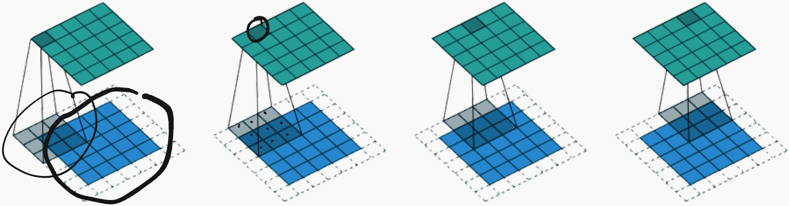
CS-GY 6923: Lecture 12
Finish Convolutional Networks, Adversarial Examples, Autoencoders

NYU Tandon School of Engineering, Prof. Christopher Musco

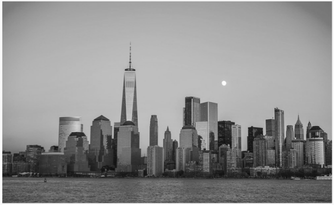
RECALL FROM LAST LECTURE

Common way of processing images, time series, audio, etc. is via convolution with a filter:



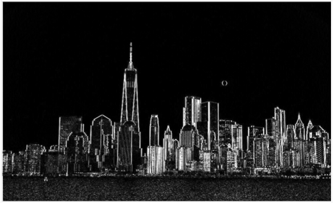
Can perform operations like smoothing, template matching, edge detection, etc.

EDGE DETECTION

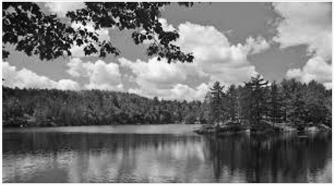


$\mathbf{I}_C$

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

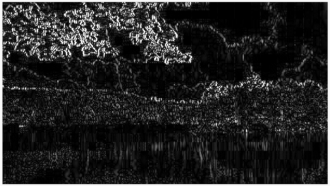


$\mathbf{E}_C$



$\mathbf{I}_L$

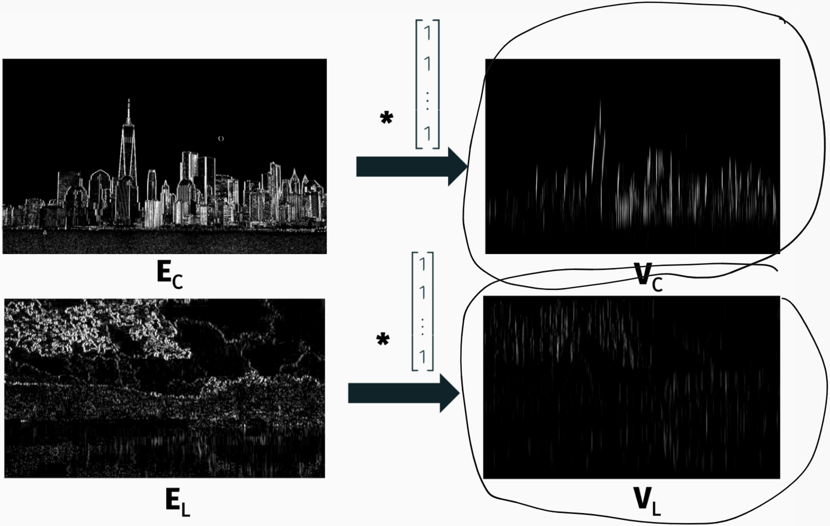
$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \longrightarrow$$



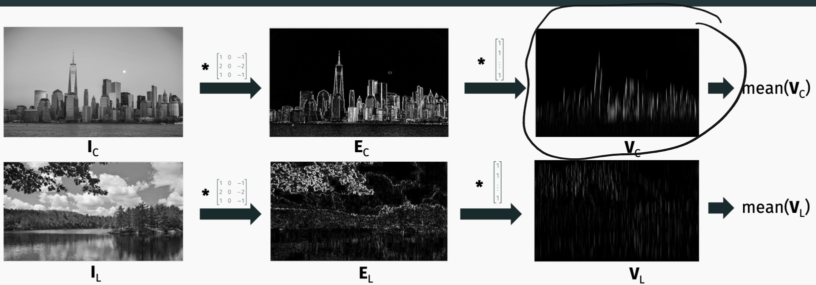
$\mathbf{E}_L$

EDGE DETECTION + PATTERN MATCHING

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


$*\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$
$*\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$
$\mathbf{E}_C$
$\mathbf{V}_C$
$\mathbf{E}_L$
$\mathbf{V}_L$

HIERARCHICAL CONVOLUTIONAL FEATURES



$\mathbf{I}_C$
$* \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$
$\mathbf{E}_C$
$* \begin{bmatrix}1\\ 1\\ \vdots\\ 1\end{bmatrix}$
$\mathbf{V}_C$
mean($\mathbf{V}_C$)
$\mathbf{I}_L$
$* \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$
$\mathbf{E}_L$
$* \begin{bmatrix}1\\ 1\\ \vdots\\ 1\end{bmatrix}$
$\mathbf{V}_L$
mean($\mathbf{V}_L$)

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

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

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

HIERARCHICAL CONVOLUTIONAL FEATURES

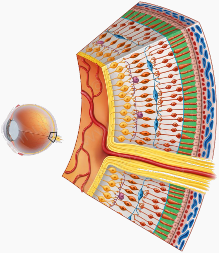
Hierarchical combinations of simple convolution filters are very powerful for understanding images.

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**.



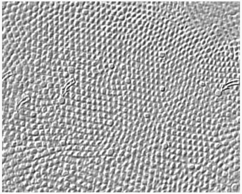
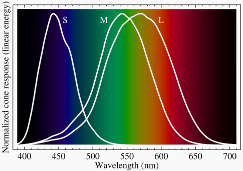


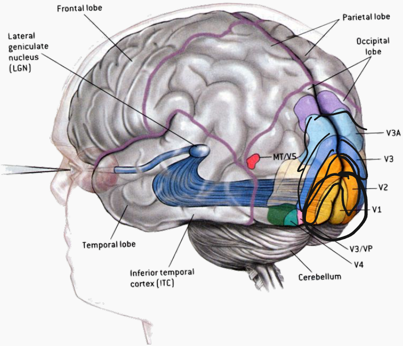
Fig. 13. Tangential section through the human fovea. Larger cones (arrows) are blue cones. *From Ahnelt et al. 1987.*

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.

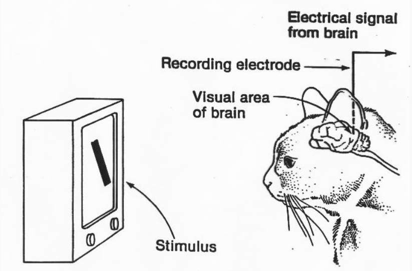


What sort of processing happens in the primary cortex?

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`.

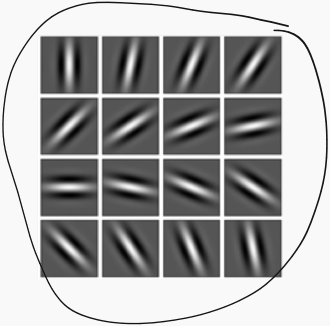
”What the Frog’s Eye Tells the Frog’s Brain”, Lettvin et al. 1959. Found explicit edge detection circuits in a frogs visual cortex.

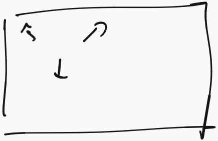
EXPLICIT FEATURE ENGINEERING

State of the art until 13 years ago:

Histogram of Oriented Features

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

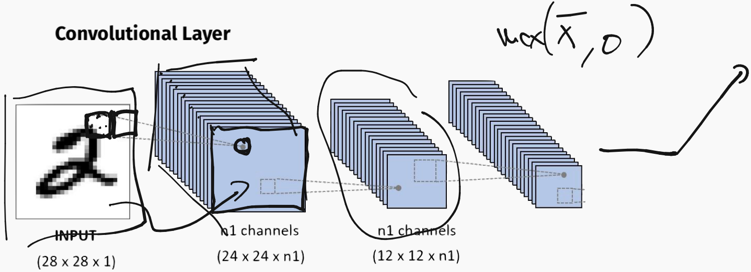




CONVOLUTIONAL NEURAL NETWORKS

Neural network approach: Learn the parameters of the convolution filters based on training data. $w_1, w_2 \ldots w_q$

$\max(\bar{x}, 0)$



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

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

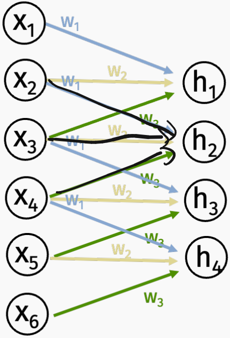
Each output in the tensor is processed with a non-linearity. Most commonly a Rectified Linear Unity (ReLU): $x = \max(\bar{x}, 0)$.

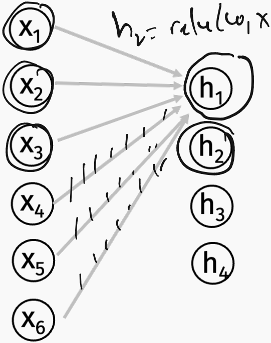
WEIGHT SHARING

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.

$h_1 = \text{relu}(w_1 x_1 + w_2 x_2 + w_3 x_3)$

$h_2 = \text{relu}(w_1 x_2 + w_2 x_3 + w_3 x_4)$





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

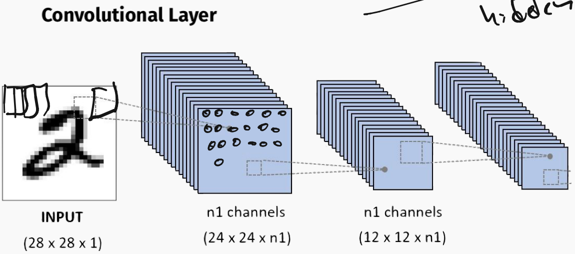
CONVOLUTIONAL NEURAL NETWORKS

$25q$ 784

For a 28 × 28 image like MNIST, a fully connected layer that extracts the same features as q, 5 × 5 filters would require $(28 \cdot 28 \cdot 24 \cdot 24) \cdot q = 451,584 \cdot q$ parameters. Compare to $25q$.

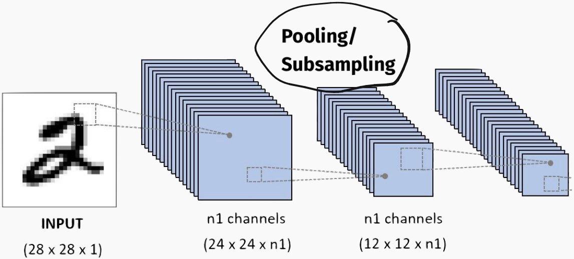
By "baking in" knowledge about what type of features matter, we greatly simplify the network.

$24 \cdot 24 \cdot q$ total hidden neurons



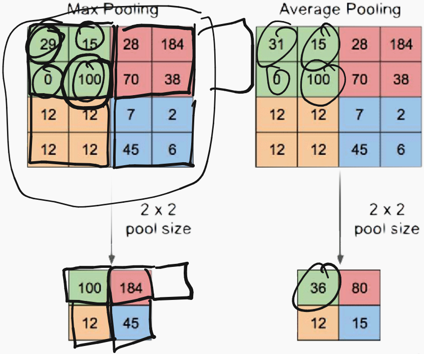
POOLING AND DOWNSAMPLING

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

2 x 2
pool size

100	184
12	45

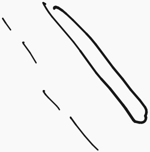
Average Pooling

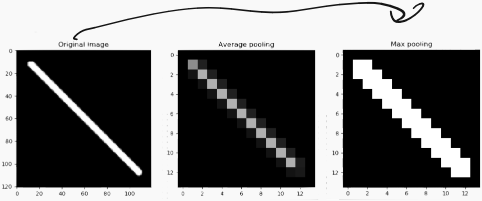
31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2
pool size

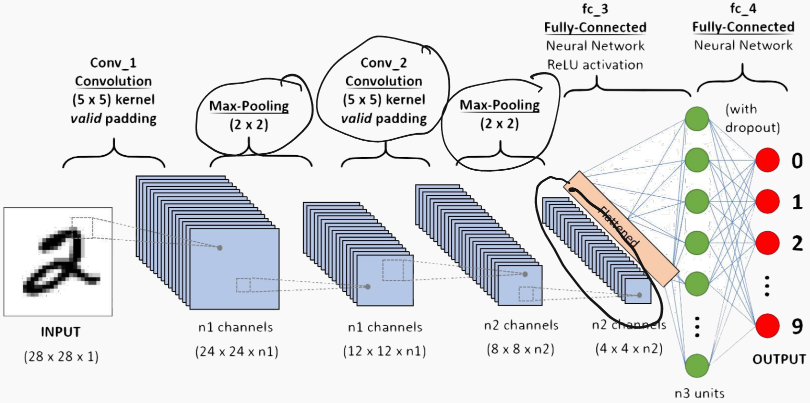
36	80
12	15

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





OVERALL NETWORK ARCHITECTURE

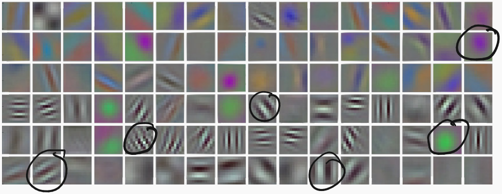


Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

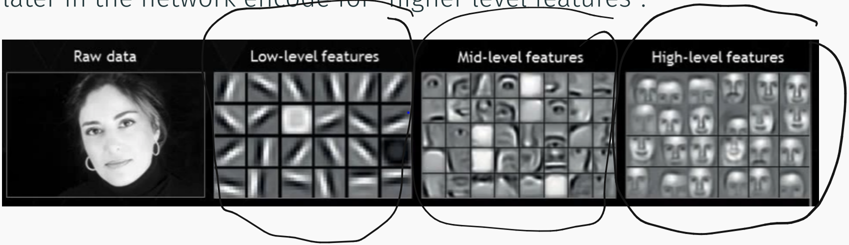
UNDERSTANDING LAYERS

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

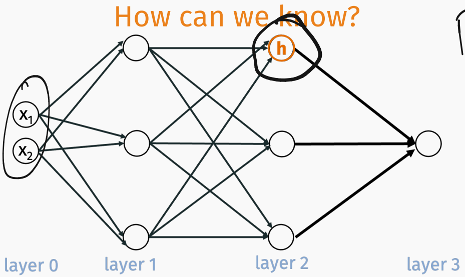
Lots of edge detectors in the first layer!



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

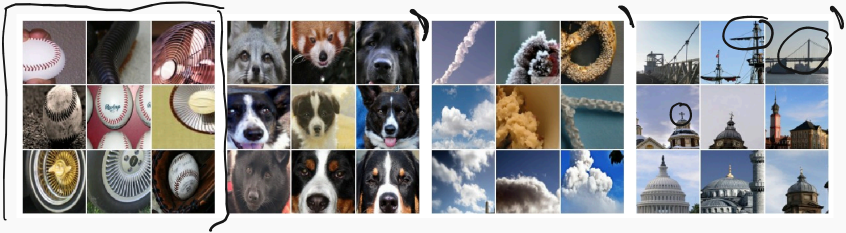


UNDERSTANDING LAYERS

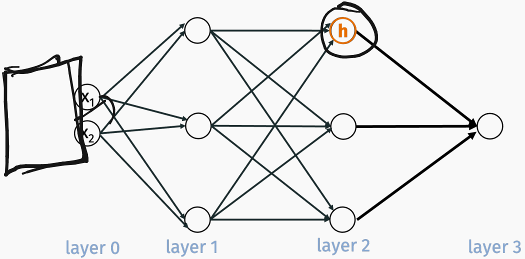


Distill

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



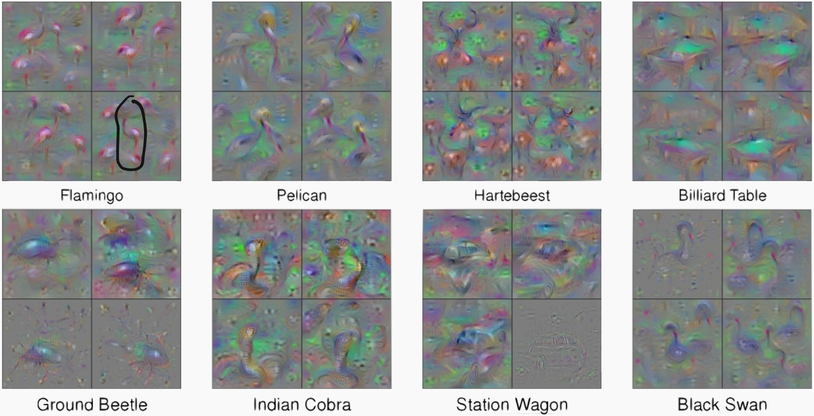
How can we know?



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

UNDERSTANDING LAYERS

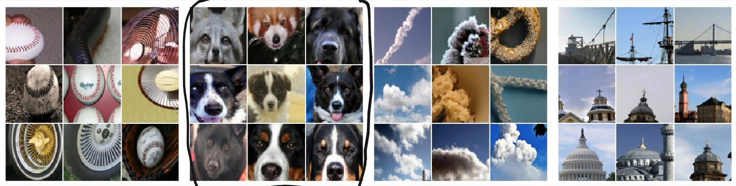
Early work had some interesting results.

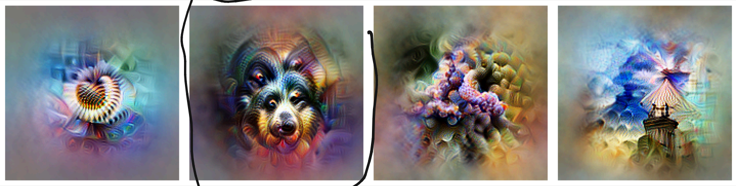


“Understanding Neural Networks Through Deep Visualization”, Yosinski, Clune, Nguyen, Fuchs, Lipson.

UNDERSTANDING LAYERS

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

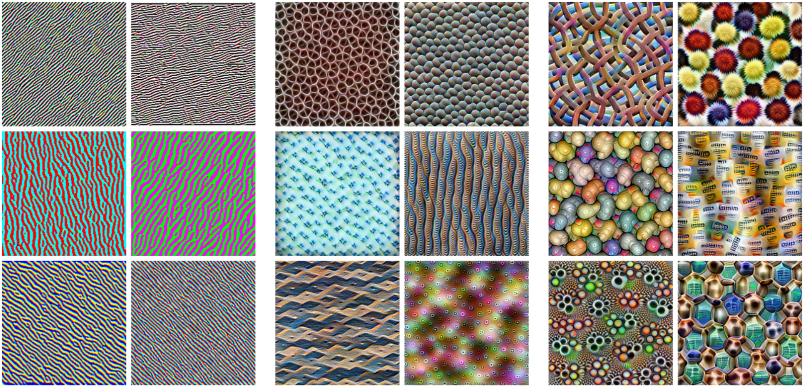




If you are interested in learning more on these techniques, there is a great Distill article at: `https://distill.pub/2017/feature-visualization/`.

UNDERSTANDING LAYERS

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



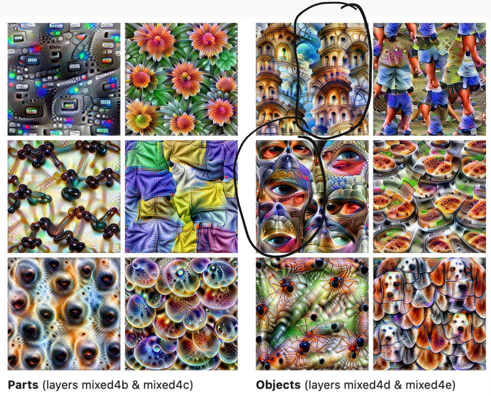
Edges (layer conv2d0)

Textures (layer mixed3a)

Patterns (layer mixed4a)

UNDERSTANDING LAYERS

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


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

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

TRICKS OF THE TRADE

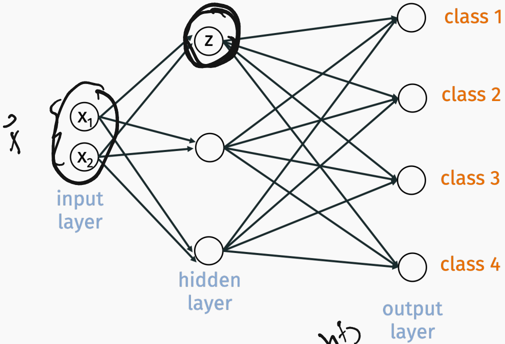
Beyond techinques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of “tricks”.

- 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 **x**,

weights

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

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

where **w**, *b*, and *s* are weights, bias, and non-linearity.

BATCH NORMALIZATION

→ z before nonlinearity $\bar{z} = w^T x + b$

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

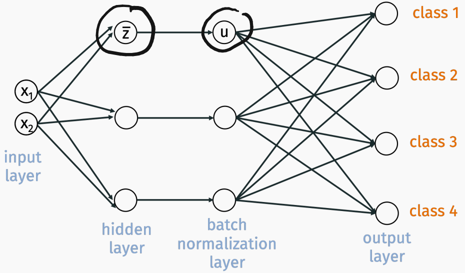
$$\underline{\mu} = \frac{1}{n}\sum_{j=1}^{n} \underline{\bar{z}(\mathbf{x}_j)}$$

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

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:



$$\underline{\bar{u}} = \frac{\bar{z} - \underline{\mu}}{\sigma}$$
$$\underline{u = s(\bar{u}).}$$

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

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

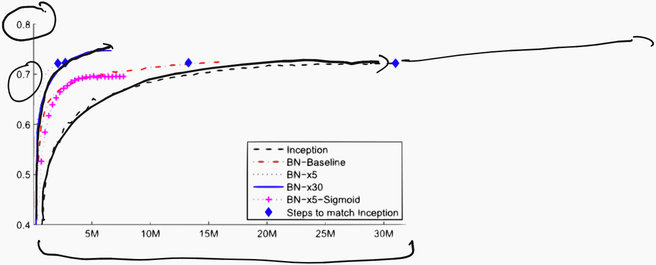
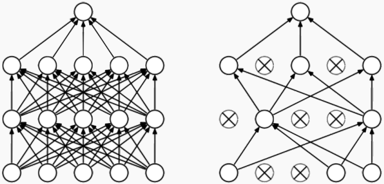


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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.

DROPOUT

Proposed in 2012: “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov:



(a) Standard Neural Net (b) After applying dropout.

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

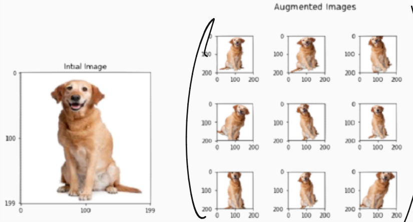
- Only used on fully connected layers.
- Simultaneously performs model regularization (model simplification) and model averaging.
- Has become less important in modern CNNs (convolutional neural nets) as the final fully connected layers become less important. But still a very helpful technique to know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

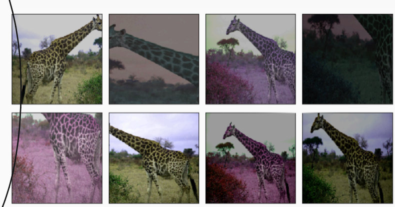
For example, will be very helpful in avoiding overfitting in the demo on convolutional nets, since we train a pretty shallow network with the last layer doing a lot of the heavy lifting.

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.

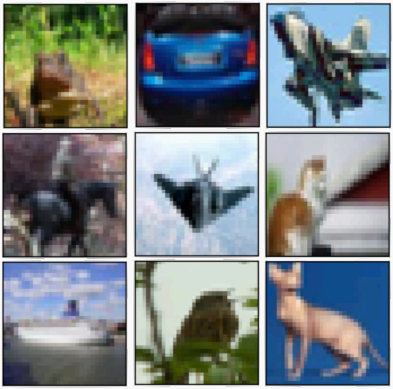




Take training images and randomly shift, flip, rotate, skew, darken, lighten, shift colors, etc. to create new training images. **Final classifier will be more robust to these transformations.**

DEEP LEARNING TRICKS

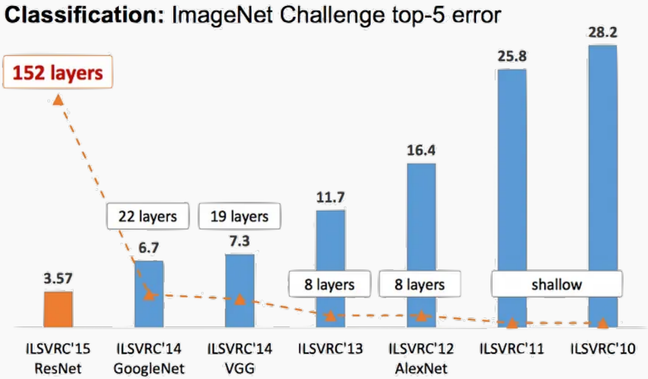
Try these techinques out in `demo_cnn_classifier.ipynb` on CIFAR-10 dataset.



airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck

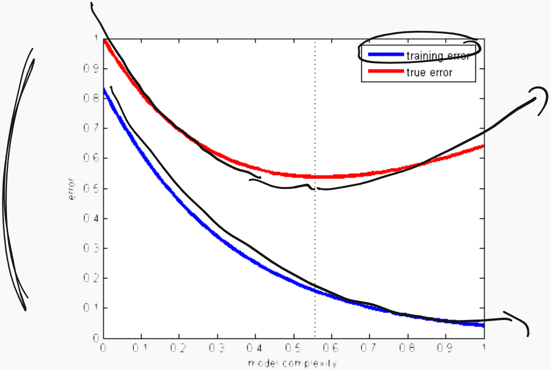
DEEPER AND DEEPER, BIGGER AND BIGGER

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



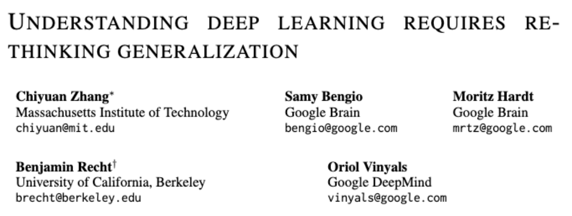
GENERALIZATION FOR NEURAL NETWORKS

Even with weight sharing, convolution, etc. modern neural networks typically have 100s of millions or billions of parameters. And we often 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 easily overfit to training data. Papers have shown that they can fit large vision data sets with random class labels to 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

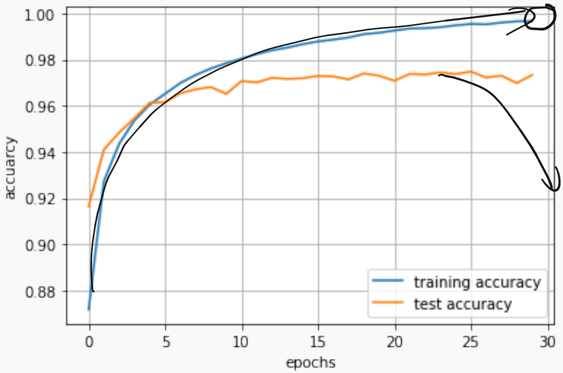
Moritz Hardt
Google Brain
mrtz@google.com

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.** For example, overfitting is common when using fully connected networks.

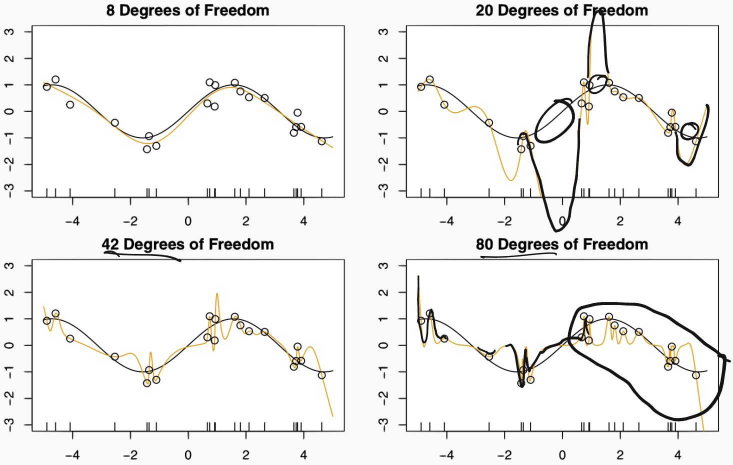
We even see this lack of overfitting for MNIST data. See `keras_demo_mnist.ipynb` that I posted on the website:



Overparameterization seems to be part of the story.

GENERALIZATION FOR NEURAL NETWORKS

Growing realization is that this phenomena doesn't only apply to neural nets – it can also be true for overparameterized polynomials.



8 Degrees of Freedom

20 Degrees of Freedom

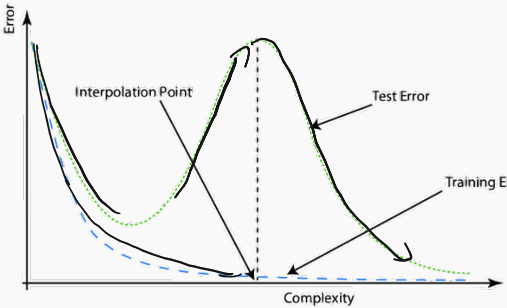
42 Degrees of Freedom

80 Degrees of Freedom

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.



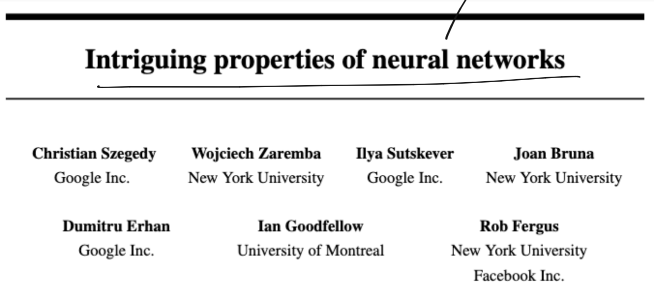
Caveat: We don’t usually see this same curve for neural networks, but maybe gives some hint about what is going on.

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?

adversarial example



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.

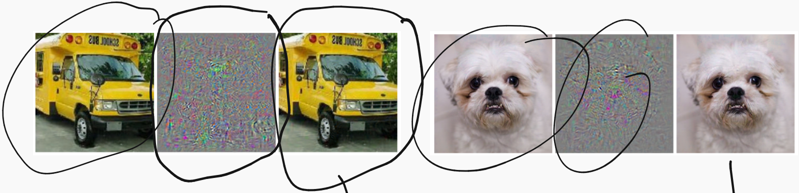
Ian Goodfellow
University of Montreal

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

$$\max \ell(\theta, x+\delta, y) + \lambda \|\delta\|_2^2$$

How to find "good" perturbations:

Fix model f_θ input $\mathbf{x}$, correct label y. Consider the loss $\ell(\theta, \mathbf{x}, \overset{\text{class}}{y})$.

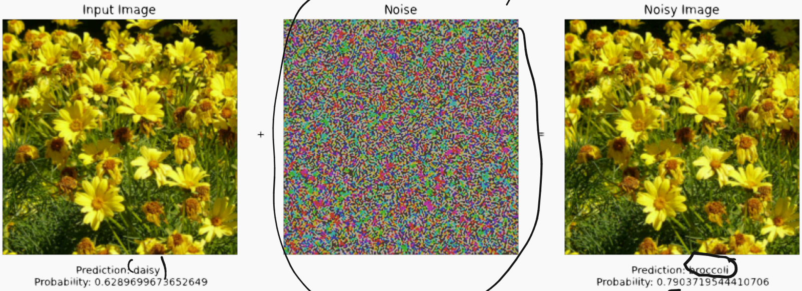
Solve the optimization problem:

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

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

We will post a lab where you can find your own adversarial examples for a model called Resnet18. The entire model + weights are available pretrained through PyTorch, so we do not need to train it ourselves.



Break until 4:05.

TRANSFER LEARNING

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 unlabeled 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

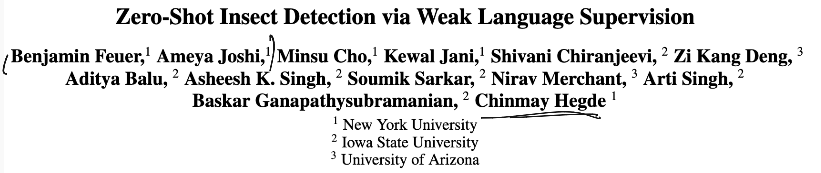
bludger

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

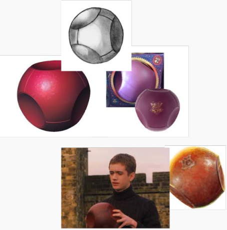
(**Benjamin Feuer,**[1] **Ameya Joshi,**[1]) **Minsu Cho,**[1] **Kewal Jani,**[1] **Shivani Chiranjeevi,** [2] **Zi Kang Deng,** [3]
Aditya Balu, [2] **Asheesh K. Singh,** [2] **Soumik Sarkar,** [2] **Nirav Merchant,** [3] **Arti Singh,** [2]
Baskar Ganapathysubramanian, [2] **Chinmay Hegde** [1]

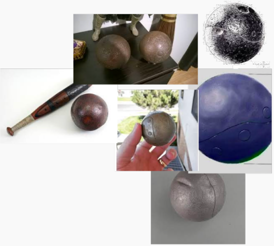
[1] New York University
[2] Iowa State University
[3] University of Arizona

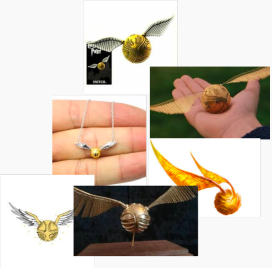
Aedes Vexans | Creatonotos Gangis | Daphnis Neril | Hypena Deceptalis | Pyralis Farinalis

ONE-SHOT LEARNING

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



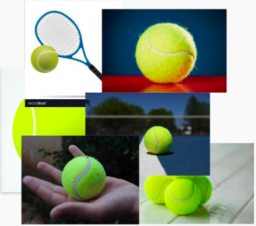




Major question in ML: How? Can we design ML algorithms which can do the same?

TRANSFER LEARNING

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



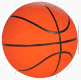
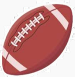
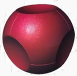
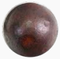
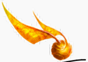
For example, we have learned from past experience that balls used in sports have consistent shapes, colors, and sizes. These features can be used to distinguish balls of different type.

FEATURE LEARNING

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

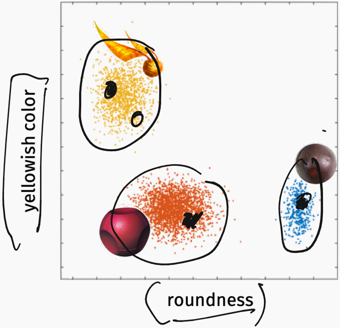
Classes

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

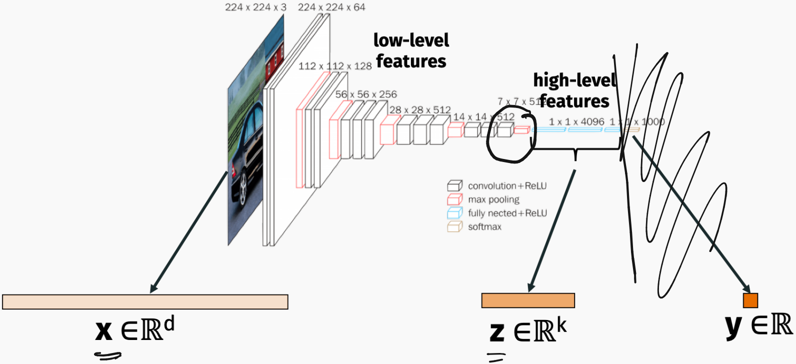
If these features are highly informative (i.e. lead to highly separable data) few training examples are needed to learn.



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 $\mathbf{z}$ 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 any 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.

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 10 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.

Next few lectures: How do we learn interesting features without access to labels?

AUTOENCODER

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

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

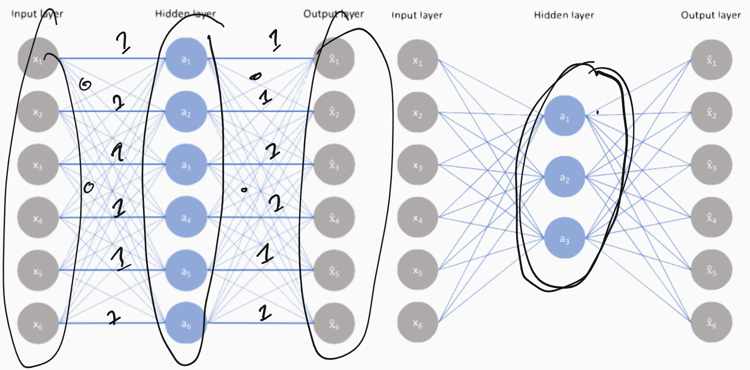
$f_{\boldsymbol{\theta}}$ is called an autoencoder. It maps input space to input space (e.g. images to images, french to french, PDE solutions to PDE solutions).

AUTOENCODER

Two examples of autoencoder architectures:

θ^*

bottleneck

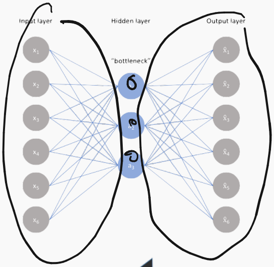

Input layer
Hidden layer
Output layer
x_1 x_2 x_3 x_4 x_5 x_6
a_1 a_2 a_3 a_4 a_5 a_6
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$
1
6
2
1
0
1
1
2
1
0
1
2
0
2
1
1
Input layer
Hidden layer
Output layer
x_1 x_2 x_3 x_4 x_5 x_6
a_1 a_2 a_3
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$

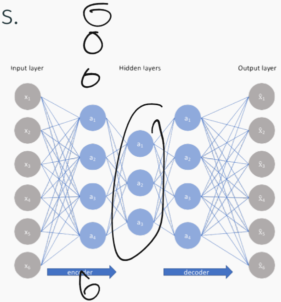
Which would lead to better feature learning?

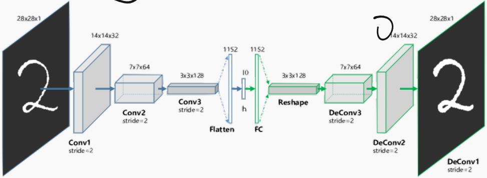
$L(\theta^*)$

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.





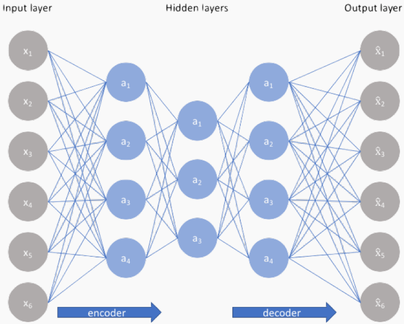


AUTOENCODER

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

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

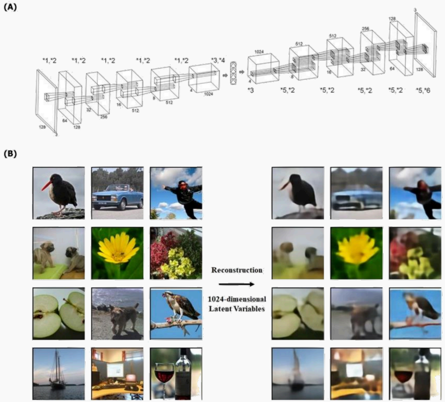
$$f(x) =$$



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]

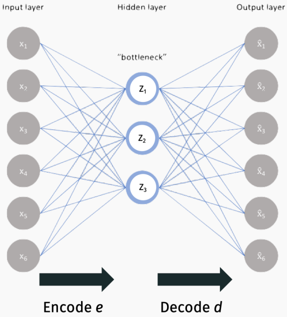
There are a lot of cool applications of autoencoders beyond feature learning!

- 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.



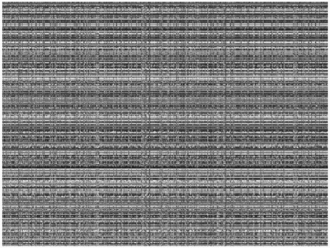
Given input $\mathbf{x}$, we can completely recover $f(\mathbf{x})$ from $\mathbf{z} = e(\mathbf{x})$. $\mathbf{z}$ typically has many fewer dimensions than $\mathbf{x}$ and for a typical image $f(\mathbf{x})$ will closely approximate $\mathbf{x}$.

AUTOENCODERS FOR IMAGE COMPRESSION

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

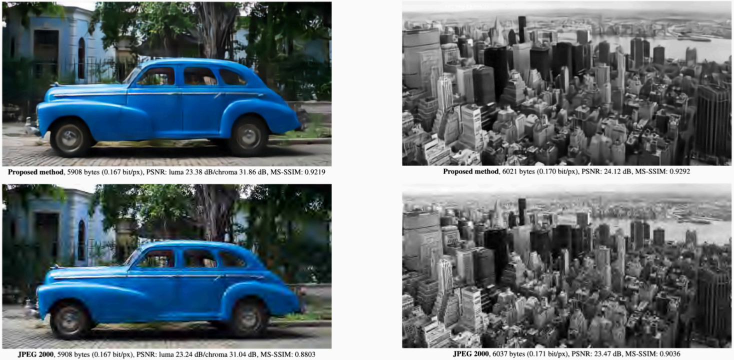
- JPEG 2000 for images
- MP3 for digital audio.
- MPEG-4 for video.

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

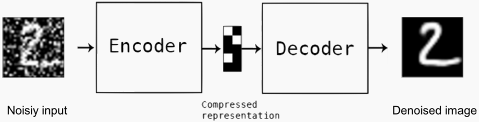


Image denoising

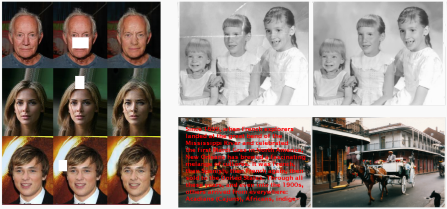
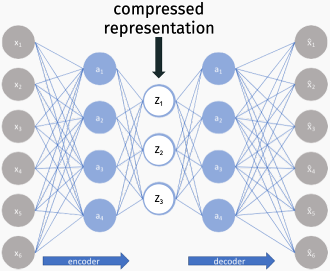

Since 1699, when French explorers
landed at the great bend of the
Mississippi River and celebrated
the first Mardi Gras in North America,
New Orleans has brewed a fascinating
melange of cultures. It was French,
then Spanish, then French again, then
sold to the United States. Through all
these years, and even into the 1900s,
others arrived from everywhere:
Acadians (Cajuns), Africans, indige-

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

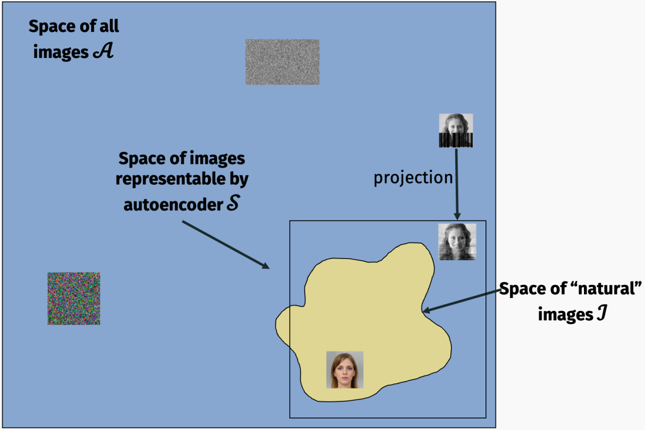
Why does this work?



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

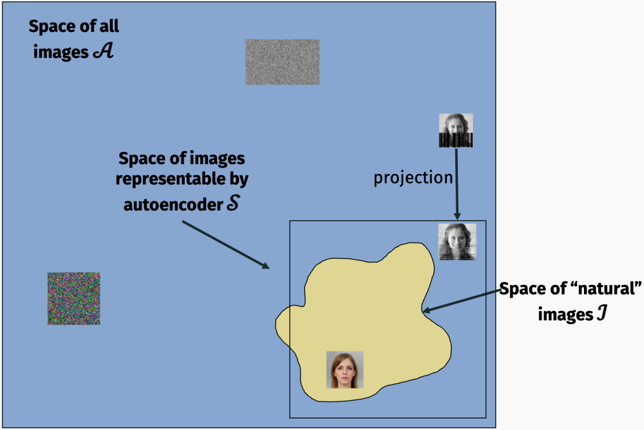
If $\mathbf{z}$ holds k values in $0, .1, .2, \ldots, 1$, how many unique images $\mathbf{w}$ can be output by the autoencoder function f?

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS


Space of all images $\mathcal{A}$
Space of images representable by autoencoder $\mathcal{S}$
projection
Space of "natural" images $\mathcal{I}$

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

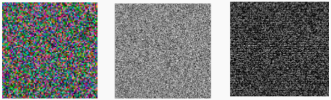

Space of all images $\mathcal{A}$
Space of images representable by autoencoder $\mathcal{S}$
projection
Space of "natural" images $\mathcal{I}$

$f(\mathbf{x}) = d(e(\mathbf{x}))$ 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 $\mathcal{A}$.

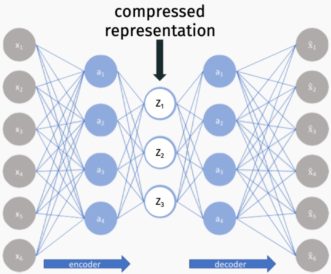


- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.

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

AUTOENCODERS FOR DATA GENERATION

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

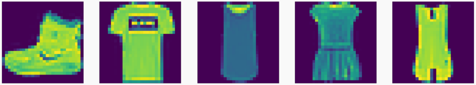


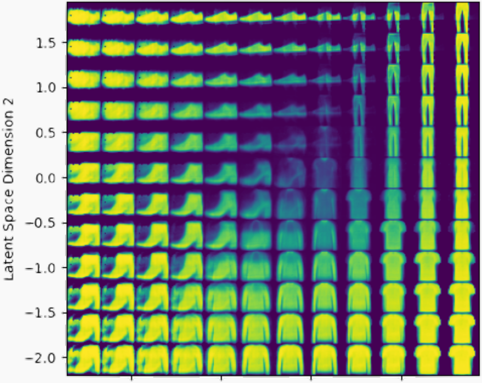
Randomly select code **z**, then set $\mathbf{x} = d(\mathbf{z})$.[2]

[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

We will upload a demo on autoencoder based image generation for the ”Fashion MNIST” data set:





PRINCIPAL COMPONENT ANALYSIS

PRINCIPAL COMPONENT ANALYSIS

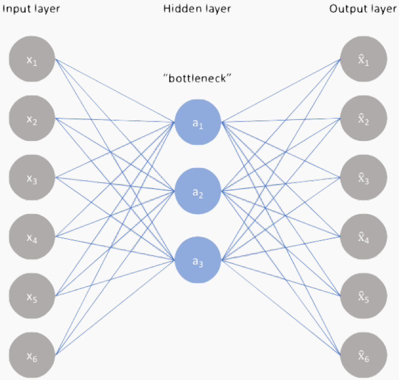
Deeper dive into understanding a simple, but powerful autoencoder architecture. Specifically we will view principal component analysis (PCA) as a type of autoencoder.

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

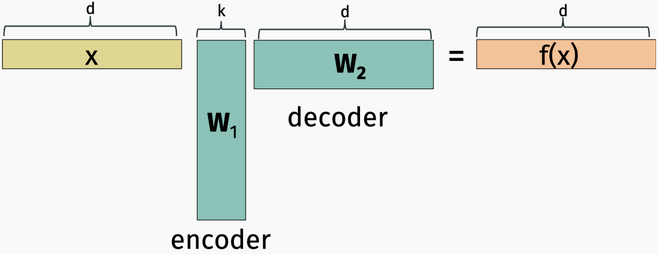
Consider the simplest possible autoencoder:



- 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 $\mathbf{W}_2 \in \mathbb{R}^{k \times d}$.

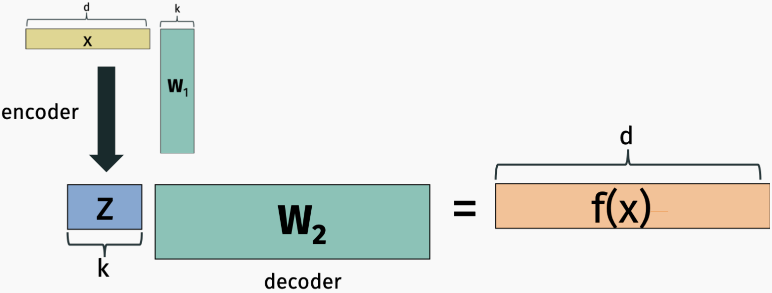
PRINCIPAL COMPONENT ANALYSIS

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



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

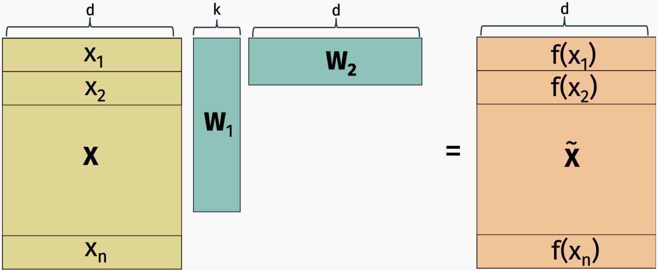
PRINCIPAL COMPONENT ANALYSIS



Encoder: $e(x) = x^T W_1$.   Decoder: $d(z) = zW_2$

PRINCIPAL COMPONENT ANALYSIS

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:

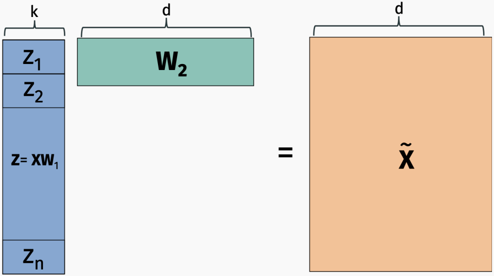
$$\begin{aligned} L(\mathbf{X}, \tilde{\mathbf{X}}) &= \sum_{i=1}^{n} \|\mathbf{x}_i - f(\mathbf{x}_i)\|_2^2 \\ &= \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{x}_i[j] - f(\mathbf{x}_i)[j])^2 \\ &= \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 \end{aligned}$$

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**.



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

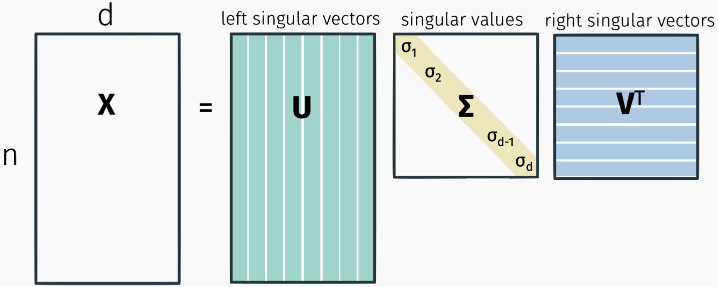
LOW-RANK APPROXIMATION

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 **X** can be written:

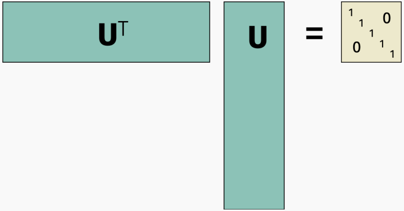


Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. **U** and **V** are orthogonal matrices.

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

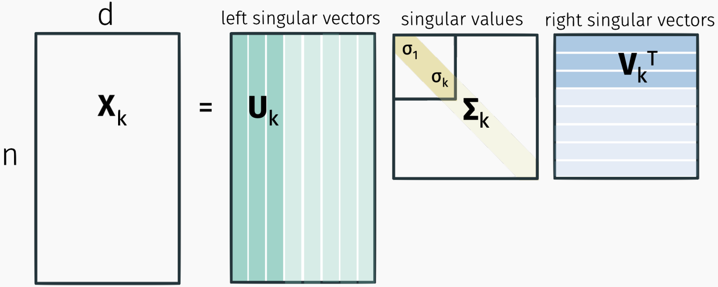
Can be computed in $O(nd^2)$ time (faster with approximation algos).

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 $\mathbf{X}$.



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

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 $\mathbf{X}$:

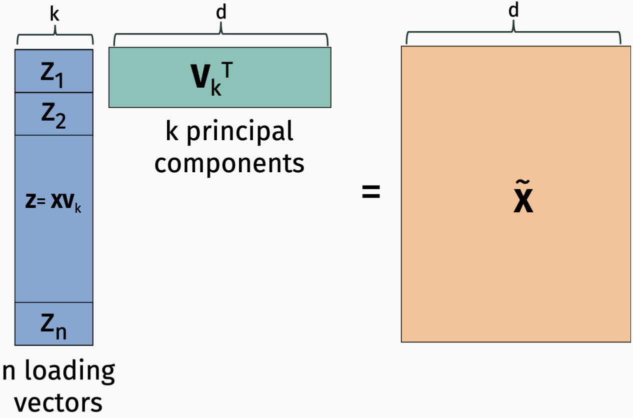
$$\mathbf{X}_k = \underset{\tilde{\mathbf{X}} \text{ with rank } \leq k}{\arg\min} \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2.$$

SINGULAR VALUE DECOMPOSITION

Claim: $X_k = U_k \Sigma_k V_k^T = XV_kV_k^T$.

So for a model with k hidden variables, we obtain an optimal autoencoder by setting $W_1 = V_k$, $W_2 = V_k^T$. $f(x) = x^T V_k V_k^T$.

PRINCIPAL COMPONENT ANALYSIS



Usually 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.

CONNECTION TO EIGENDECOMPOSITION

Recall that for a matrix $\mathbf{M} \in \mathbb{R}^{p \times p}$, $\mathbf{q}$ is an eigenvector of $\mathbf{M}$ if $\lambda \mathbf{q} = \mathbf{M}\mathbf{q}$ for any scalar λ.

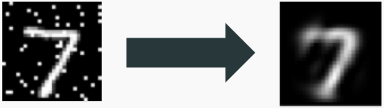
- $\mathbf{U}$'s columns (the left singular vectors) are the orthonormal eigenvectors of $\mathbf{X}\mathbf{X}^T$.
- $\mathbf{V}$'s columns (the right singular vectors) are the orthonormal eigenvectors of $\mathbf{X}^T\mathbf{X}$.
- $\sigma_i^2 = \lambda_i(\mathbf{X}\mathbf{X}^T) = \lambda_i(\mathbf{X}^T\mathbf{X})$

Exercise: Verify this directly. This means you can use any eigensolver for computing the SVD.

PCA APPLICATIONS

Like any autoencoder, PCA can be used for:

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

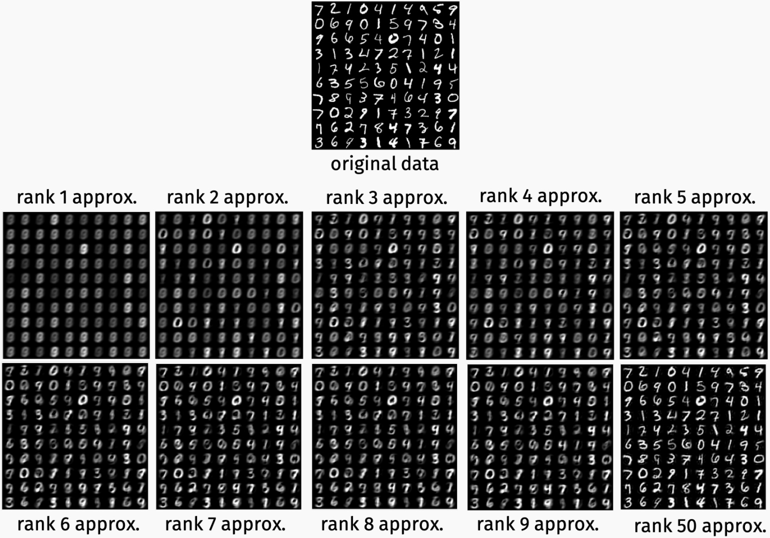


denoising

synthetic data generation

LOW-RANK APPROXIMATION

The larger we set *k*, the better approximation we get.



original data

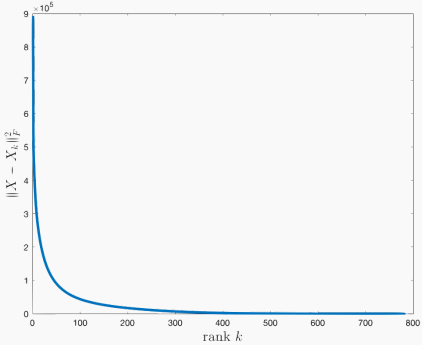
rank 1 approx. rank 2 approx. rank 3 approx. rank 4 approx. rank 5 approx.

rank 6 approx. rank 7 approx. rank 8 approx. rank 9 approx. rank 50 approx.

LOW RANK APPROXIMATION

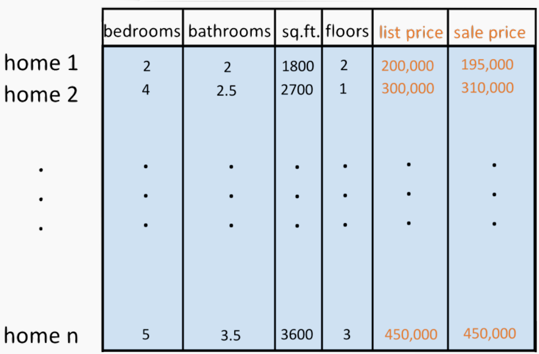
Error vs. k is dictated by X's singular values. The singular values are often called the **spectrum** of X.

$$\|\mathbf{X} - \mathbf{X}_k\|_F^2 = \sum_{i=k}^{d} \sigma_i^2.$$



COLUMN REDUNDANCY

Colinearity of data features leads to an approximately low-rank data matrix.



	bedrooms	bathrooms	sq.ft.	floors	list price	sale price
home 1	2	2	1800	2	200,000	195,000
home 2	4	2.5	2700	1	300,000	310,000
⋮	⋮	⋮	⋮	⋮	⋮	⋮
home n	5	3.5	3600	3	450,000	450,000

sale price $\approx 1.05 \cdot$ list price.

property tax $\approx .01 \cdot$ list price.

COLUMN REDUNDANCY

Sometimes these relationships are simple, other times more complex. But as long as there exists linear relationships between features, we will have a lower rank matrix.

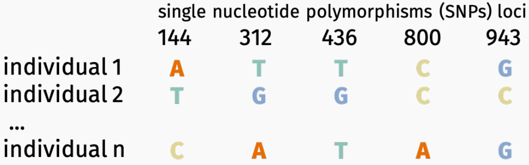
$$\text{yard size} \approx \text{lot size} - \frac{1}{2} \cdot \text{square footage}.$$

$$\begin{aligned}\text{cumulative GPA} \approx & \frac{1}{4} \cdot \text{year 1 GPA} + \frac{1}{4} \cdot \text{year 2 GPA} \\ & + \frac{1}{4} \cdot \text{year 3 GPA} + \frac{1}{4} \cdot \text{year 4 GPA}.\end{aligned}$$

LOW-RANK INTUITION

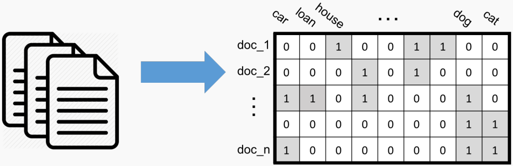
Two other examples of data with good low-rank approximations:

1. Genetic data:



	single nucleotide polymorphisms (SNPs) loci				
	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

2. “Term-document” matrix with bag-of-words data:



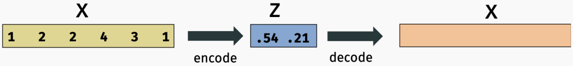
	car	loan	house		. . .			dog	cat
doc_1	0	0	1	0	0	1	1	0	0
doc_2	0	0	0	1	0	1	0	0	0
⋮	1	1	0	1	0	0	0	1	0
	0	0	0	0	0	0	0	1	1
doc_n	1	0	0	0	0	0	0	1	1

EXAMPLES OF LOW-RANK STRUCTURE

SNPs matrices tend to be very low-rank.

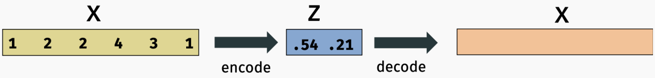
	single nucleotide polymorphisms (SNPs) loci				
	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

Most of the information in **x** is explained by just a few **latent variable**.



EXAMPLES OF LOW-RANK STRUCTURE

“Genes Mirror Geography Within Europe” – Nature, 2008.



In data collected from European populations, latent variables capture information about geography.

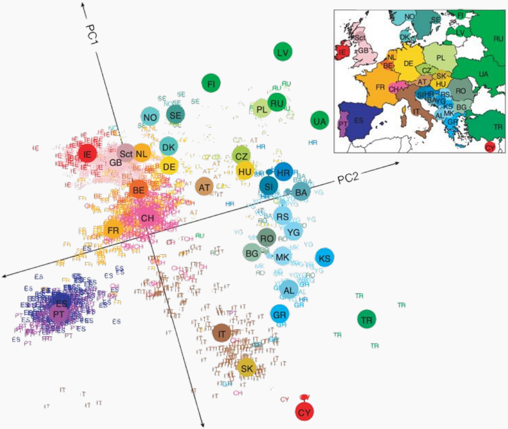
z[1] ≈ relative north-south position of birth place

z[2] ≈ relative east-west position of birth place

Individuals born in similar places tend to have similar genes.

PCA FOR DATA VISUALIZATION

“Genes Mirror Geography Within Europe” – Nature, 2008.



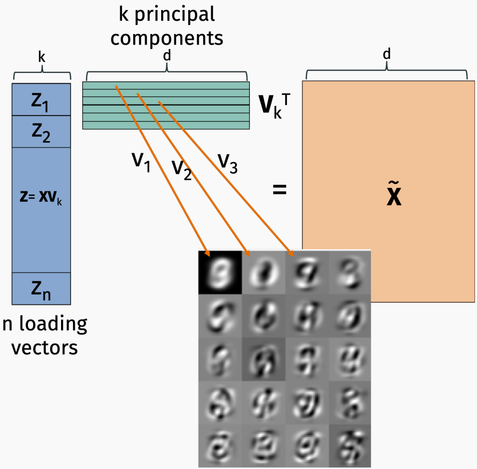
Genetic data can be nicely visualized using PCA! Plot each data example **x** using two loading variables in **z**.

PRINCIPAL COMPONENTS

For more complex data, what do principal components and loading vectors look like?

PRINCIPAL COMPONENTS

MNIST **principal components**:

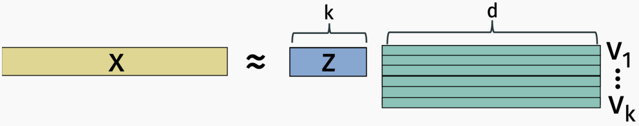


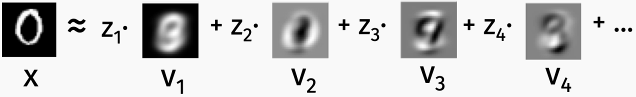
Often principal components are difficult to interpret.

LOADING VECTORS

What do the **loading vectors** looks like?

The loading vector $\mathbf{z}$ for an example $\mathbf{x}$ contains coefficients which recombine the top k principal components $\mathbf{v}_1, \ldots, \mathbf{v}_k$ to approximately reconstruct $\mathbf{x}$.

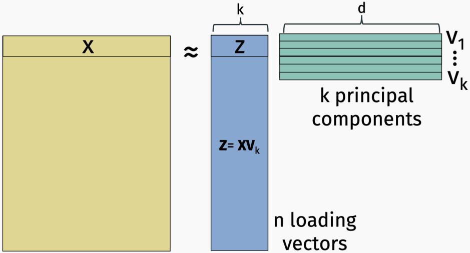


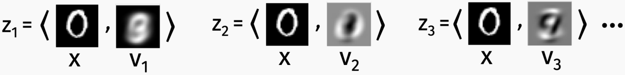


Provide a short “finger print” for any image $\mathbf{x}$ which can be used to reconstruct that image.

LOADING VECTORS: SIMILARITY VIEW

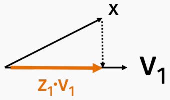
For any $\mathbf{x}$ with loading vector $\mathbf{z}$, $\mathbf{z}_i$ is the inner product similarity between $\mathbf{x}$ and the i^{th} principal component $\mathbf{v}_i$.

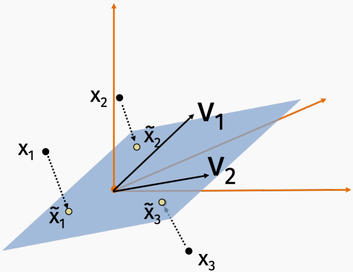




LOADING VECTORS: PROJECTION VIEW

So we approximate $\mathbf{x} \approx \tilde{\mathbf{x}} = \langle \mathbf{x}, \mathbf{v}_1 \rangle \cdot \mathbf{v}_1 + \ldots + \langle \mathbf{x}, \mathbf{v}_k \rangle \cdot \mathbf{v}_k$.



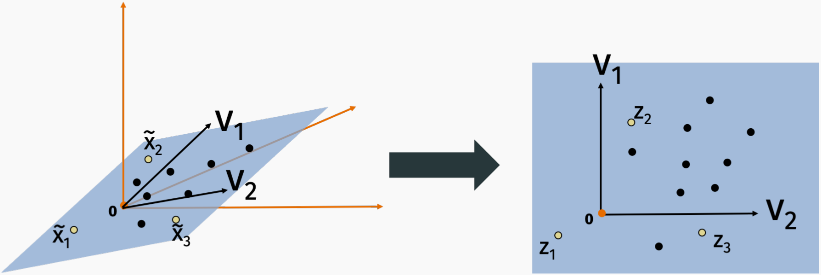


Since $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are orthonormal, this operation is a **projection** onto first k principal components.

I.e. we are projecting $\mathbf{x}$ onto the k-dimensional subspace spanned by $\mathbf{v}_1, \ldots, \mathbf{v}_k$.

LOADING VECTORS: PROJECTION VIEW

For an example $\mathbf{x}_i$, the loading vector $\mathbf{z}_i$ contains the coordinates in the projection space:



SIMILARITY PRESERVATION

Important takeaway for data visualization and more: Latent feature vectors preserve similarity and distance information in the original data.

Let $\mathbf{x}_1 \ldots, \mathbf{x}_n \in \mathbb{R}^d$ be our original data vectors, $\mathbf{z}_1 \ldots, \mathbf{z}_n \in \mathbb{R}^k$ be our loading vectors (encoding), and $\tilde{\mathbf{x}}_1 \ldots, \tilde{\mathbf{x}}_n \in \mathbb{R}^d$ be our low-rank approximated data.

We have:

$$\|\tilde{\mathbf{x}}_i\|_2^2 = \|\mathbf{z}_i\|_2^2$$
$$\langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j \rangle = \langle \mathbf{z}_i, \mathbf{z}_j \rangle$$
$$\|\tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_j\|_2^2 = \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

SIMILARITY PRESERVATION

Conclusion: If our data had a good low rank approximation, we expect that:

$$\|\mathbf{x}_i\|_2^2 \approx \|\mathbf{z}_i\|_2^2$$
$$\langle \mathbf{x}_i, \mathbf{x}_j \rangle \approx \langle \mathbf{z}_i, \mathbf{z}_j \rangle$$
$$\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \approx \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

When we come back from break, will use this to motivate semantic embeddings.