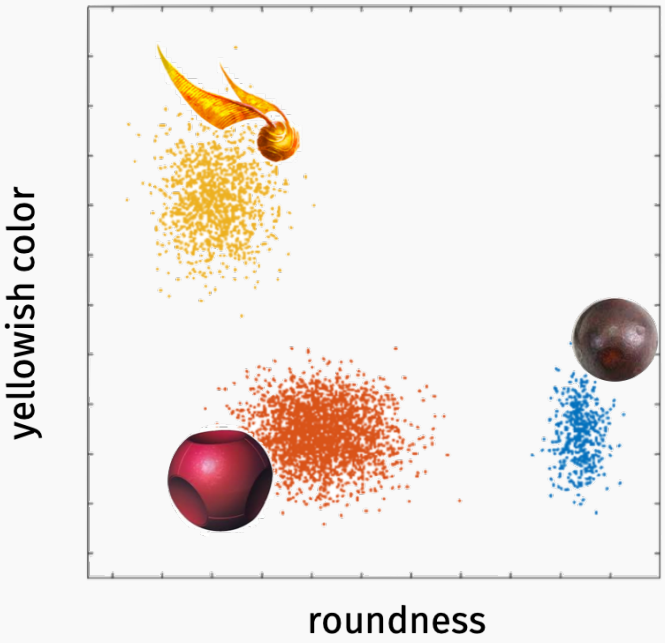
Feature learning

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

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

Feature learning

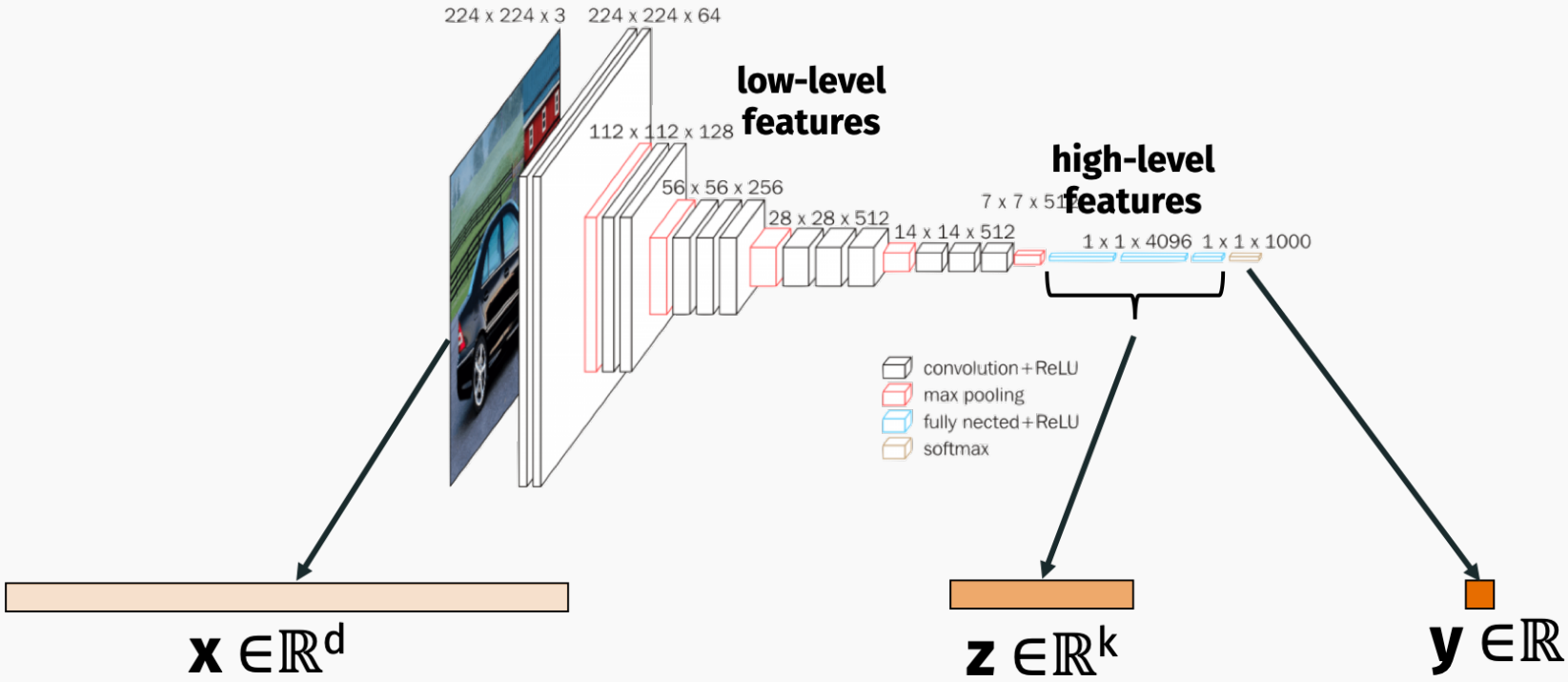
If these features are highly informative (i.e. lead to highly separable data) few training examples are needed to learn.



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.

But what if we don't even have labeled data for a sufficiently related problem?

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 have been really good at for the past 8 years.
- **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 have recently made machines a lot better at.

Autoencoder

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

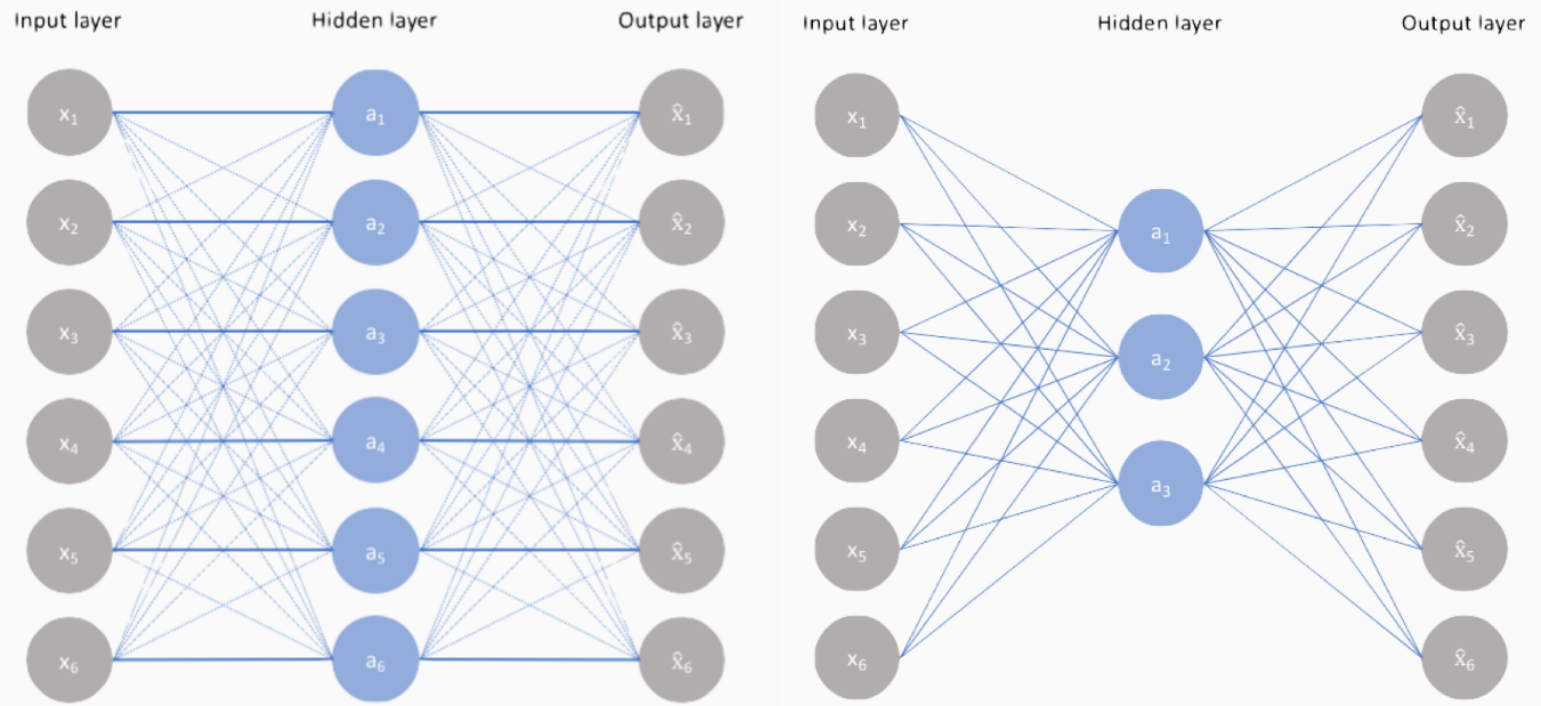
- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$ be our model.
- Let $L_{\boldsymbol{\theta}}$ be a loss function. E.g. squared loss: $L_{\boldsymbol{\theta}}(\mathbf{x}) = \|\mathbf{x} - f_{\boldsymbol{\theta}}(\mathbf{x})\|_2^2$.
- Train model: $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}}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

If $f_{\boldsymbol{\theta}}$ is a model that incorporates feature learning, then these features can be used for supervised tasks.

$f_{\boldsymbol{\theta}}$ is called an **autoencoder**. It maps input space to input space (e.g. images to images, french to french, PDE solutions to PDE solutions).

Autoencoder

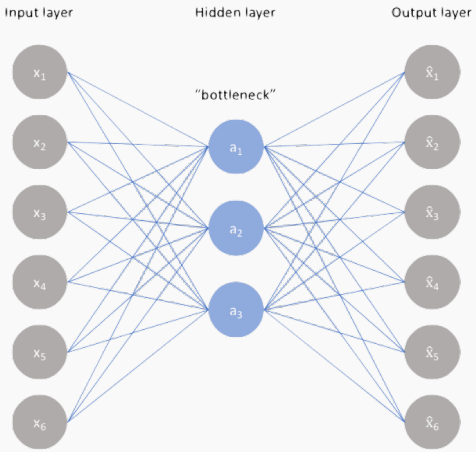
Two examples of autoencoder architectures:

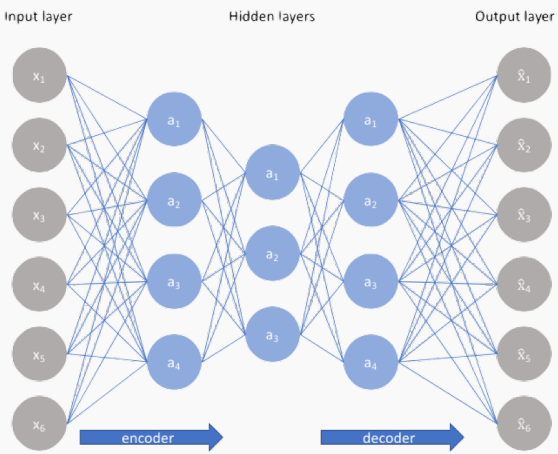


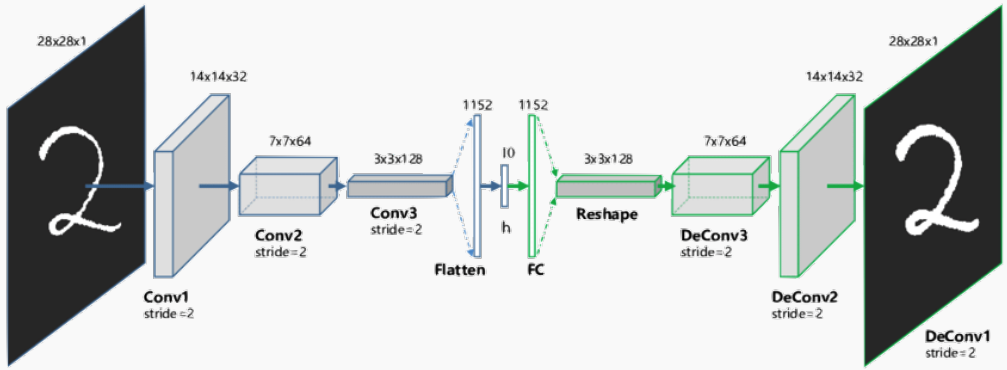
Which would lead to better feature learning?

Autoencoder

Important property of autoencoders: no matter the architecture, 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.





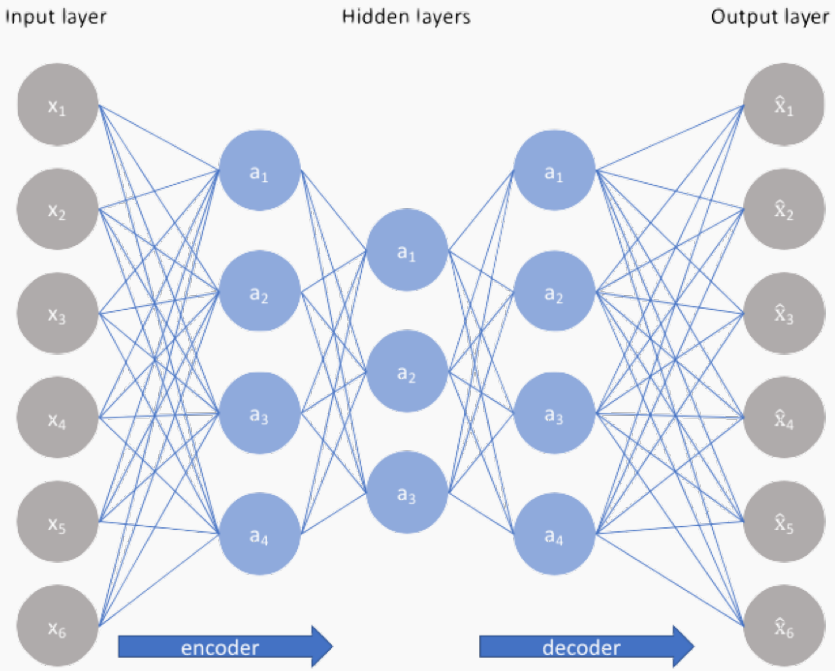


Autoencoder

Separately name the mapping from input to bottleneck and from bottleneck to output.

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

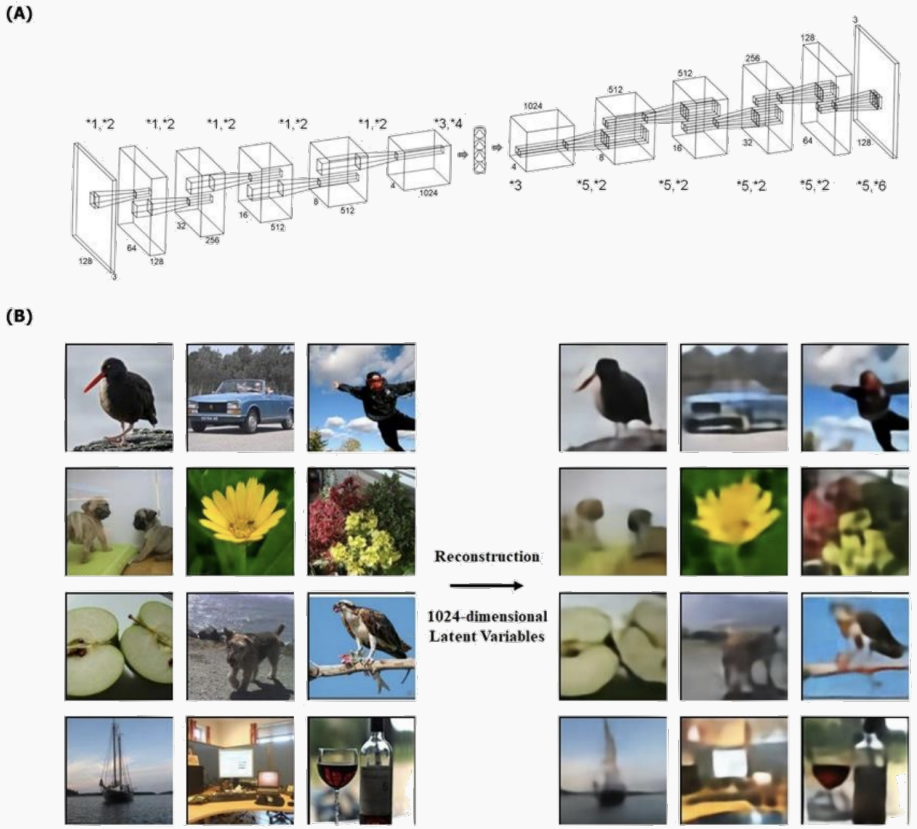
$$f(\mathbf{x}) =$$



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 supervised methods for feature extraction, but they require no labeled data.[1]

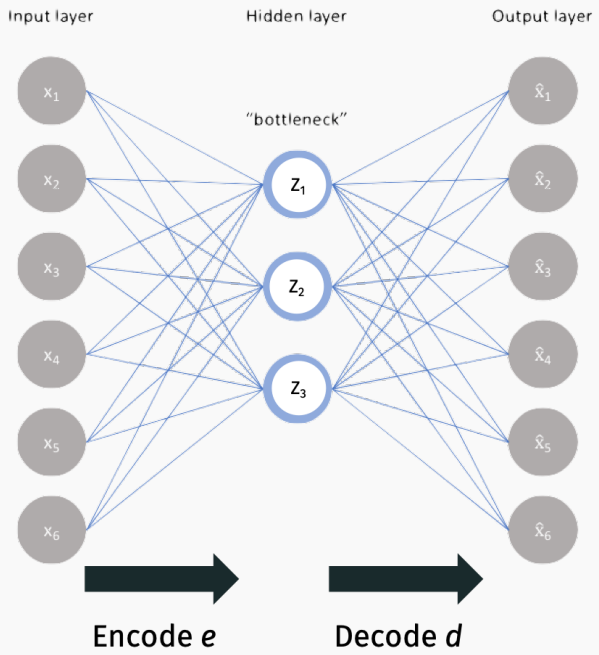
There are a lot of cool applications of autoencoders beyond feature learning!

- Learned data compression.
- Denoising and in-painting.
- Data/image synthesis.

[1] Recent progress on **self-supervised** learning achieves the best of both worlds – state-of-the-art feature learning with no labeled data.

Autoencoders for data compression

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



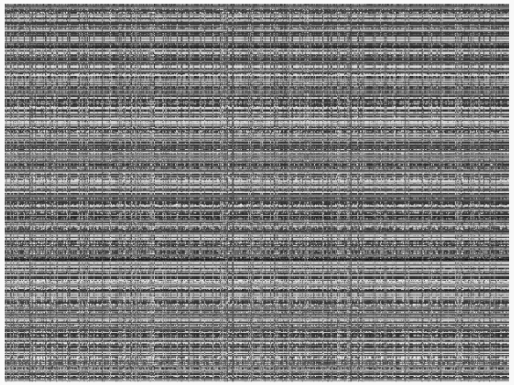
Given input $\mathbf{x}$, we can completely recover $f(\mathbf{x})$ from $\mathbf{z} = e(\mathbf{x})$. $\mathbf{z}$ typically has many fewer dimensions than $\mathbf{x}$ and for a typical image $f(\mathbf{x})$ will closely approximate $\mathbf{x}$.

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

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

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

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 correction

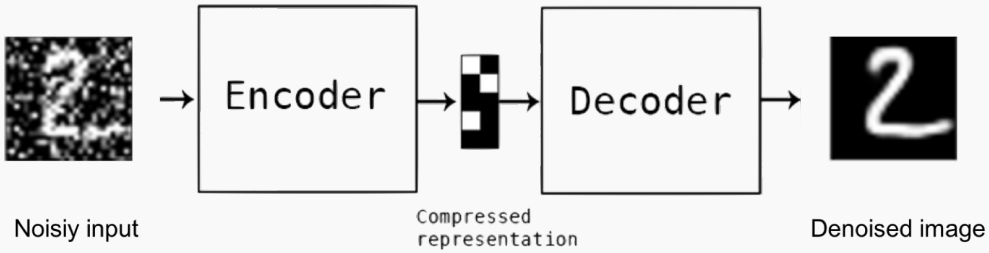


Image denoising

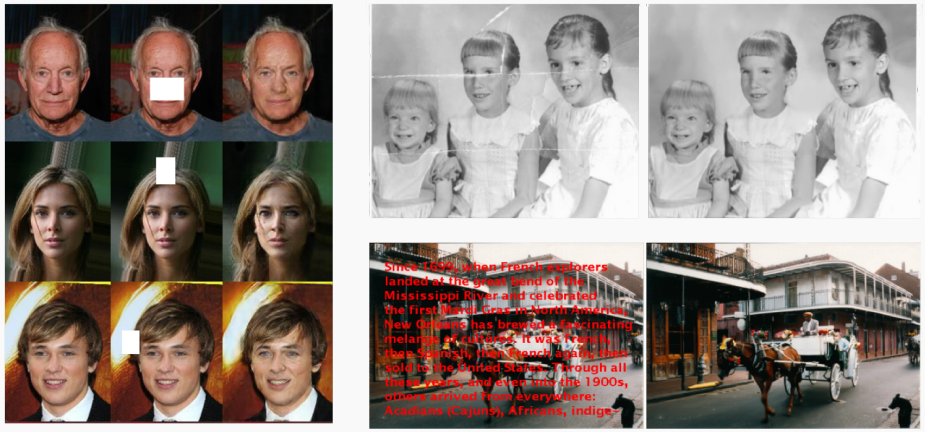
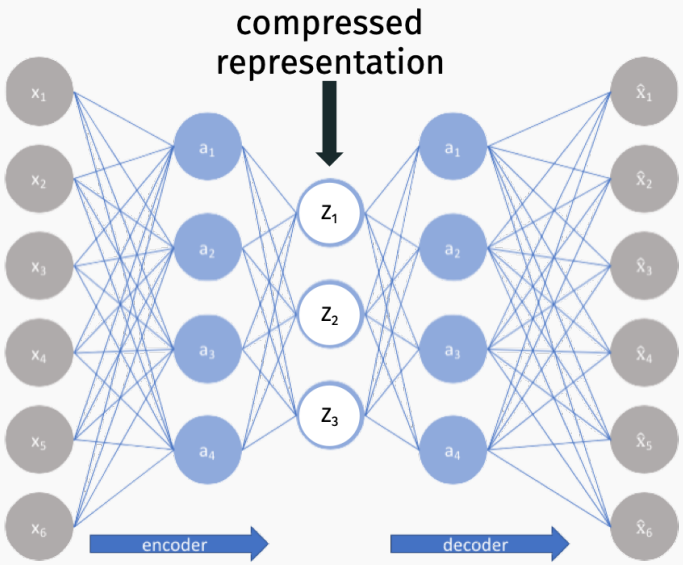


Image inpainting

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

Autoencoders learn compressed representations

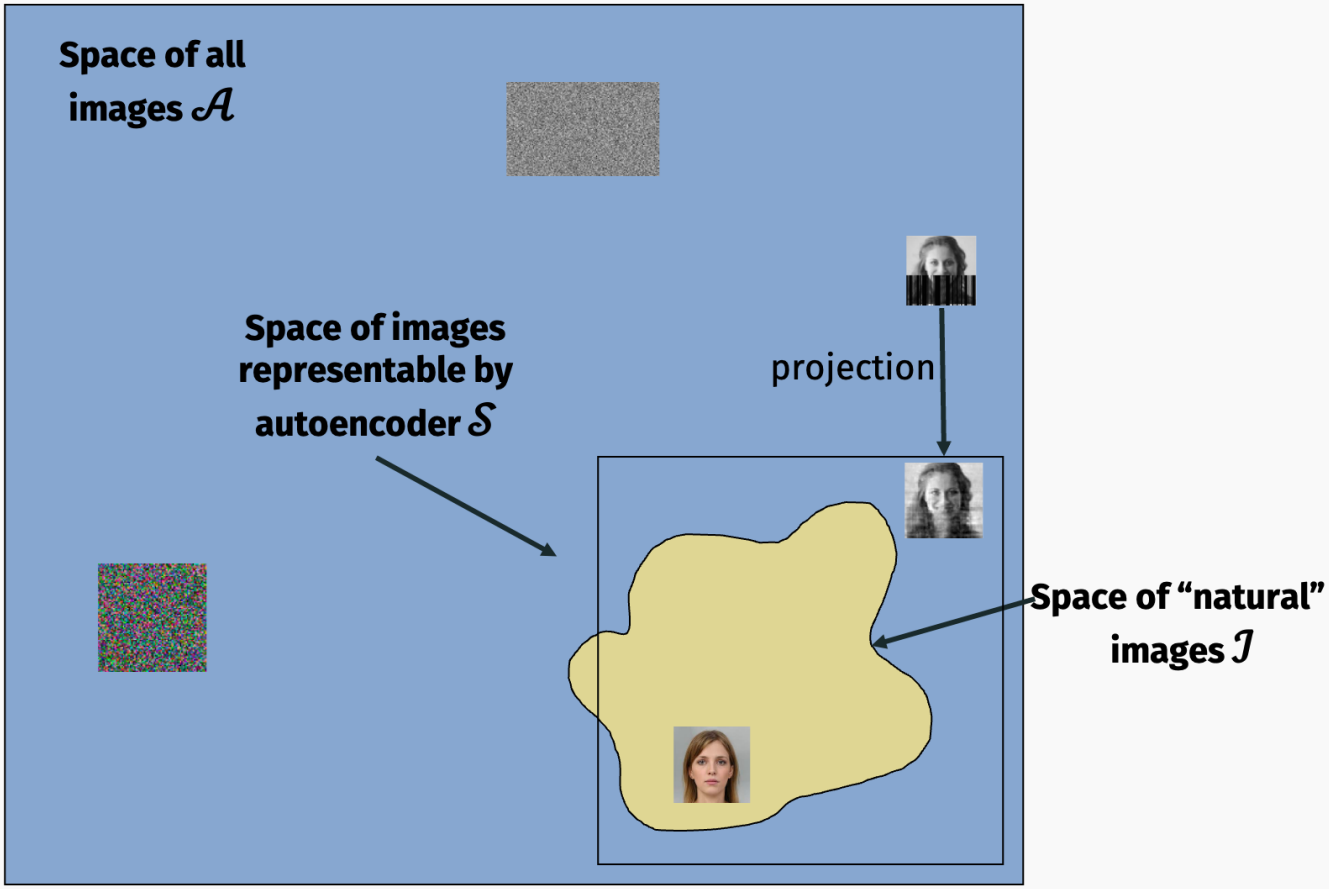
Why does this work?



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

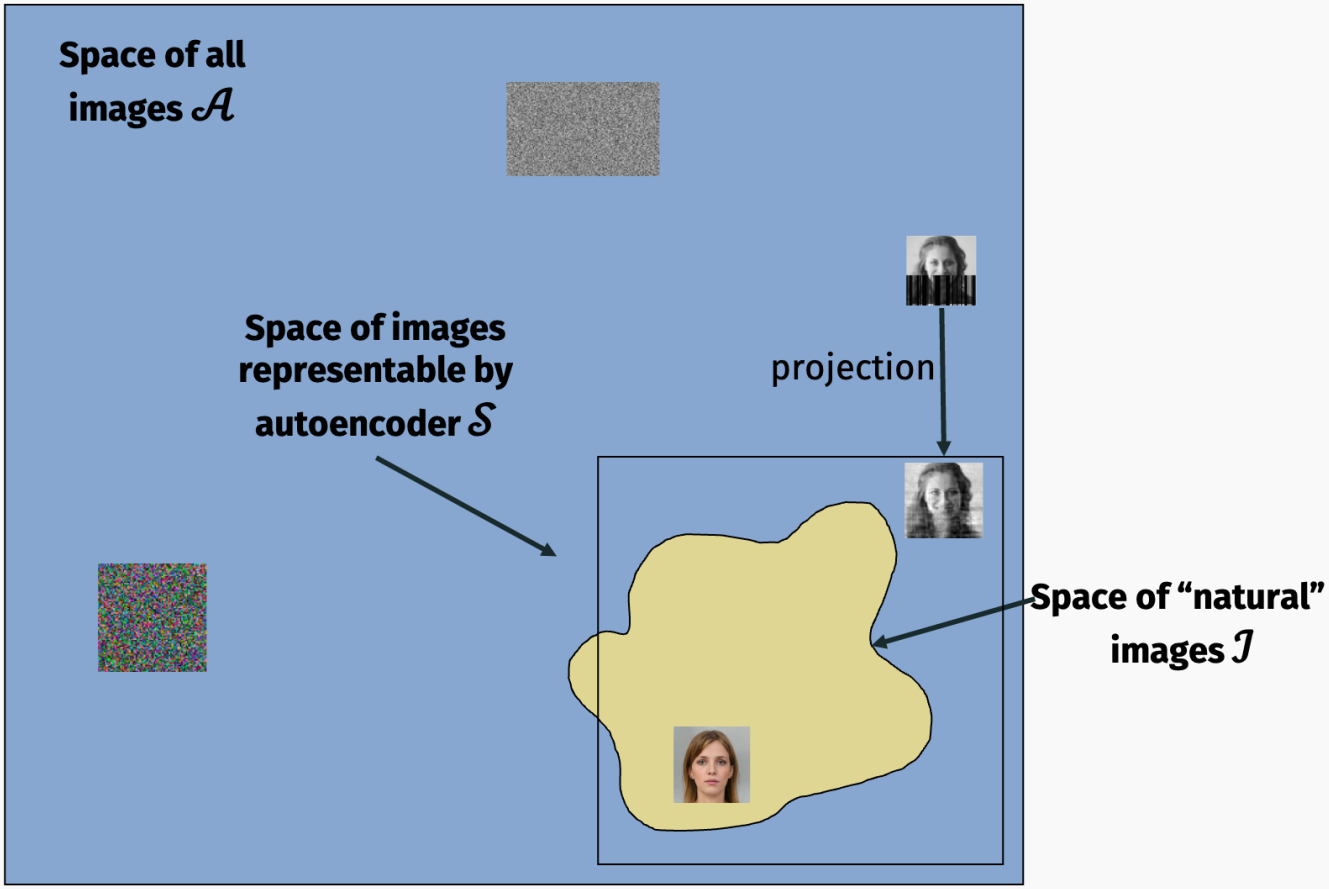
If $\mathbf{z}$ holds k values in $0, .1, .2, \ldots, 1$, how many unique images $\mathbf{w}$ can be output by the autoencoder function f?

Autoencoders learn compressed representations



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

Autoencoders learn compressed representations

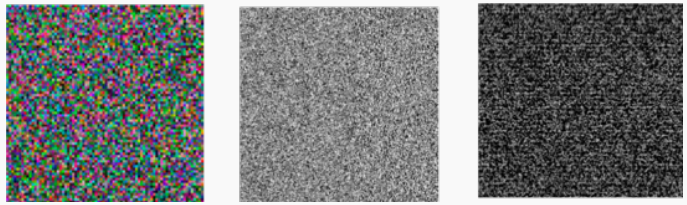


$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

Autoencoders for data generation

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

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

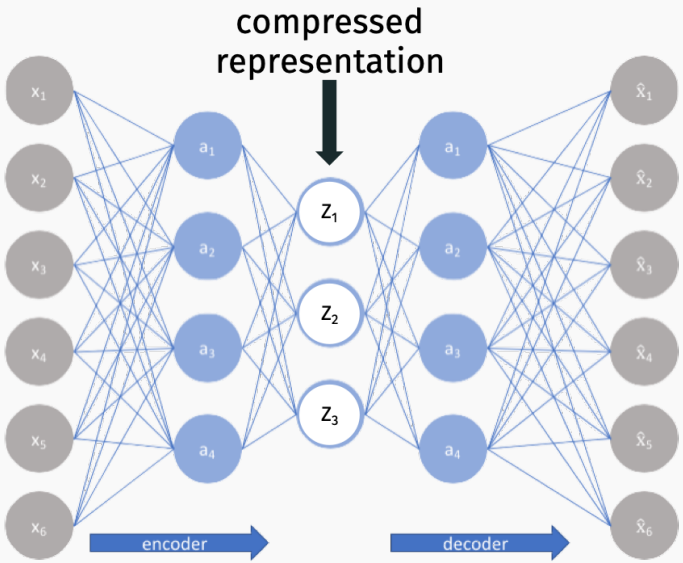


- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.

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

Autoencoders for data generation

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

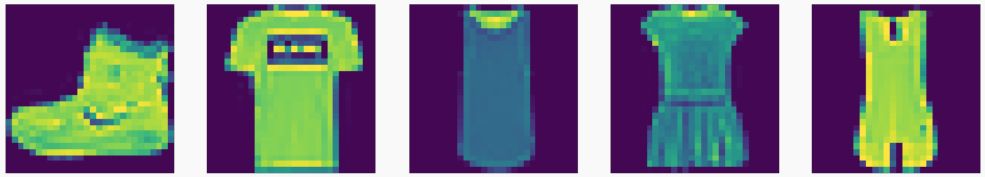


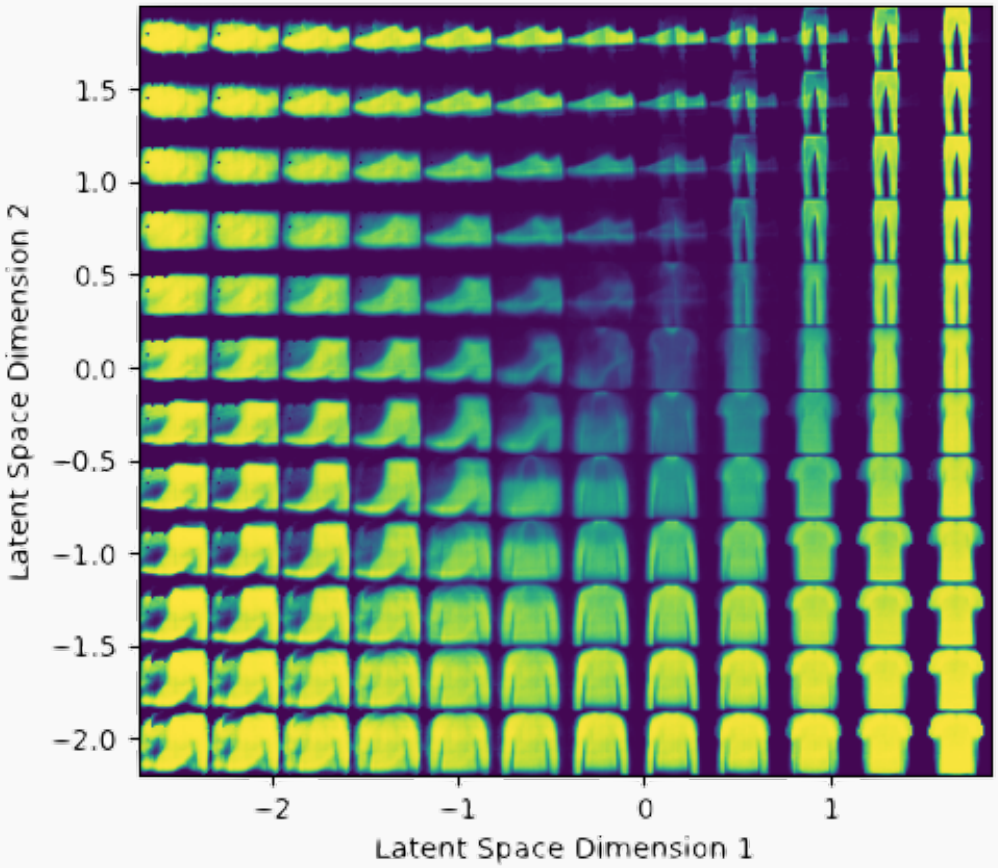
Randomly select code $\mathbf{z}$, then set $\mathbf{x} = d(\mathbf{z})$.[2]

[2] Lots of details to think about here. In reality, people use "variational autoencoders" (VAEs), which are a natural modification of AEs.

Autoencoders for data generation demo

Teal created a demo for the "Fashion MNIST" data set:





Principal Component Analysis (PCA)

Principal Component Analysis (PCA)

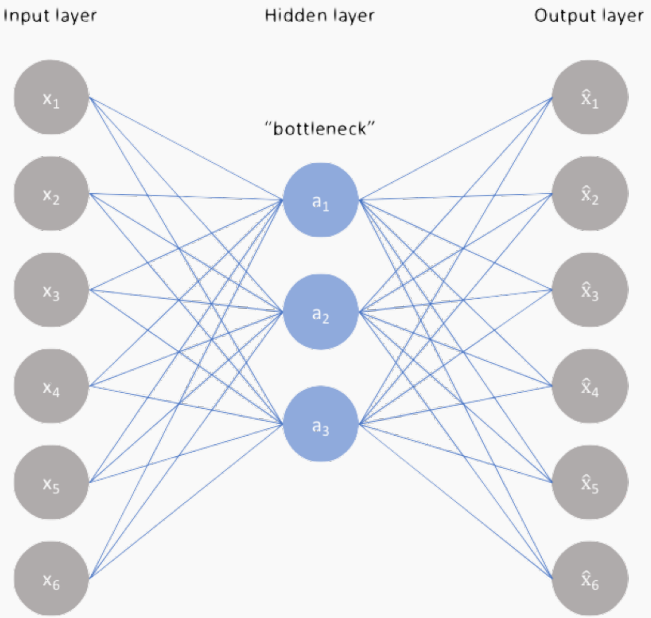
Rest of lecture: Deeper dive into understanding a simple, but powerful autoencoder architecture. Specifically we will view **Principal Component Analysis (PCA)** as a type of autoencoder.

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.

Principal Component Analysis (PCA)

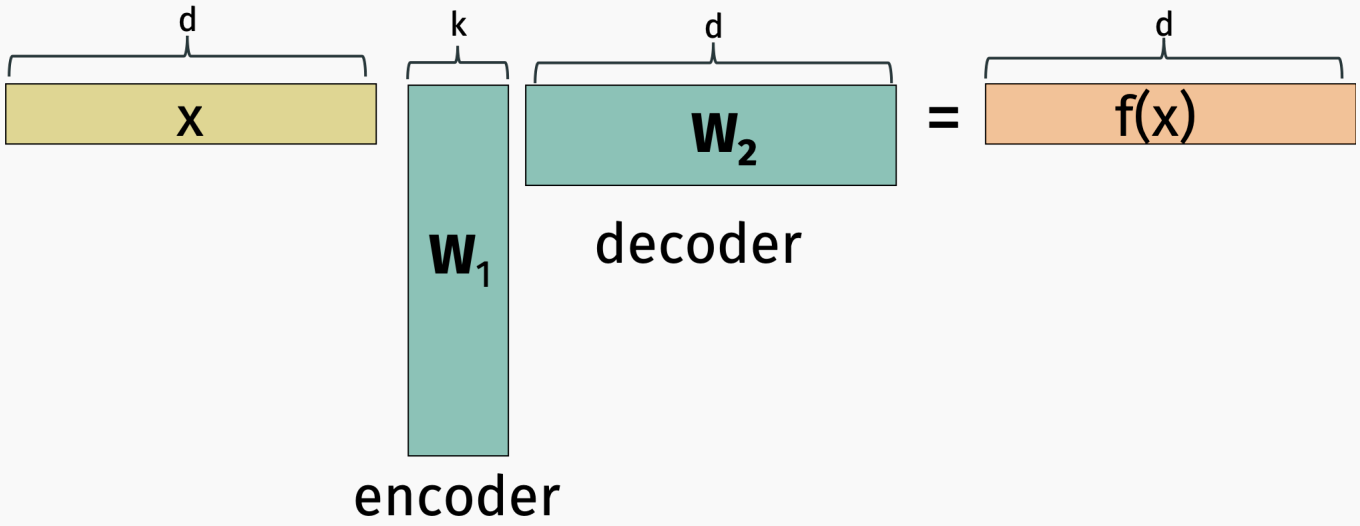
Consider the simplest possible autoencoder:



- One hidden layer. No non-linearity. No biases.
- Latent space of dimension k.
- Weight matrices are $\mathbf{W}_1 \in \mathbb{R}^{d \times k}$ and $\mathbf{W}_2 \in \mathbb{R}^{k \times d}$.

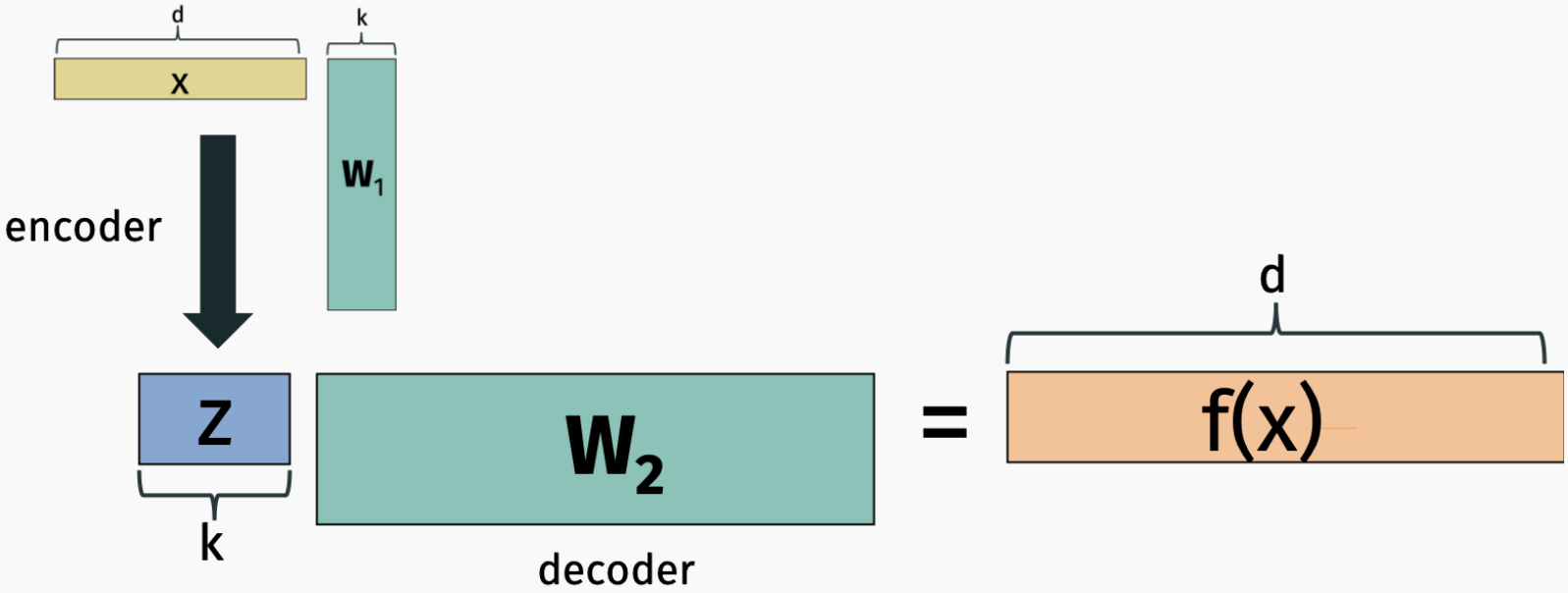
Principal Component Analysis (PCA)

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



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

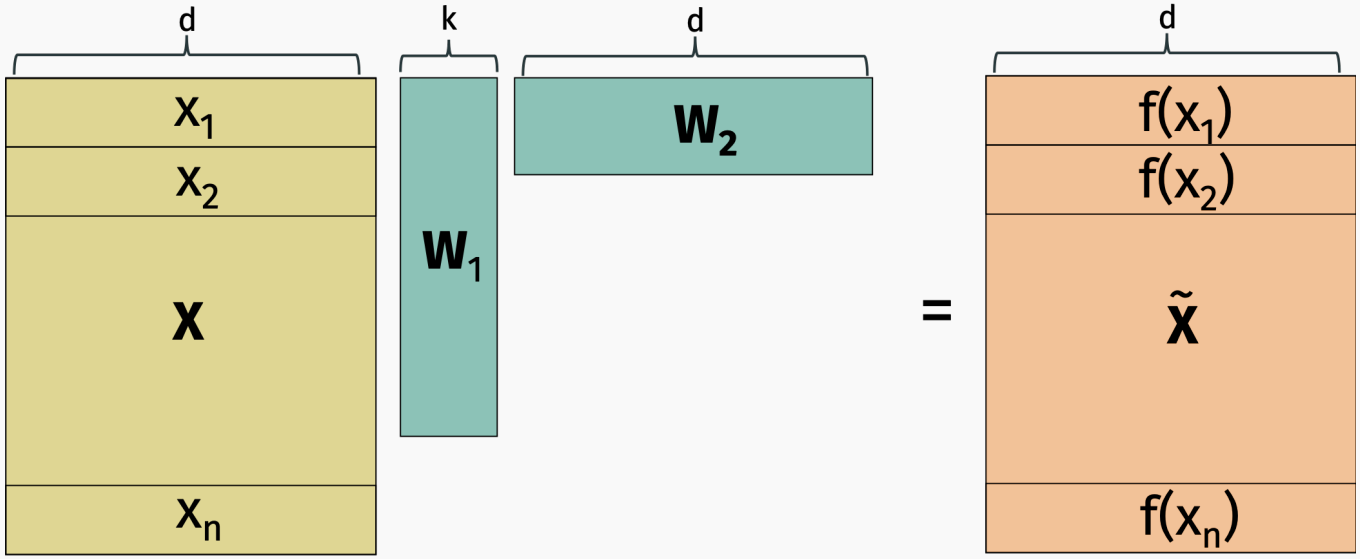
Principal Component Analysis (PCA)



Encoder: $e(\mathbf{x}) = \mathbf{x}^T \mathbf{W}_1$. **Decoder:** $d(\mathbf{z}) = \mathbf{z}\mathbf{W}_2$

Principal Component Analysis (PCA)

Given training data set $\mathbf{x}_1, \ldots, \mathbf{x}_n$, let $\mathbf{X}$ denote our data matrix.
Let $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$.



Frobenius norm

Natural squared autoencoder loss: Minimize $L(\mathbf{X}, \tilde{\mathbf{X}})$ where:

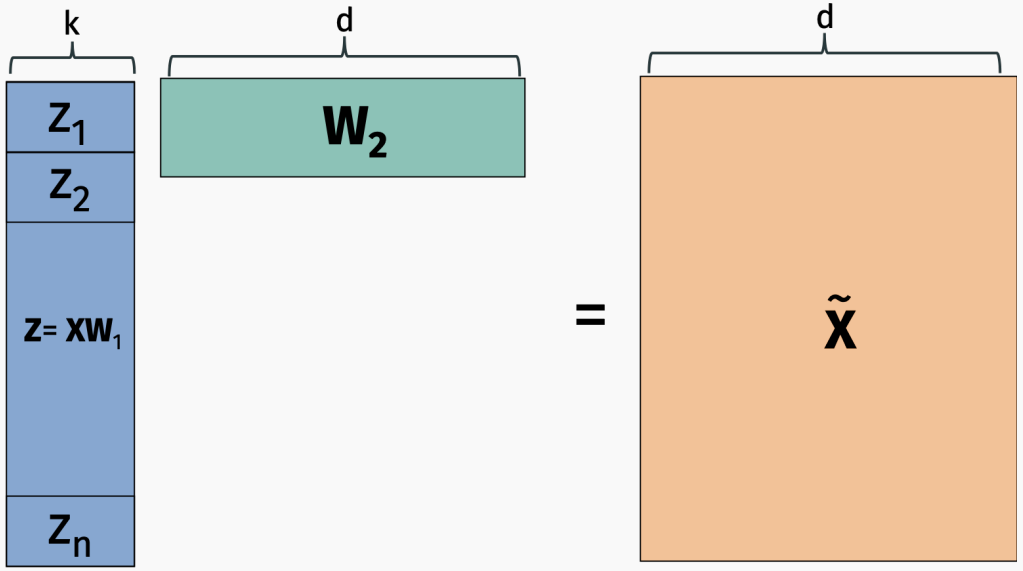
$$\begin{aligned} L(\mathbf{X}, \tilde{\mathbf{X}}) &= \sum_{i=1}^{n} \|\mathbf{x}_i - f(\mathbf{x}_i)\|_2^2 \\ &= \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{x}_i[j] - f(\mathbf{x}_i)[j])^2 \\ &= \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 \end{aligned}$$

Goal: Find $\mathbf{W}_1, \mathbf{W}_2$ to minimize the Frobenius norm loss $\|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 = \|\mathbf{X} - \mathbf{X}\mathbf{W}_1\mathbf{W}_2\|_F^2$ (sum of squared entries).

Low-rank approximation

Rank in linear algebra:

- The columns of a matrix with column rank k can all be written as linear combinations of just k columns.
- The rows of a matrix with row rank k can all be written as linear combinations of k rows.
- Column rank $=$ row rank $=$ **rank**.



$\tilde{\mathbf{X}}$ is a **low-rank matrix**. It only has rank k for $k \ll d$.

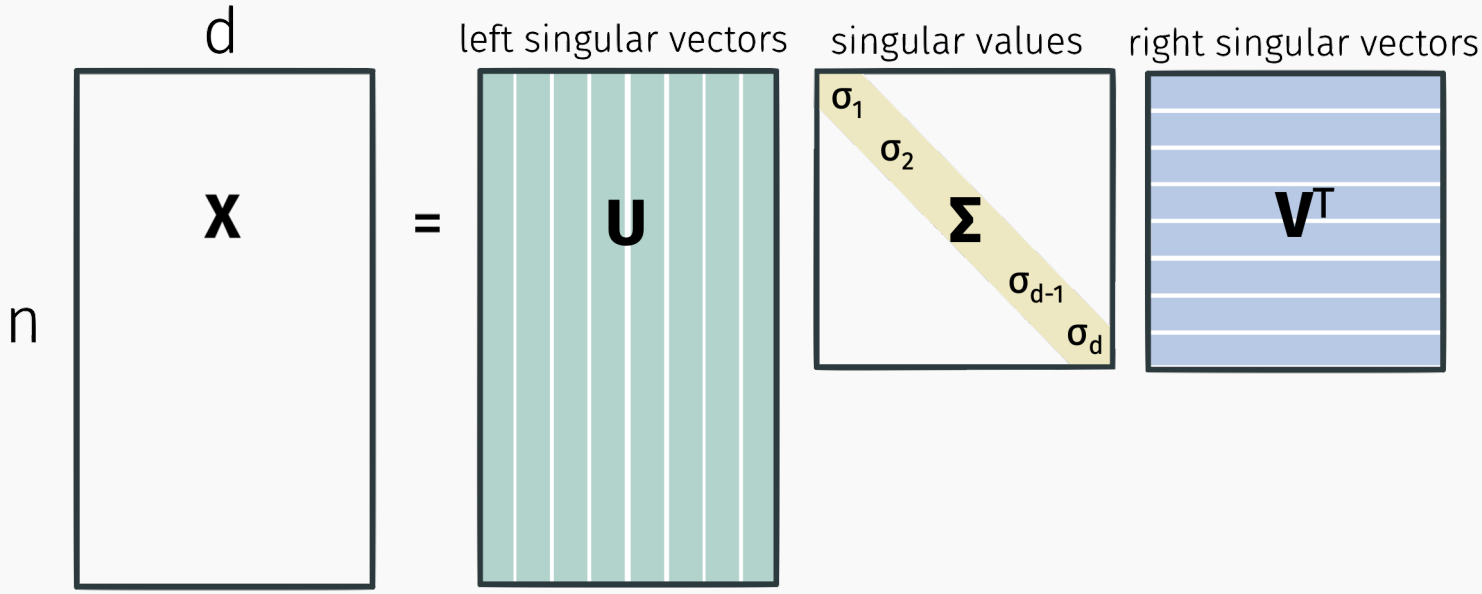
Low-rank approximation

Principal component analysis is the task of finding $\mathbf{W}_1$, $\mathbf{W}_2$, which amounts to finding a rank k matrix $\tilde{\mathbf{X}}$ which approximates the data matrix $\mathbf{X}$ as closely as possible.

Finding the best $\mathbf{W}_1$ and $\mathbf{W}_2$ is a non-convex problem. We could try running an iterative method like gradient descent anyway. But there is also a direct algorithm!

Singular value decomposition

Any matrix $\mathbf{X}$ can be written:



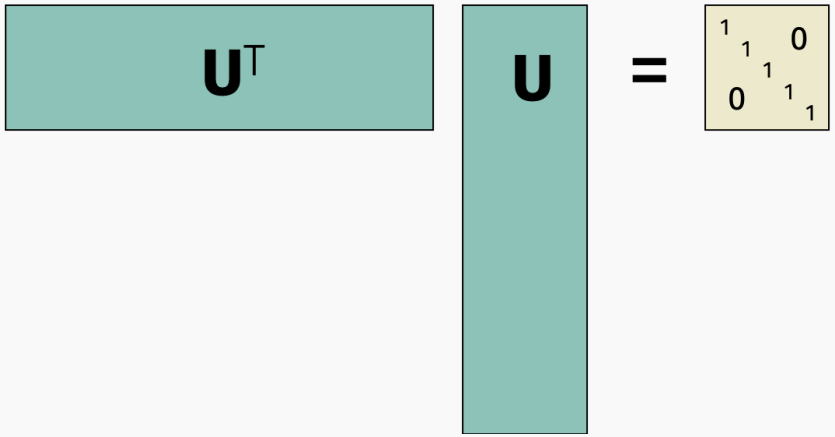
Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices.

This is called the **singular value decomposition.**

Can be computed in $O(nd^2)$ time (faster with approximation algos).

Orthogonal matrices

Let $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathbb{R}^n$ denote the columns of $\mathbf{U}$. I.e. the left singular vectors of $\mathbf{X}$.

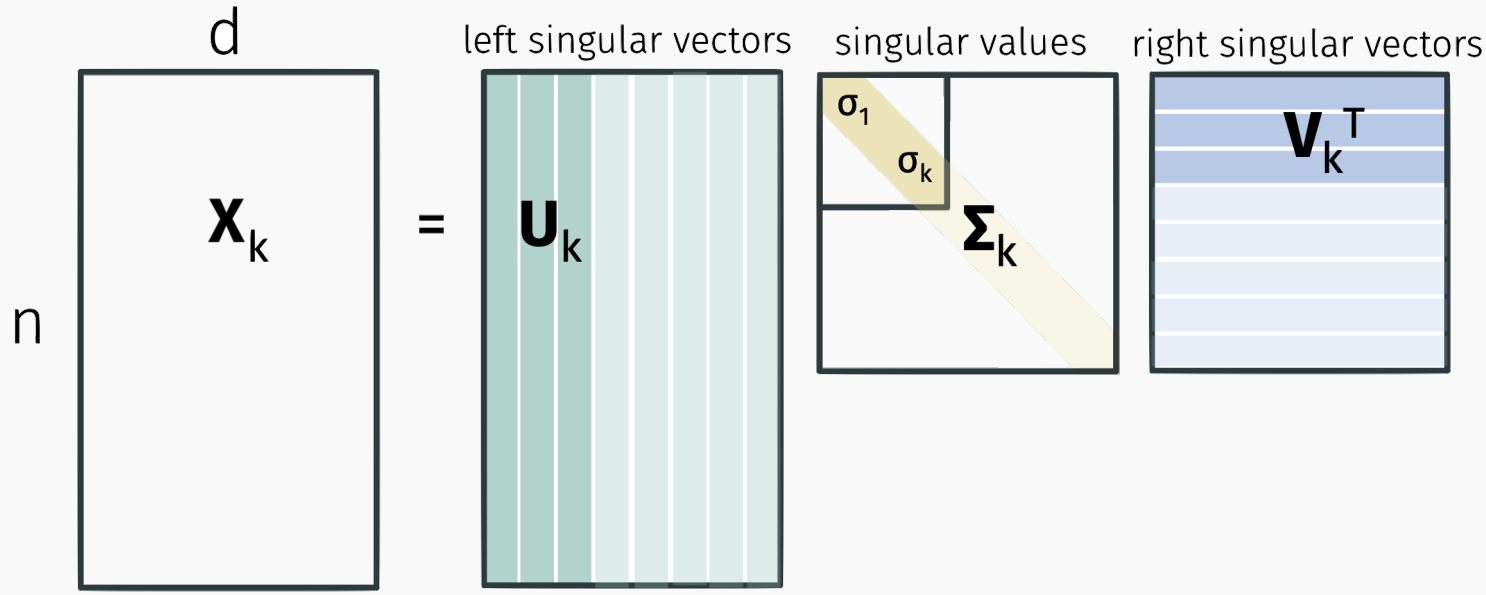


$$\|\mathbf{u}_i\|_2^2 =$$

$$\mathbf{u}_i^T \mathbf{u}_j =$$

Singular value decomposition

Can read off optimal low-rank approximations from the SVD:



Eckart–Young–Mirsky Theorem: For any $k \leq d$,
$\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

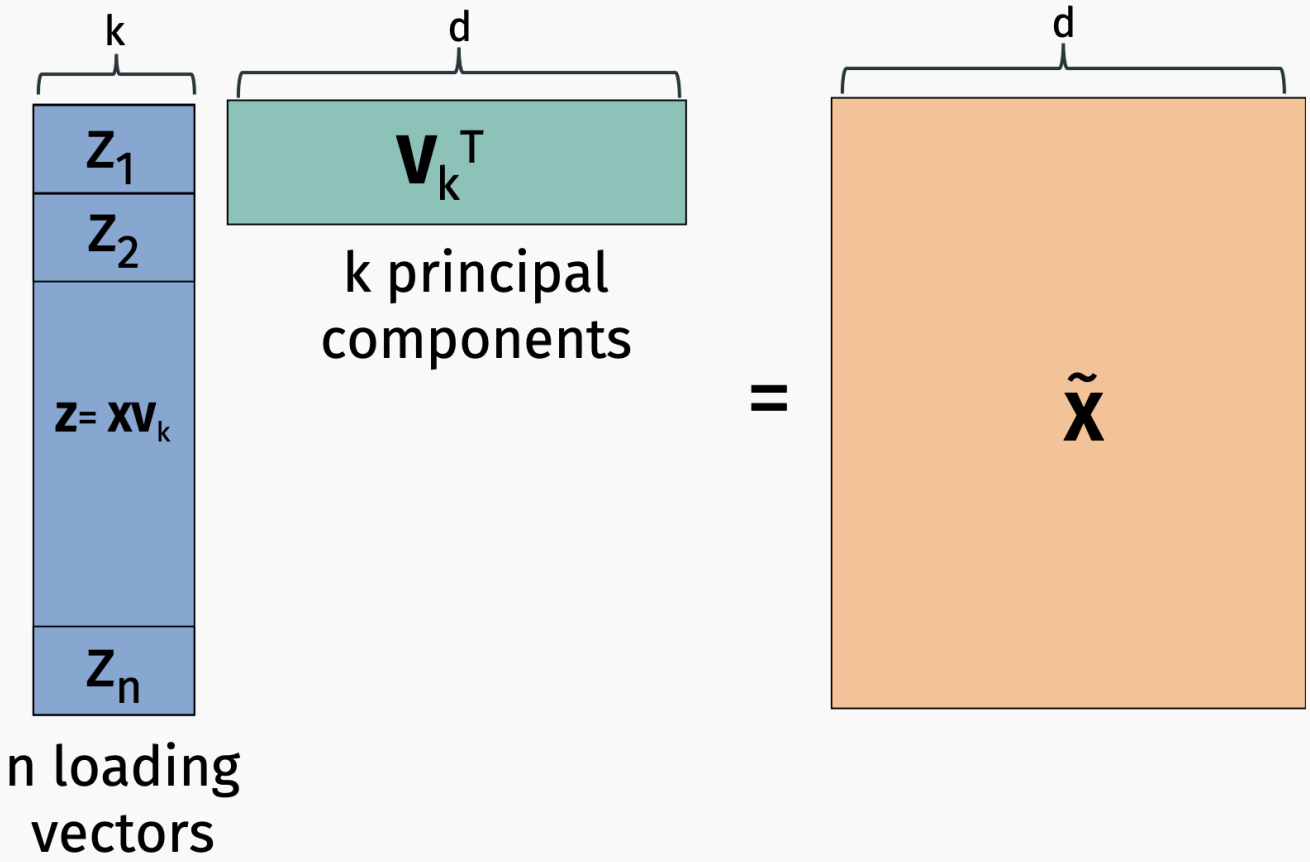
$$\mathbf{X}_k = \underset{\tilde{\mathbf{X}} \text{ with rank} \leq k}{\arg\min} \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2.$$

Singular value decomposition

Claim: $\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T = \mathbf{X}\mathbf{V}_k\mathbf{V}_k^T$.

So for a model with k hidden variables, we obtain an optimal autoencoder by setting $\mathbf{W}_1 = \mathbf{V}_k$, $\mathbf{W}_2 = \mathbf{V}_k^T$. $f(\mathbf{x}) = \mathbf{x}\mathbf{V}_k\mathbf{V}_k^T$.

Principal Component Analysis (PCA)



Usually $\mathbf{x}$'s columns (features) are mean centered and normalized to variance 1 before computing principal components.

Singular value decomposition

Computing the SVD.

- Full SVD:

  `U,S,V = scipy.linalg.svd(X)`.

  Runs in $O(nd^2)$ time.

- Just the top k components:

  `U,S,V = scipy.sparse.linalg.svds(X, k)`.

  Runs in roughly $O(ndk)$ time.