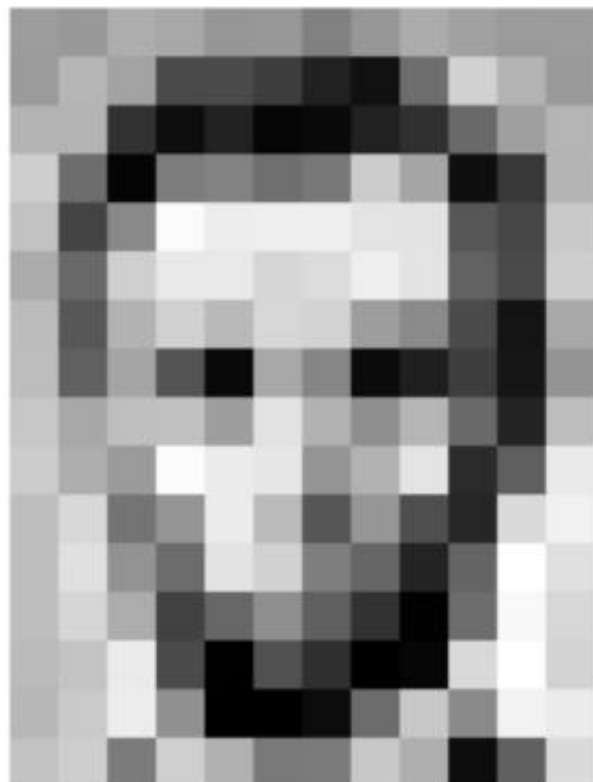


Introduction to Deep Computer Vision

Contents

1. Intro to CV and tasks in CV
2. Intro to DL
3. Convolutional Neural Networks
4. DL limitations

Images are Numbers



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

What the computer sees

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

An image is just a matrix of numbers $[0,255]$!
i.e., $1080 \times 1080 \times 3$ for an RGB image

Tasks in Computer Vision



Input Image



167	193	174	168	150	192	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	195	292	236	231	149	178	228	43	96	234
190	216	116	149	226	187	86	190	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	90	2	109	249	215
187	196	236	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	129	207	177	121	129	200	175	13	96	218

Pixel Representation



classification

Lincoln

Washington

Jefferson

Obama

$$\begin{bmatrix} 0.8 \\ 0.1 \\ 0.05 \\ 0.05 \end{bmatrix}$$

- **Regression:** output variable takes continuous value
- **Classification:** output variable takes class label. Can produce probability of belonging to a particular class

Image Classification



Classification task: produce a list of object categories present in image. 1000 categories.
“Top 5 error”: rate at which the model does not output correct label in top 5 predictions

Beyond Classification

Semantic Segmentation



GRASS, CAT,
TREE, SKY

No objects, just pixels

**Classification
+ Localization**



CAT

Single Object

**Object
Detection**



DOG, DOG, CAT

Multiple Object

**Instance
Segmentation**



DOG, DOG, CAT

This image is CC0 public domain

Instance Segmentation and Pose Estimation



Image Domain Transfer



Low-res to high-res



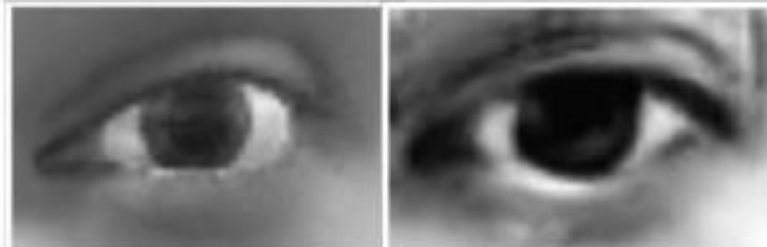
Blurry to sharp



Image to painting



LDR to HDR



Synthetic to real



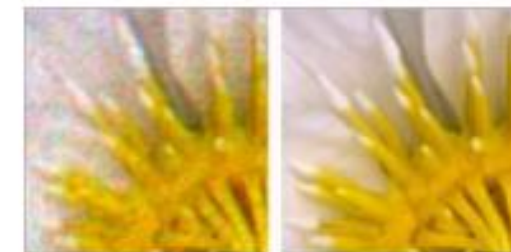
Thermal to color



Day to night



Summer to winter

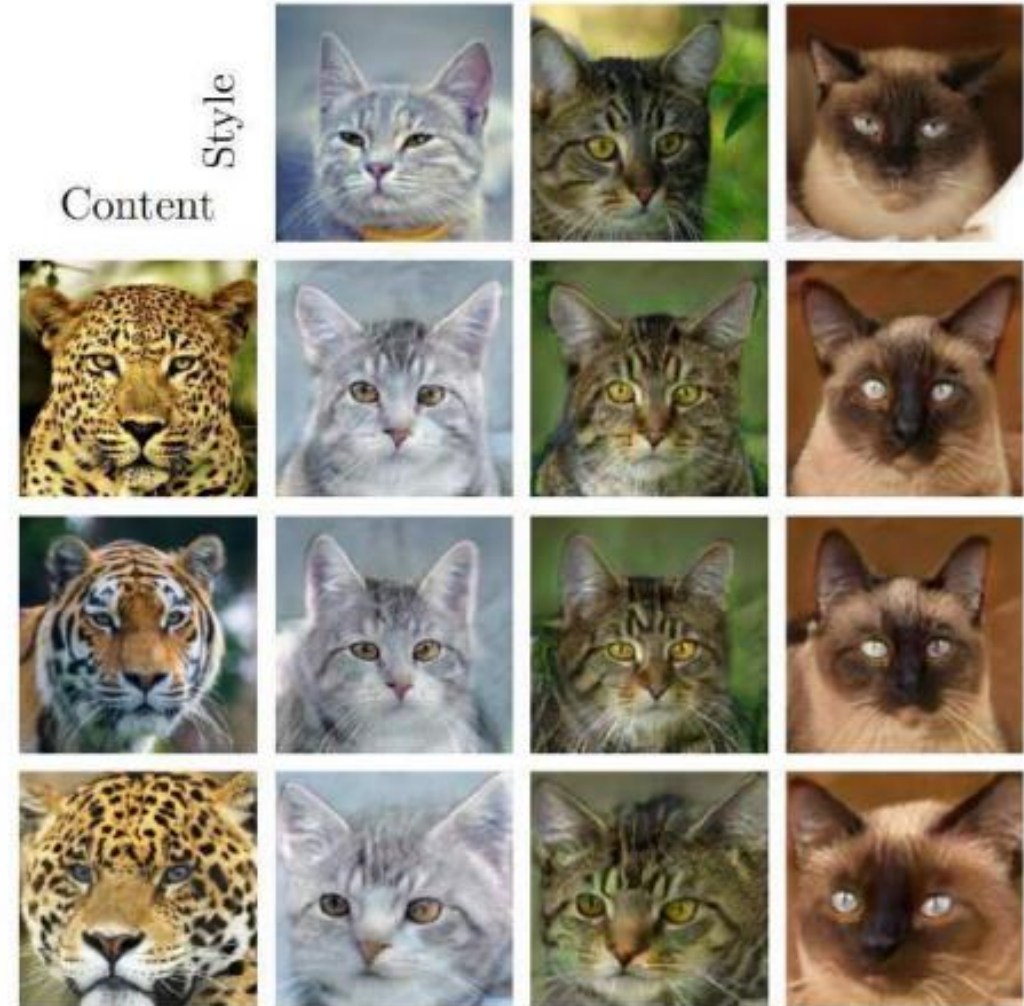


Noisy to clean

Multimodal Image Domain Transfer



(b) edges \rightarrow shoes



(b) big cats \rightarrow house cats

Manual Feature Extraction

Domain knowledge

Define features

Detect features
to classify

Viewpoint variation



Scale variation



Deformation



Occlusion



Illumination conditions



Background clutter



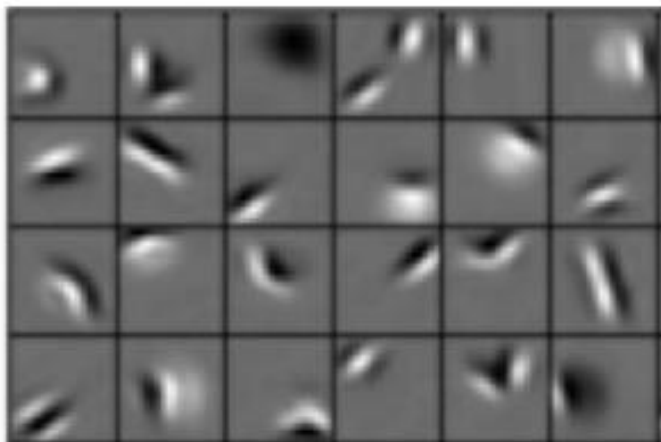
Intra-class variation



Learning Feature Representations

Can we learn a **hierarchy of features** directly from the data instead of hand engineering?

Low level features



Edges, dark spots

Mid level features



Eyes, ears, nose

High level features



Facial structure

The Rise of Deep Learning

'Deep Voice' Software Can Clone Anyone's Voice With Just 3.7 Seconds of Audio

Using snippets of voices, Baidu's 'Deep Voice' can generate new speech, accents, and tones.



DEEPMIND STARCRAFT TRIUMPH

Let There Be Sight: How Deep Learning Is Helping the Blind 'See'



Technology outpacing security measures

1. Social Researchers 2. Features and their uses

AI beats docs in cancer spotting

A new study provides a fresh example of machine learning as an import diagnostic tool. Paul Hager reports.



AI Can Help In Predicting Cryptocurrency Value



'Creative' AlphaZero leads way for chess computers and, maybe, science

Former chess world champion Garry Kasparov likes what he sees of computer that could be used to find cures for diseases



How an A.I. 'Cat-and-Mouse Game' Generates Believable Fake Photos

By CLIVE WITTE and KATHY GOLAN JAN 5, 2018



Stock Predictions Based On AI: Is the Market Truly Predictable?



Complex of bacteria infecting new patients modeled in CASP-13. The complex can that were modeled individually. Microscopy image

Google's DeepMind aces protein folding

By Robert F. Service | Dec. 6, 2019, 12:00 PM

After Millions of Trials, These Simulated Humans Learned to Do Perfect Backflips and Cartwheels

Example: Backflip. After 100,000 trials, the system generated 10 million episodes over 10 days.



Neural networks everywhere

New chip reduces neural networks' power consumption by up to 95 percent, making them practical for battery-powered devices.

Work, study/visit - Bottom - Comment - By Ravi Varma - Digital Reporter - @RaviDMagazine



Researchers introduce a deep learning method that converts mono audio recordings into 3D sounds using video scenes

AI faces show how far AI image generation has come in just four years

As we see the right ones? Well, they're the product of machine learning.



Automation And Algorithms: De-Risking Manufacturing With Artificial Intelligence

Barish Goshima | Entrepreneur | Manufacturing | Views on the relationship of automation and AI

TWEET THIS
The two key applications of AI in manufacturing are pricing and manufacturability feedback

ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



MACHINE LEARNING

Ability to learn without explicitly being programmed

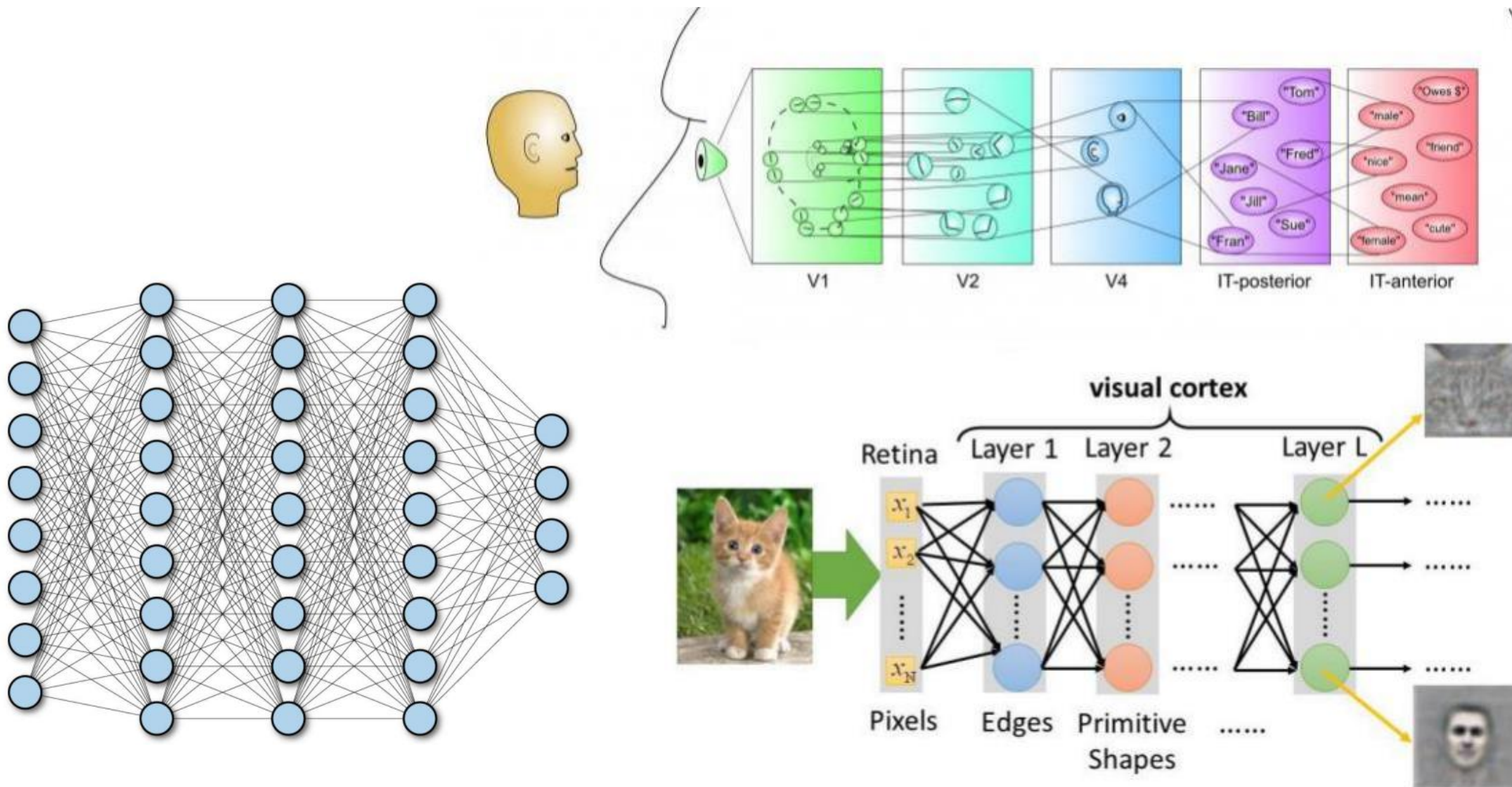


DEEP LEARNING

Extract patterns from data using neural networks

3 1 3 4 7 2
1 2 4 2 3 5

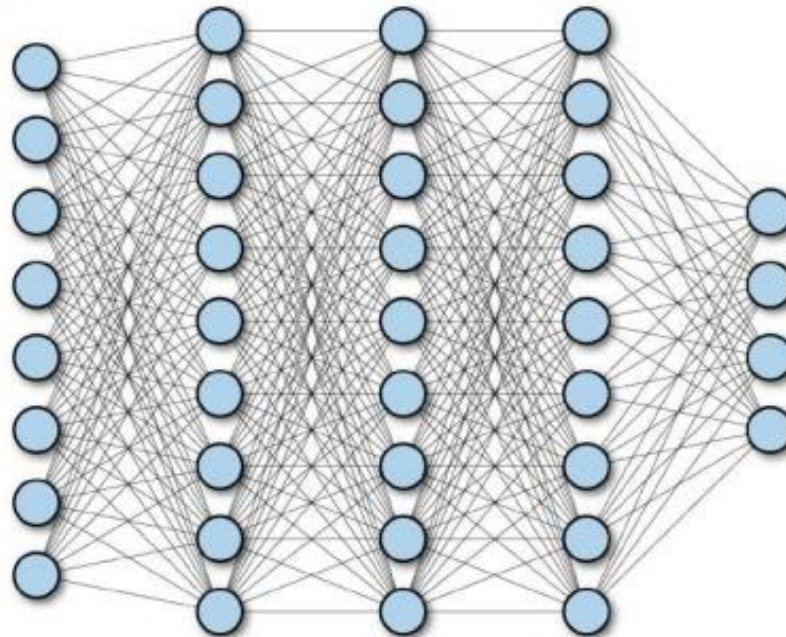
Visual Cortex and FFNN



Power of Neural Nets

Universal Approximation Theorem

A feedforward network with a single layer is sufficient to approximate, to an arbitrary precision, any continuous function.



Hornik et al. *Neural Networks*. (1989)

Power of Neural Nets

Universal Approximation Theorem

A feedforward network with a single layer is sufficient to approximate, to an arbitrary precision, any continuous function.

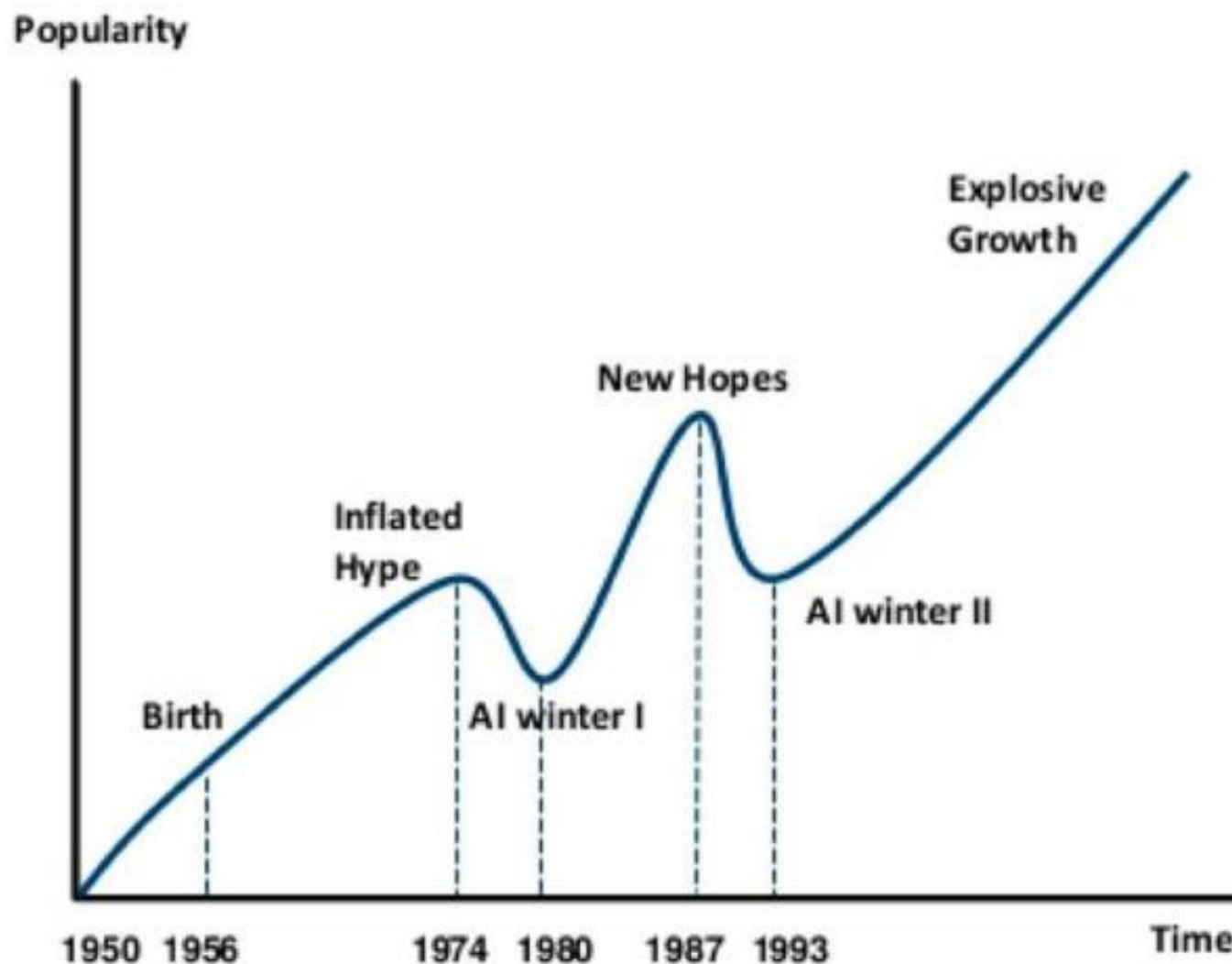
Caveats:

The number of hidden units may be infeasibly large

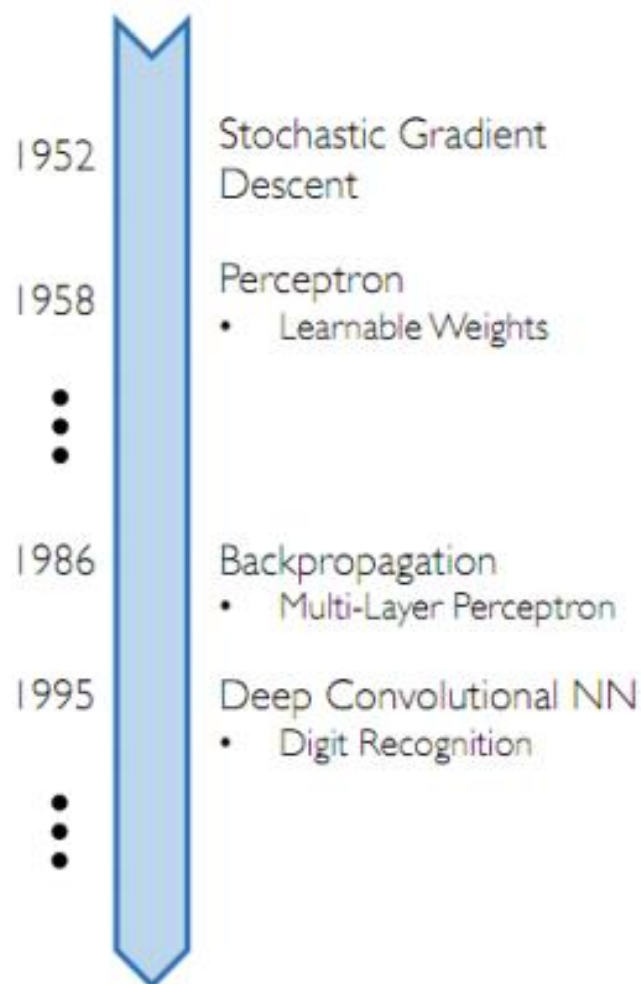
The resulting model may not generalize

Hornik et al. *Neural Networks*. (1989)

Artificial Intelligence “Hype”: Historical Perspective



Why now?



Neural Networks date back decades, so why the resurgence?

1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



WIKIPEDIA
The Free Encyclopedia



2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



3. Software

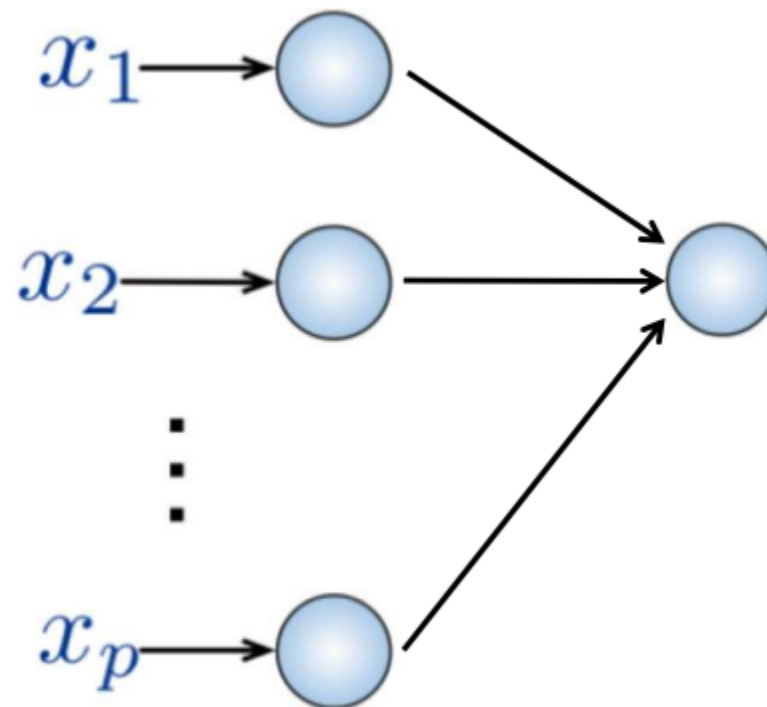
- Improved Techniques
- New Models
- Toolboxes



Fully Connected Neural Networks

Input:

- 2D image
- Vector of pixel values



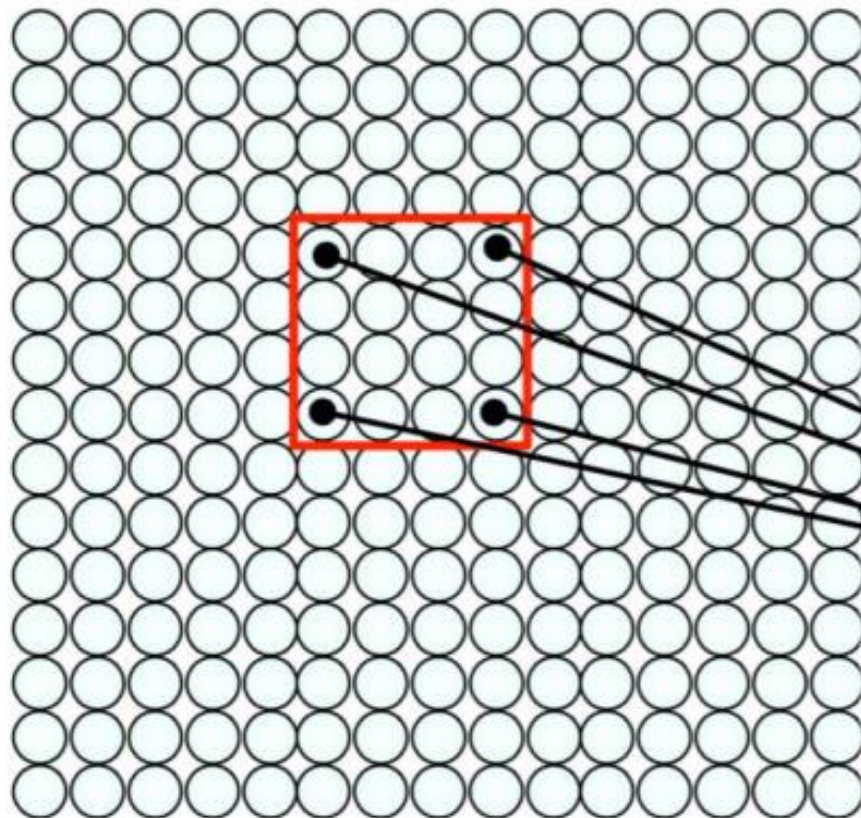
Fully Connected:

- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- And many, many parameters!

How can we use **spatial structure** in the input to inform the architecture of the network?

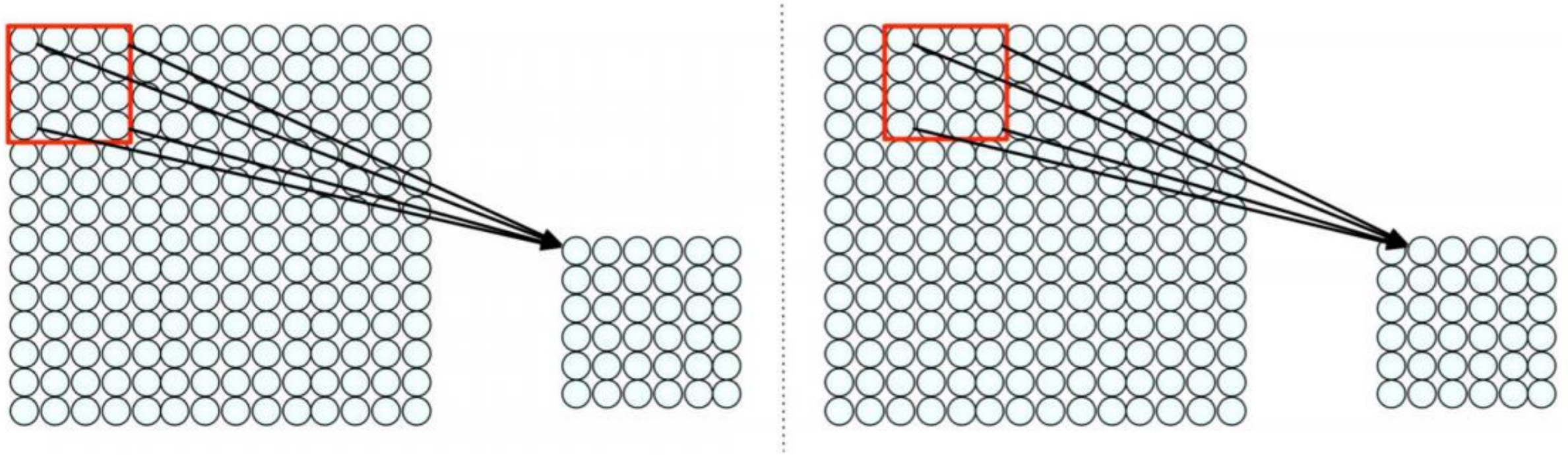
Using Spatial Structure

Input: 2D image.
Array of pixel values



Idea: connect patches of input
to neurons in hidden layer.
Neuron connected to region of
input. Only "sees" these values.

Using Spatial Structure



Connect patch in input layer to a single neuron in subsequent layer.

Use a sliding window to define connections.

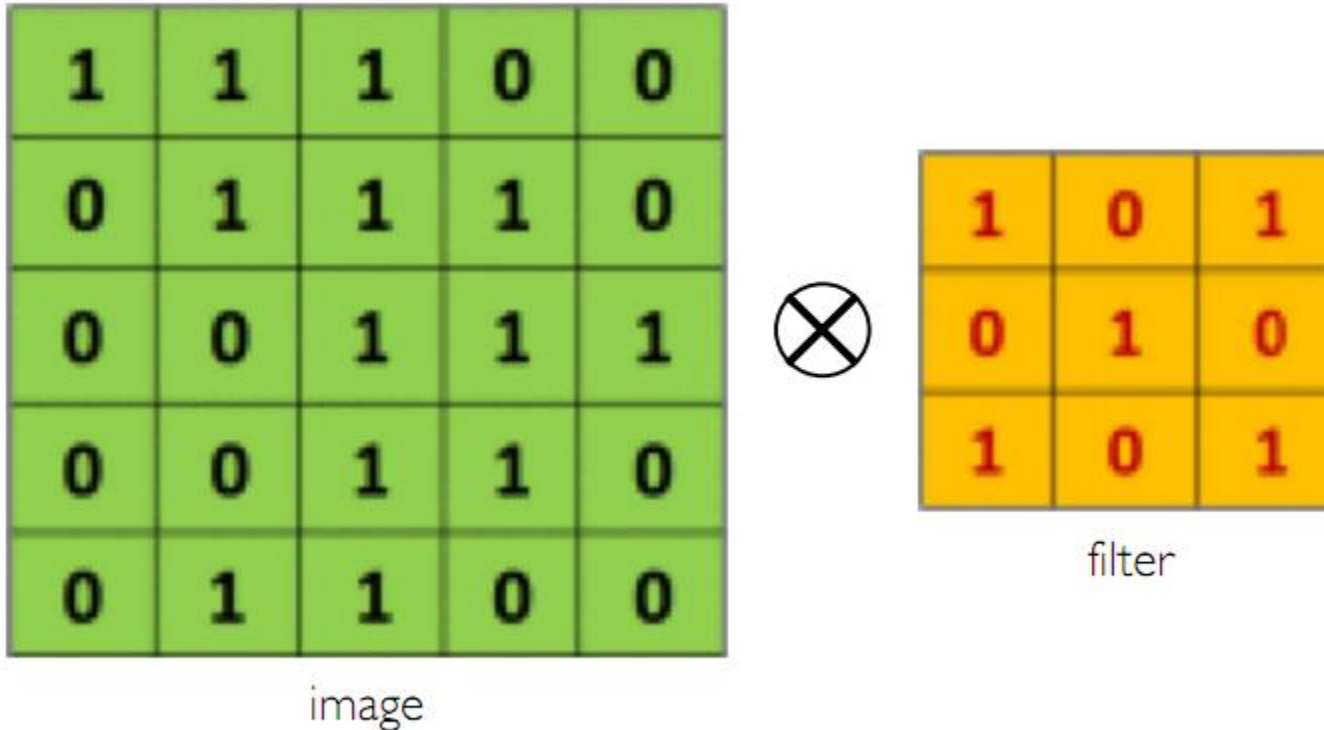
How can we **weight** the patch to detect particular features?

Applying Filters to Extract Features

- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) Spatially **share** parameters of each filter
(features that matter in one part of the input should matter elsewhere)

The Convolution Operation

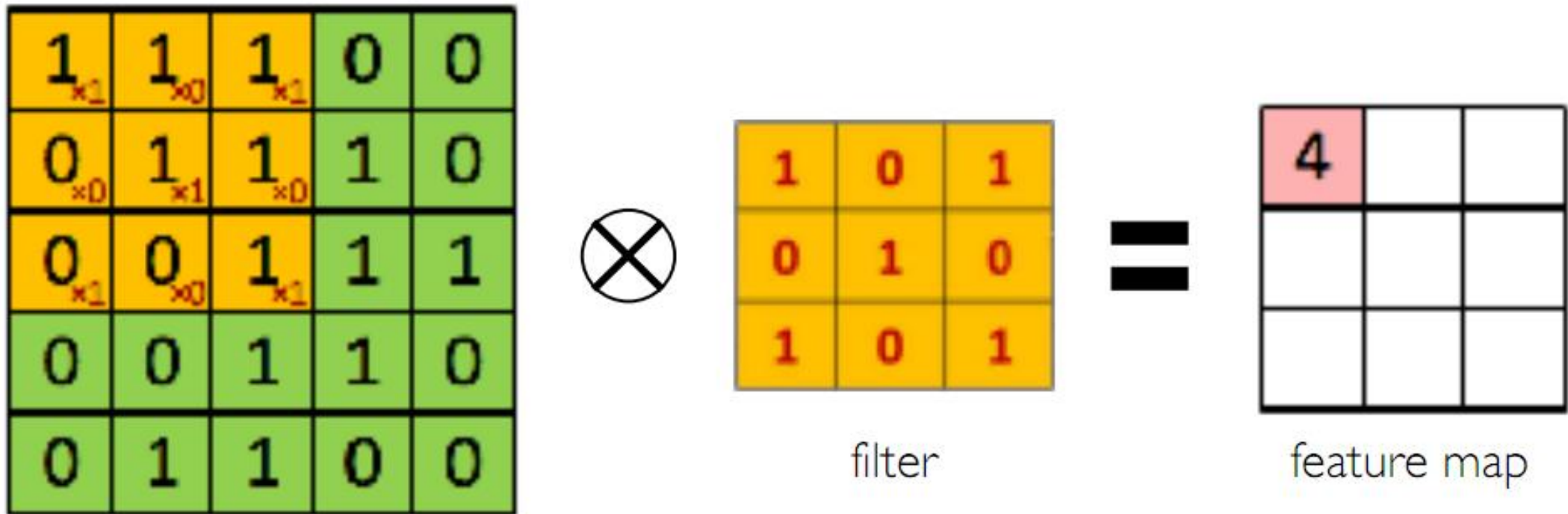
Suppose we want to compute the convolution of a 5x5 image and a 3x3 filter:



We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs...

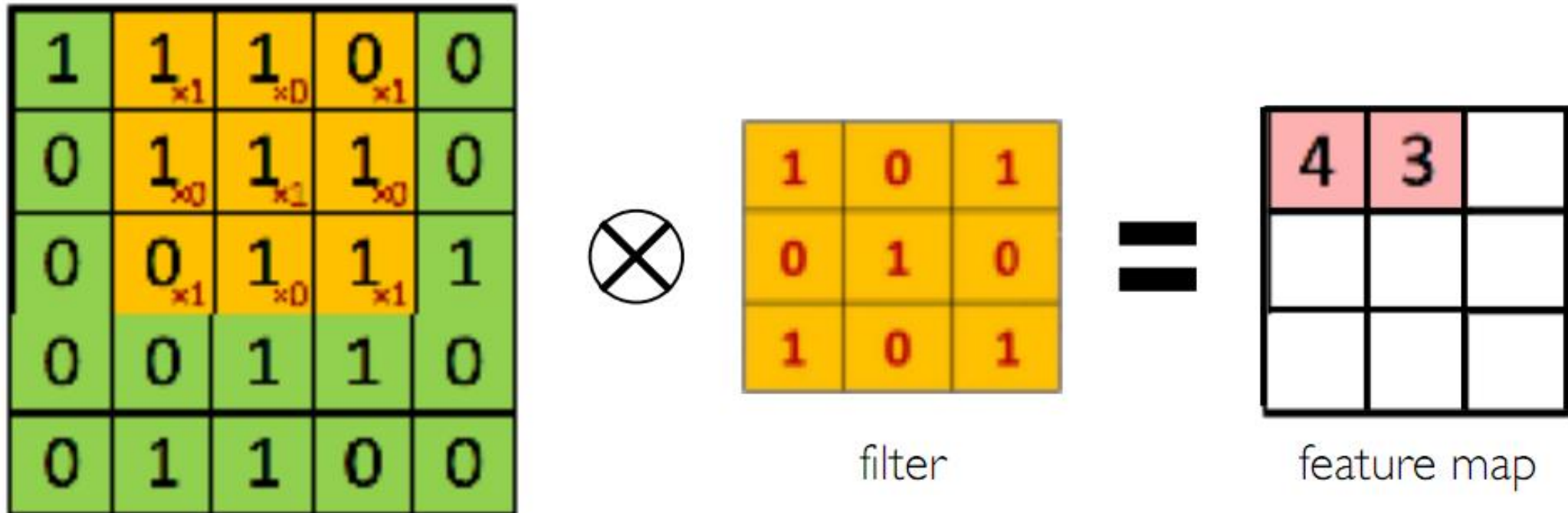
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



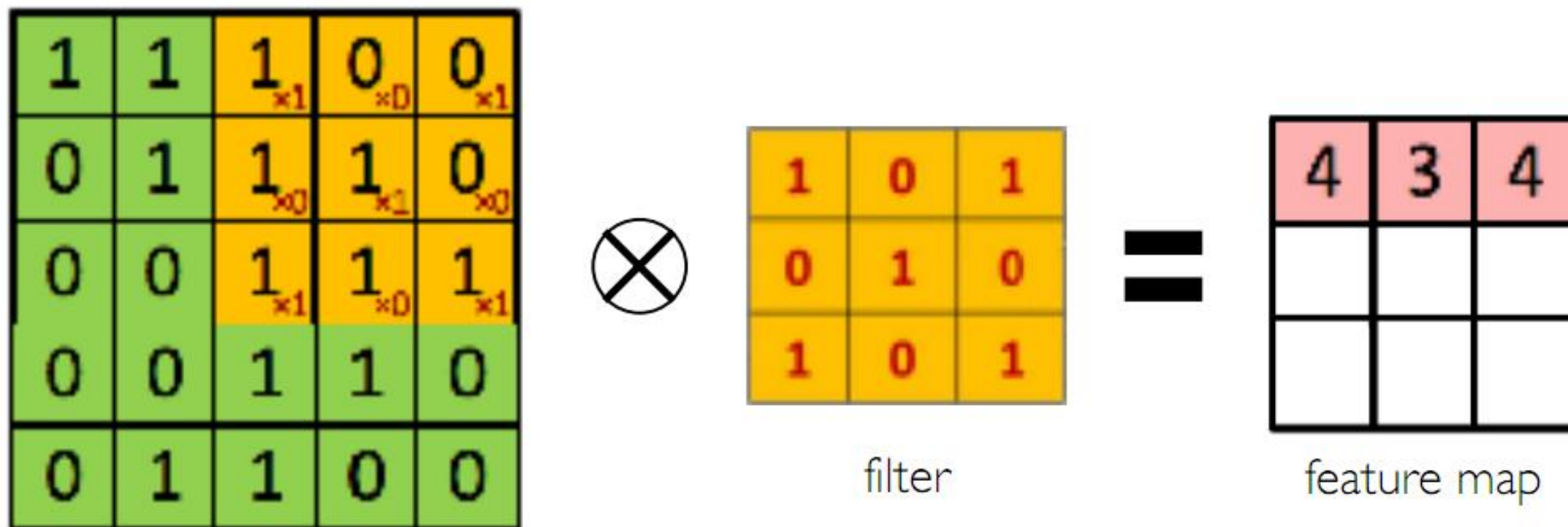
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



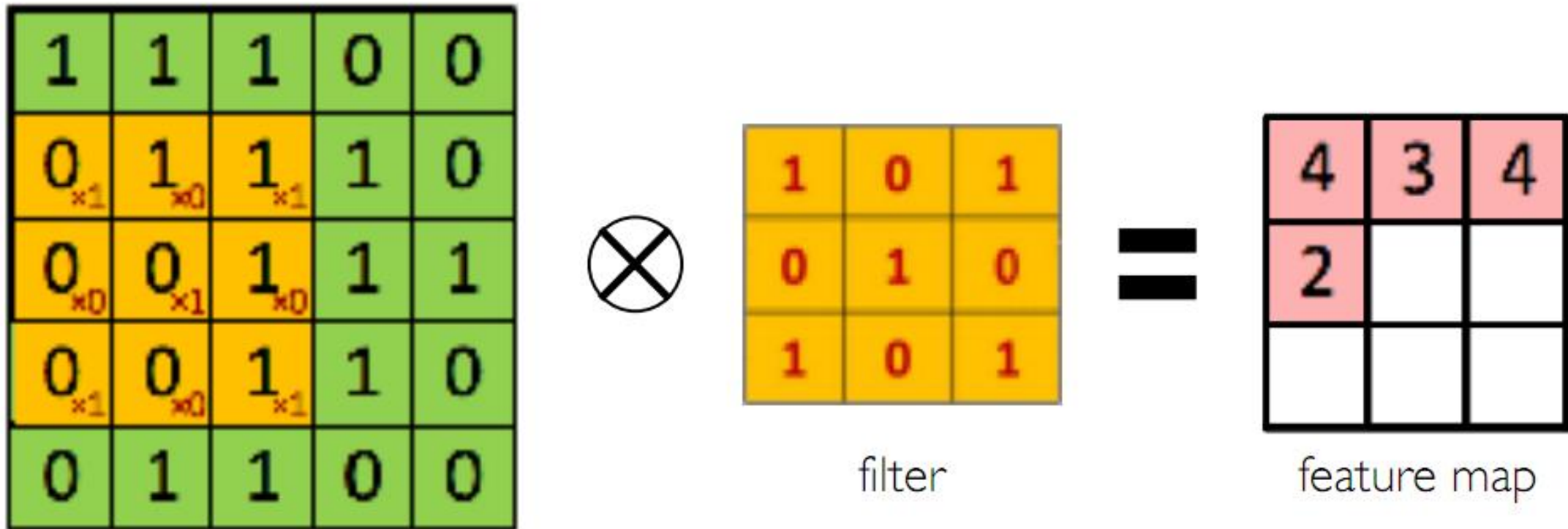
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



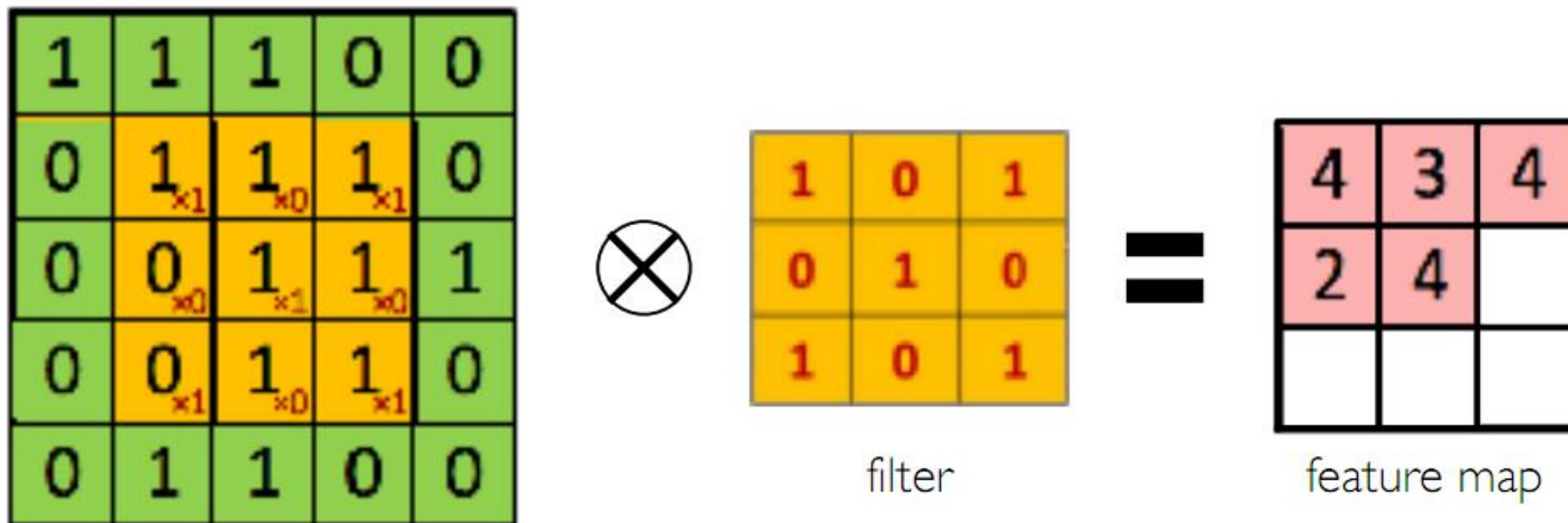
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



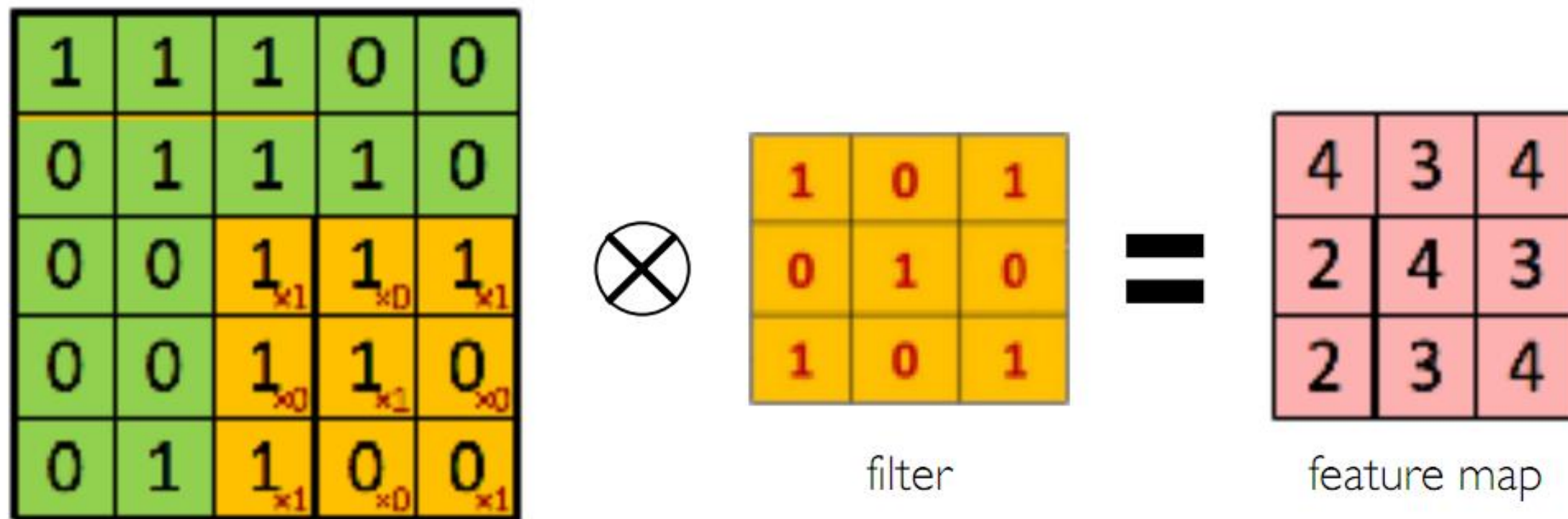
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



Producing Feature Maps



Original



Sharpen



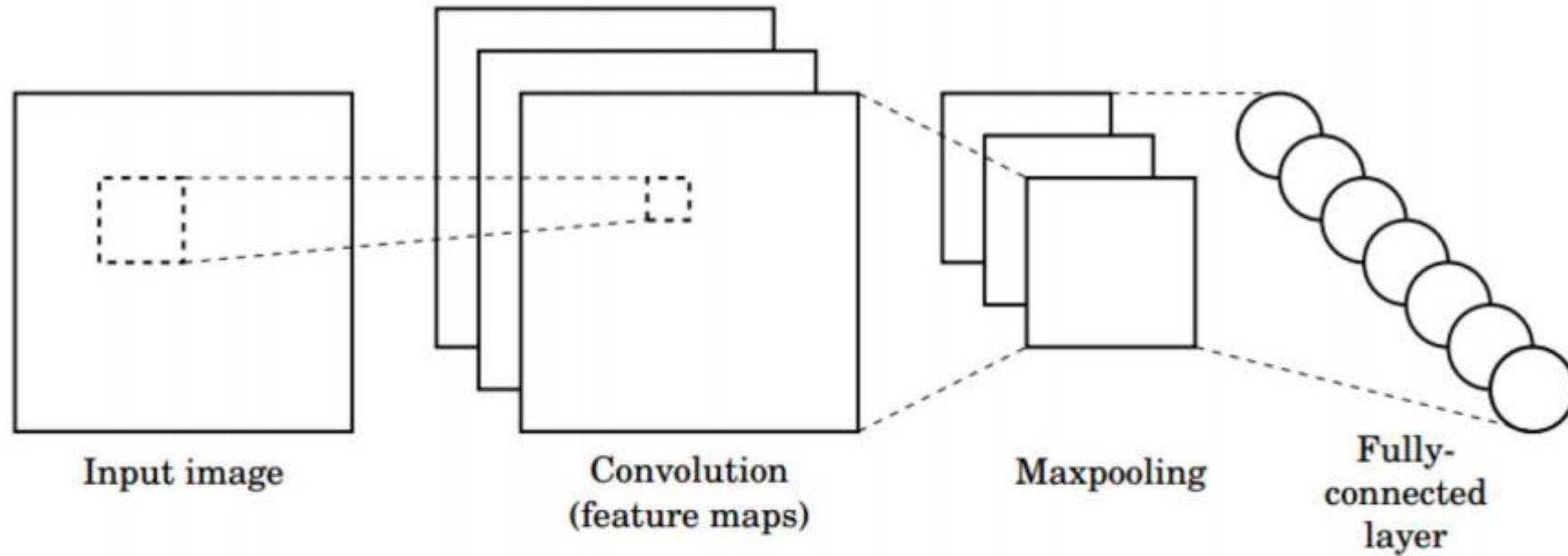
Edge Detect



“Strong” Edge
Detect

Convolutional Neural Networks (CNNs)

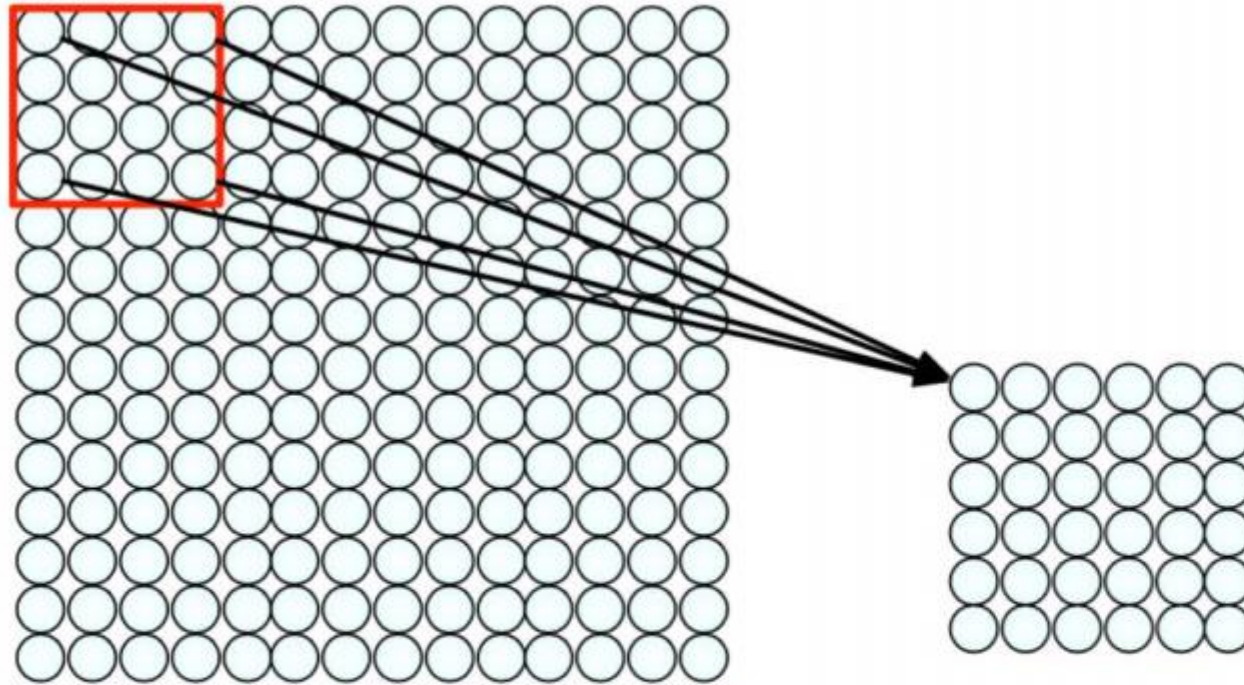
Producing Feature Maps



1. **Convolution:** Apply filters with learned weights to generate feature maps.
2. **Non-linearity:** Often ReLU.
3. **Pooling:** Downsampling operation on each feature map.

Train model with image data.
Learn weights of filters in convolutional layers.

Producing Feature Maps



4x4 filter: matrix
of weights w_{ij}

$$\sum_{i=1}^4 \sum_{j=1}^4 w_{ij} x_{i+p,j+q} + b$$

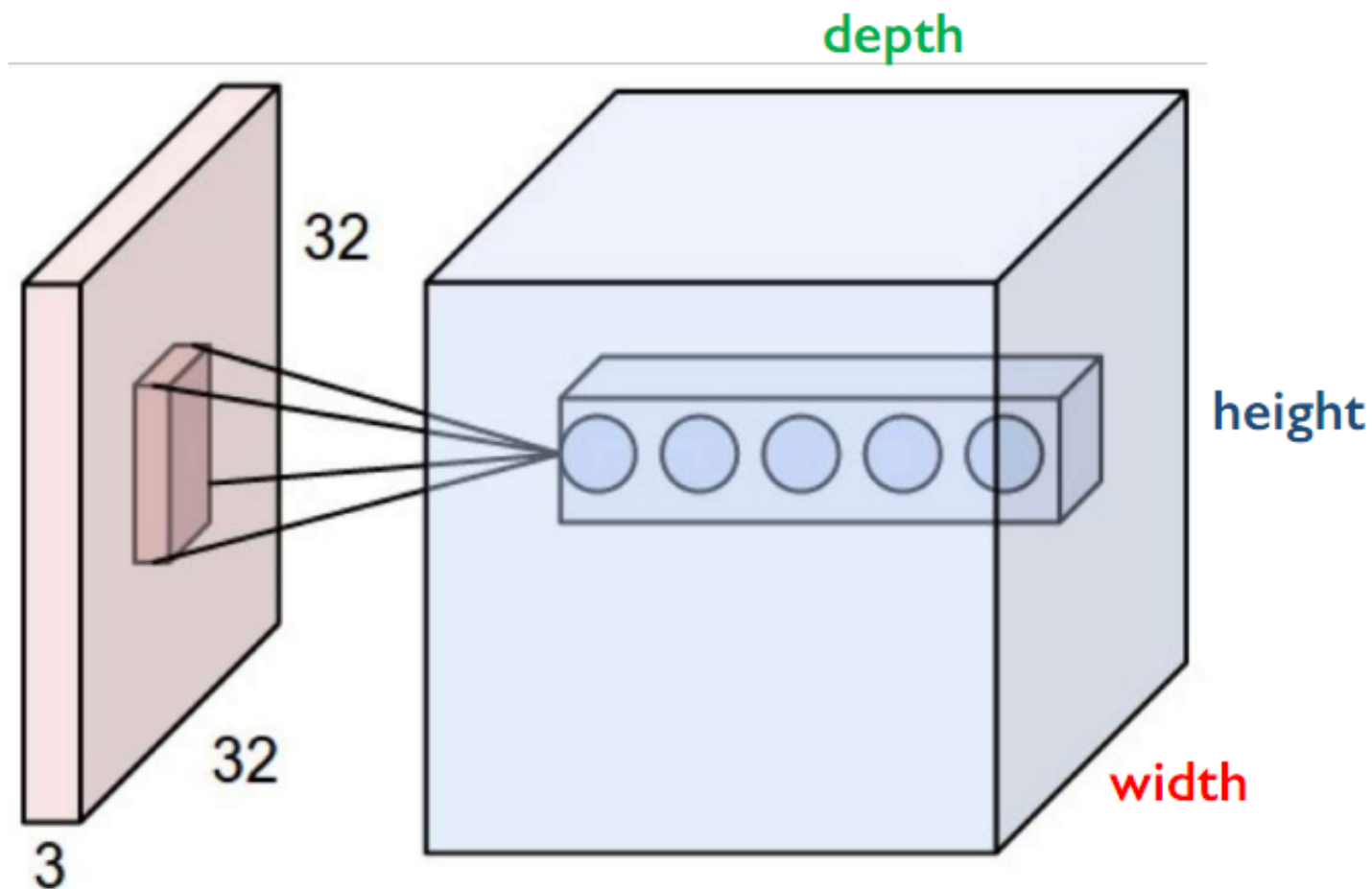
for neuron (p,q) in hidden layer

For a neuron in hidden layer:

- Take inputs from patch
- Compute weighted sum
- Apply bias

- 1) applying a window of weights
- 2) computing linear combinations
- 3) activating with non-linear function

CNNs: Spatial Arrangement of Output Volume



Layer Dimensions:

$$h \times w \times d$$

where h and w are spatial dimensions
 d (depth) = number of filters

Stride:

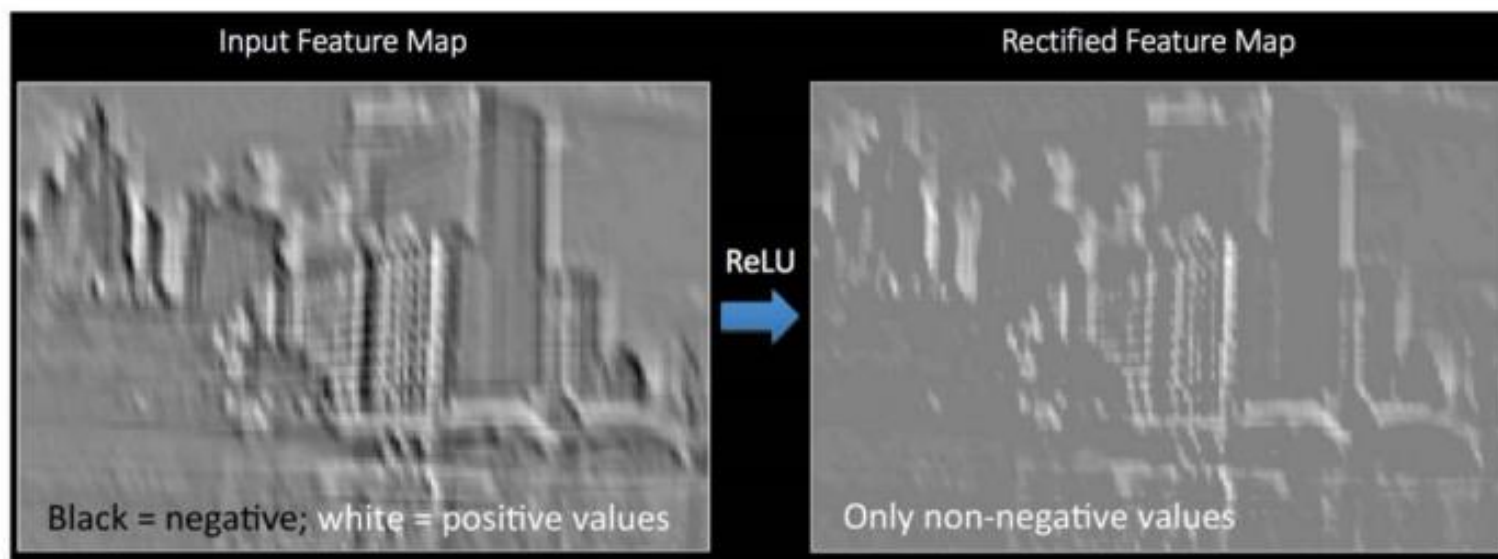
Filter step size

Receptive Field:

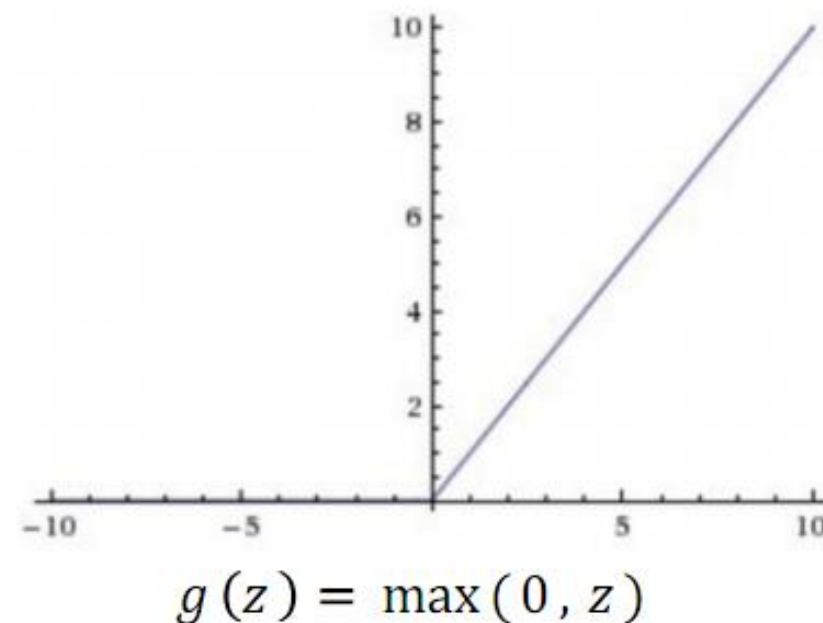
Locations in input image that
a node is path connected to

Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**



Rectified Linear Unit (ReLU)



Pooling

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

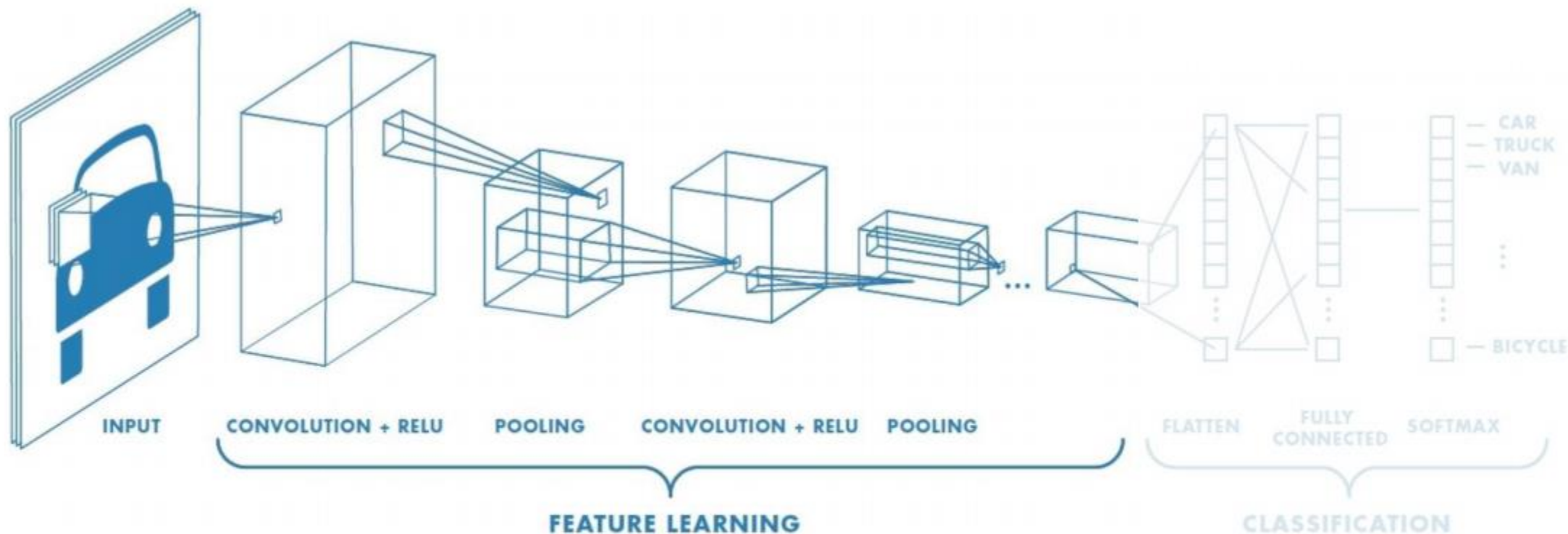
max pool with 2x2 filters
and stride 2



6	8
3	4

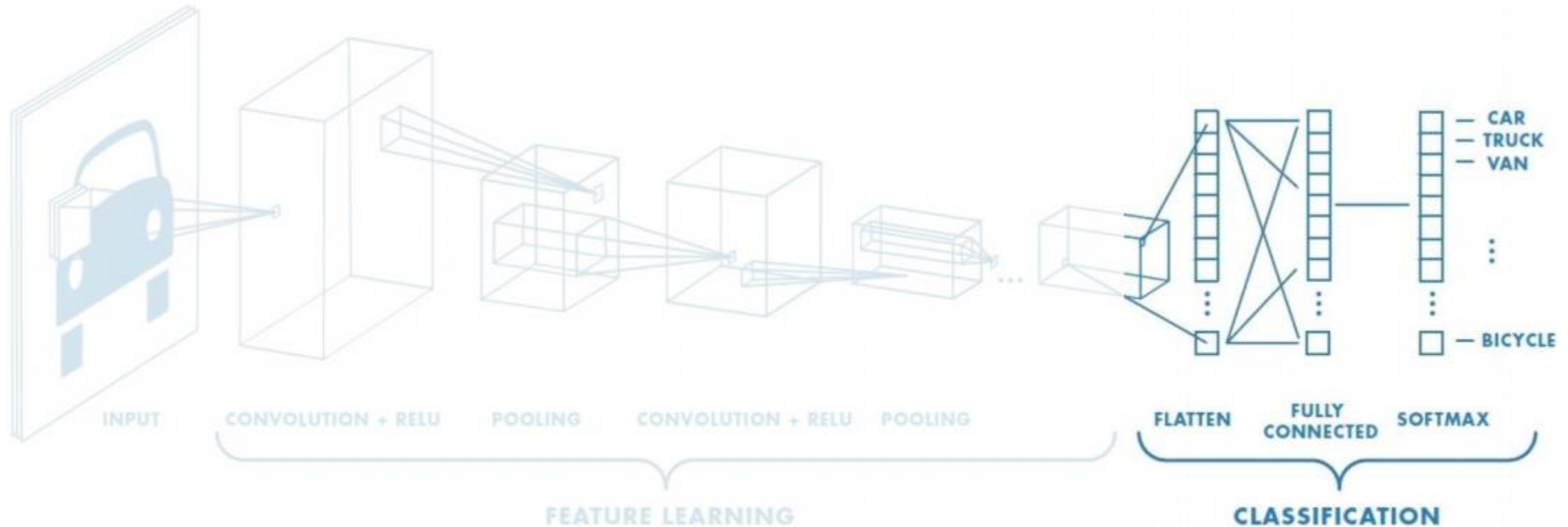
- 1) Reduced dimensionality
- 2) Spatial invariance

CNNs for Classification: Feature Learning



1. Learn features in input image through **convolution**
2. Introduce **non-linearity** through activation function (real-world data is non-linear!)
3. Reduce dimensionality and preserve spatial invariance with **pooling**

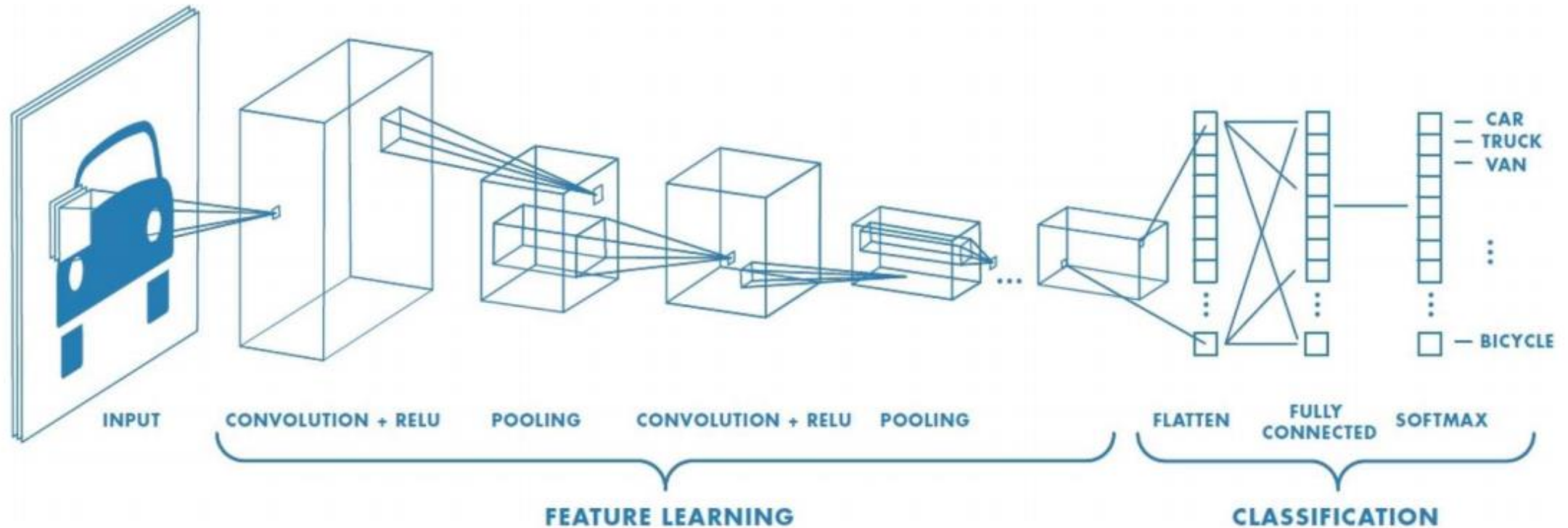
CNNs for Classification: Feature Learning



- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

CNNs for Classification: Feature Learning

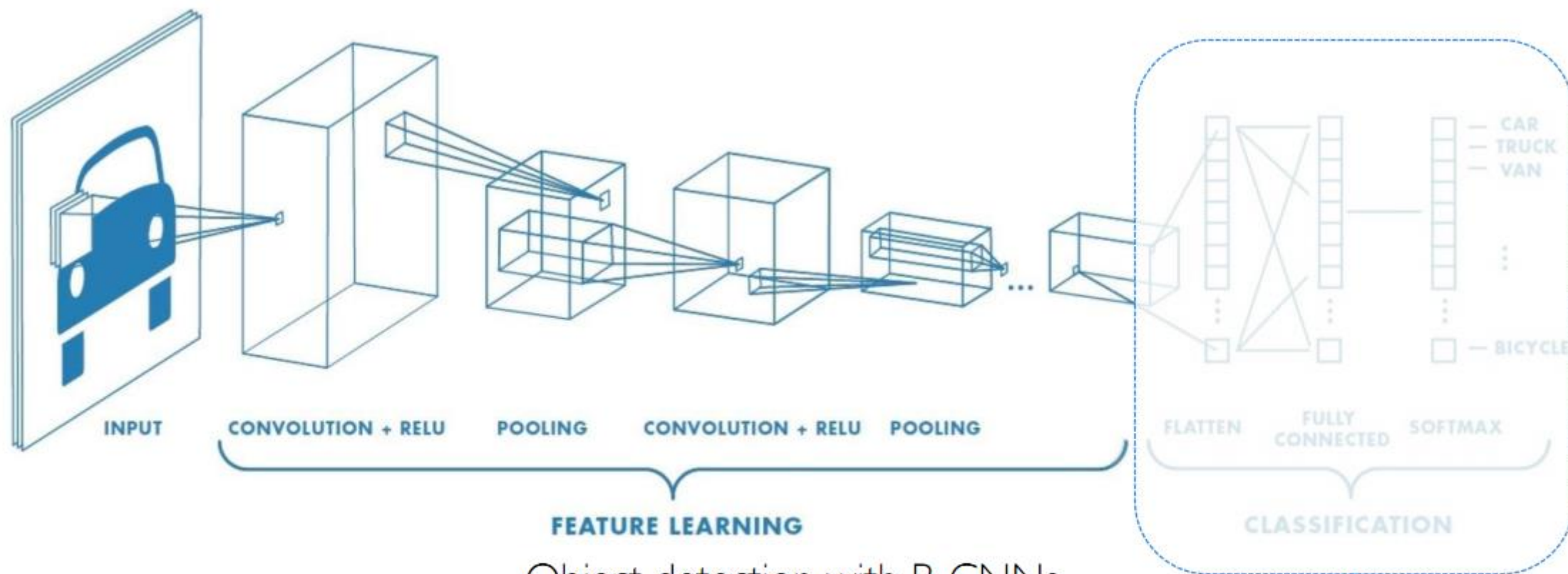


Learn weights for convolutional filters and fully connected layers

Backpropagation: cross-entropy loss

$$J(\theta) = \sum_i y^{(i)} \log(\hat{y}^{(i)})$$

An Architecture for Many Applications

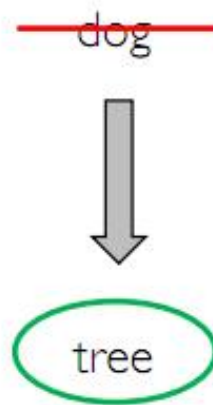
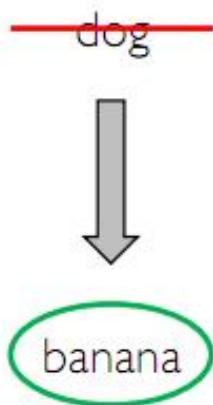


Object detection with R-CNNs
Segmentation with fully convolutional networks
Image captioning with RNNs

Limitations

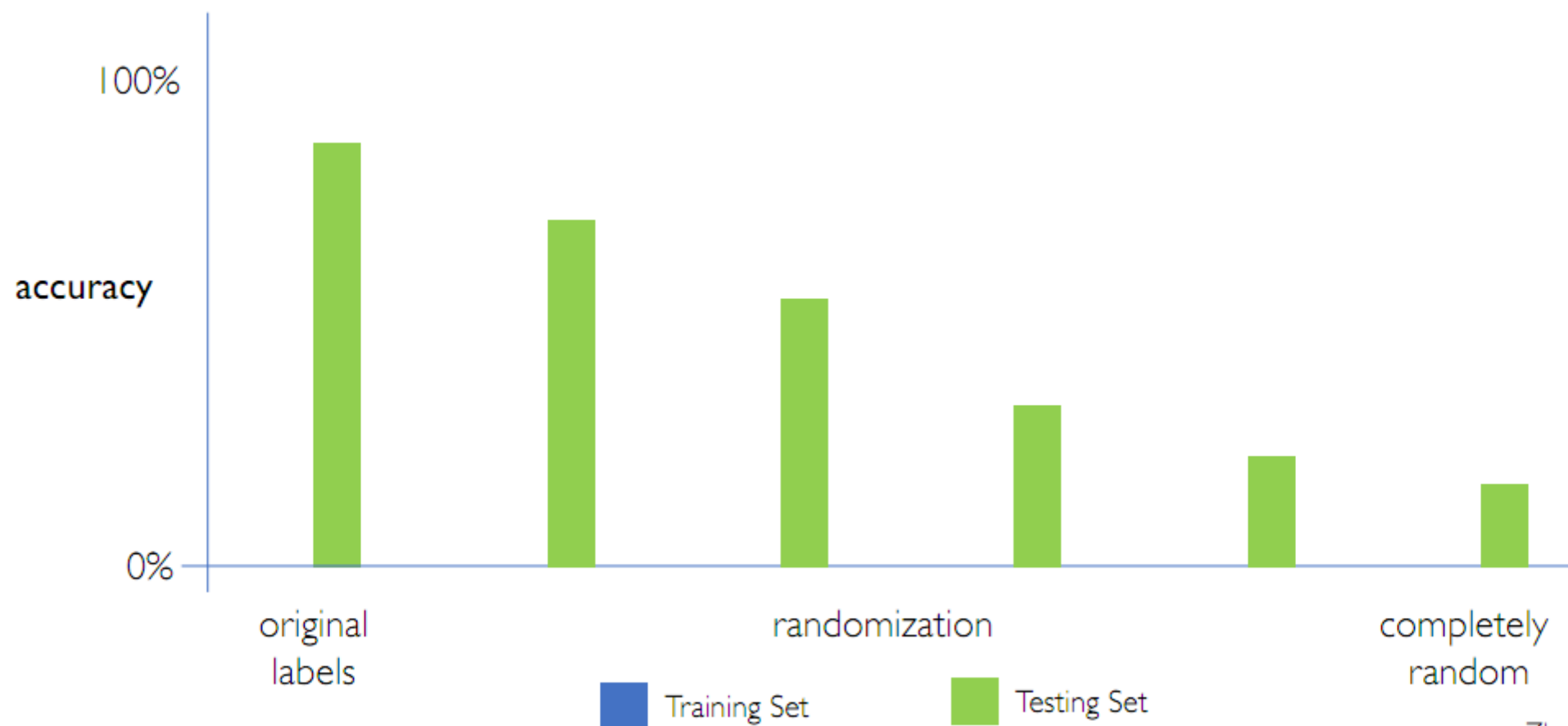
Rethinking Generalization

“Understanding Deep Neural Networks Requires Rethinking Generalization”



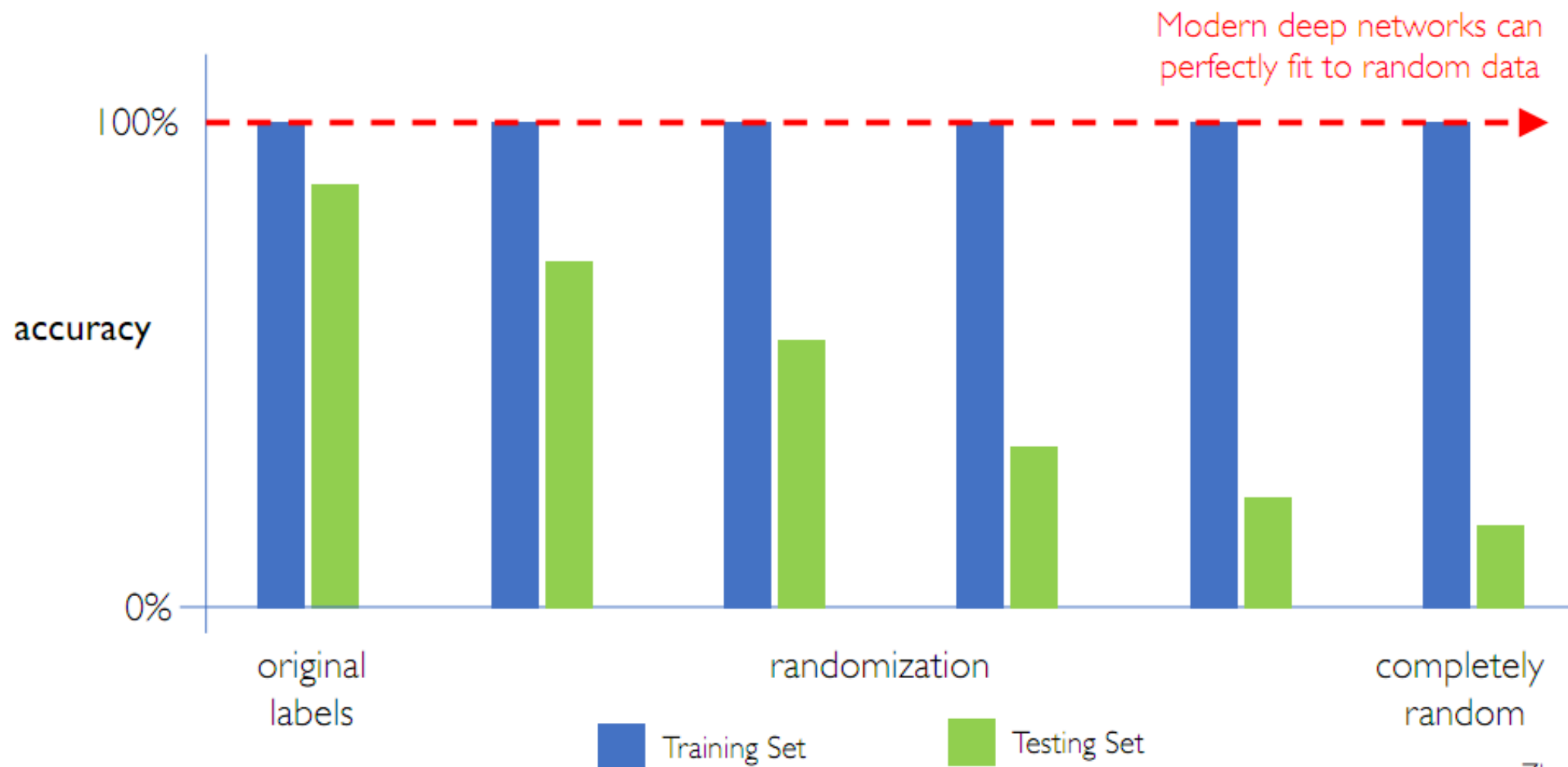
Zhang et al. ICLR (2017)

Capacity of Deep Neural Networks



Zhang et al. ICLR. (2017)

Capacity of Deep Neural Networks



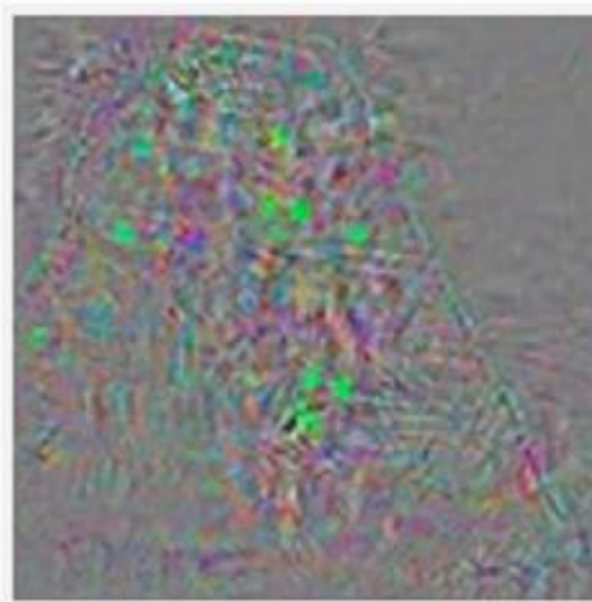
Zhang et al. ICLR. (2017)

Adversarial Attacks on Neural Networks

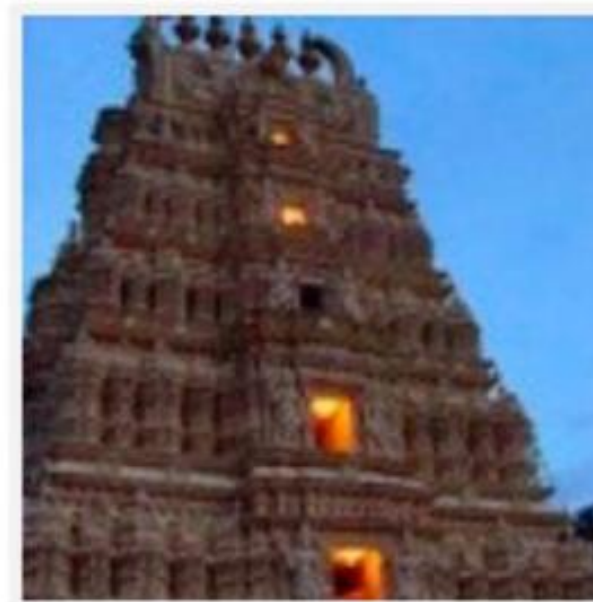


Original image

Temple (97%)



Perturbations



Adversarial example

Ostrich (98%)

Adversarial Attacks on Neural Networks



■ classified as turtle ■ classified as rifle
■ classified as other

Neural Network Limitations...

- Very **data hungry** (eg. often millions of examples)
- **Computationally intensive** to train and deploy (tractably requires GPUs)
- Easily fooled by **adversarial examples**
- Uninterpretable **black boxes**, difficult to trust
- **Finicky to optimize**: non-convex, choice of architecture, learning parameters
- Often require **expert knowledge** to design, fine tune architectures

Training Neural Networks

Contents

1. Activation Functions
2. Weight Initialization
3. Dropout
4. Batch Norm
5. Optimization and Learning rate Schedules
6. Data Augmentation

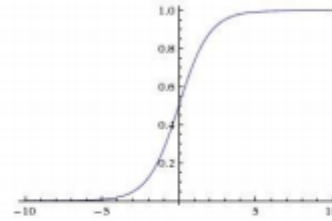
Activation Functions

Activation Functions

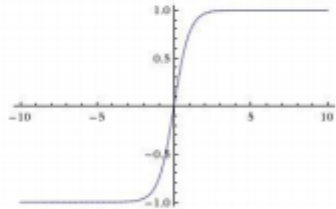
Activation Functions

Sigmoid

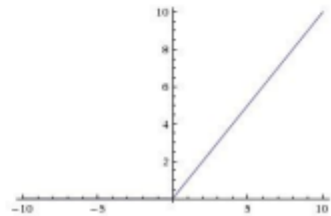
$$\sigma(x) = 1/(1 + e^{-x})$$



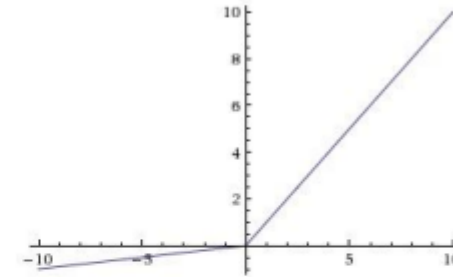
tanh tanh(x)



ReLU max(0,x)



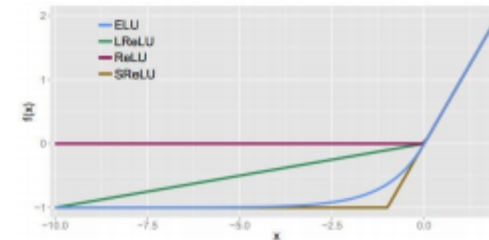
Leaky ReLU $\max(0.1x, x)$



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

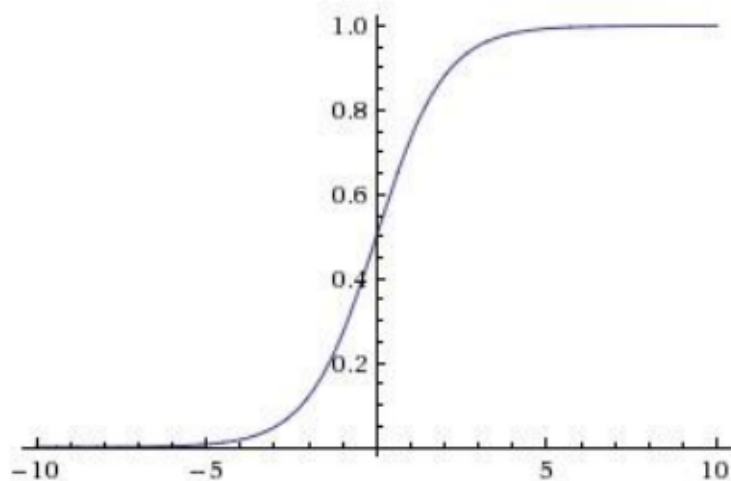
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Activation Functions

Activation Functions



Sigmoid

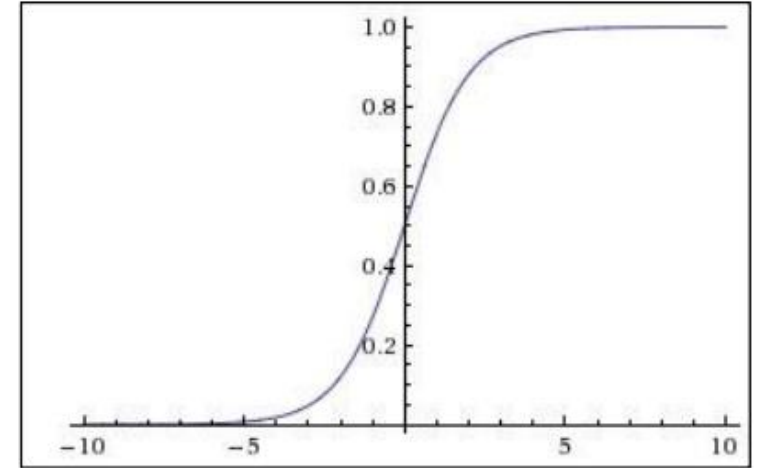
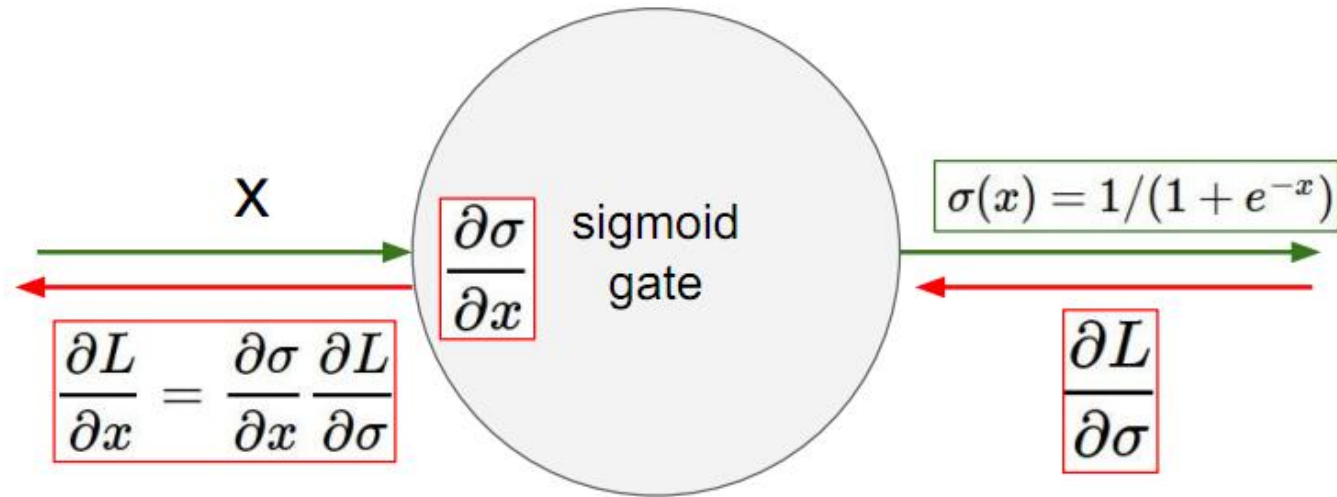
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

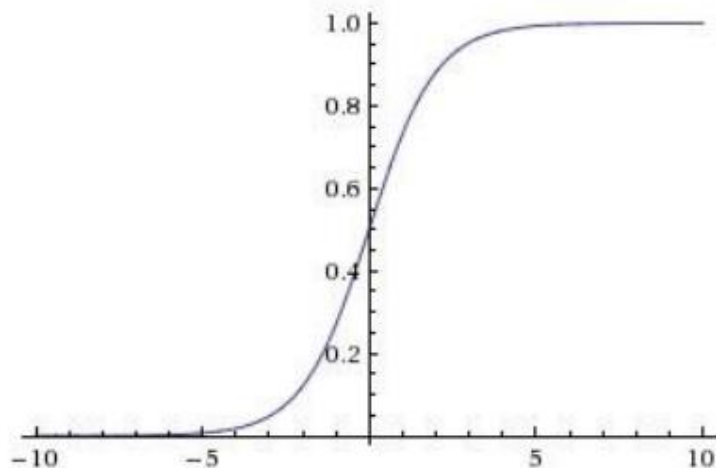
Activation Functions



What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?

Activation Functions

Activation Functions



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

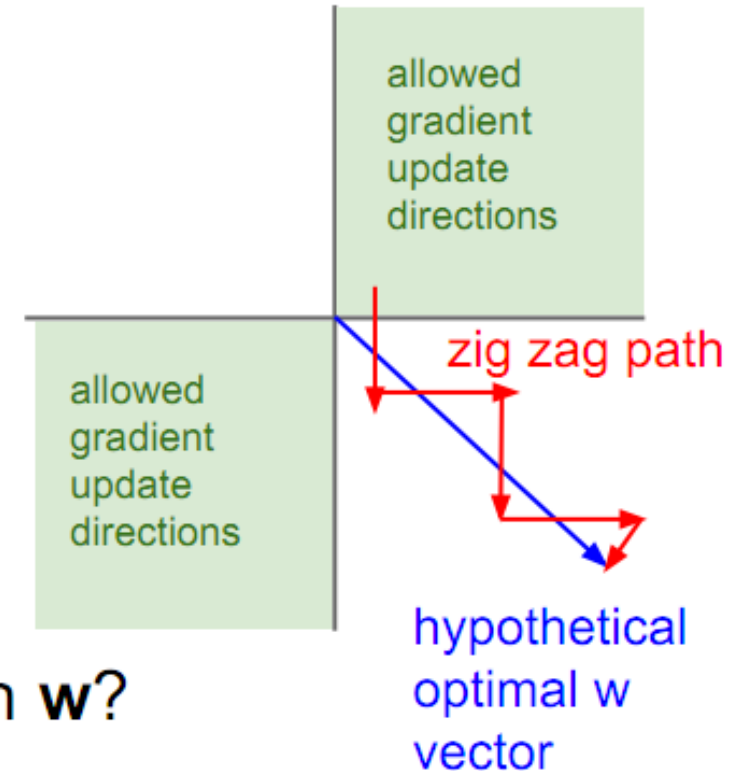
3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Activation Functions

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



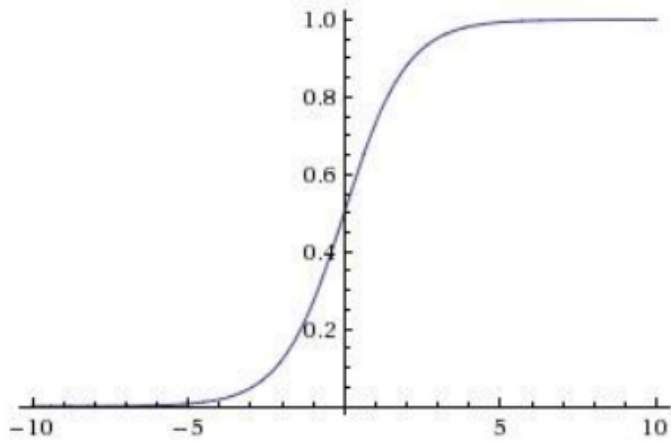
What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Activation Functions

Activation Functions



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

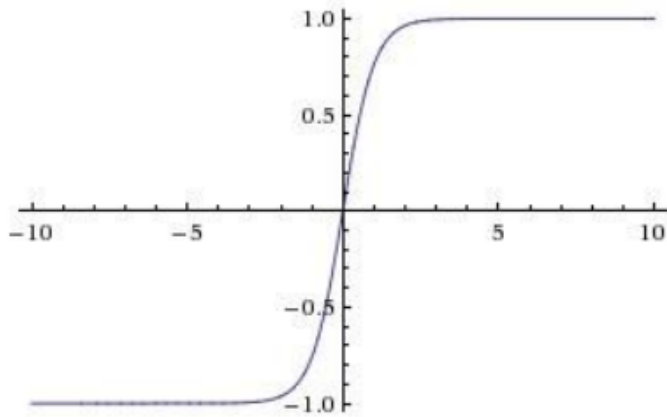
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. $\exp()$ is a bit compute expensive

Activation Functions

Activation Functions



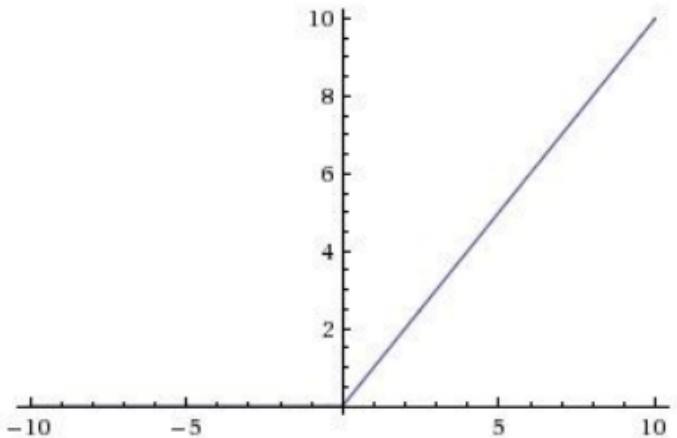
$\tanh(x)$

- Squashes numbers to range $[-1,1]$
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Activation Functions

Activation Functions



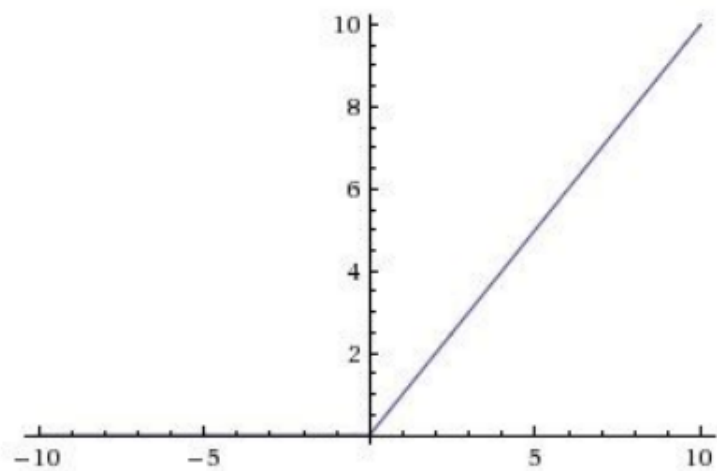
ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

Activation Functions

Activation Functions



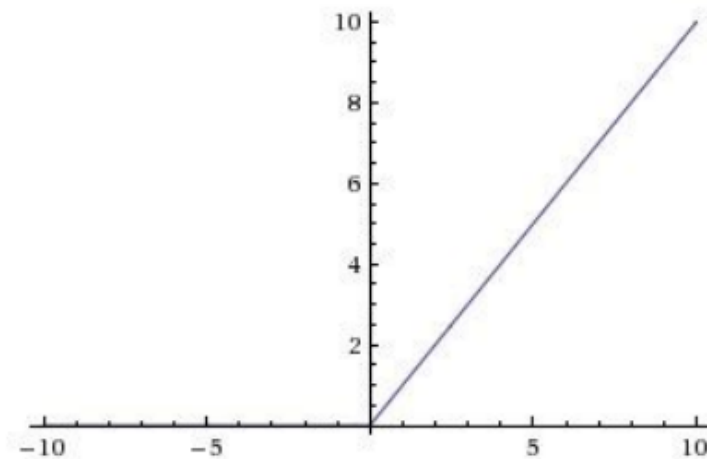
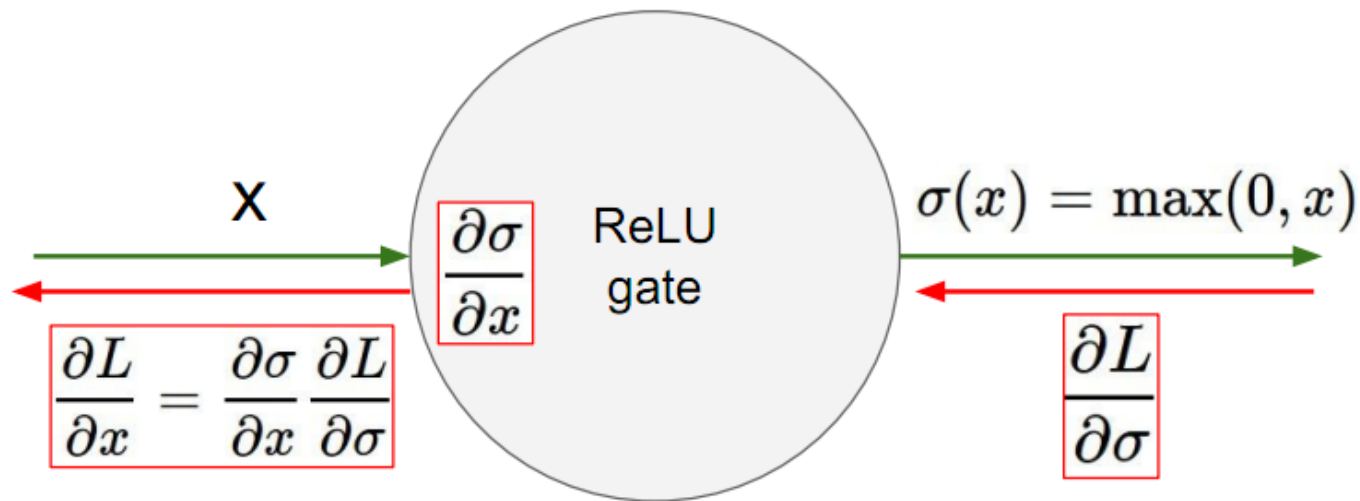
ReLU

(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

Activation Functions

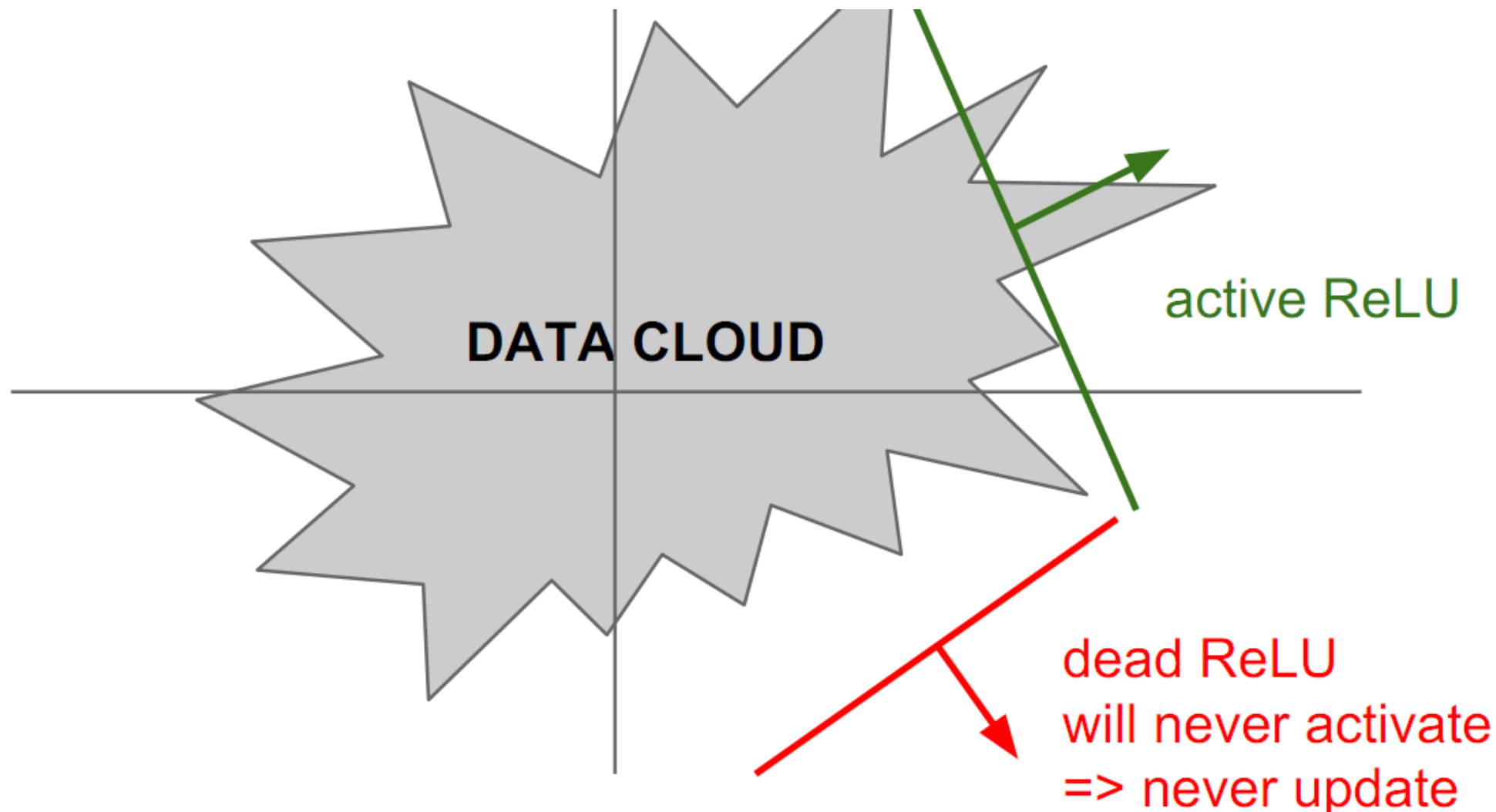


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

Activation Functions

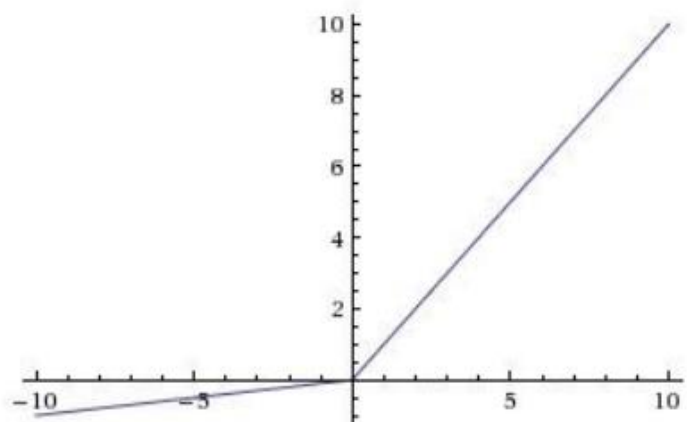


Activation Functions

Activation Functions

[Mass et al., 2013]

[He et al., 2015]



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

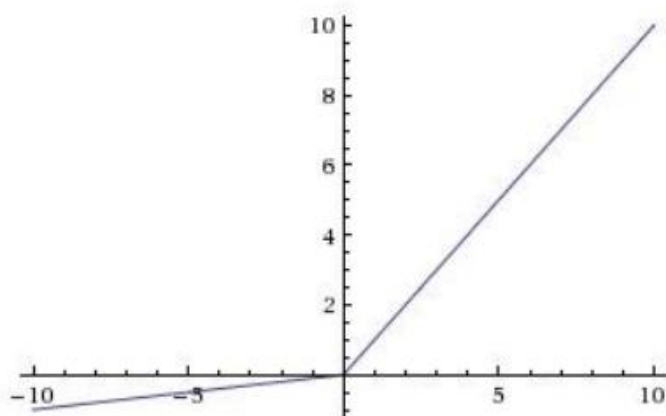
$$f(x) = \max(0.01x, x)$$

Activation Functions

Activation Functions

[Mass et al., 2013]

[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

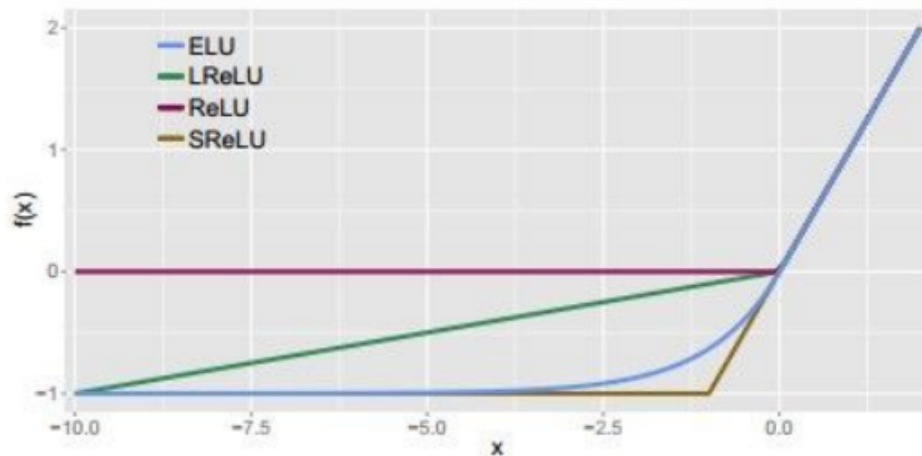
backprop into α
(parameter)

Activation Functions

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires $\exp()$

Activation Functions

Maxout “Neuron”

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

Activation Functions

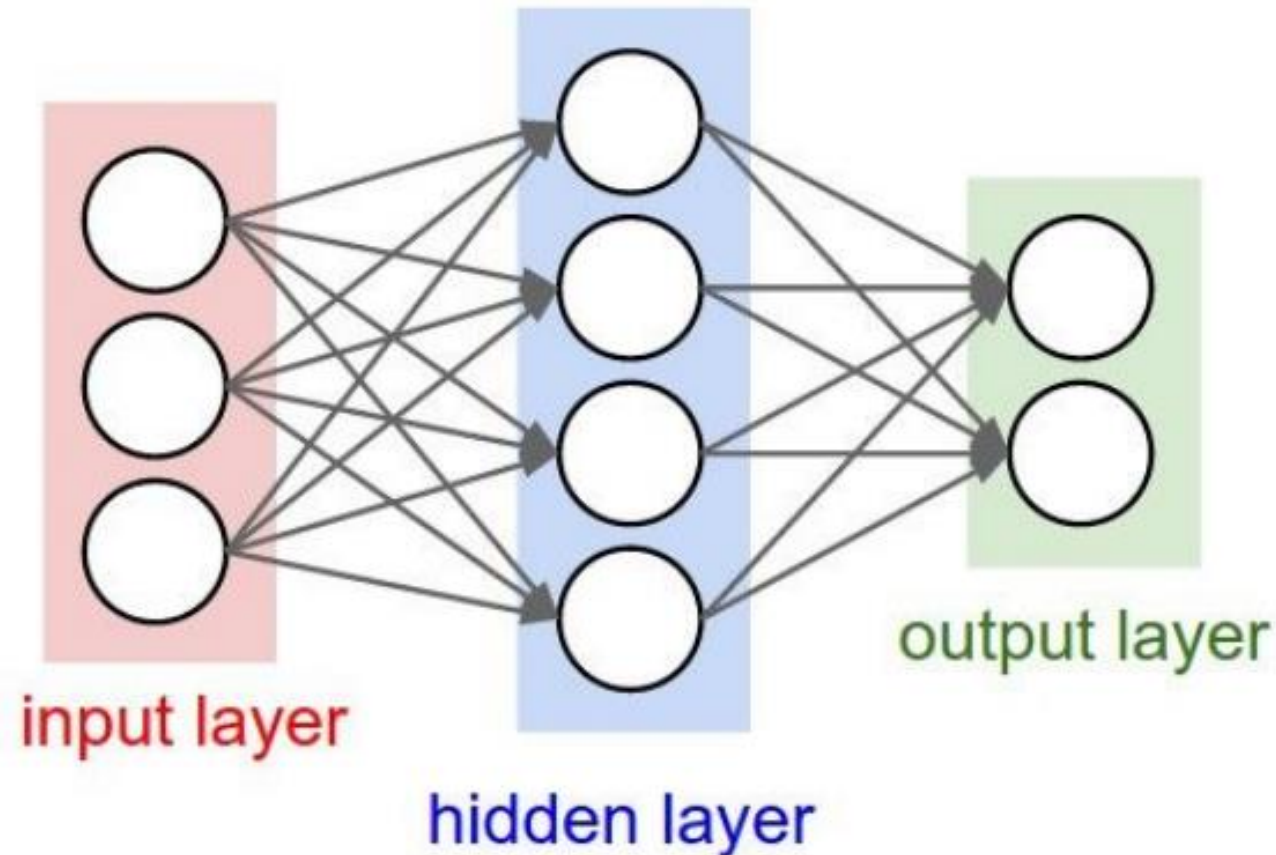
In practice:

- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU / Maxout / ELU**
- Try out **tanh** but don't expect much
- **Don't use sigmoid**

Weight Initialization

Weight Initialization

- Q: what happens when $W=0$ init is used?



Weight Initialization

- First idea: **Small random numbers**
(gaussian with zero mean and $1e-2$ standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Weight Initialization

- First idea: **Small random numbers**
(gaussian with zero mean and $1e-2$ standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Weight Initialization

Lets look at
some
activation
statistics

E.g. 10-layer net with
500 neurons on each
layer, using tanh non-
linearities, and
initializing as
described in last slide.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer

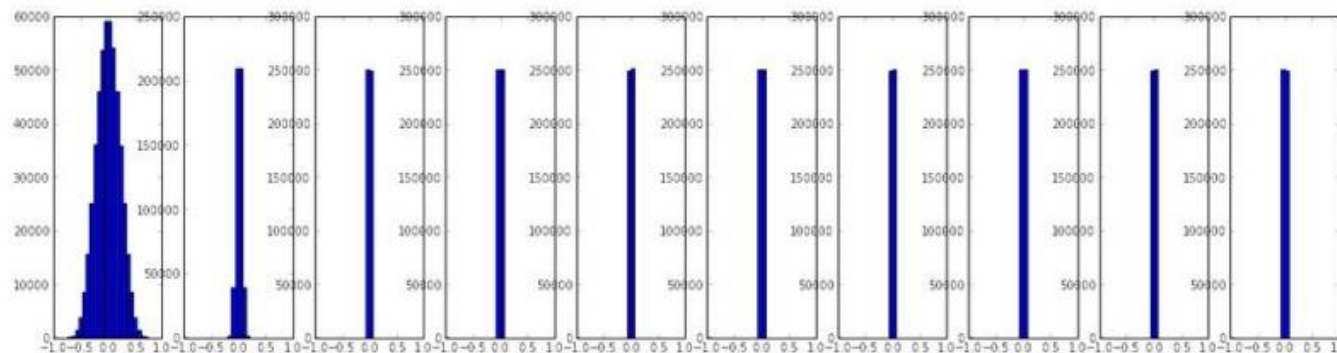
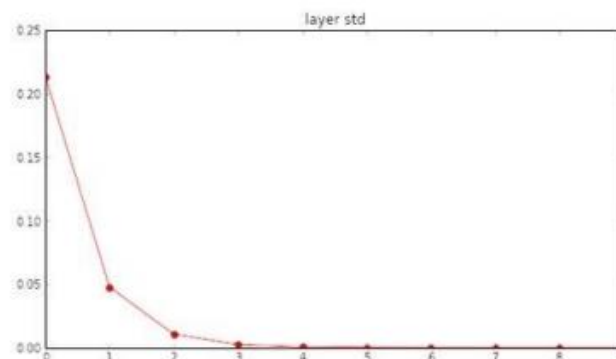
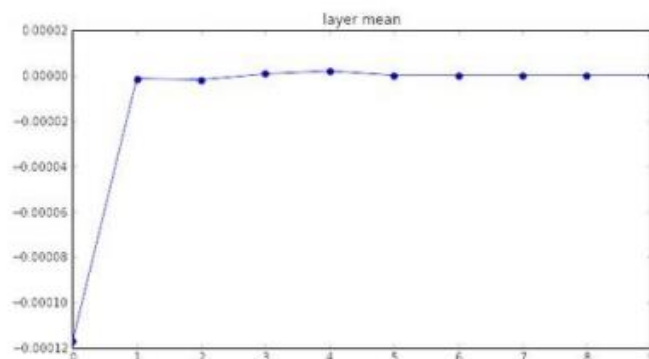
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Weight Initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations
become zero!

Q: think about the
backward pass.
What do the
gradients look like?

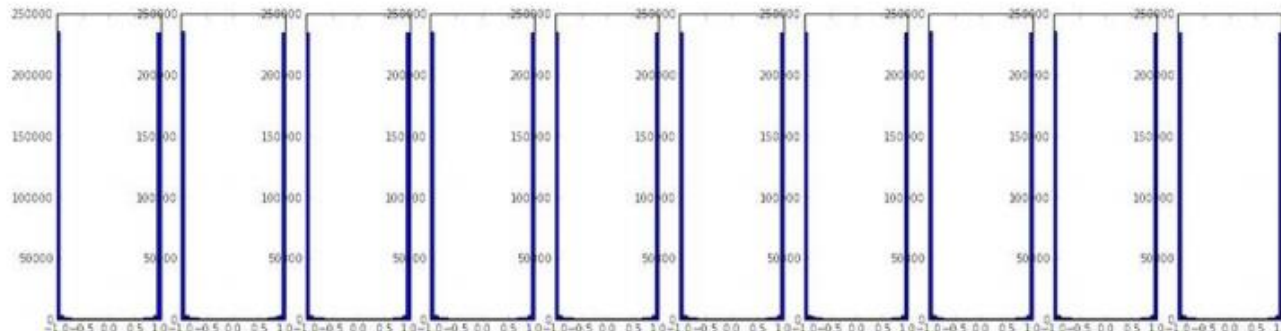
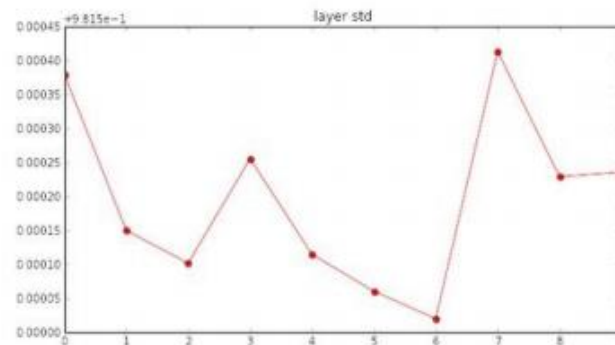
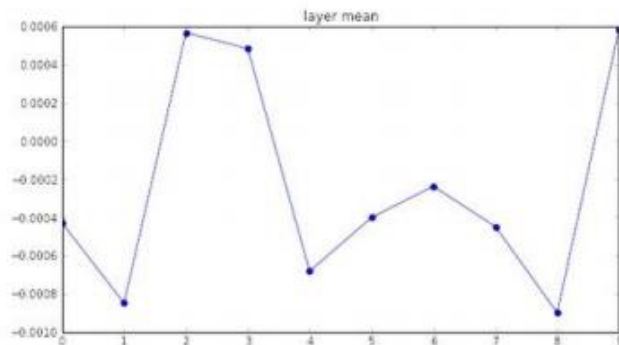
Hint: think about backward
pass for a $W \cdot X$ gate.

Weight Initialization

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

input layer had mean 0.001800 and std 1.001311
 hidden layer 1 had mean -0.000430 and std 0.981879
 hidden layer 2 had mean -0.000849 and std 0.981649
 hidden layer 3 had mean 0.000566 and std 0.981601
 hidden layer 4 had mean 0.000483 and std 0.981755
 hidden layer 5 had mean -0.000682 and std 0.981614
 hidden layer 6 had mean -0.000401 and std 0.981560
 hidden layer 7 had mean -0.000237 and std 0.981520
 hidden layer 8 had mean -0.000448 and std 0.981913
 hidden layer 9 had mean -0.000899 and std 0.981728
 hidden layer 10 had mean 0.000584 and std 0.981736

*1.0 instead of *0.01



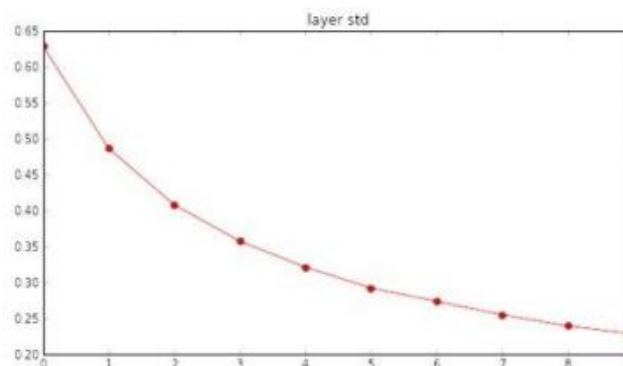
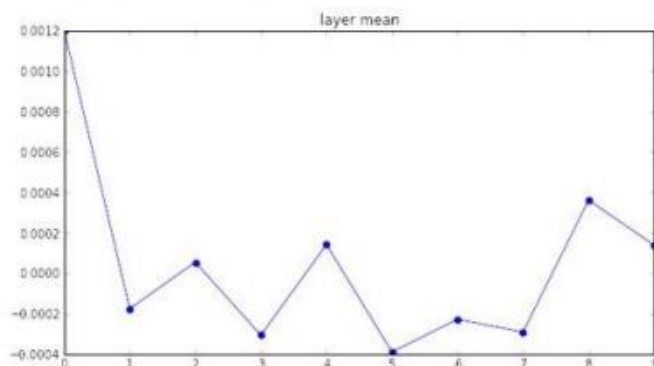
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Weight Initialization

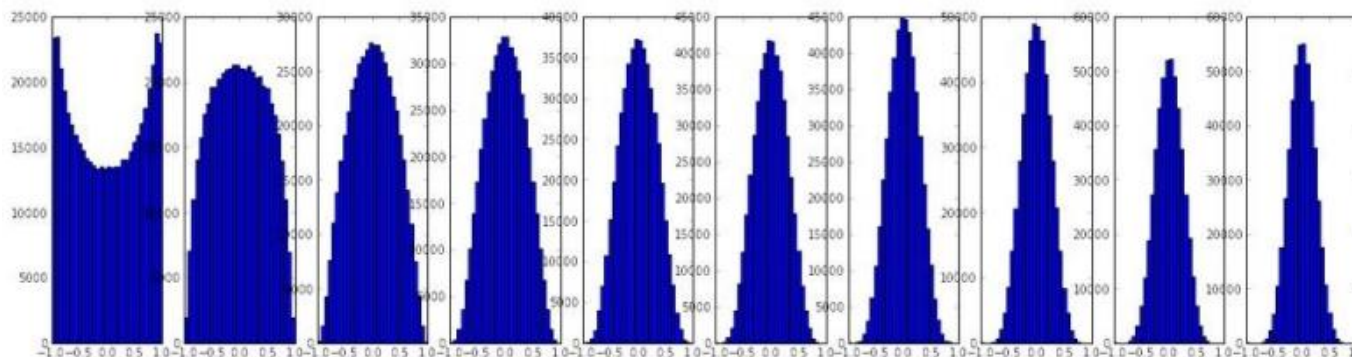
input layer had mean 0.001800 and std 1.001311
 hidden layer 1 had mean 0.001198 and std 0.627953
 hidden layer 2 had mean -0.000175 and std 0.486051
 hidden layer 3 had mean 0.000055 and std 0.407723
 hidden layer 4 had mean -0.000306 and std 0.357108
 hidden layer 5 had mean 0.000142 and std 0.320917
 hidden layer 6 had mean -0.000389 and std 0.292116
 hidden layer 7 had mean -0.000228 and std 0.273387
 hidden layer 8 had mean -0.000291 and std 0.254935
 hidden layer 9 had mean 0.000361 and std 0.239266
 hidden layer 10 had mean 0.000139 and std 0.228008

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

“Xavier initialization”
 [Glorot et al., 2010]



Reasonable initialization.
 (Mathematical derivation
 assumes linear activations)

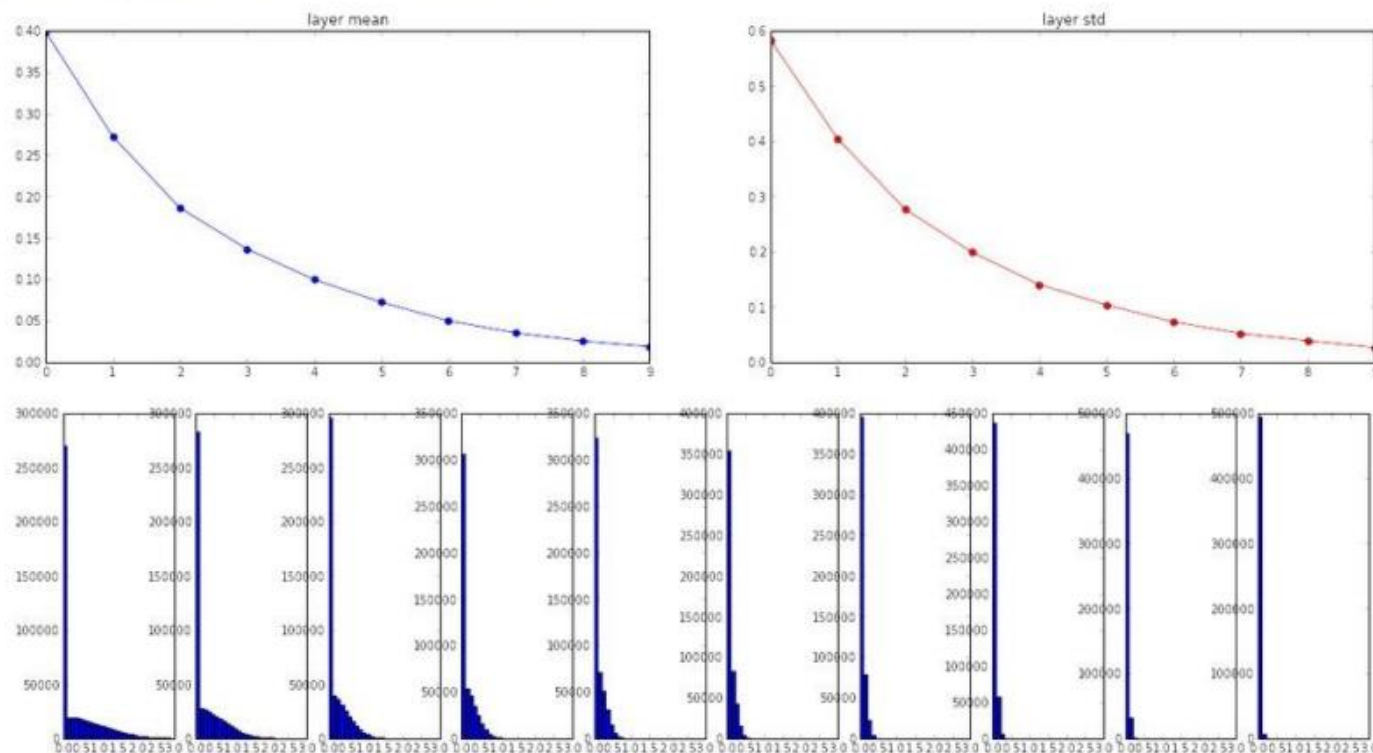


Weight Initialization

input layer had mean 0.000501 and std 0.999444
 hidden layer 1 had mean 0.398623 and std 0.582273
 hidden layer 2 had mean 0.272352 and std 0.403795
 hidden layer 3 had mean 0.186076 and std 0.276912
 hidden layer 4 had mean 0.136442 and std 0.198685
 hidden layer 5 had mean 0.099568 and std 0.140299
 hidden layer 6 had mean 0.072234 and std 0.103280
 hidden layer 7 had mean 0.049775 and std 0.072748
 hidden layer 8 had mean 0.035138 and std 0.051572
 hidden layer 9 had mean 0.025404 and std 0.038583
 hidden layer 10 had mean 0.018408 and std 0.026076

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.

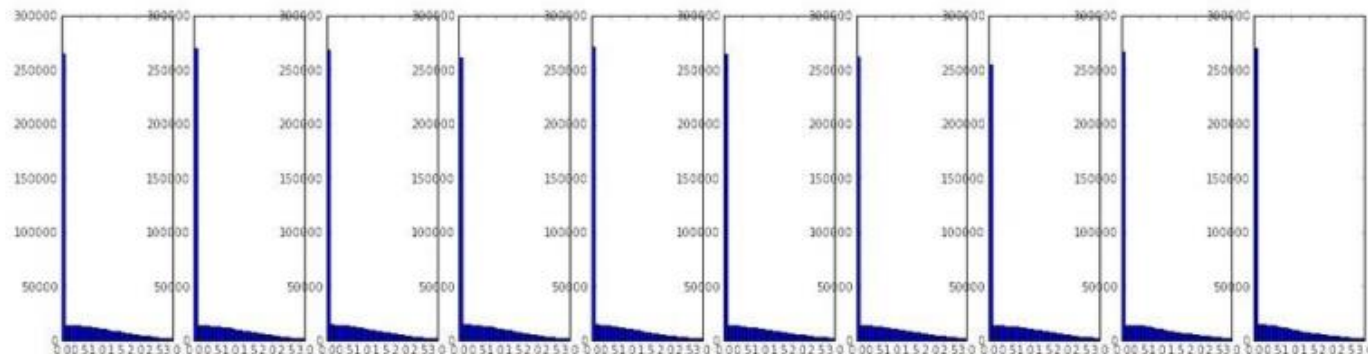
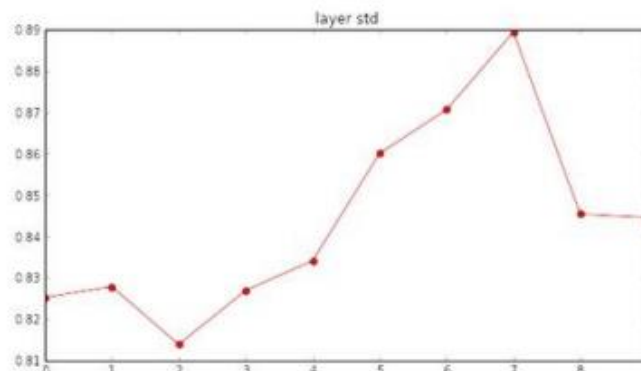
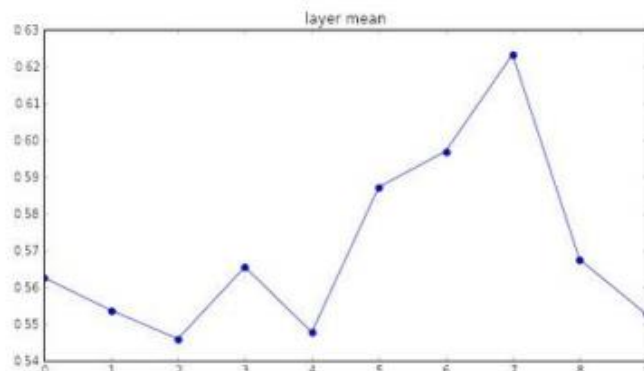


Weight Initialization

input layer had mean 0.000501 and std 0.999444
 hidden layer 1 had mean 0.562488 and std 0.825232
 hidden layer 2 had mean 0.553614 and std 0.827835
 hidden layer 3 had mean 0.545867 and std 0.813855
 hidden layer 4 had mean 0.565396 and std 0.826902
 hidden layer 5 had mean 0.547678 and std 0.834092
 hidden layer 6 had mean 0.587103 and std 0.860035
 hidden layer 7 had mean 0.596867 and std 0.870610
 hidden layer 8 had mean 0.623214 and std 0.889348
 hidden layer 9 had mean 0.567498 and std 0.845357
 hidden layer 10 had mean 0.552531 and std 0.844523

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

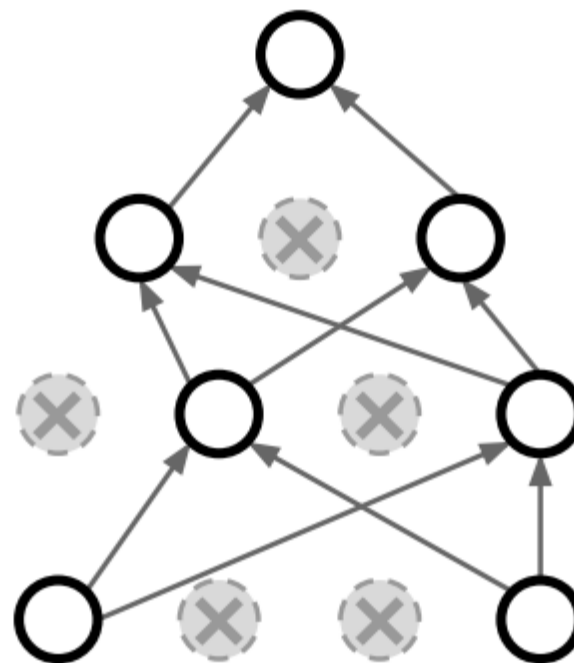
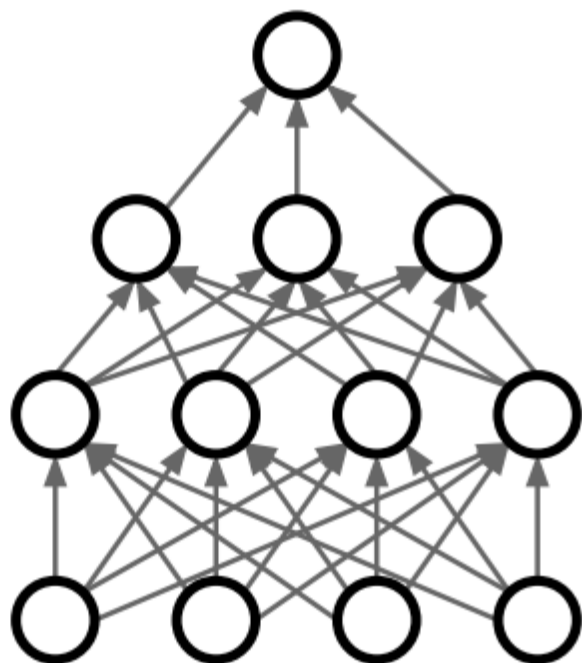
He et al., 2015
 (note additional /2)



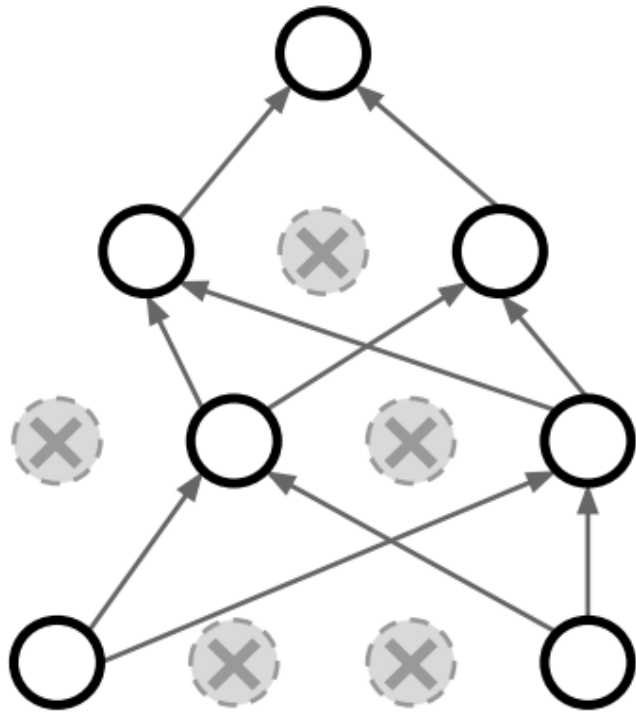
Dropout

Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



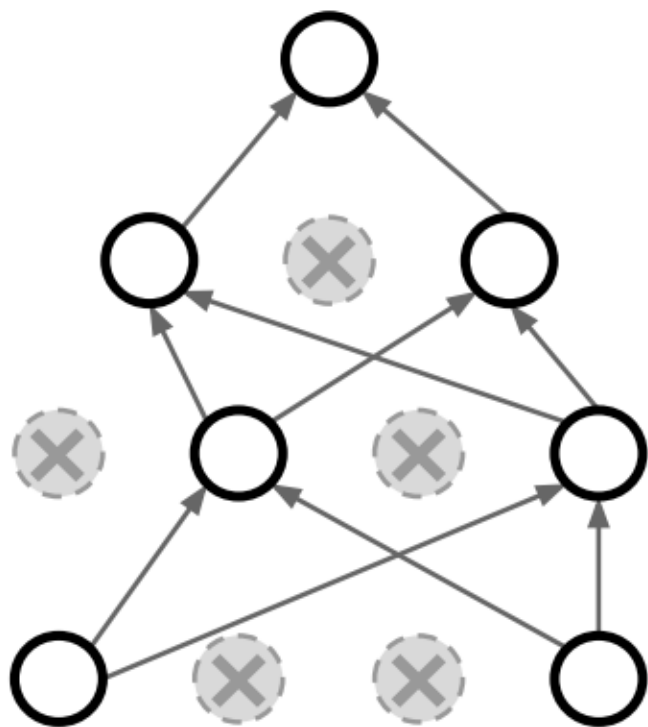
Dropout



Forces the network to have a redundant representation;
Prevents co-adaptation of features



Dropout



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Batch Normalization

Batch Normalization

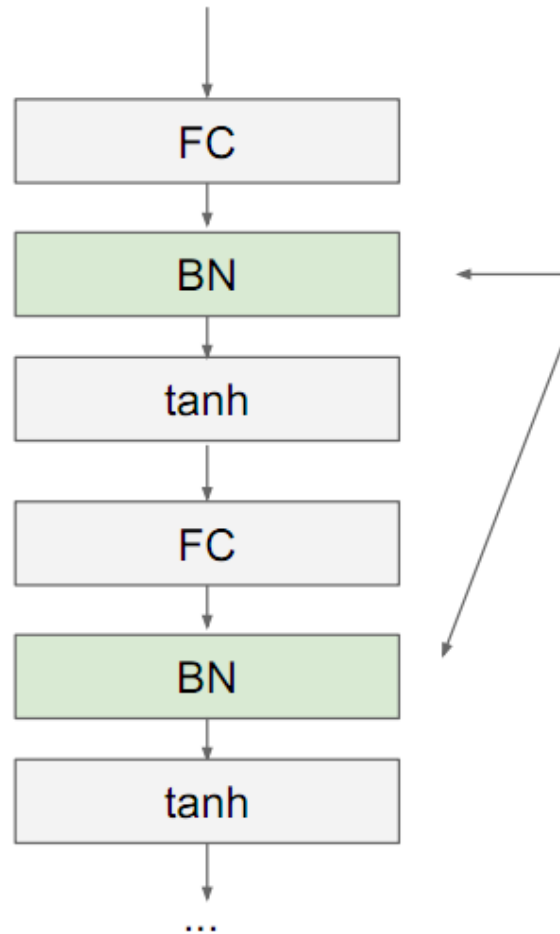
“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

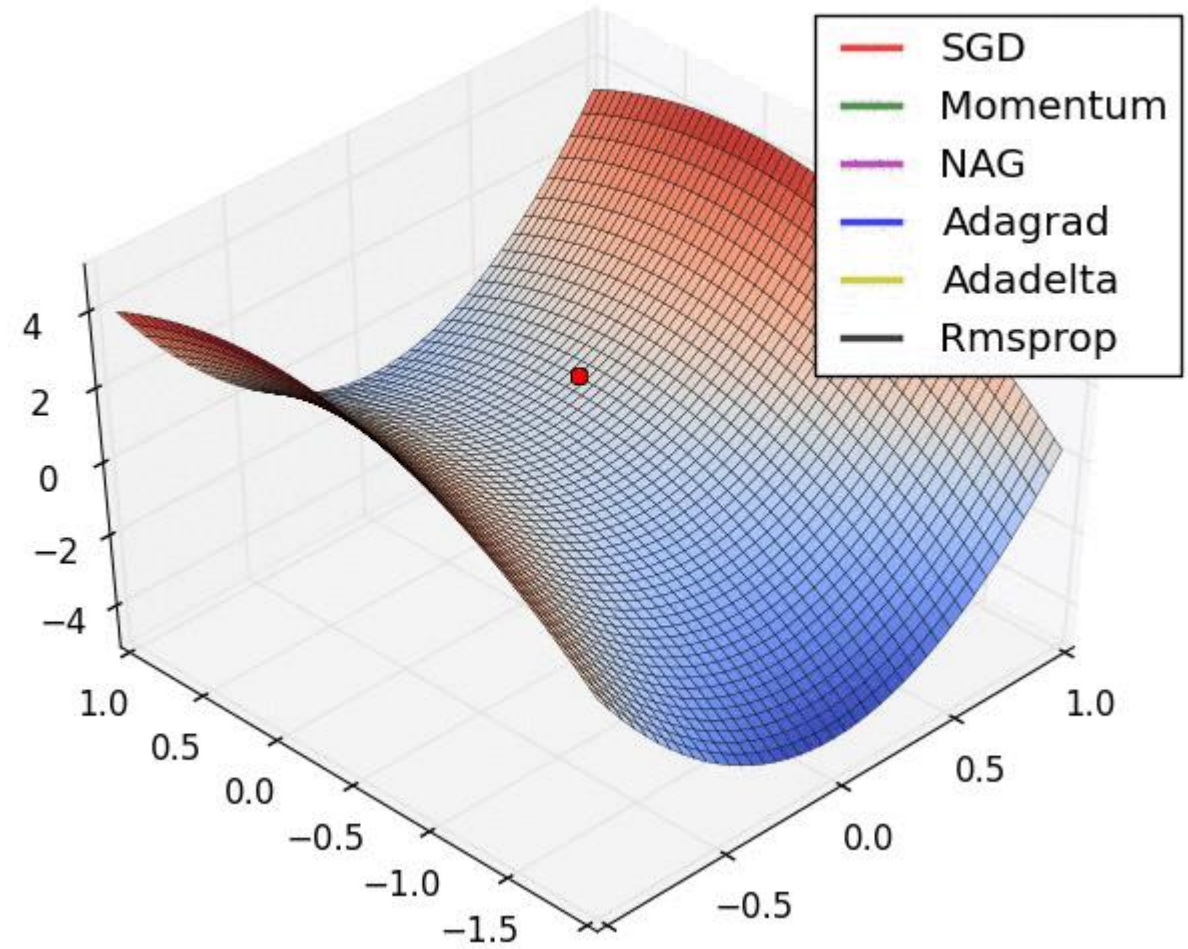
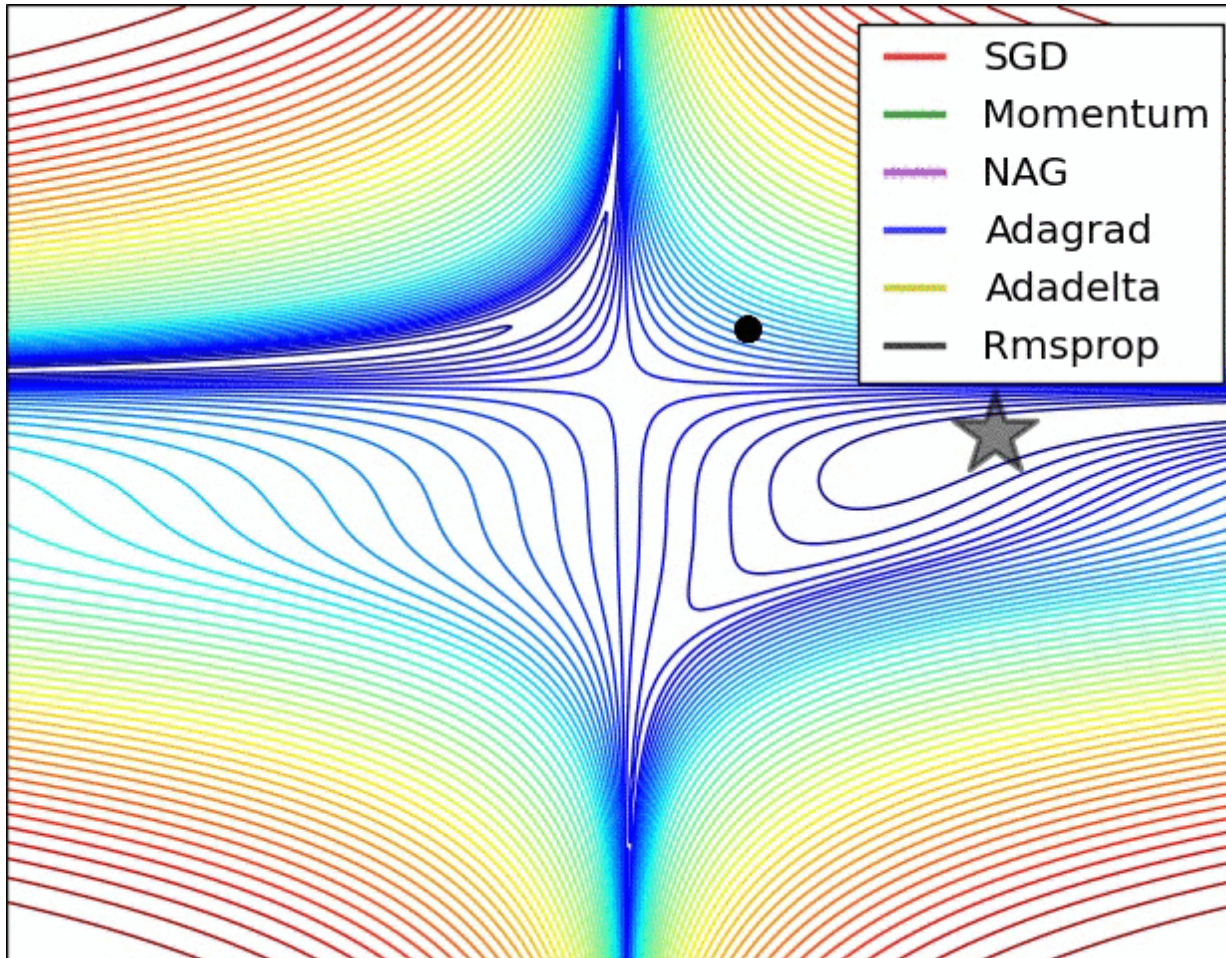
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

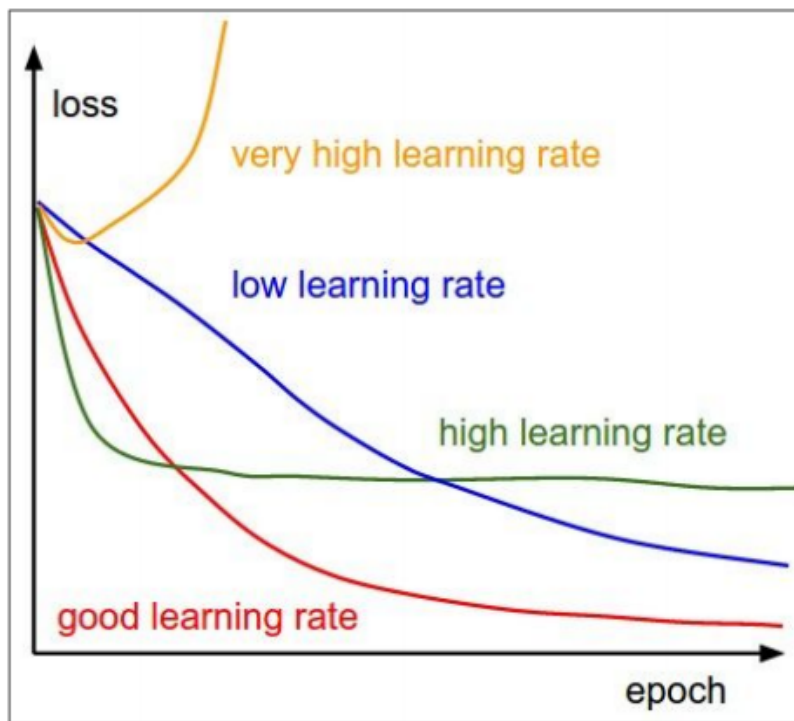
Optimization and Learning rate Schedules

Optimization



Optimization

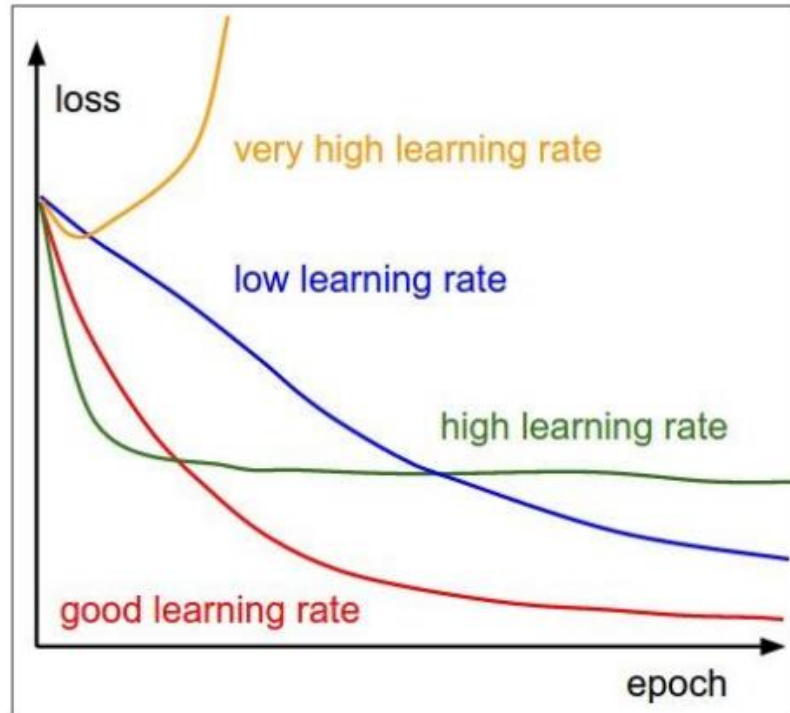
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning rate Schedules

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> **Learning rate decay over time!**

step decay:

e.g. decay learning rate by half every few epochs.

exponential decay:

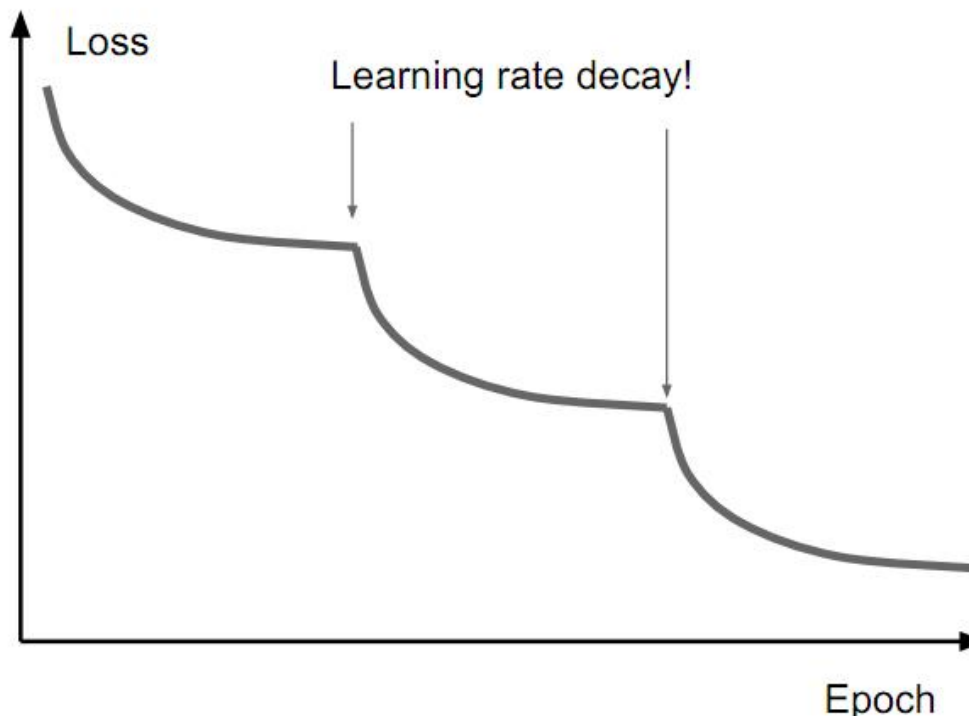
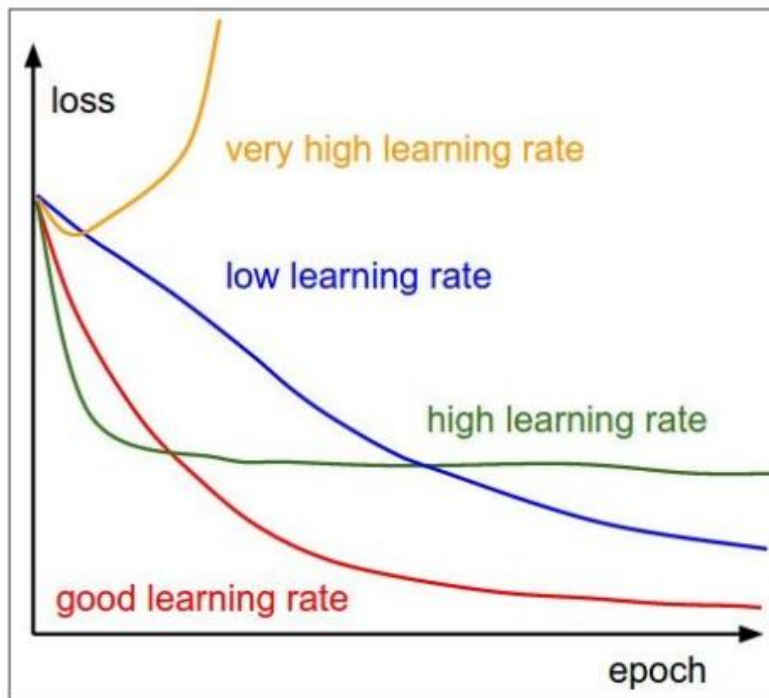
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Learning rate Schedules

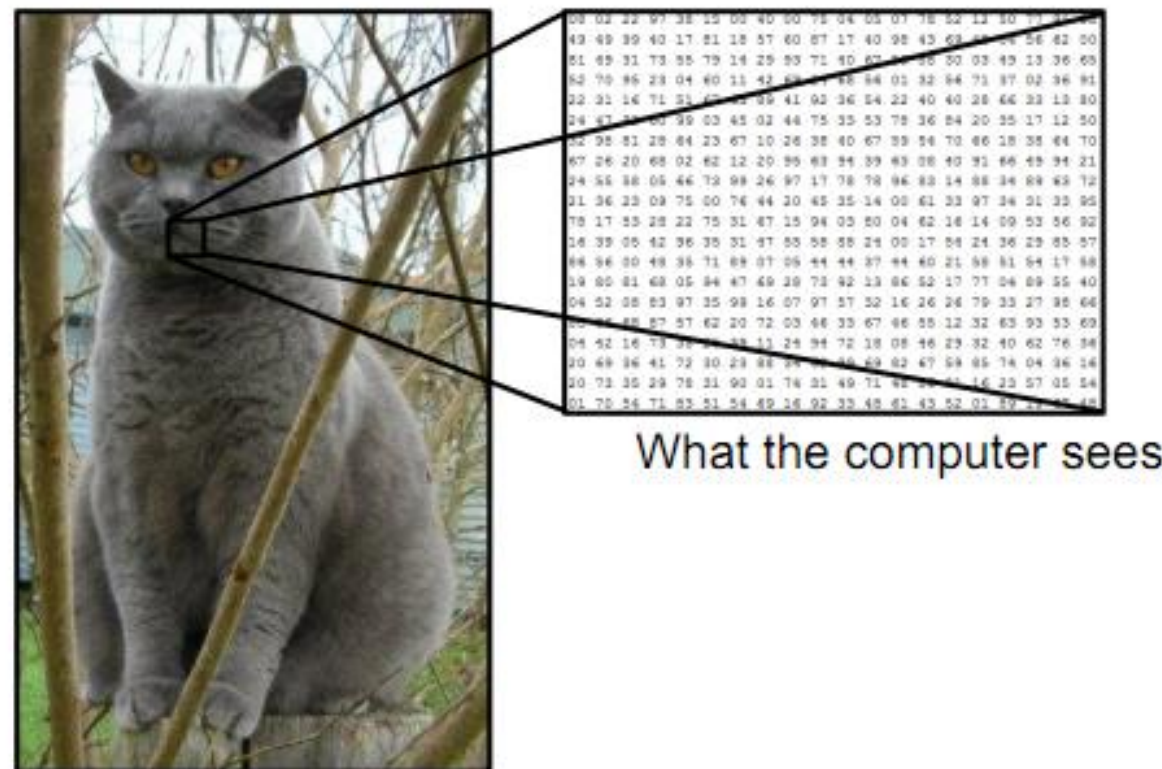
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Data Augmentation

Data Augmentation

- i.e. simulating “fake” data
- explicitly encoding image transformations that shouldn't change object identity.



What the computer sees

Data Augmentation

Flip horizontally



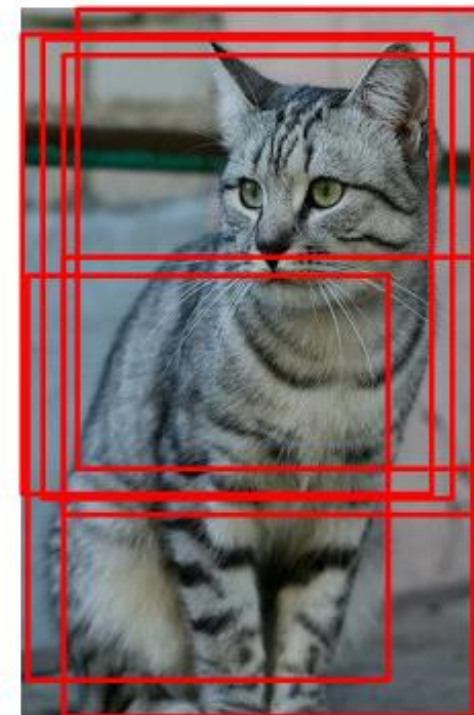
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range $[256, 480]$
2. Resize training image, short side = L
3. Sample random 224×224 patch



Data Augmentation

Color jittering

(maybe even contrast jittering, etc.)

- Simple: Change contrast small amounts, jitter the color distributions, etc.
- Vignette,... (go crazy)



Data Augmentation

Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

Tài liệu tham khảo

1. Stanford cs231n: CNNs for Visual Recognition

<http://cs231n.stanford.edu/>

2. Xavier Initialization

<https://mnsgrg.com/2017/12/21/xavier-initialization/>