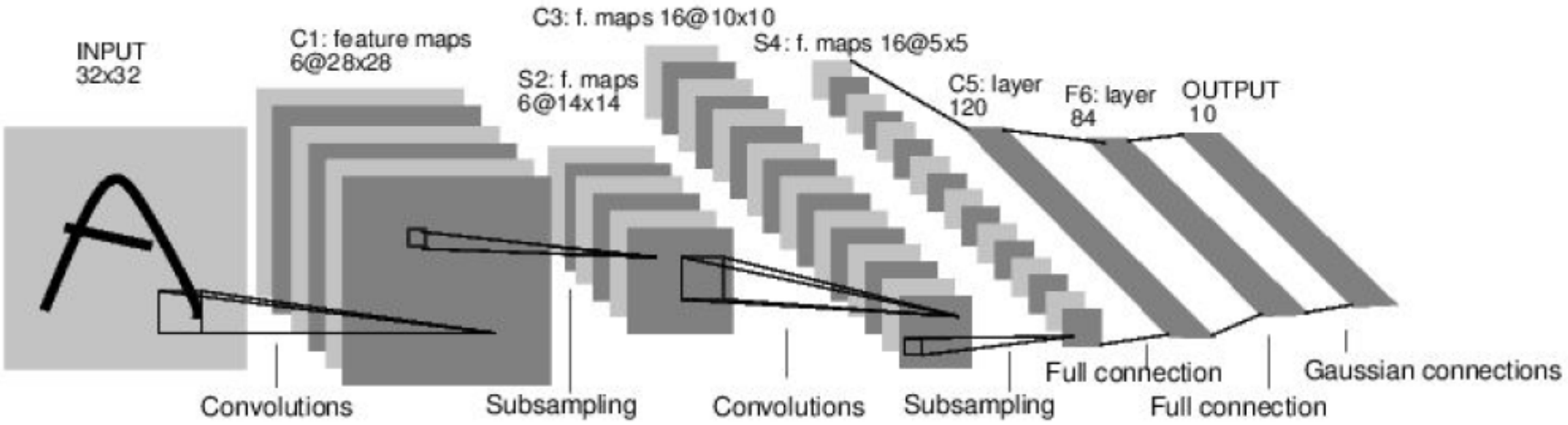
Advanced Practical Machine Learning – 67750

Convolutional Neural Networks: A Practical Introduction

Or Sharir

Slides from:
Nadav Cohen, Fei-Fei Li, Andrej Karpathy,
Justin Johnson, Yann LeCun, Kaiming He

Convolutional Neural Networks (ConvNets)



[LeNet-5, LeCun 1990]

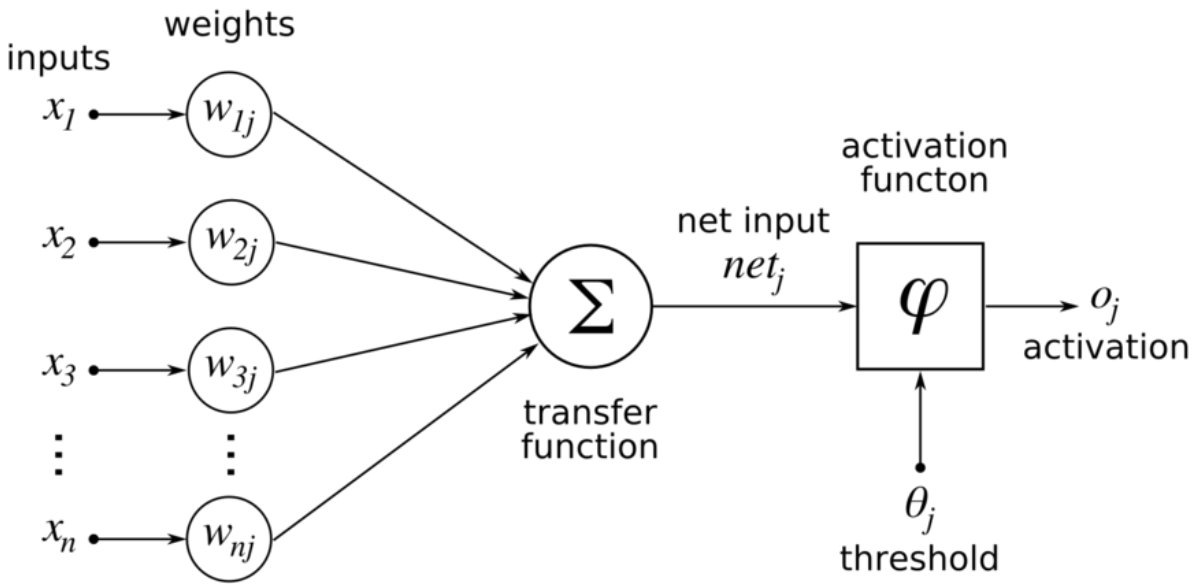
Have been around for a while…

On the Agenda

- The basics:
  - What are ConvNets?
  - How to efficiently train them?
  - How they differ from classical models?
- Advanced architectures
- End-to-end learning principle
- ConvNets as generative models (if we'll have time)

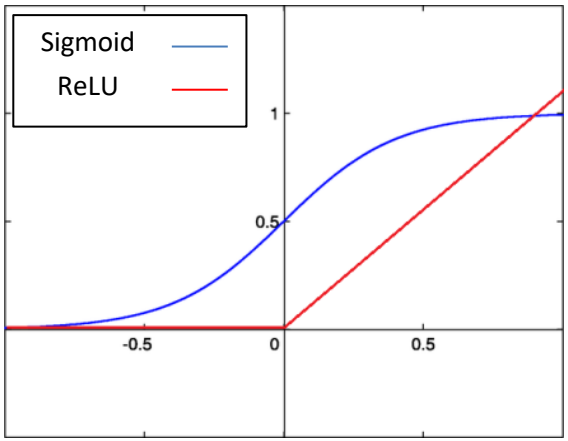
Recap – Neural Networks

Artificial Neuron



$$\Leftrightarrow o_j \stackrel{\text{def}}{=} \varphi\left(\sum_{i=1}^{n} w_i x_i\right)$$

Activation functions:



Sigmoid

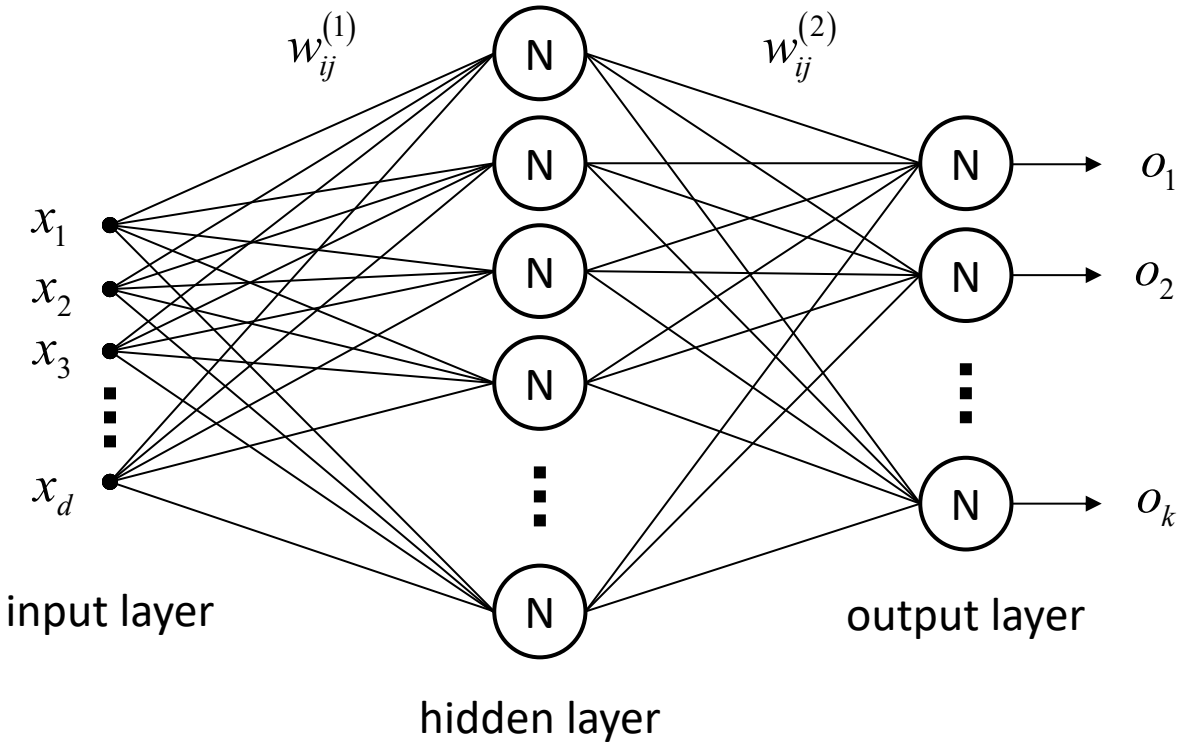
$$\varphi(z) = \frac{1}{1+e^{-z}}$$

ReLU

$$\varphi(z) = \max(0, z)$$

most common in ConvNets

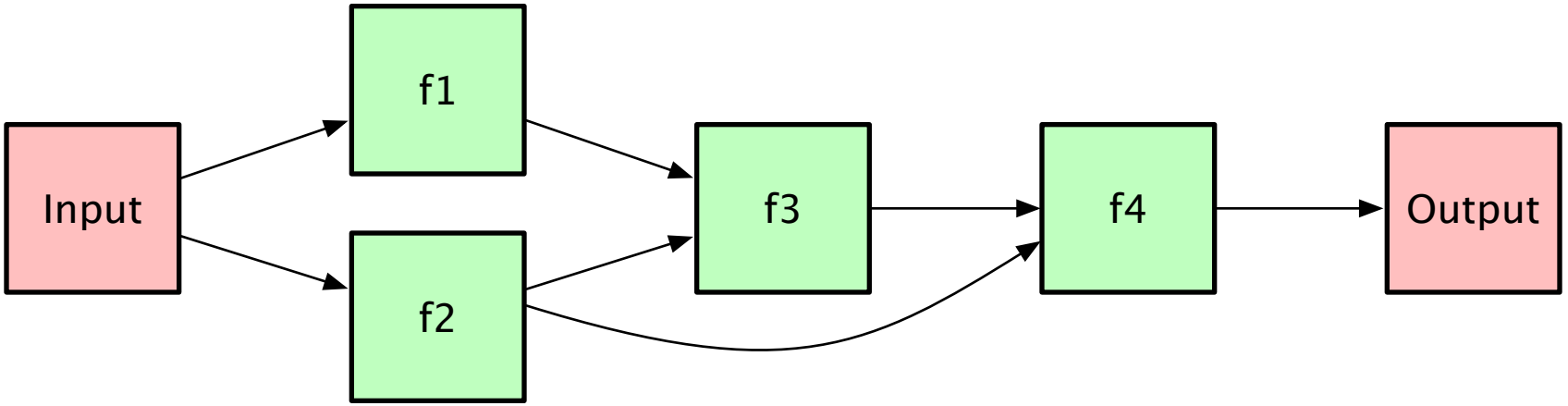
Artificial Neural Network



Can approximate any function (given sufficient size), even with a single hidden layer

A Broader Definition

Any directed acyclic graph of (sub-)differentiable operations:



- Each node is called a “layer” operating on multi-dimensional arrays called “tensors”.
- A layer could simply be a group of neurons with some predetermined connectivity pattern.
- … but not limited to such operations!

Neural Network – Training

Weights $\left\{w_{ij}^{(l)}\right\}$ are learned via minimization of training loss that reflects task at hand.

$S = \left\{\left(x^m, y^m\right)\right\}_{m=1}^{M}$ - Given labeled instances (training set).

$L_S\left(w_{ij}^{(l)}\right)$ - Training loss. Examples:

$$L_S\left(\left\{w_{ij}^{(l)}\right\}\right) = \sum_{m=1}^{M}\left\|o\left(x^m\right) - y^m\right\|_2^2$$ Square Loss (regression)

$$L_S\left(\left\{w_{ij}^{(l)}\right\}\right) = \sum_{m=1}^{M}\left\{\log\left(\sum_{r=1}^{k} e^{o_r\left(x^m\right)}\right) - o_{y^m}\left(x^m\right)\right\}$$ Softmax Loss (classification)

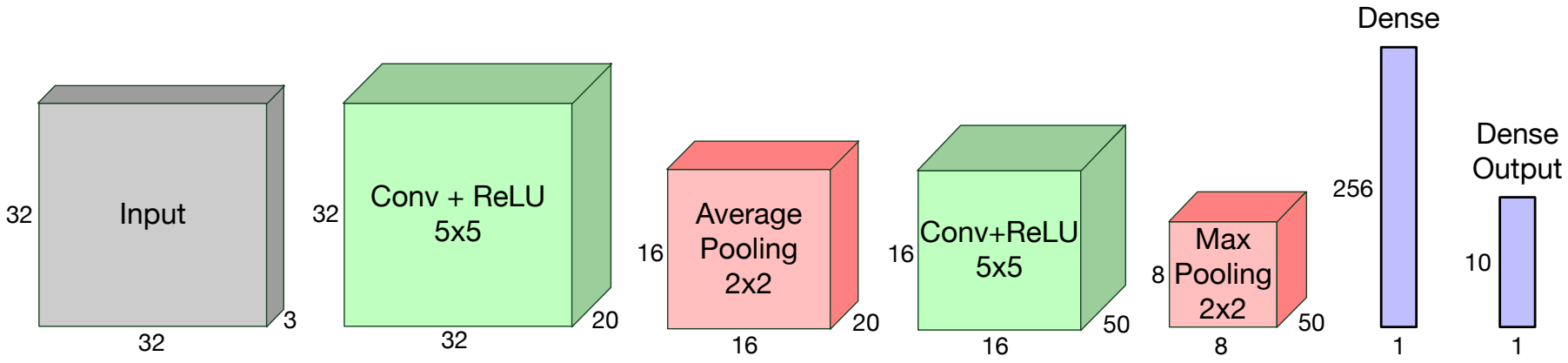
$\min_{w_{ij}^{(l)}} L_S\left(w_{ij}^{(l)}\right)$ - Non-convex program. Typically tackled with Stochastic Gradient Descent (via back-propagation).

Classic
ConvNet Architecture

Basic Architecture

- The most common layers:
  - Convolutional
  - Average / Max Pooling
  - Dense (a.k.a. Fully Connected)

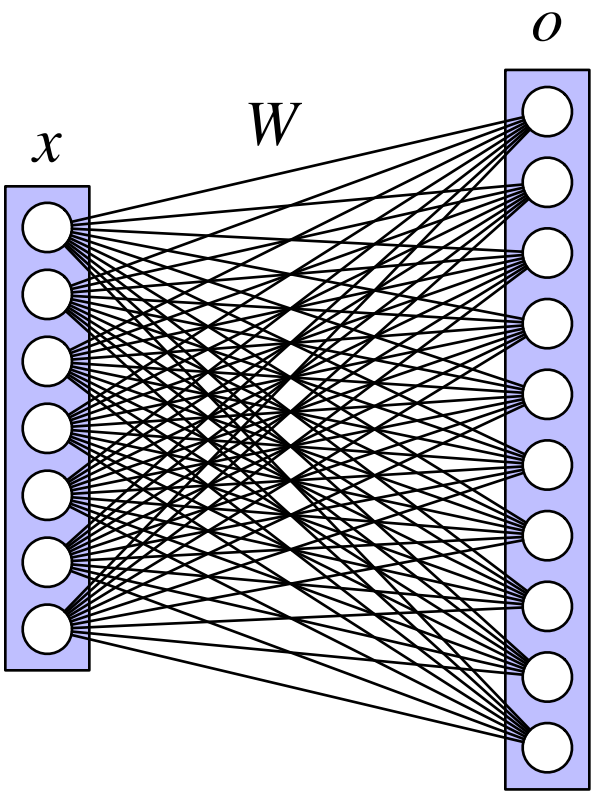
- The typical architecture: alternating conv and pooling layers, followed by a sequence of dense layers at the end.



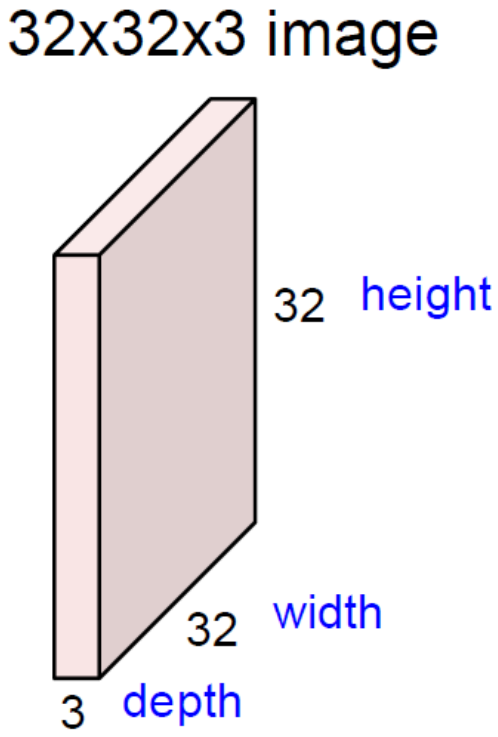
Dense (Fully Connected) Layer

- Contains neurons that connect to the entire input volume, as in ordinary neural networks

$$o = \varphi(Wx)$$

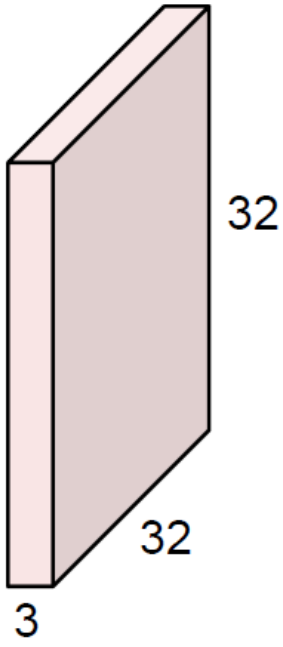


Convolutional Layer

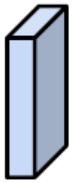


Convolutional Layer

32x32x3 image

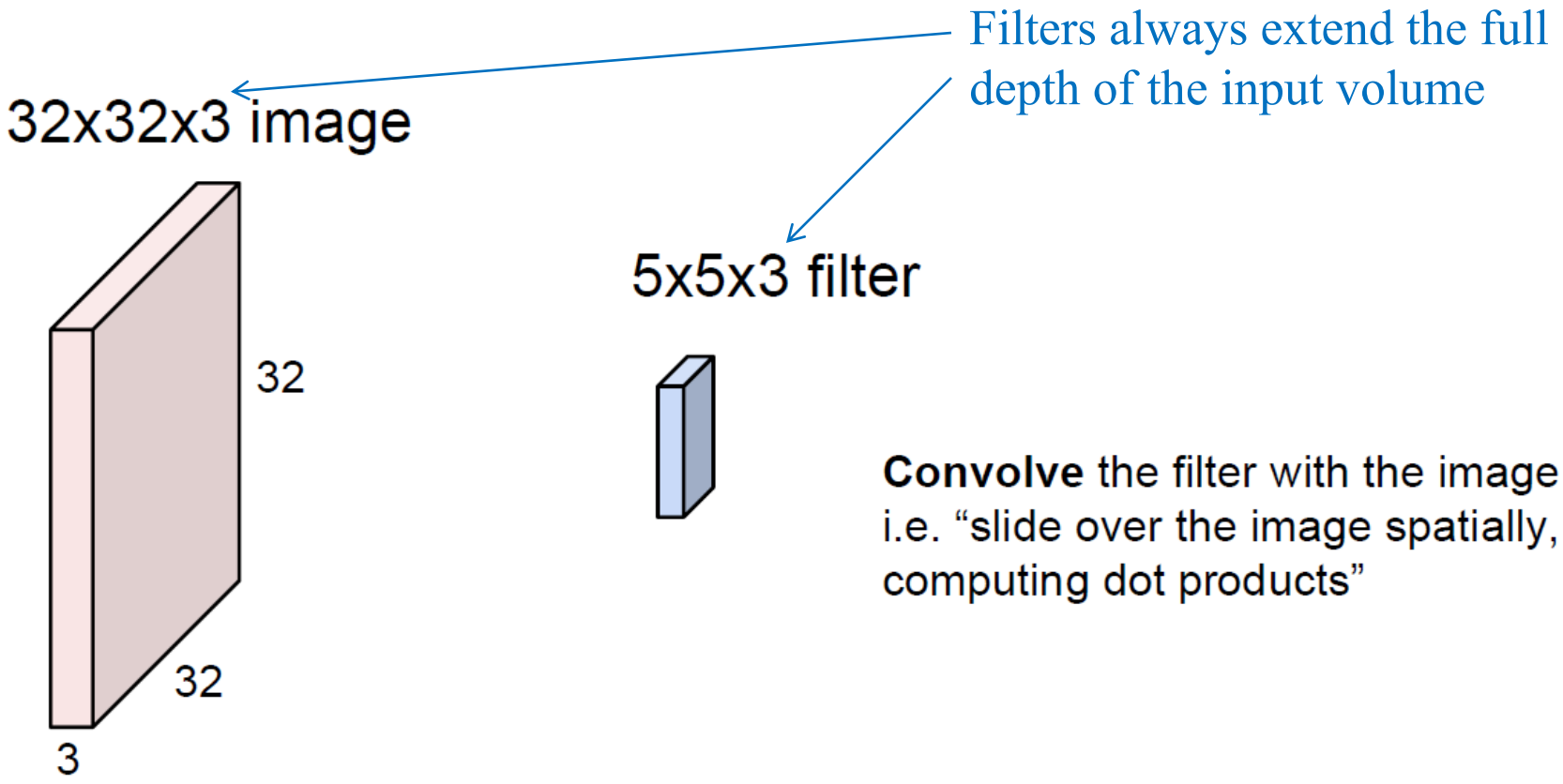


5x5x3 filter

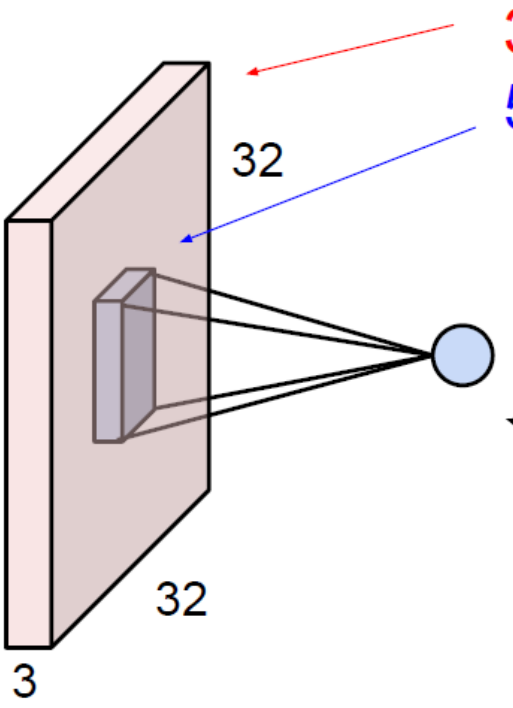


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolutional Layer



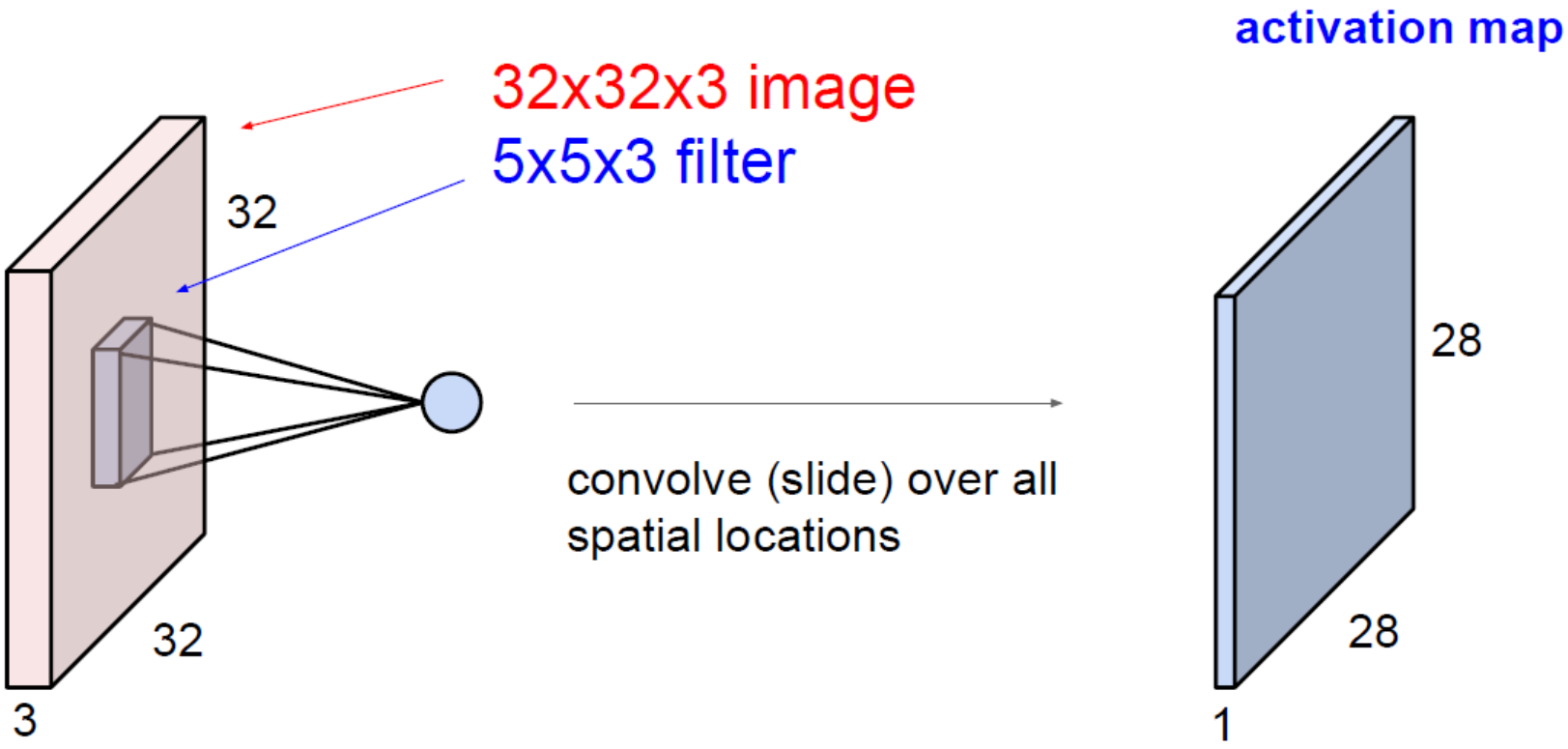
Convolutional Layer



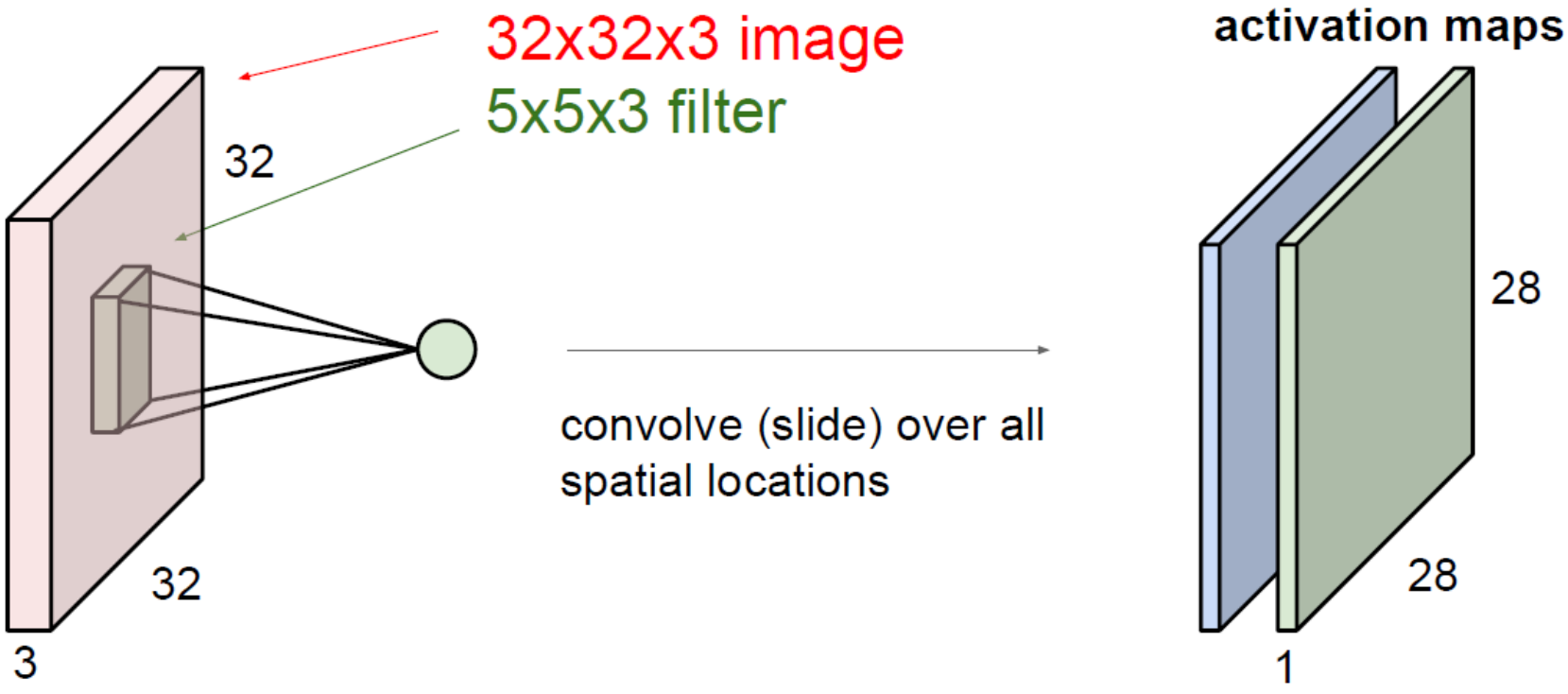
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Convolutional Layer



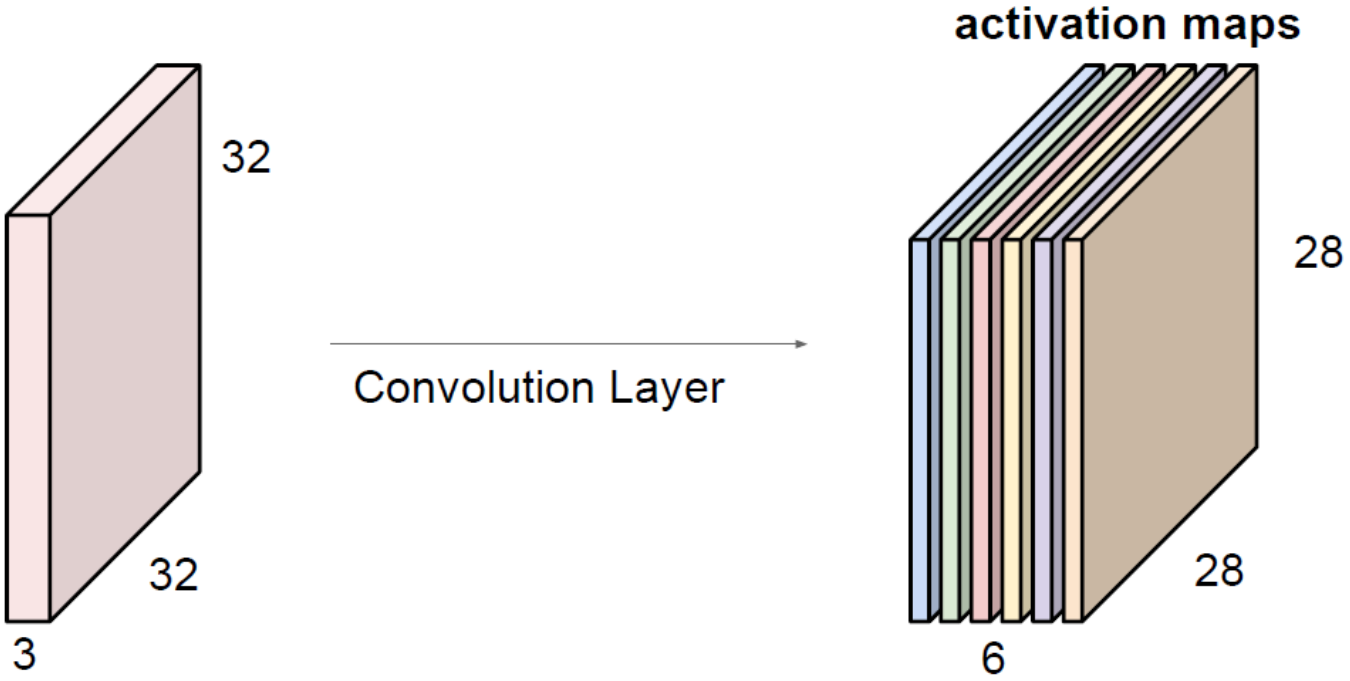
Convolutional Layer



consider a second, green filter

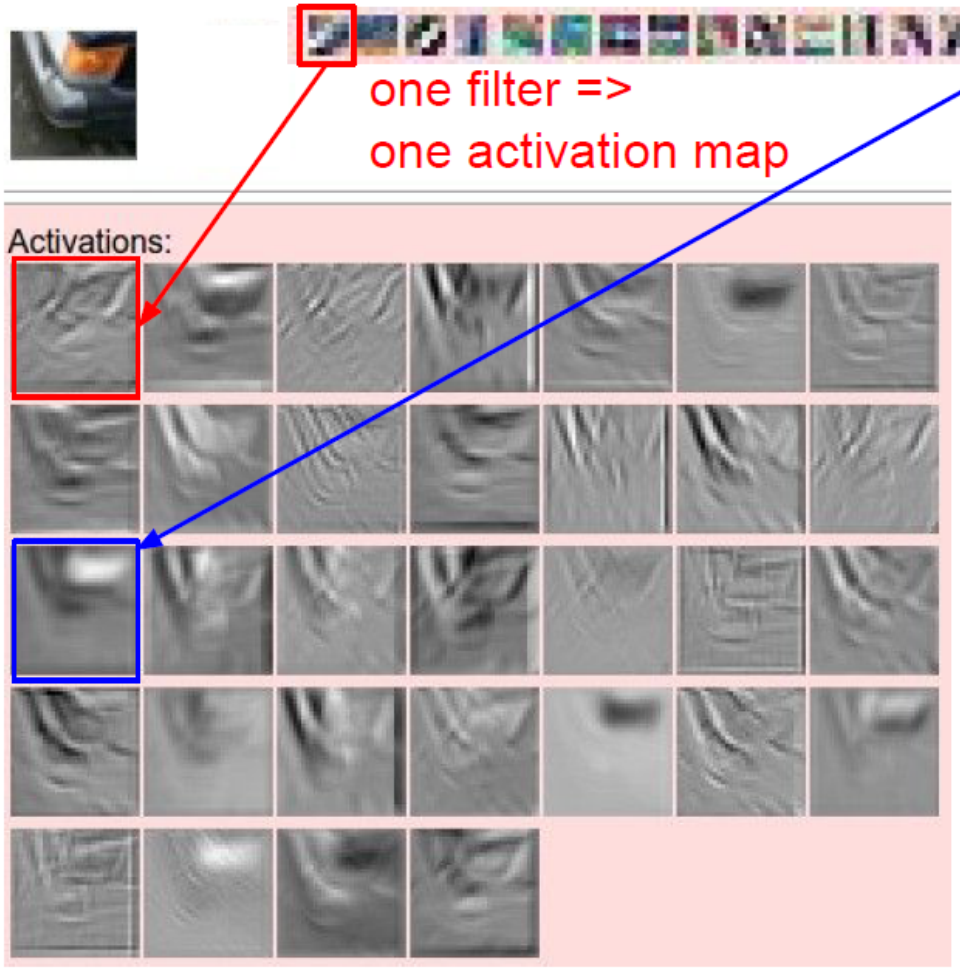
Convolutional Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

Convolutional Layer



example 5x5 filters
(32 total)

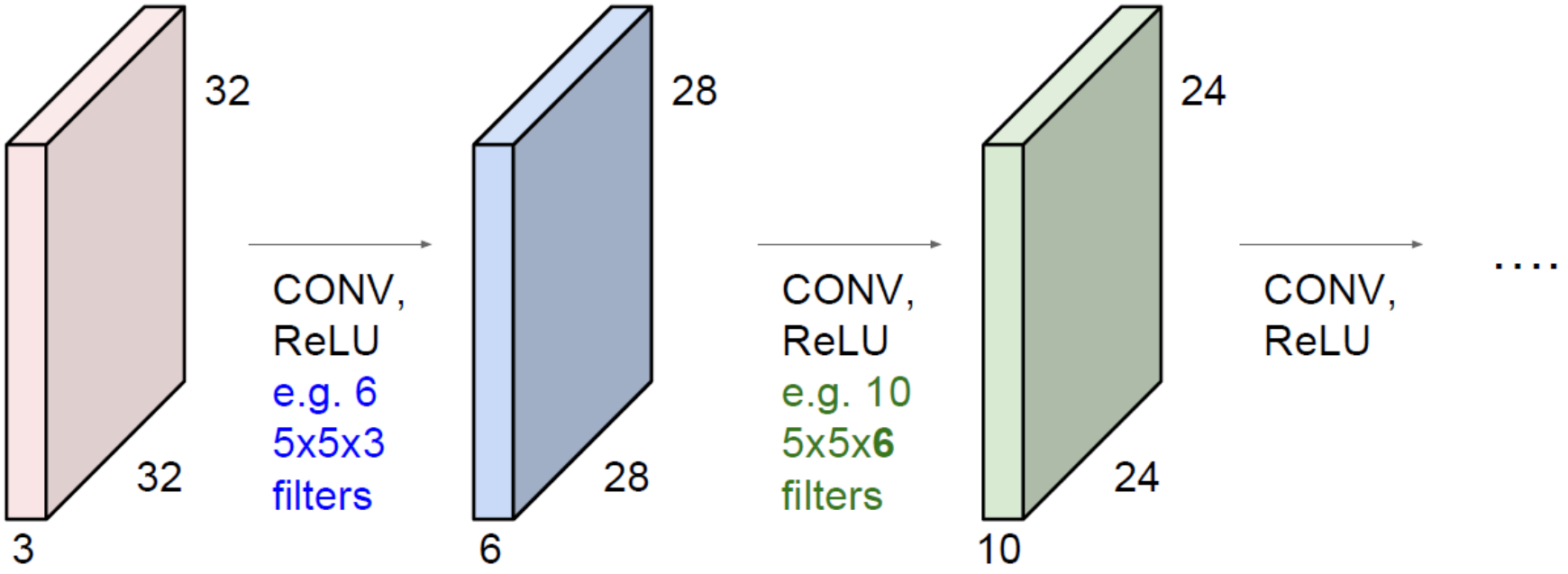
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

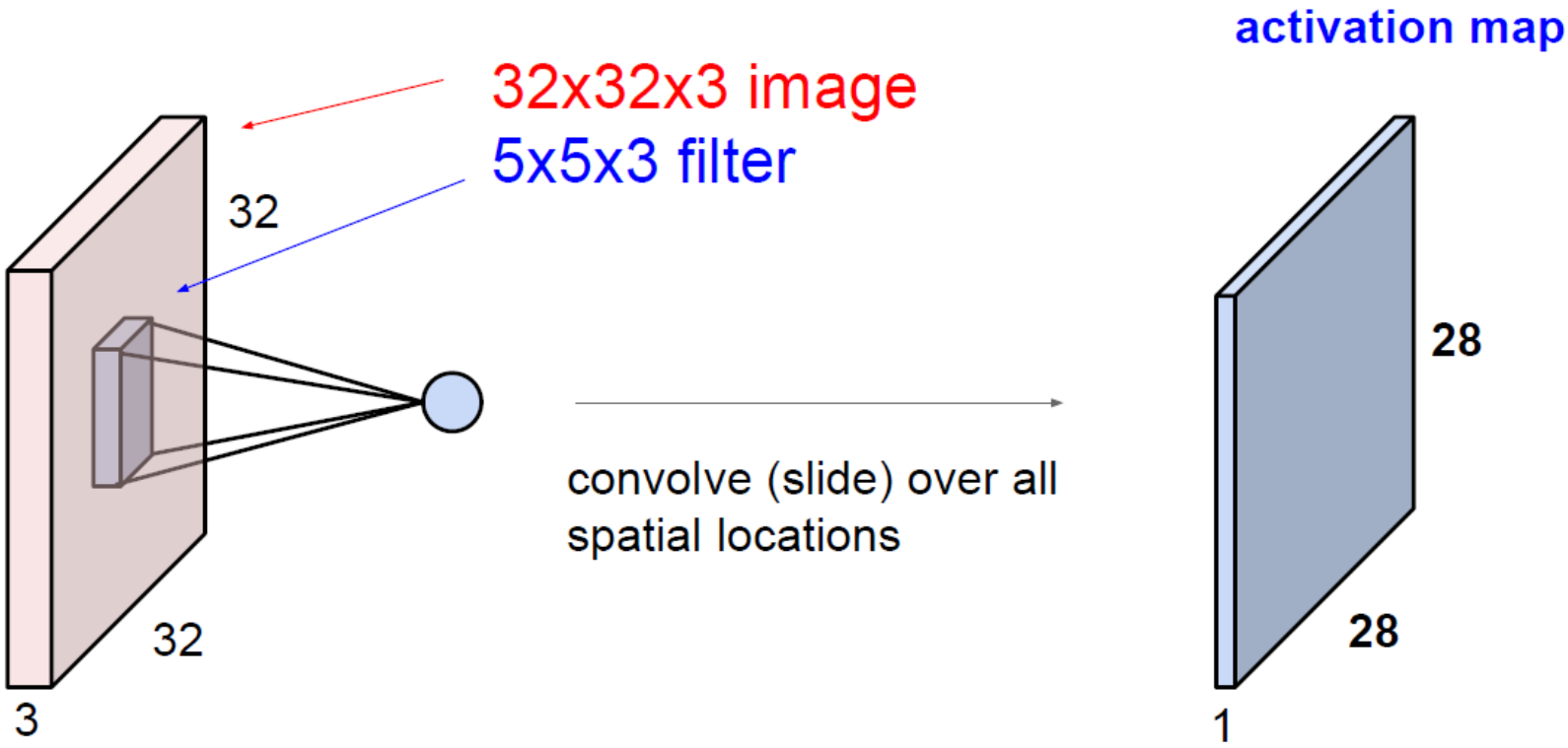
Convolutional Layer

Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



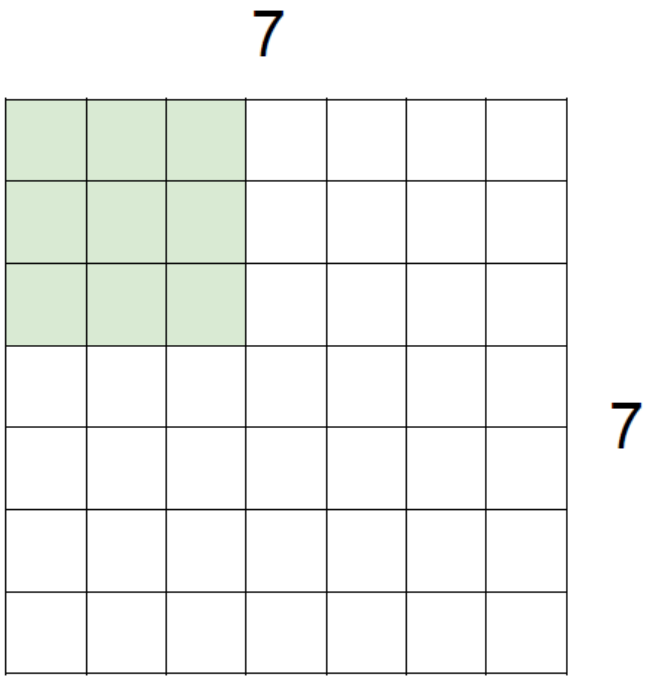
Convolutional Layer

A closer look at spatial dimensions:



Convolutional Layer

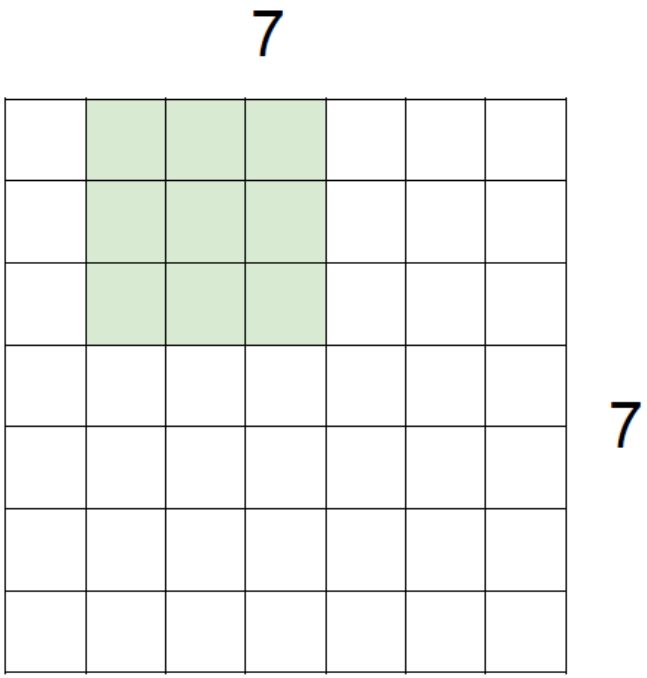
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

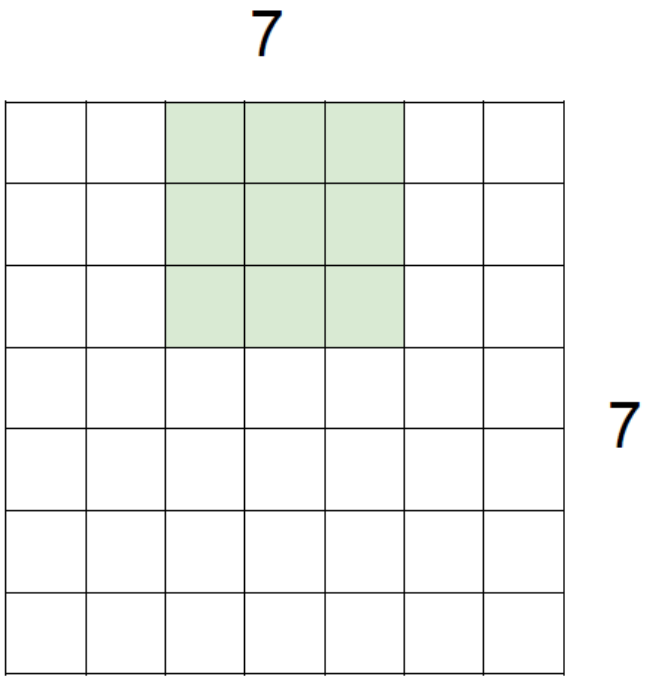
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

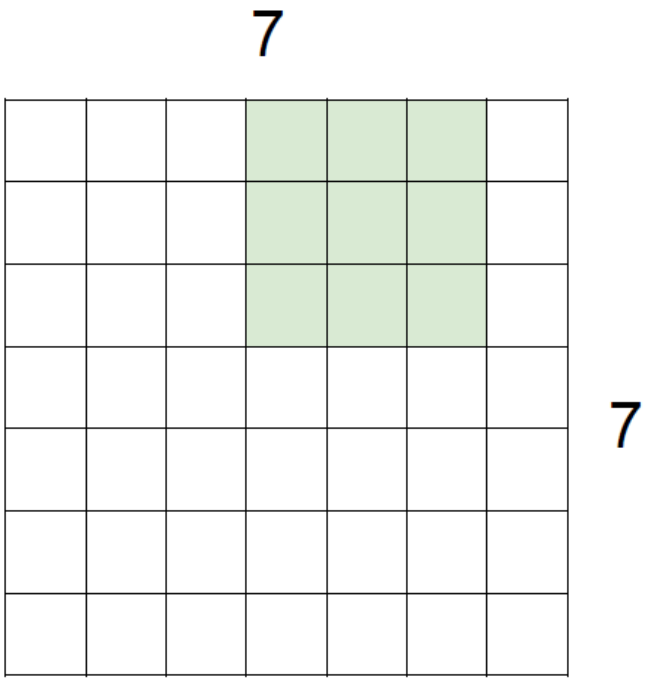
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

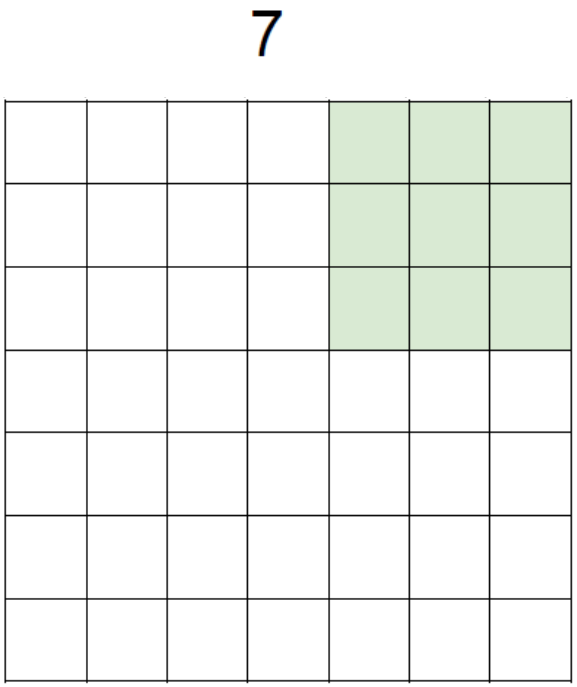
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

Convolutional Layer

A closer look at spatial dimensions:

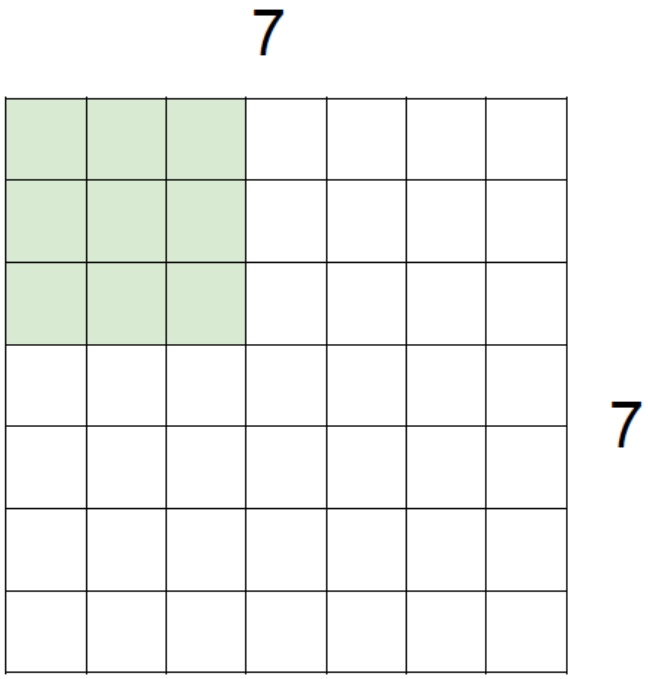


7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

Convolutional Layer

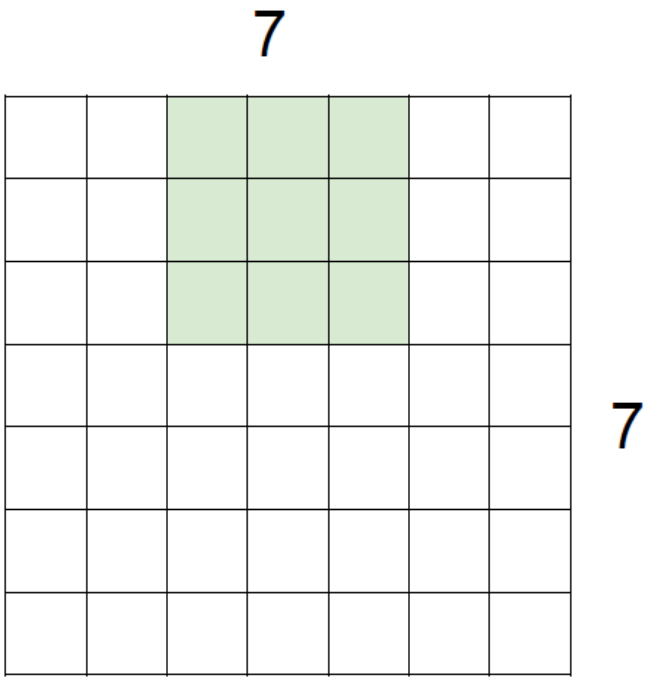
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Convolutional Layer

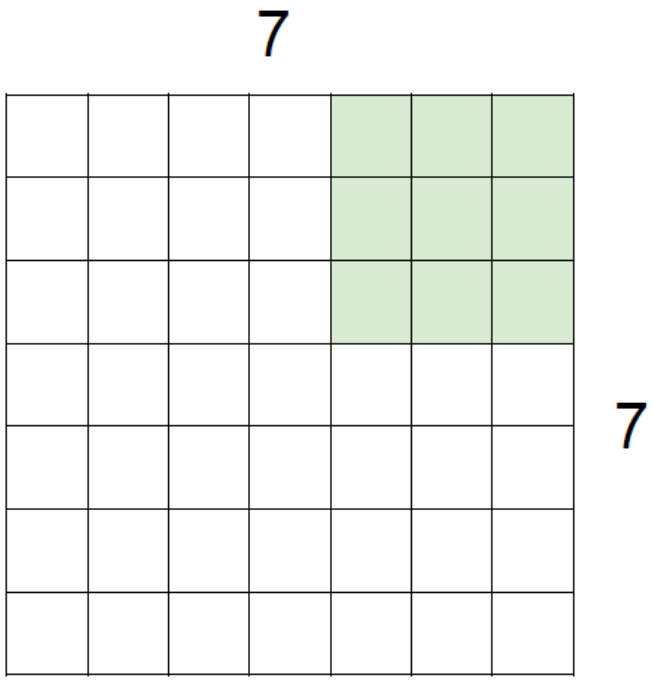
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Convolutional Layer

A closer look at spatial dimensions:

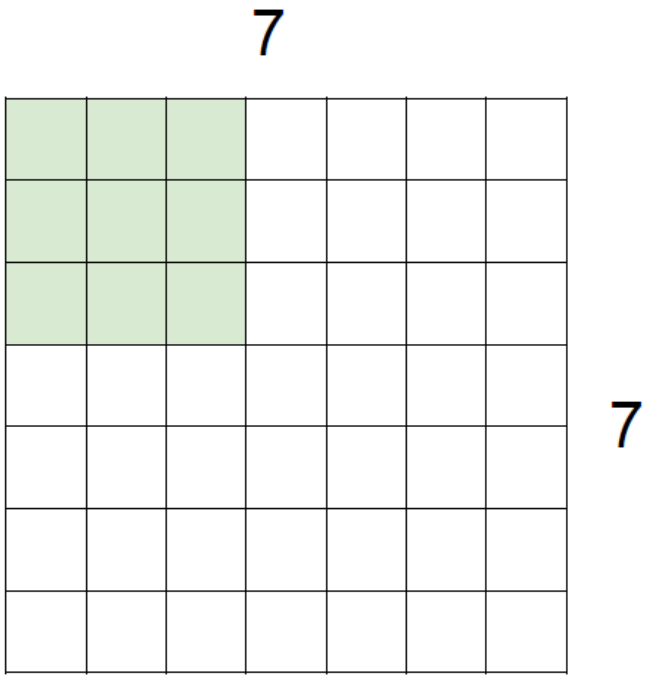


7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

=> 3x3 output!

Convolutional Layer

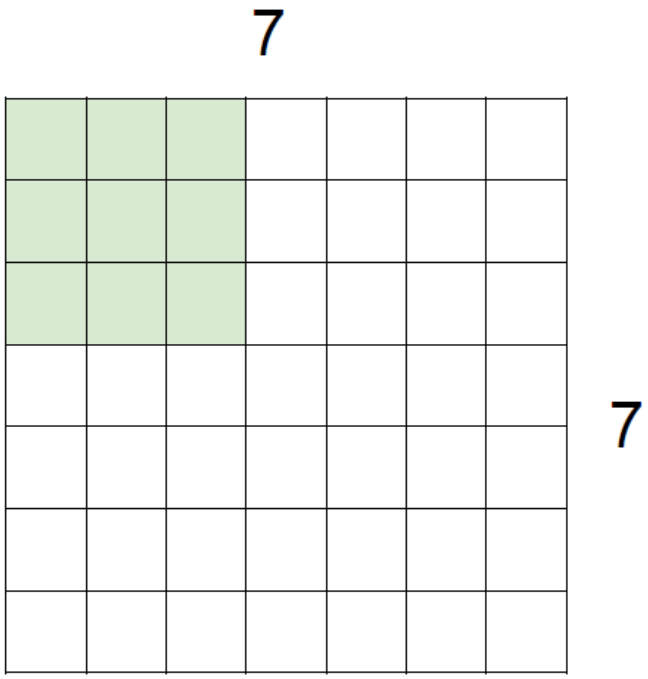
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

Convolutional Layer

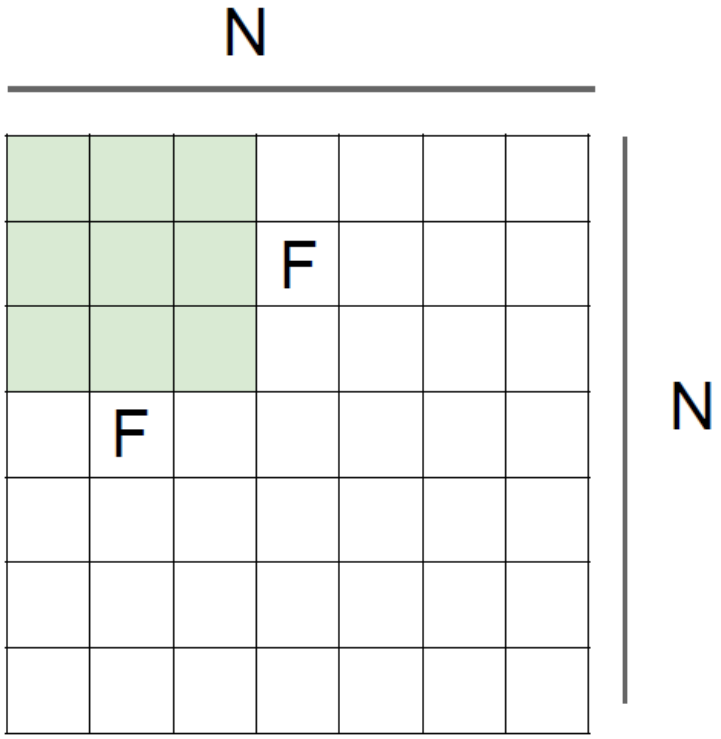
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

Convolutional Layer

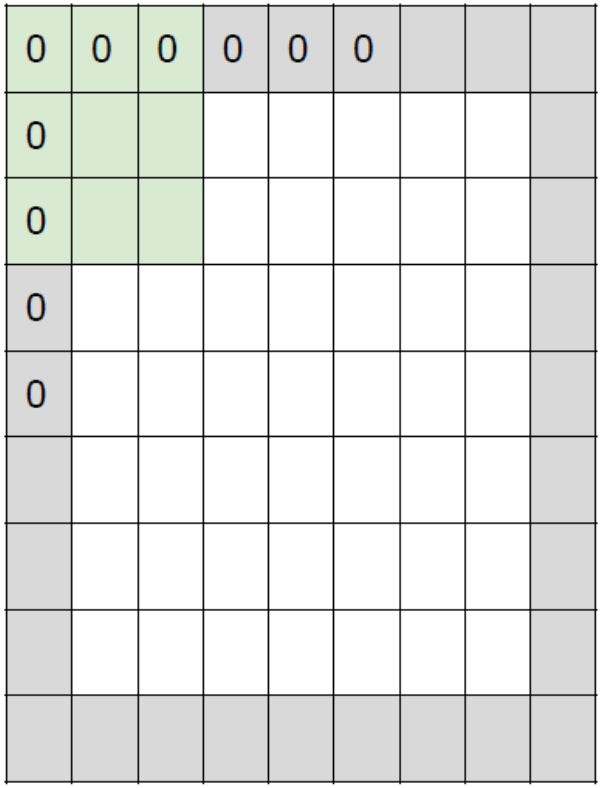


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

Convolutional Layer

In practice: Common to zero pad the border

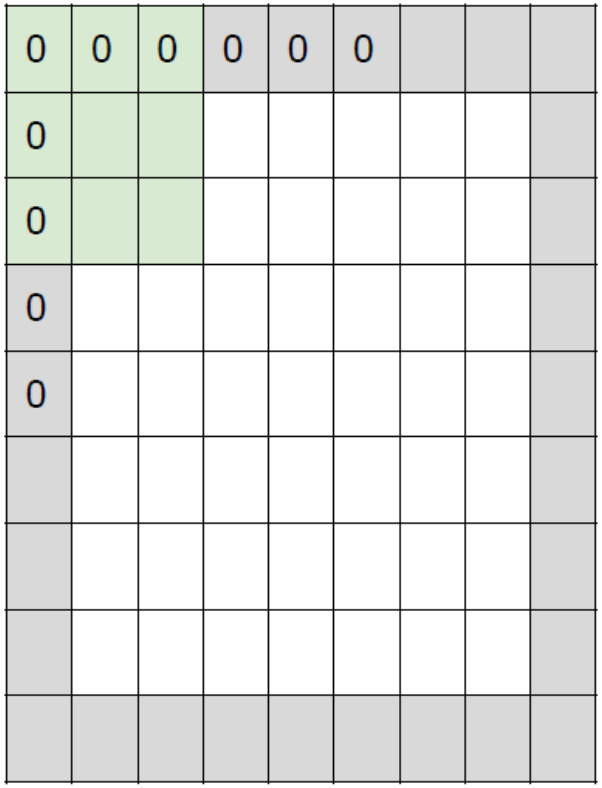


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

Convolutional Layer

In practice: Common to zero pad the border

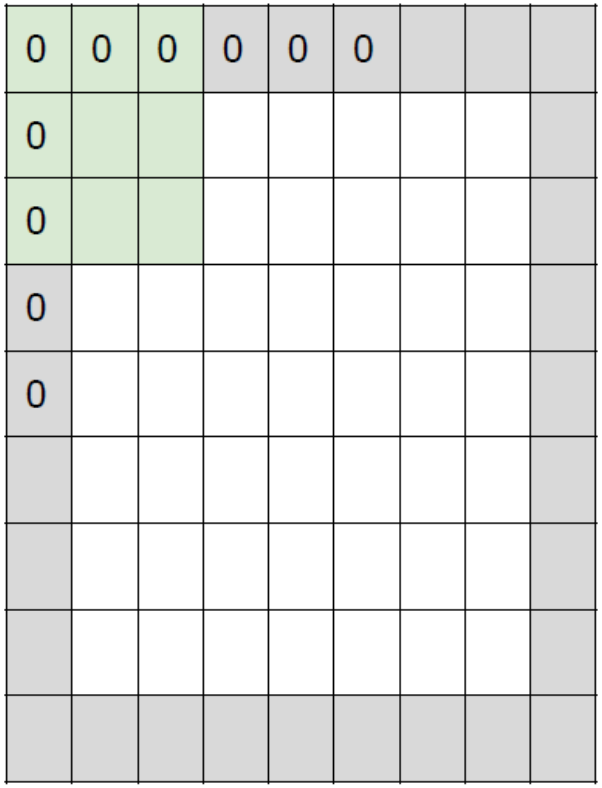


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

Convolutional Layer

In practice: Common to zero pad the border



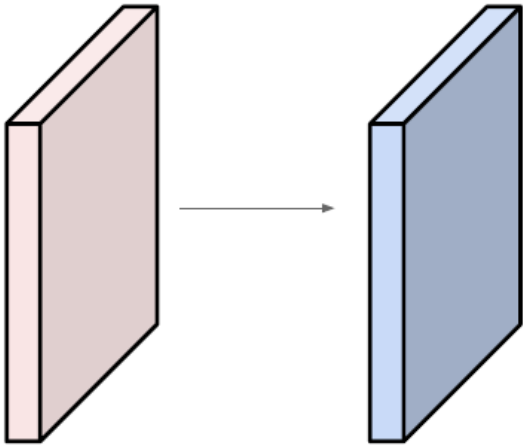
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Convolutional Layer

Examples time:

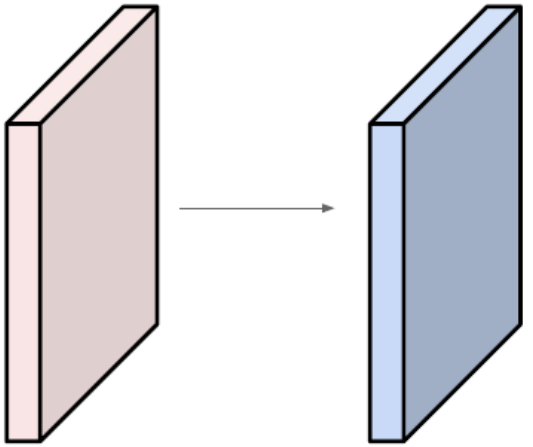


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:

Convolutional Layer

Examples time:

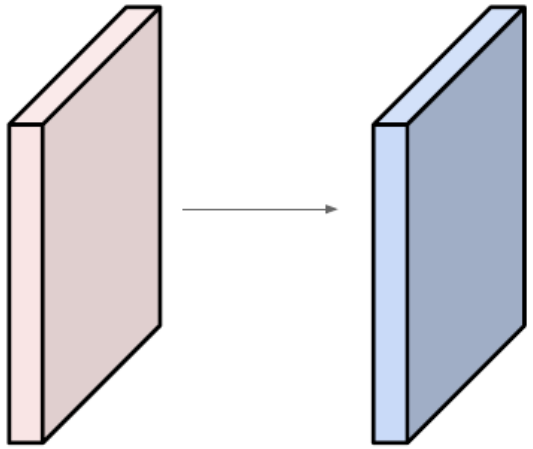


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Convolutional Layer

Examples time:

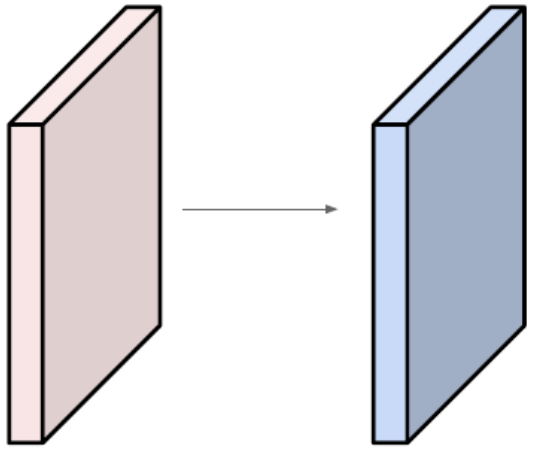


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Convolutional Layer

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Convolutional Layer

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Convolutional Layer

Summary. To summarize, the Conv Layer:

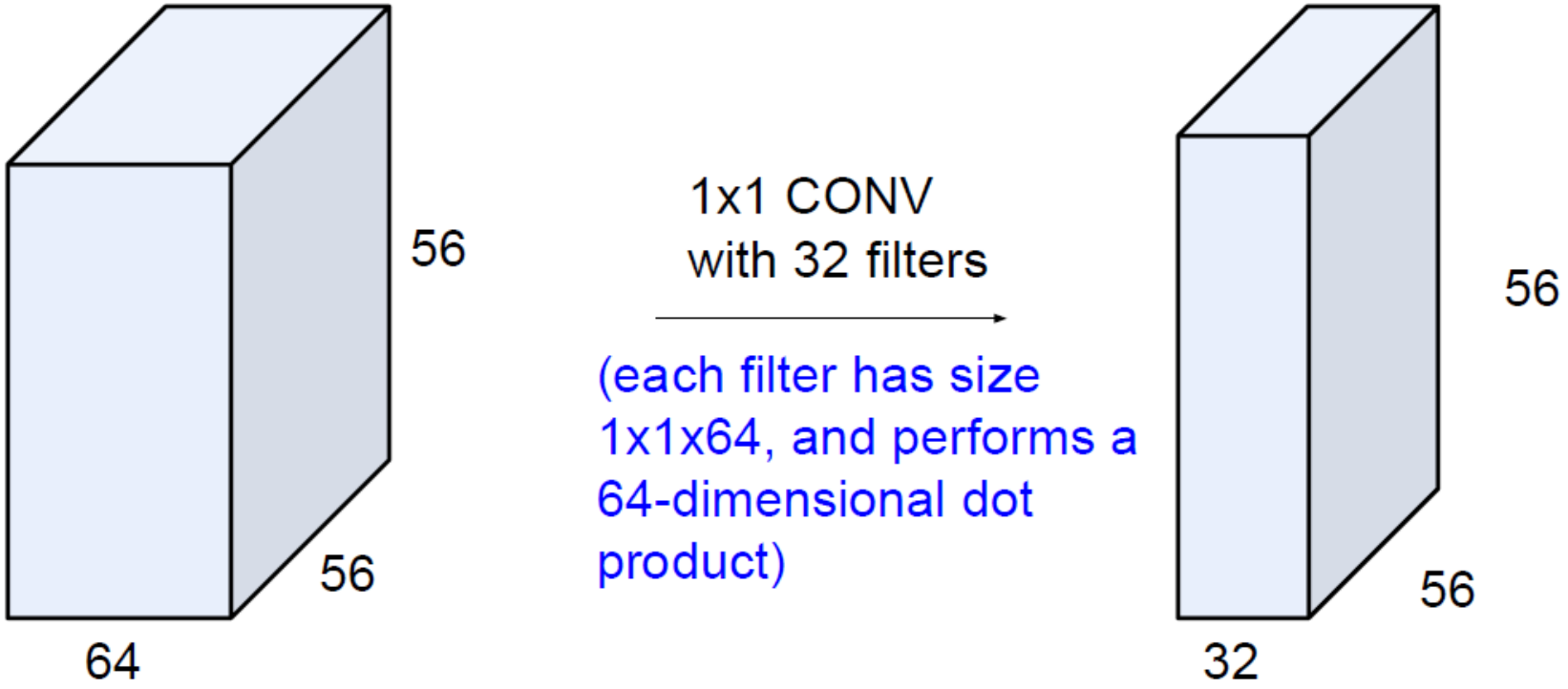
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

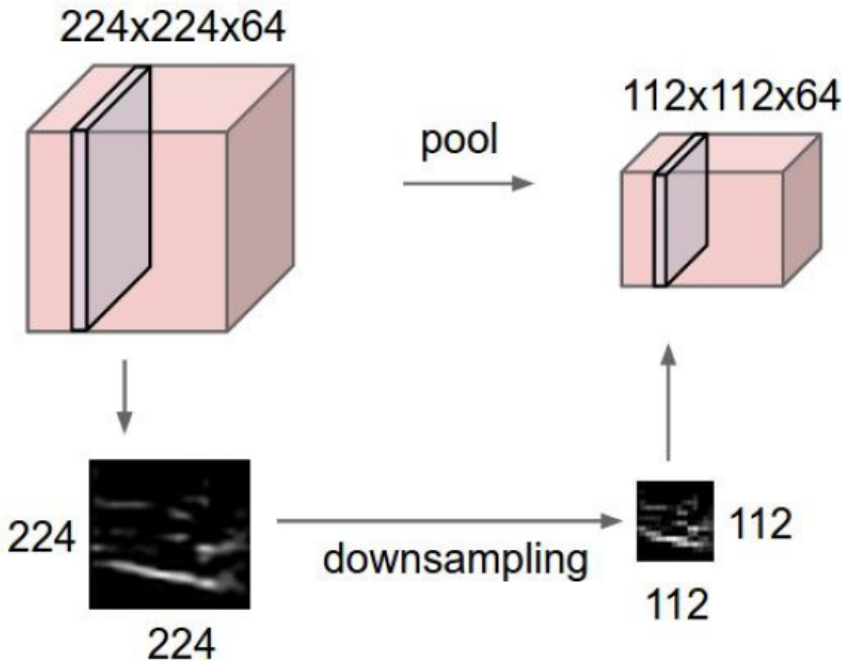
Convolutional Layer

(btw, 1x1 convolution layers make perfect sense)



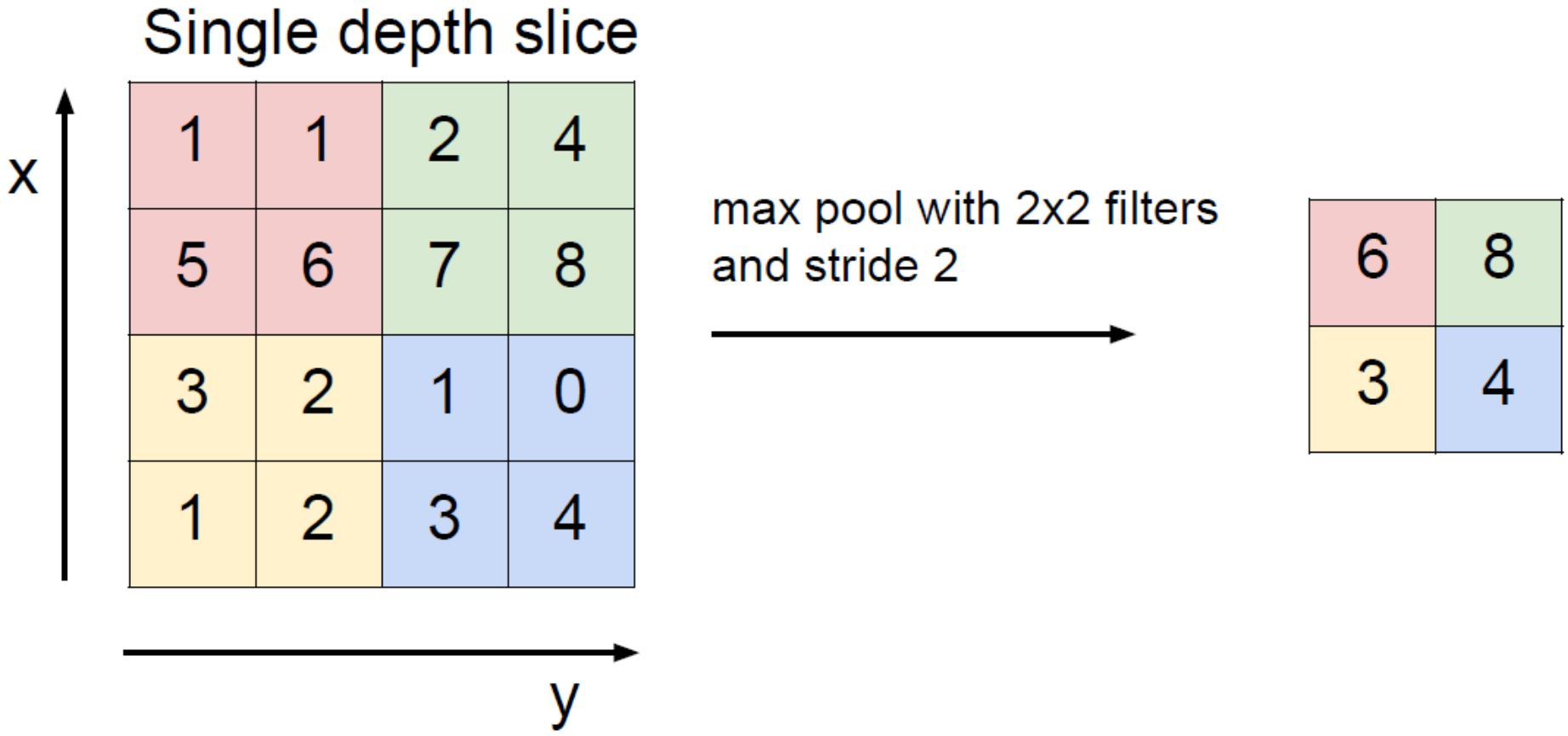
Pooling Layer

- Makes the representations smaller and more manageable
- Operates over each activation map independently



Pooling Layer

Common choice – max pooling:



Pooling Layer

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

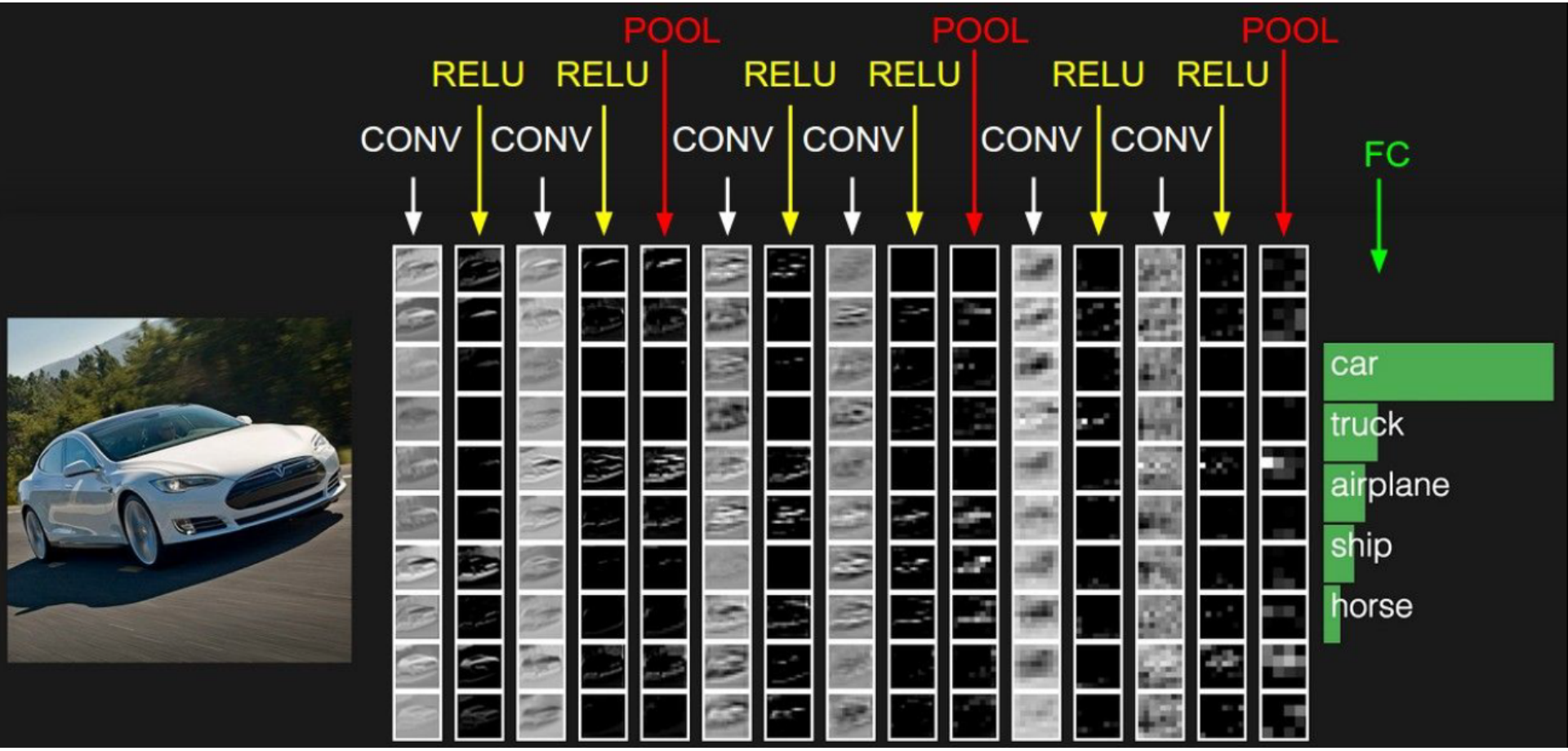
Pooling Layer

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Common settings:

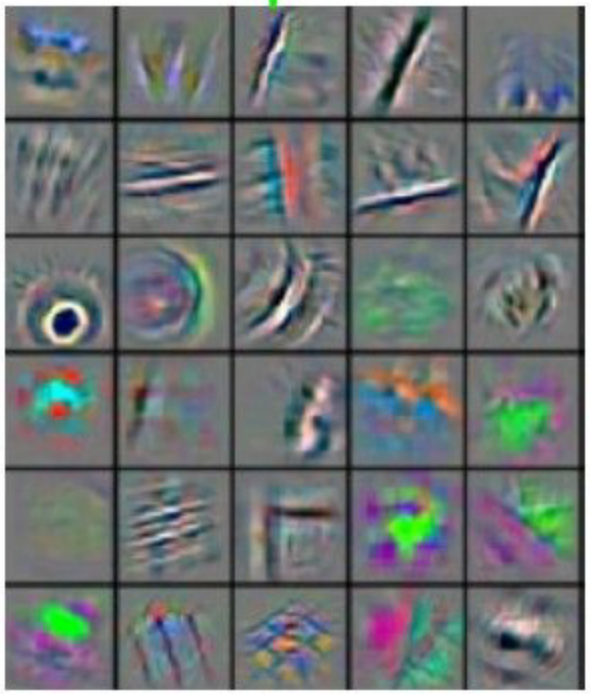
F = 2, S = 2
F = 3, S = 2

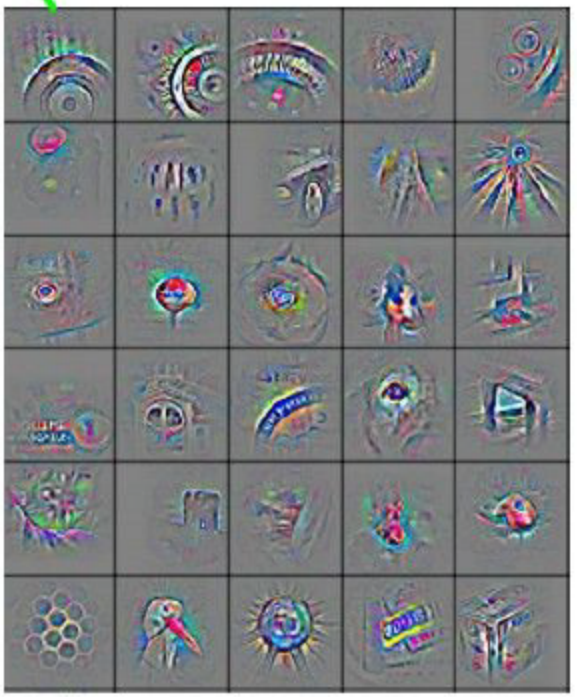
Classic ConvNet



Convolutional Layer

Low-Level Feature → Mid-Level Feature → High-Level Feature → Trainable Classifier

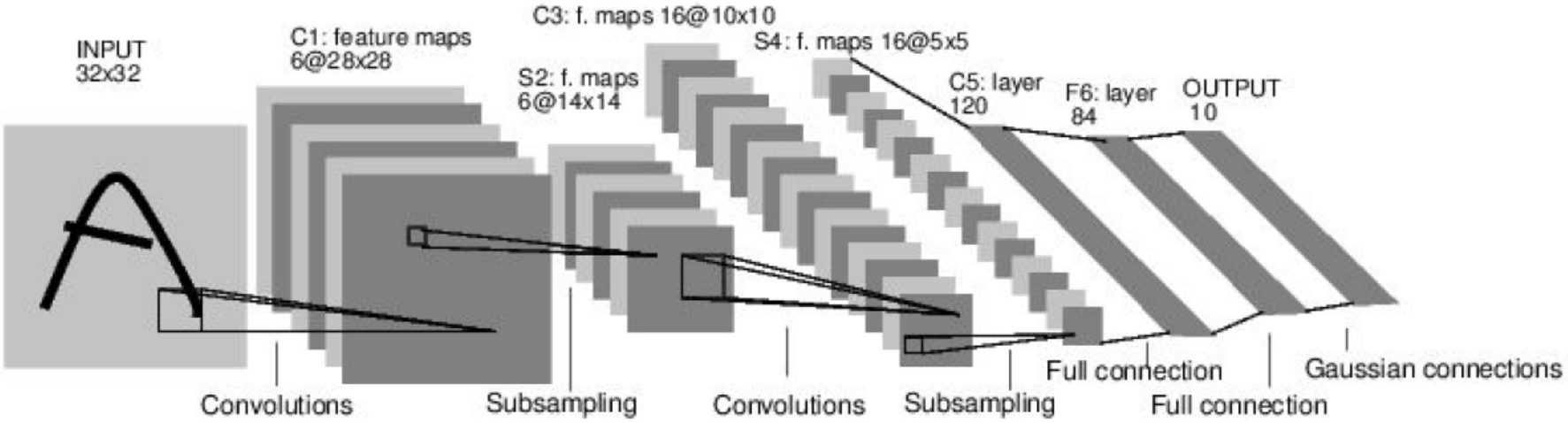




Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

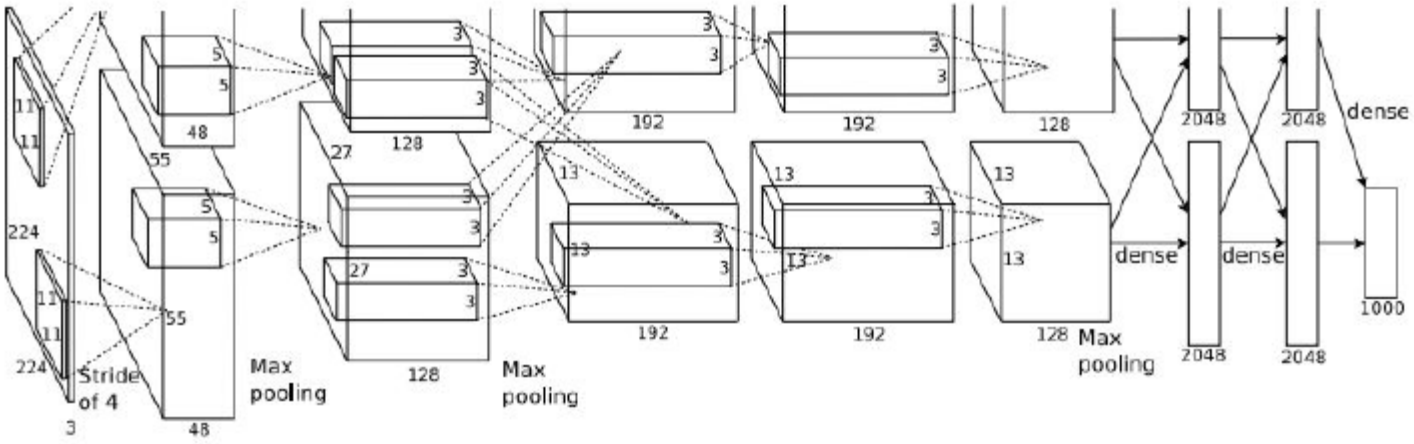
Case Studies

LeNet-5 [LeCun et al. 1998]



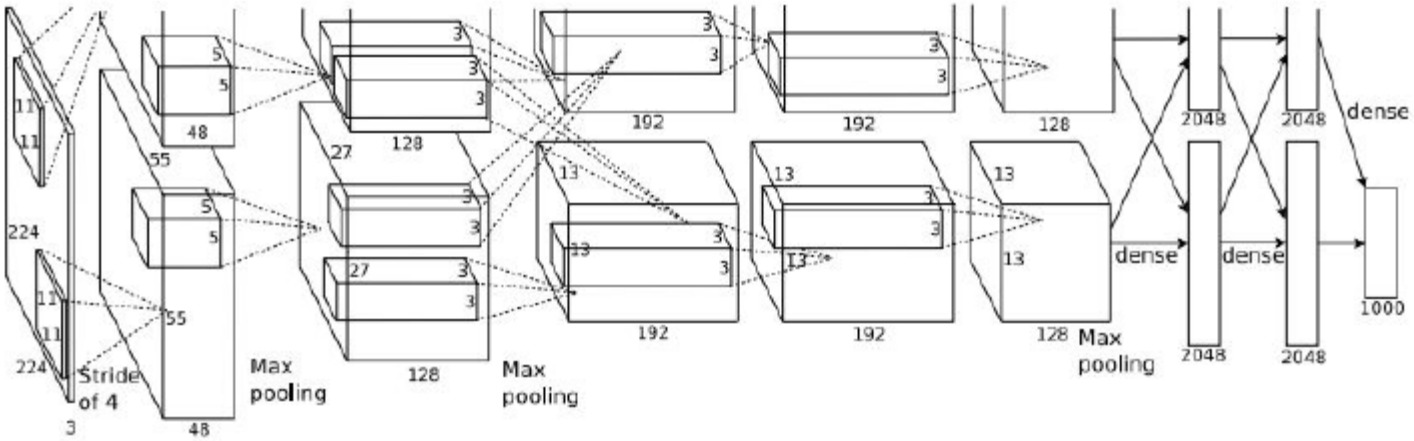
Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

AlexNet [Krizhevsky et al. 2012]



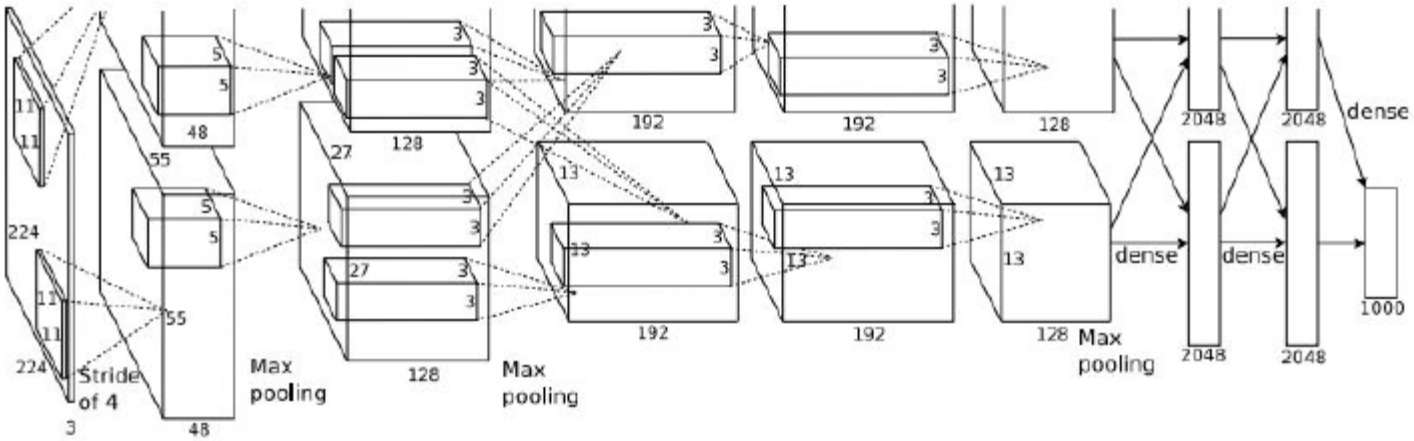
- Trained on ImageNet. 1M images. 1000 classes.
- Large network ~60M parameters
- Improved the error from 25% (top-5) down to just 16%!
- The point at which ConvNets took over!

AlexNet [Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

AlexNet [Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%
- 60M parameters

VGG [Simonyan and Zisserman 2014]

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

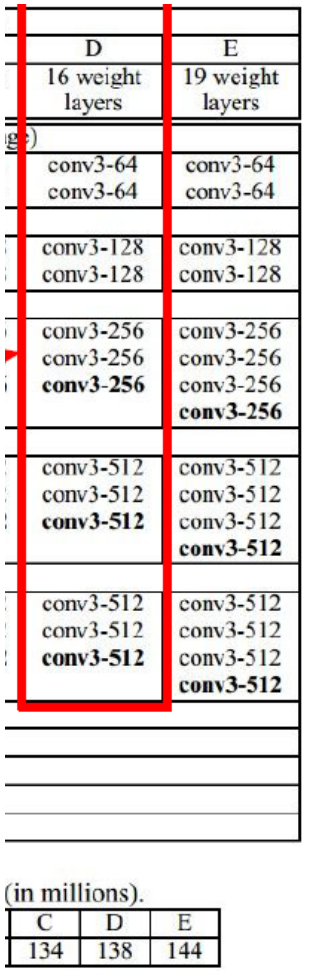
Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

VGG [Simonyan and Zisserman 2014]

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000



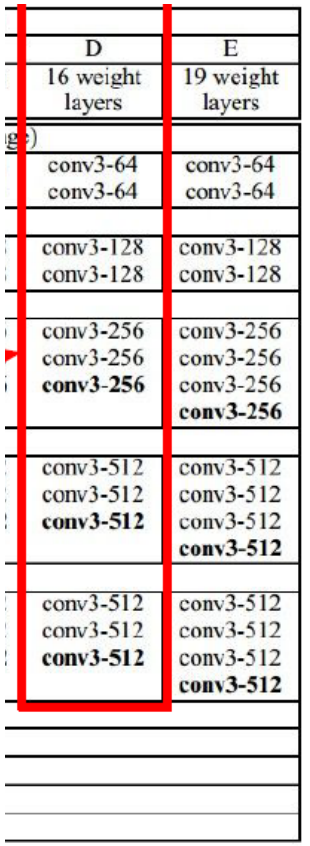
D	E
16 weight layers	19 weight layers
conv3-64 conv3-64	conv3-64 conv3-64
conv3-128 conv3-128	conv3-128 conv3-128
conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**

(in millions).

C	D	E
134	138	144

VGG [Simonyan and Zisserman 2014]

INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000



D	E
16 weight layers	19 weight layers
conv3-64 conv3-64	conv3-64 conv3-64
conv3-128 conv3-128	conv3-128 conv3-128
conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**

(in millions).

C	D	E
134	138	144

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

VGG [Simonyan and Zisserman 2014]

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

Note:

Most memory is in early CONV

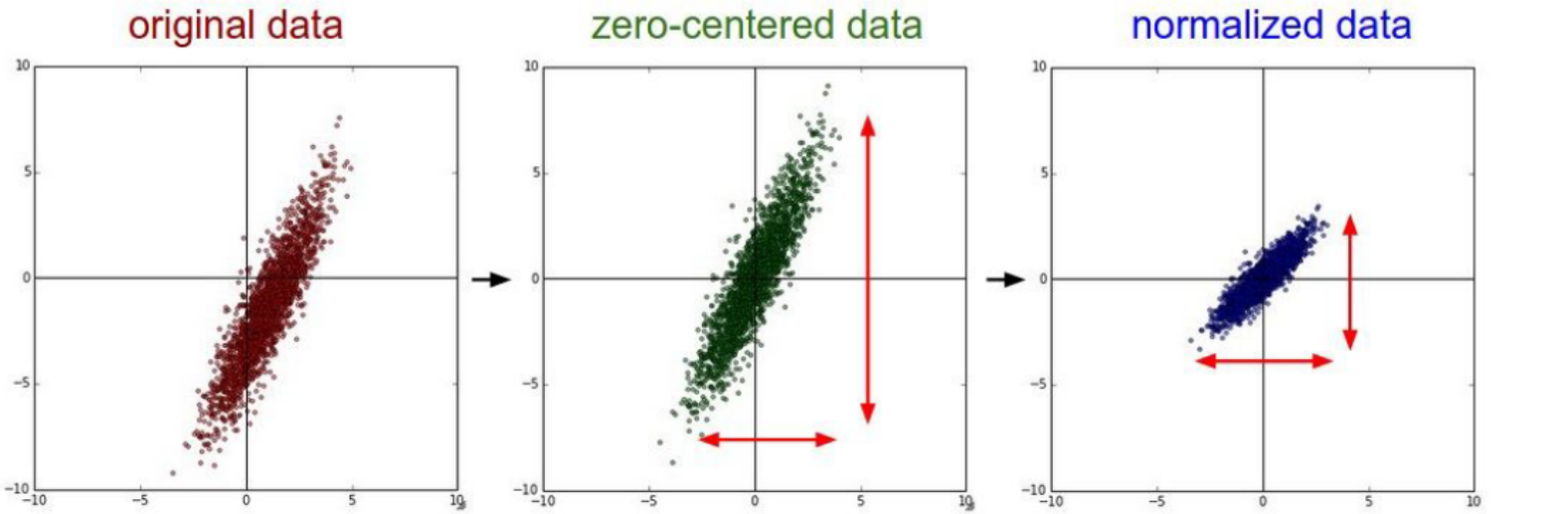
Most params are in late FC

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Training

Data Preprocessing

ConvNets are typically easier to train when data is well conditioned



Weight Initialization

Question

What happens if weights (and biases) are initialized to 0:

- With ReLU activation?
- In general?

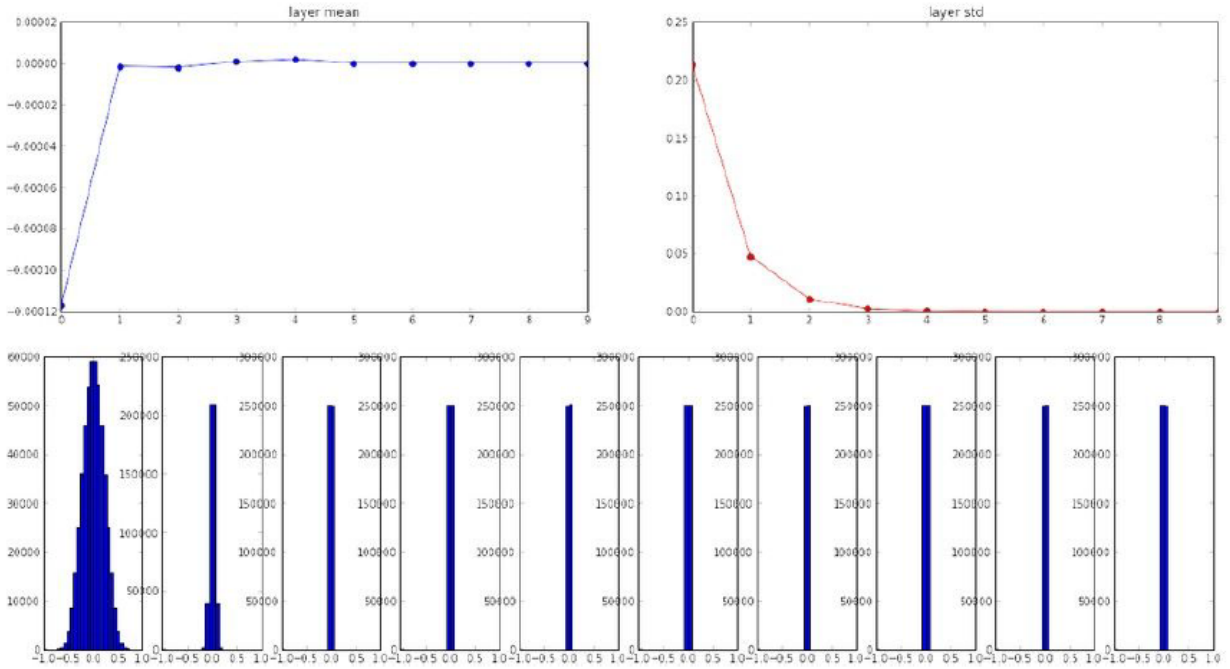
Weight Initialization

Option 1:

Fixed random distribution

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

Works OK for small networks, but can lead to non-homogeneous distributions of activations across layers



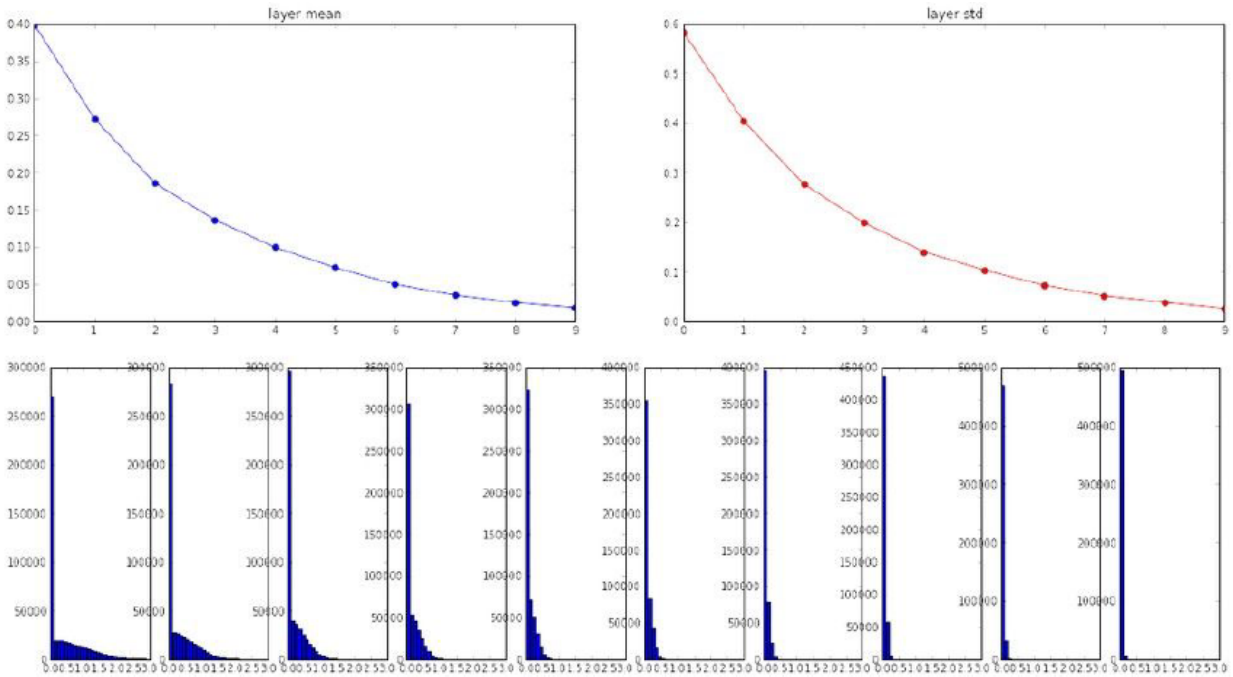
Weight Initialization

Option 2:

"Xavier" [Glorot et al. 2010]

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

Typically works better



Batch Normalization

[Ioffe and Szegedy 2015]

"you want unit gaussian activations?
just make them so."

activation

mean across mini-batch

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Improves gradient flow at training

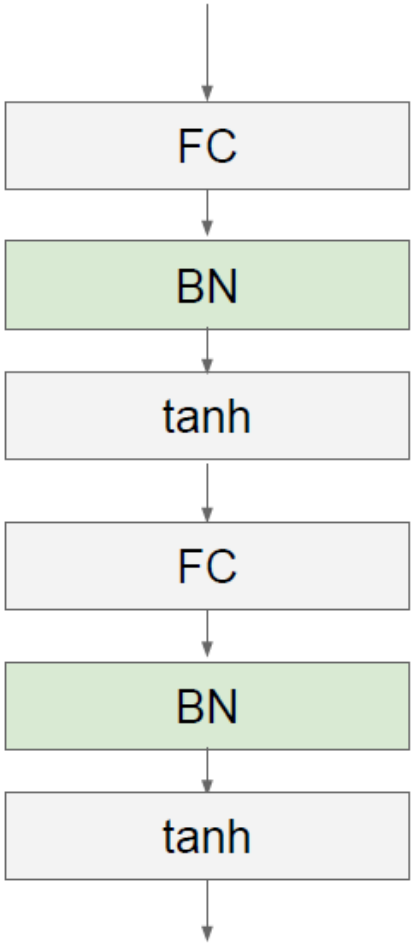
Allows higher learning rates

Reduces dependence on initialization

At test time:
Mean and std fixed (based on training data)

Batch Normalization

[Ioffe and Szegedy 2015]



$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Usually inserted after fully connected or convolutional layers

Loss Function

Typical choices:

- Classification – cross-entropy loss

network output r

$$L(X, y) = \log\left(\sum_r \exp\left(o_r(X) - o_y(X)\right)\right)$$

$$[o_r(X) \propto \log P(X, Y = r)] \Rightarrow = -\log \frac{P(X, y)}{\sum_r P(X, Y = r)} = -\log P(y|X)$$

- Regression – square loss

network output

$$L(X, y) = \left(o(X) - y\right)^2$$

- Other applications (e.g. semantic segmentation) require specialized choices

Large Scale Optimization

Large scale settings:

- Many network parameters (e.g. 10^8)
  → computing Hessian (2^{nd} order derivatives) is expensive
- Many training examples (e.g. 10^6)
  → computing full objective at every iteration is expensive

Optimization methods must be:

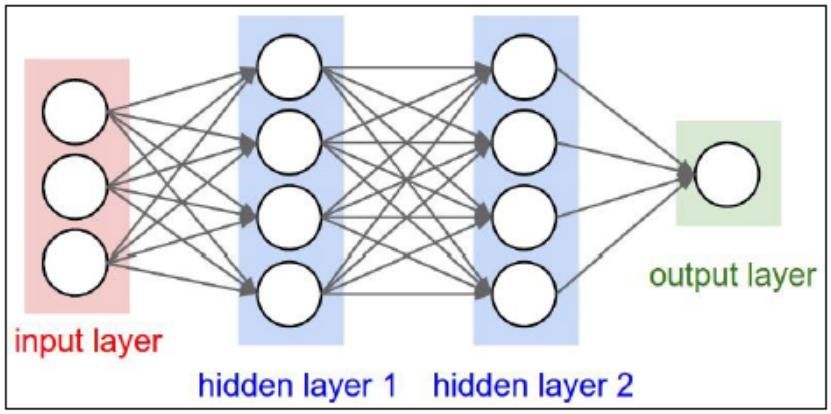
- **1^{st}-order** – based on objective value and gradient only
- **Stochastic** – each update based on subset of training examples

Mini-batch SGD

Mini-batch Stochastic Gradient Descent (SGD):

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

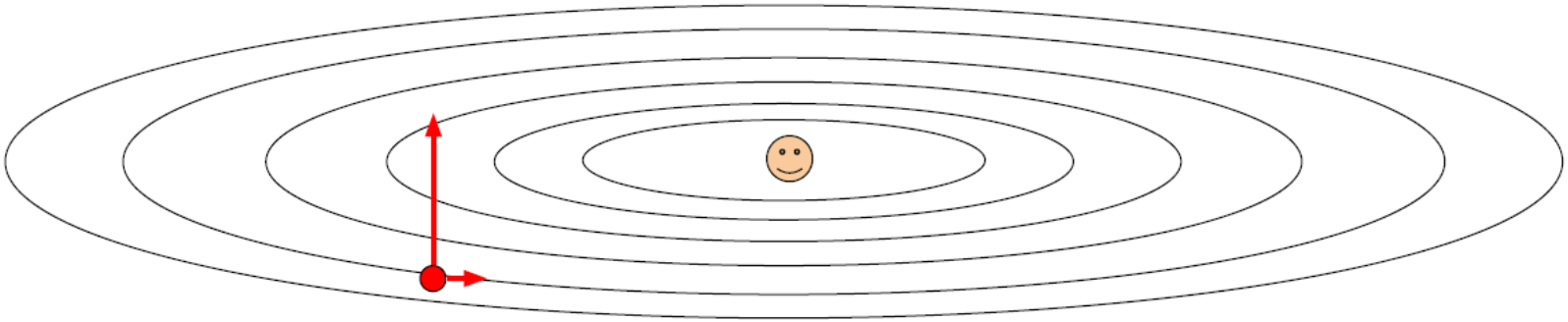


Basic Parameter Update

Basic gradient descent update:

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Suppose loss is steep vertically but shallow horizontally:



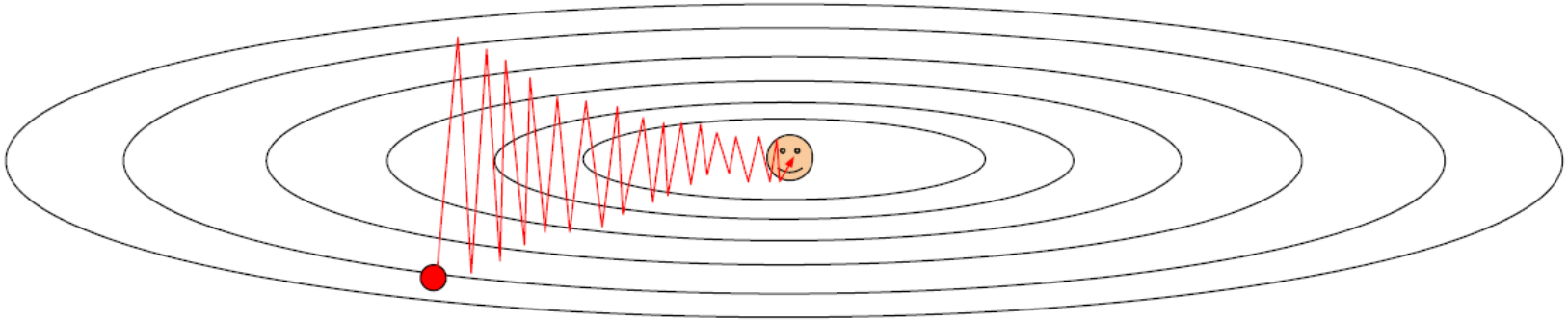
What trajectory will basic SGD follow?

Basic Parameter Update

Basic gradient descent update:

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Suppose loss is steep vertically but shallow horizontally:

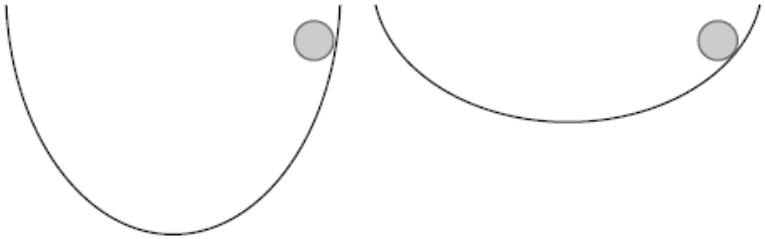


What trajectory will basic SGD follow?

Slow progress along flat direction, jitter along steep one

Momentum Update

```
# Gradient descent update
x += - learning_rate * dx
```



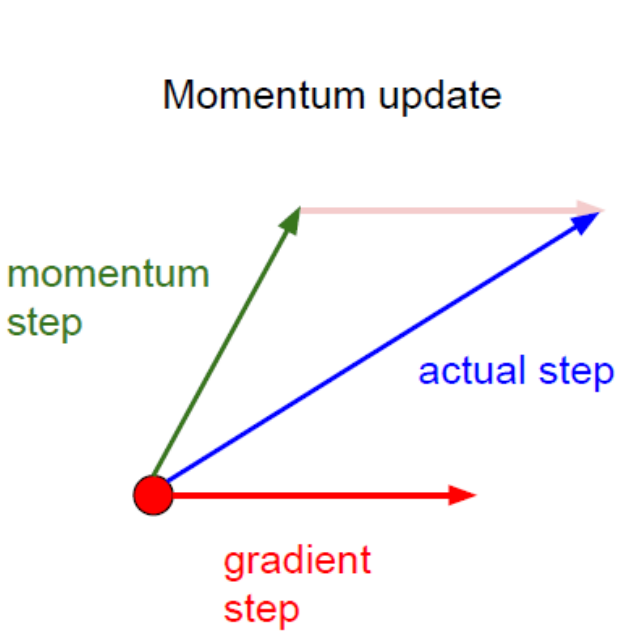
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

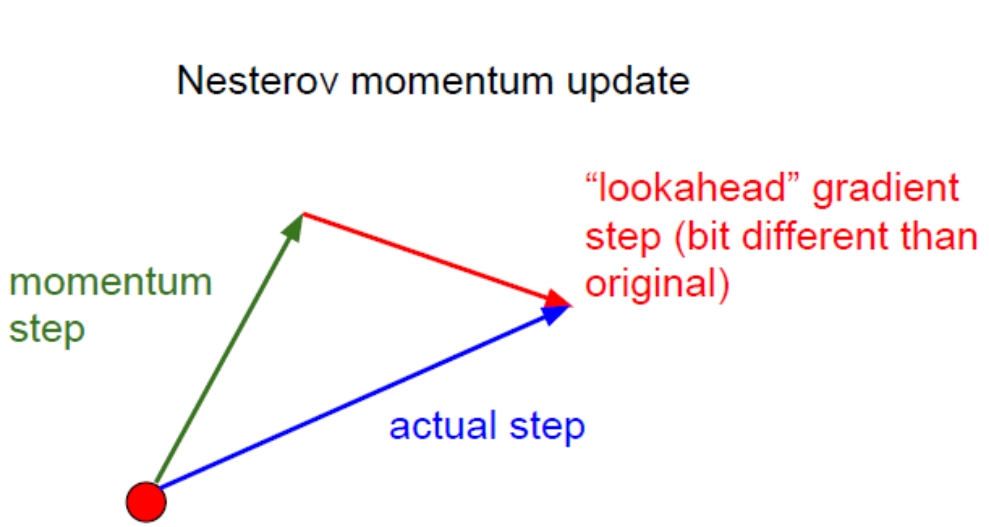
(typical choices for mu: 0.9, 0.95, 0.99)

Momentum update:

- Allows “velocity” to build-up along shallow directions
- Suppresses velocity in steep directions due to changing signs

Nesterov Momentum Update





Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

"Lookahead" reduces fluctuations

Adam Update [Kingma and Ba 2014]

momentum

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

normalized update

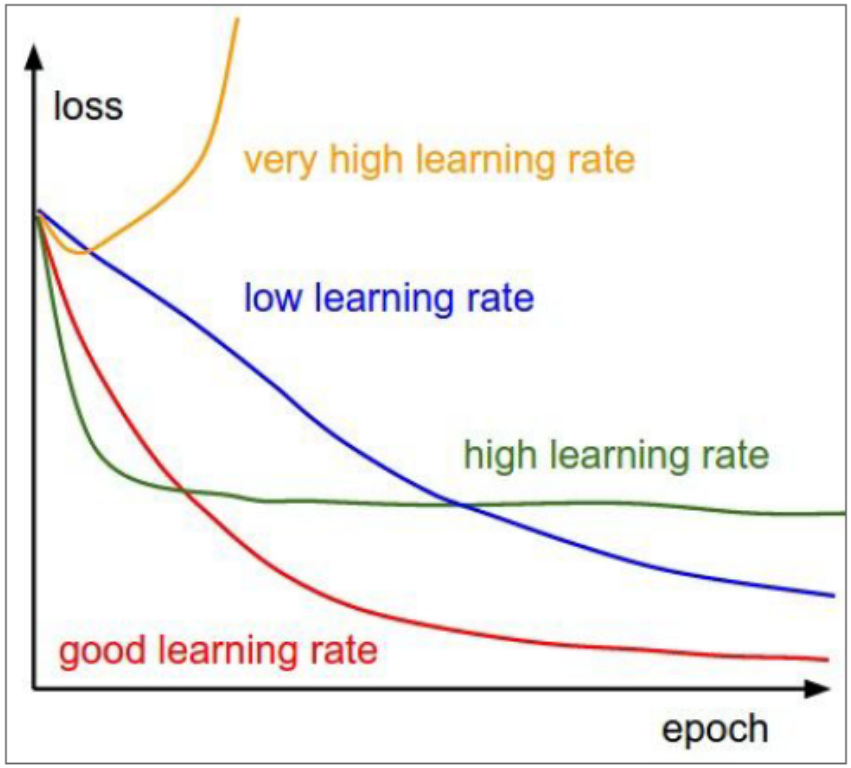
momentum over squared gradient

In practice, 1st and 2nd momentum variables are scaled to compensate for zero initialization

Adam typically works well and requires less hyper-parameter tuning than other methods

Learning Rate

All variants of SGD have learning rate as a hyper-parameter, and properly setting it is **crucial**

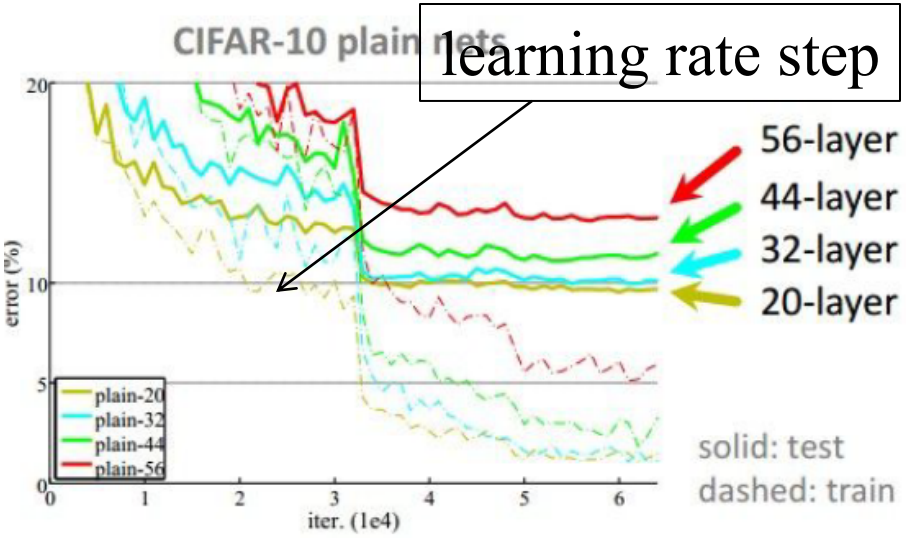


Learning Rate Schedule

Decaying learning rate over time leads to better convergence. Common schedules:

- **Step decay**: drop learning rate by constant factor after fixed number of epochs or when validation error plateaus
- **Exponential decay**: $\alpha = \alpha_0 e^{-kt}$
- **1/t decay**: $\alpha = \alpha_0 / (1 + kt)$
- **Linear decay:**

$$\alpha = \frac{(t - t_{start})\alpha_{end} + (t_{end} - t)\alpha_{start}}{t_{end} - t_{start}}$$



L^2 Penalty (Weight Decay)

Optimized objective typically includes L^2 penalty:

$$f\left(weights\right) = \frac{1}{\left|S\right|} \sum_{(X,y)\in S} L\left(X,y\right) \; + \frac{\lambda}{2} \cdot \sum_{W \in weights} \left\|W\right\|_{Fro}^2$$

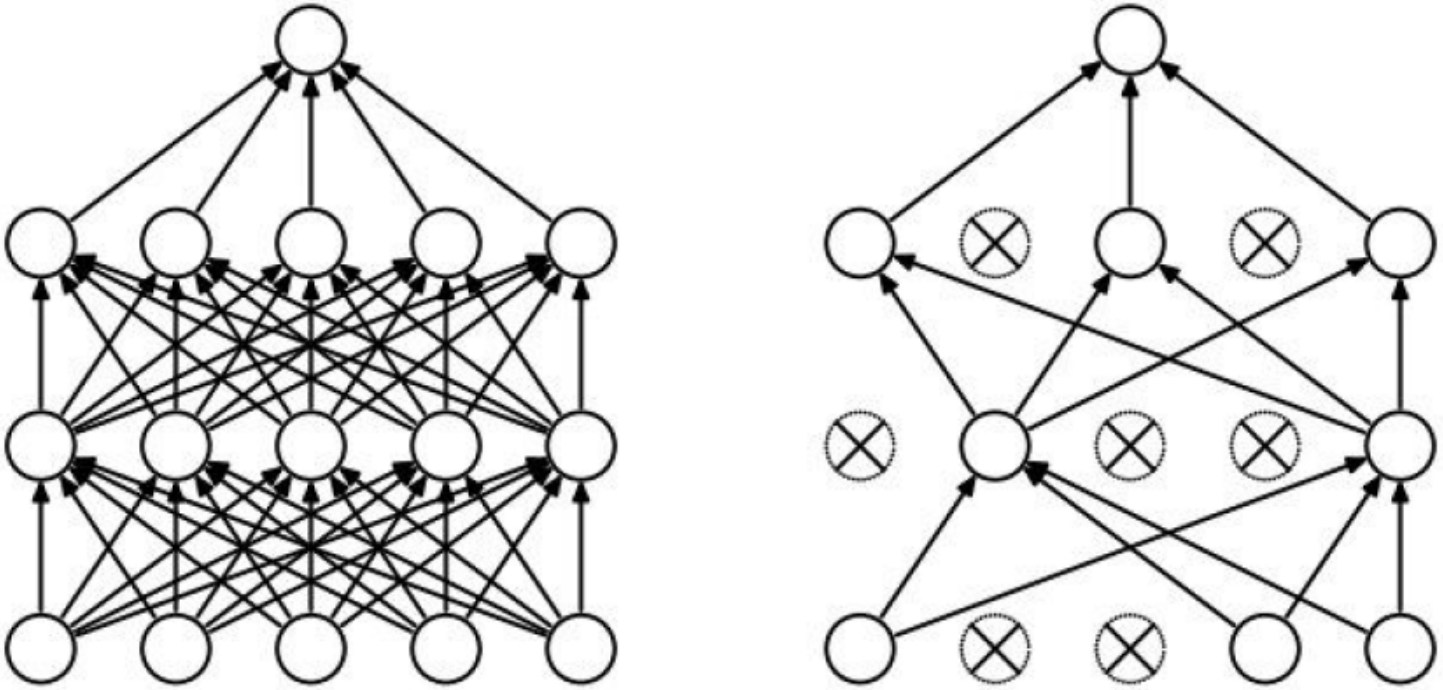
This translates into decay terms in updates:

$$-\nabla_W f = -\frac{1}{\left|S\right|} \sum_{(X,y)\in S} \nabla_W L\left(X,y\right) \; - \lambda \cdot W$$

Parameter λ is accordingly named **weight decay**

Dropout Regularization

Randomly set some neurons to 0 in forward pass:



(a) Standard Neural Net

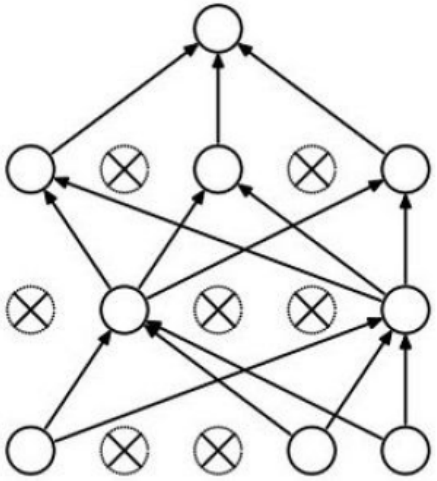
(b) After applying dropout.

[Srivastava et al. 2014]

Dropout Regularization

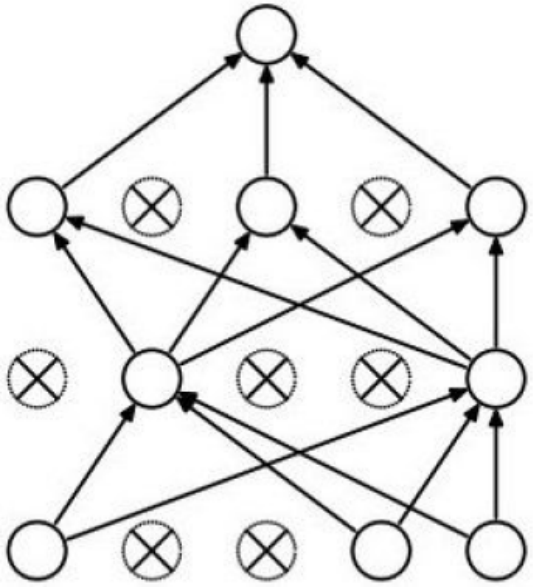
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Dropout Regularization

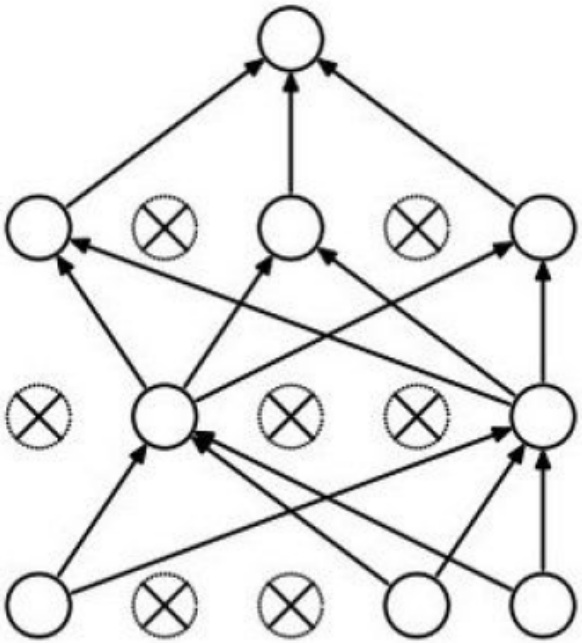
Intuition 1



Forces the network to have a redundant representation.

Dropout Regularization

Intuition 2

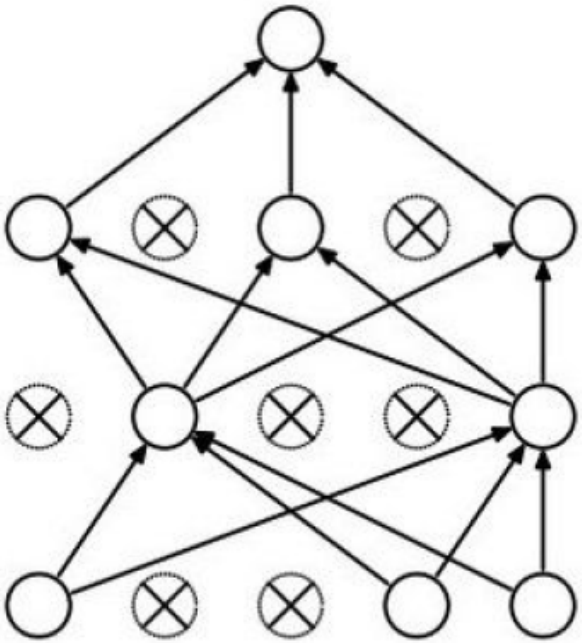


Trains a large ensemble of models with shared parameters – each binary mask is one model.

Ensembles improve generalization.

Dropout Regularization

Intuition 3

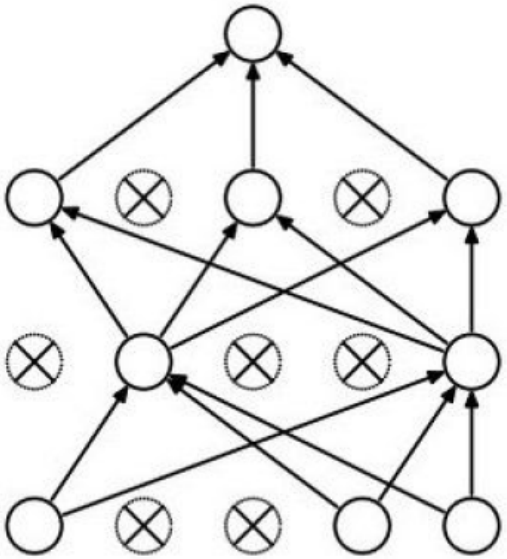


Trains network while injecting noise.

Robustness to noise improves generalization.

Dropout Regularization

At test time:



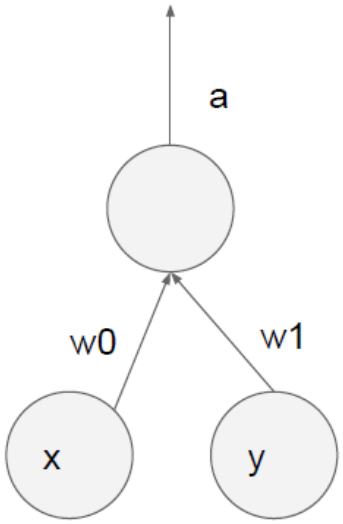
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

Dropout Regularization

Can approximate this in single forward pass – leave all neurons turned on while scaling activations by p (probability of not being dropped)

In the linear case this is exact:



during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= ½ * **(w0*x + w1*y)**

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

Dropout Regularization

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

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
```

Dropout Summary

drop in forward pass

scale at test time

Dropout Regularization

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Dropout Regularization

Alternatives to classic dropout (work just as well):

– Multiplicative Gaussian noise:

$$h \rightarrow h \cdot r \quad , \quad r \sim N\left(1, \sigma^2\right)$$

– Additive Gaussian noise:

$$h \rightarrow h + r \quad , \quad r \sim N\left(0, \sigma^2\right)$$

Both variants do not admit redundant representation and ensemble interpretations, only noise injection

Data Augmentation

- Leveraging inherent invariances of the data:
  - Translations
  - Mirroring
  - Rotations,
  - Color (hue, lighting, etc.)
  - Noise

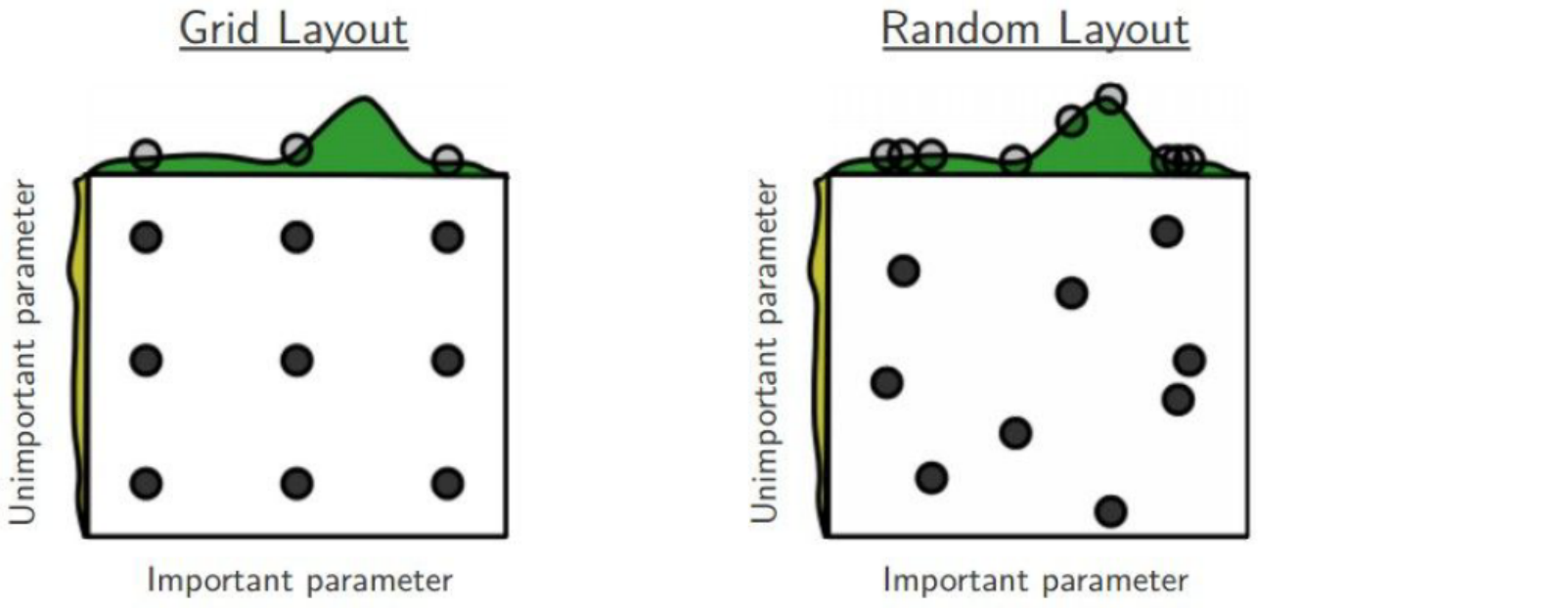
Hyper-parameter Search

Many hyper-parameters:

Network Architecture	**Training**
depth	optimizer type
breadth	learning rate
layer types	weight decay
window sizes	initialization
strides	loss function
…	…

Hyper-parameter Search

Despite many suggested algorithms for hyper-parameter search, most popular choices remain grid and random:



Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hyper-parameter Search

My personal preference for cross-validation:

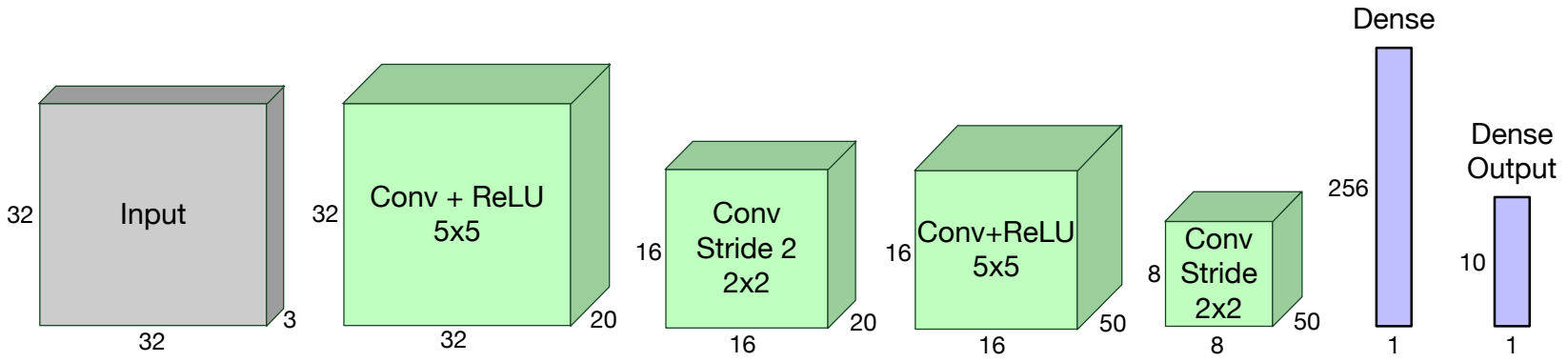
coarse → fine

1. Coarse:
   - Random search
   - Short runs to identify potentially good settings
2. Fine:
   - Grid search around potentially good settings
   - Long runs to identify best configuration

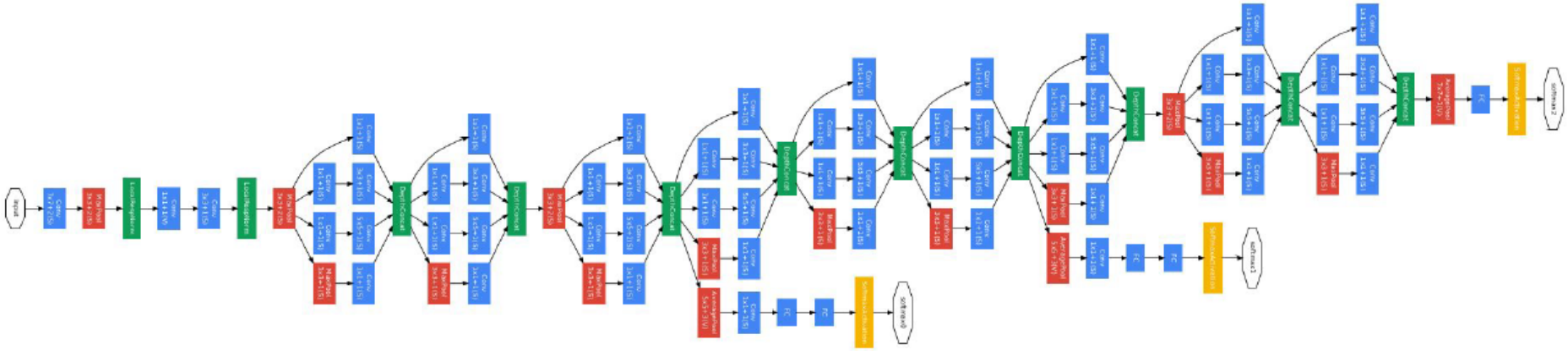
Advanced Architectures

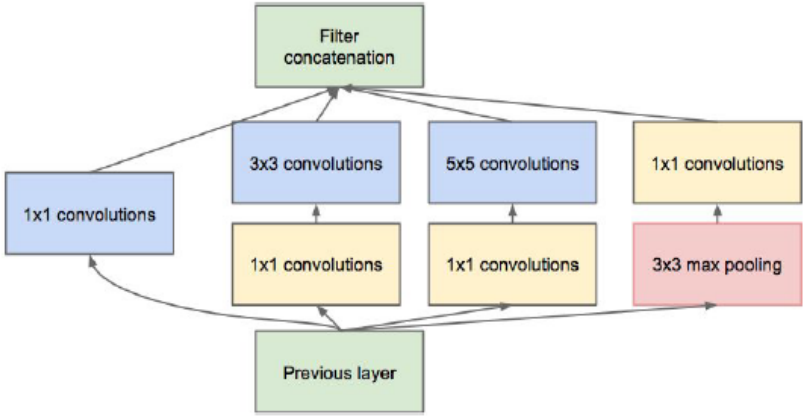
All Convolutional Net

Replace pooling layers with convolutions with stride > 1



GoogLeNet [Szegedy et al. 2014]





Inception module

ILSVRC 2014 winner (6.7% top 5 error)

GoogLeNet [Szegedy et al. 2014]

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

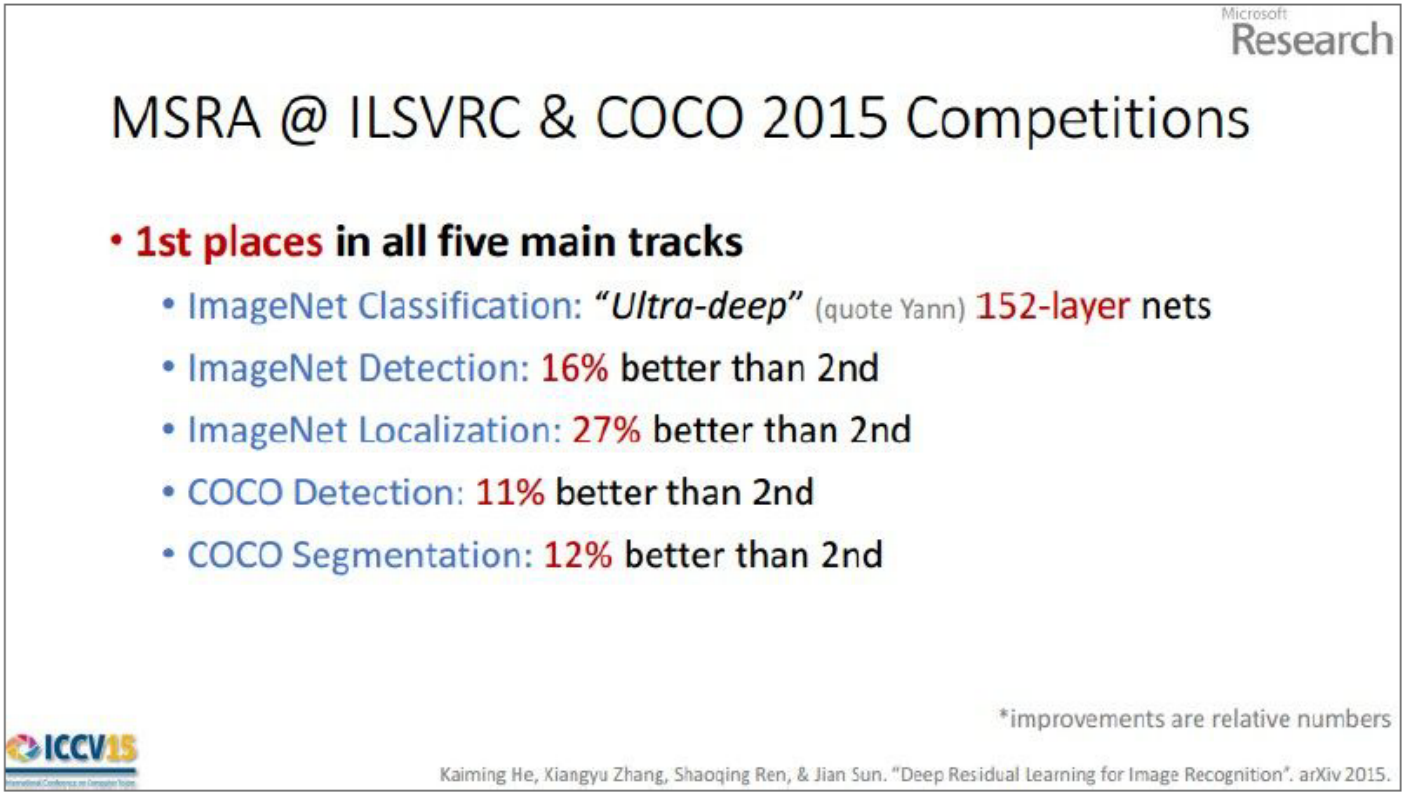
Fun features:

- Only 5 million params! (Removes FC layers completely)

Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

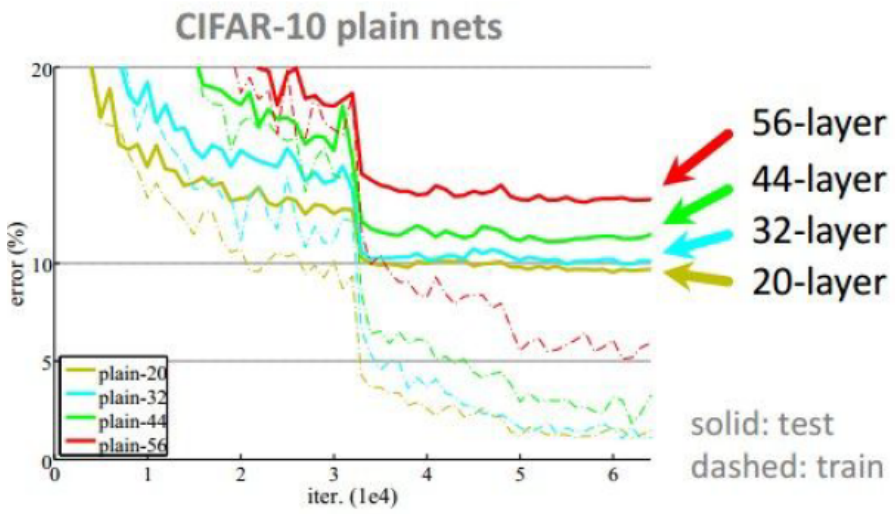
ResNet [He et al. 2015]

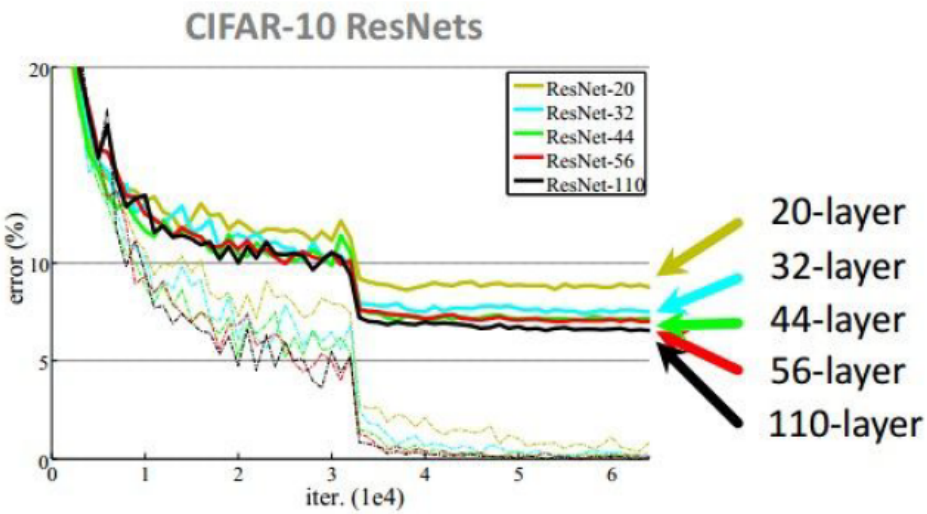
ILSVRC 2015 winner (3.6% top 5 error)



ResNet [He et al. 2015]

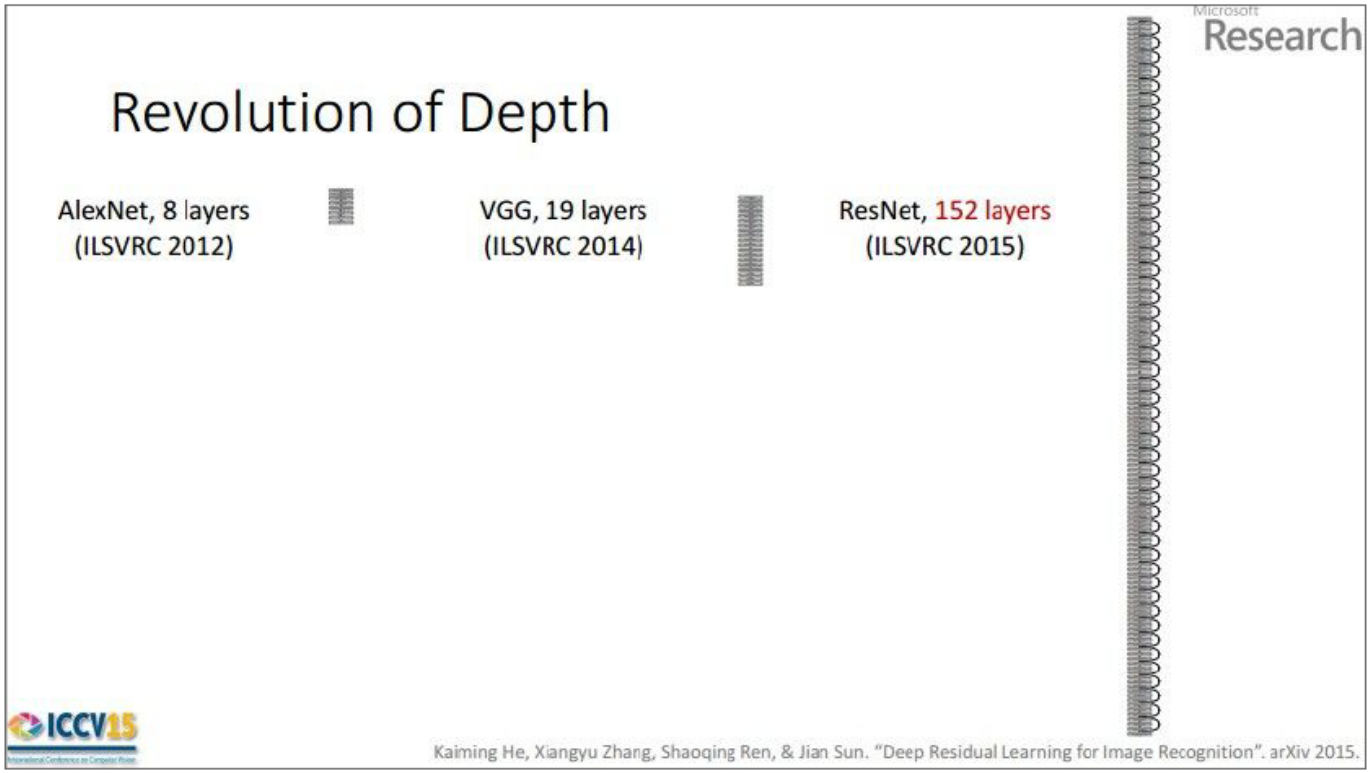
CIFAR-10 experiments





ResNet [He et al. 2015]

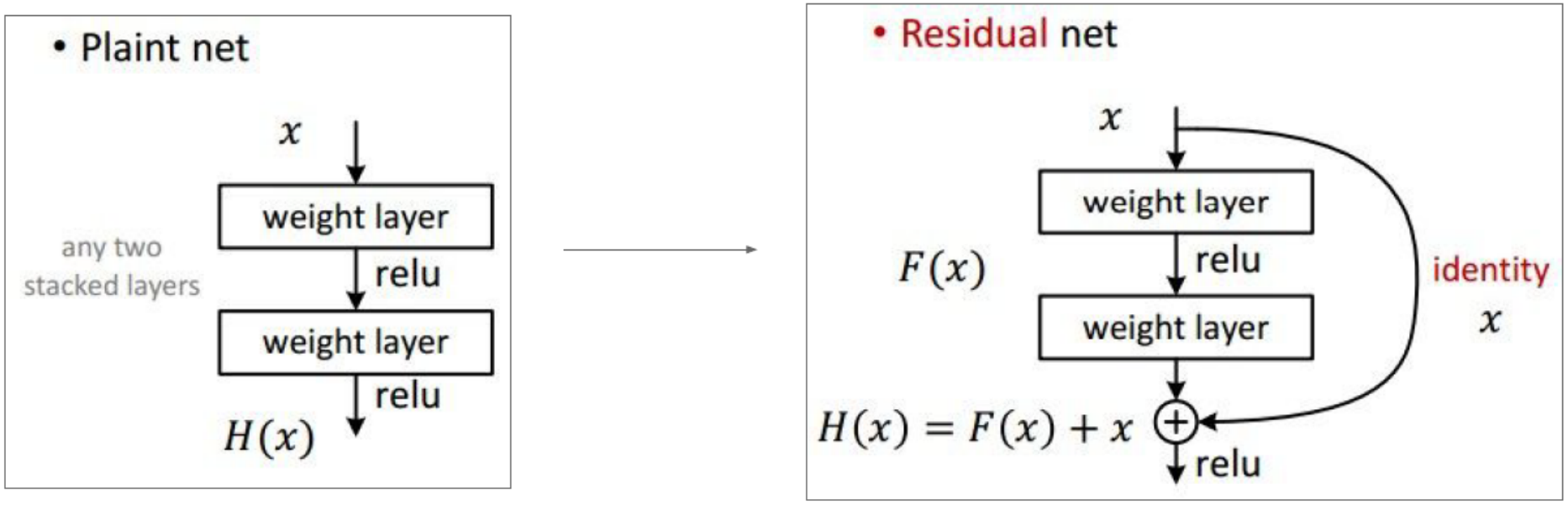
ILSVRC 2015 winner (3.6% top 5 error)



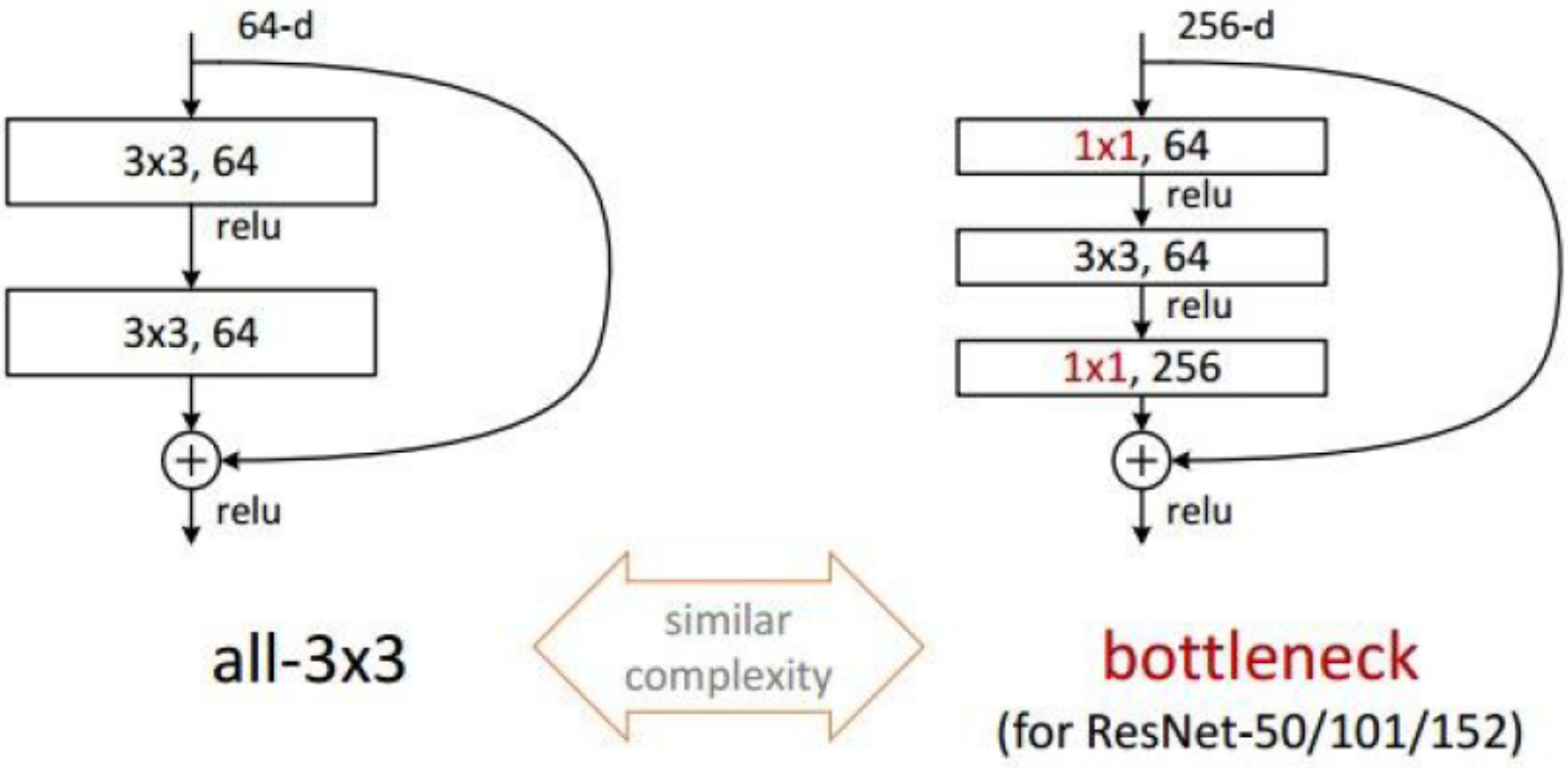
2-3 weeks of training on 8 GPU machine

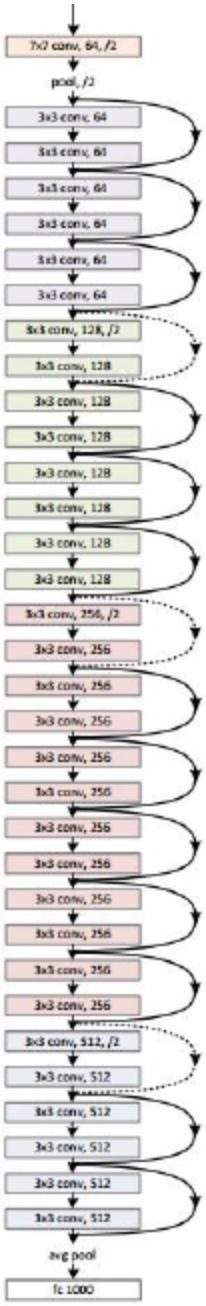
at runtime: faster than a VGGNet! (even though it has 8x more layers)

ResNet [He et al. 2015]



ResNet [He et al. 2015]

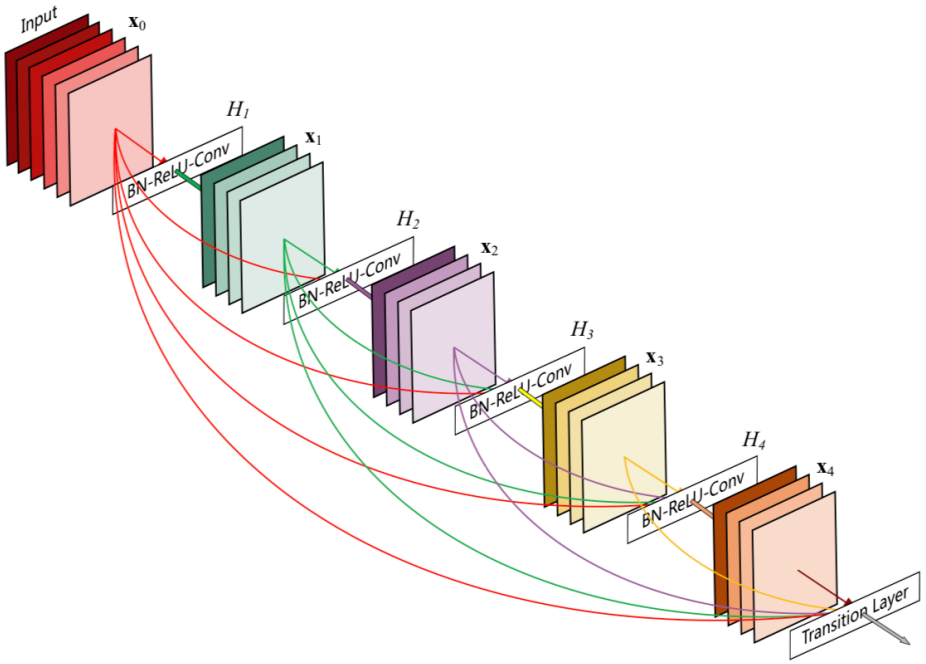




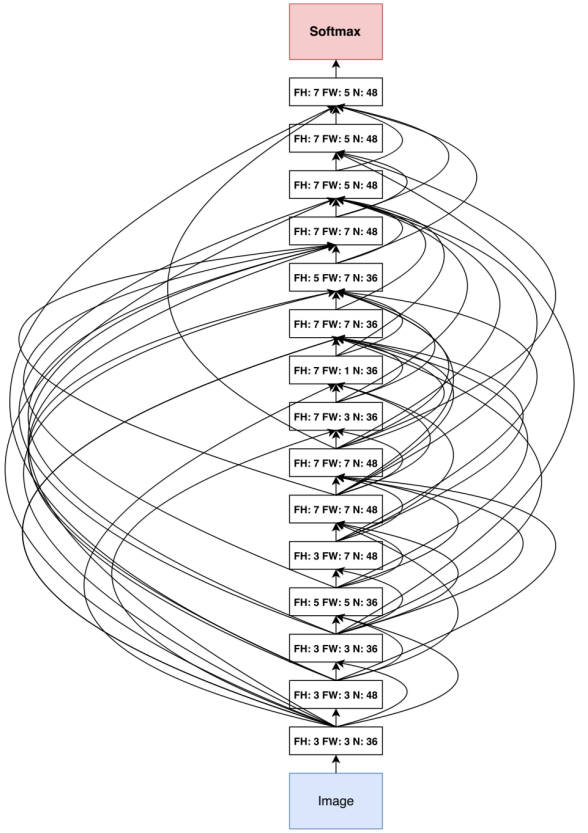
ResNet [He et al. 2015]

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Even More Intricate Architectures



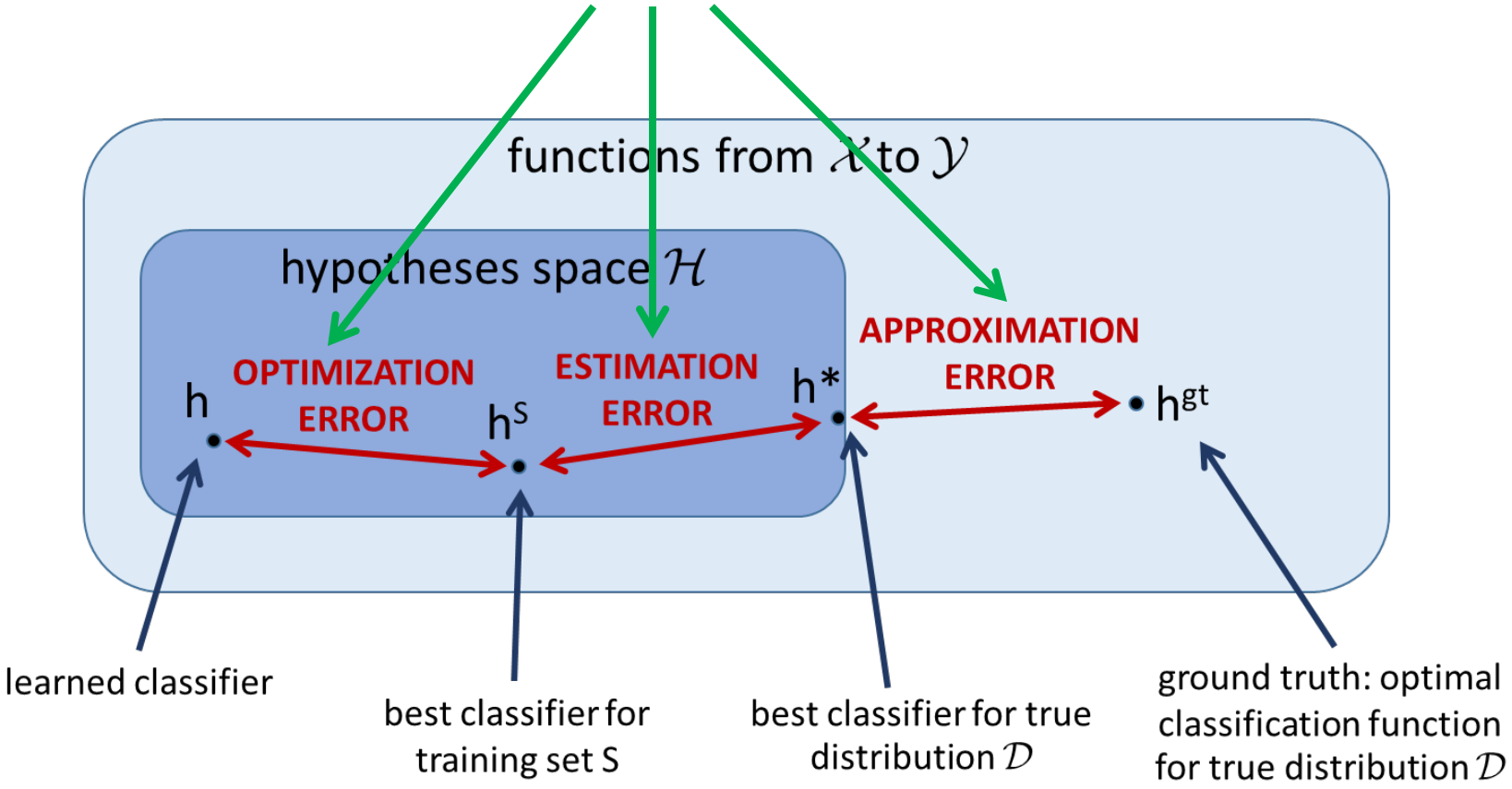
DenseNet (Huang, 2016)



RL Search (Zoph, 2016)

Theoretical Questions and Odd Phenomena

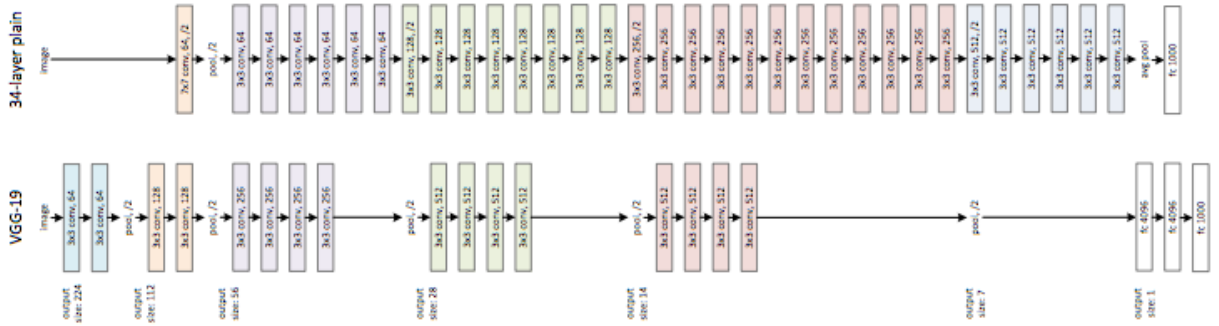
Reminder:
Three Pillars of Machine Learning



In classical machine learning (e.g. SVM) we have a decent understanding of these, in deep learning we do not

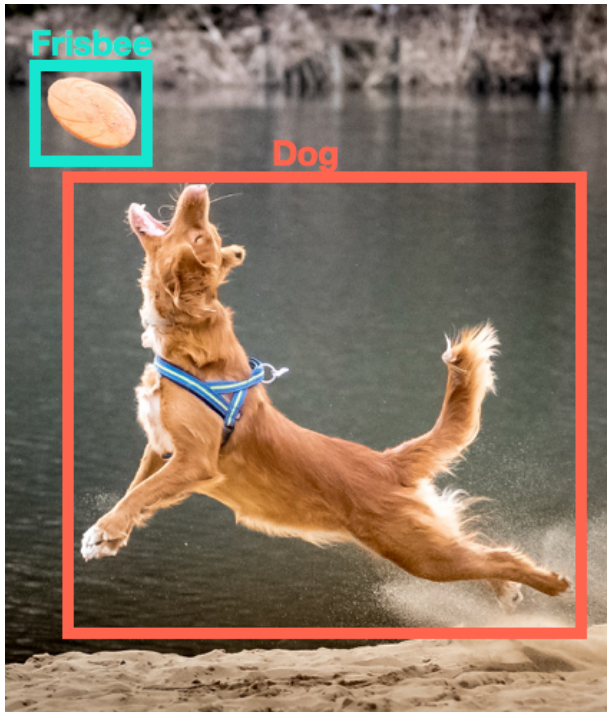
Approximation Error: Fundamental Questions

What is the expressive gain brought forth by depth?



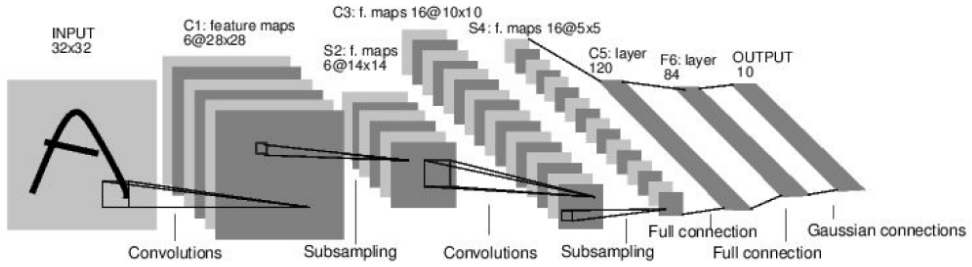
Approximation Error: Fundamental Questions

Why are functions realized by polynomially sized ConvNets suitable for images?



Approximation Error: Fundamental Questions

What is the effect of different architectural features (depth, breadth, etc.)?

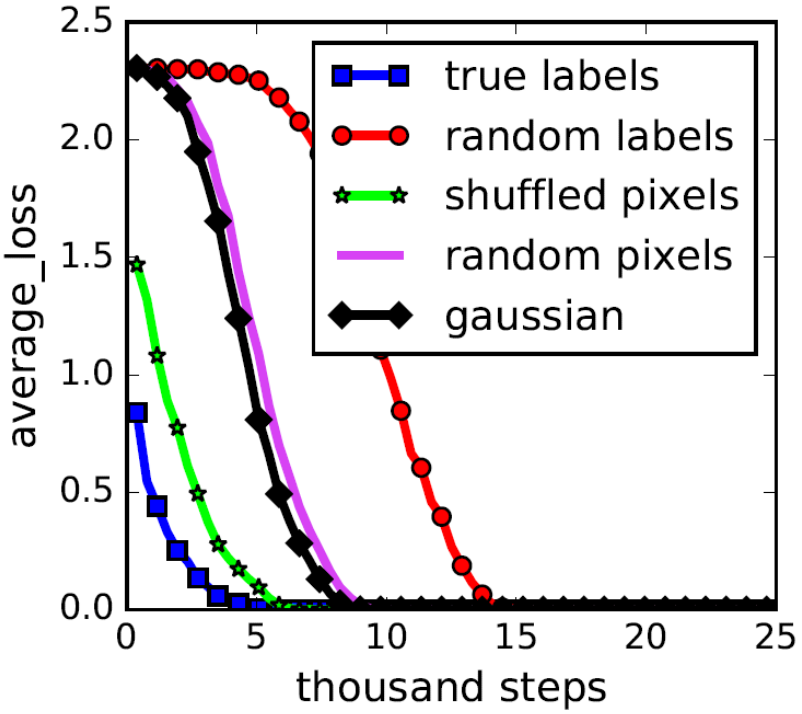


[LeNet-5, LeCun 1980]

Estimation Error: Fundamental Questions

How can models with millions of parameters generalize?

NN can practically memorize the training set…



Estimation Error: Fundamental Questions

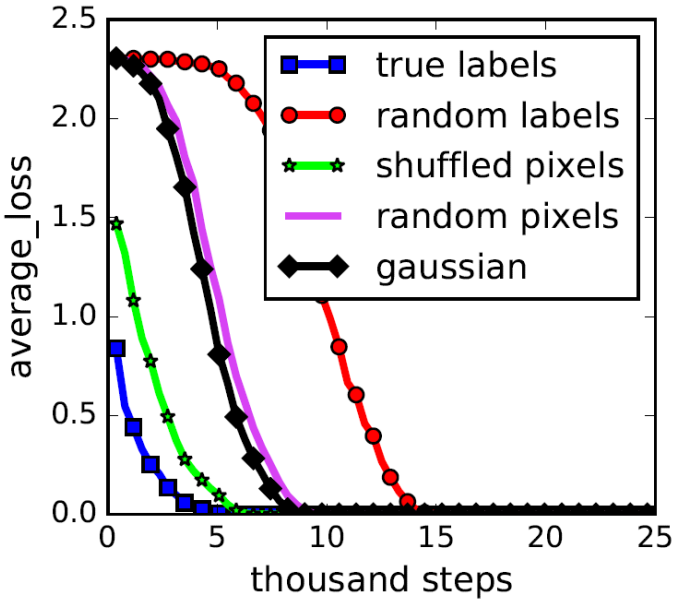
Why do large networks generalize better than small ones?

This somewhat contradicts classical machine learning…

Estimation Error: Fundamental Questions

Does generalization depend on data distribution?

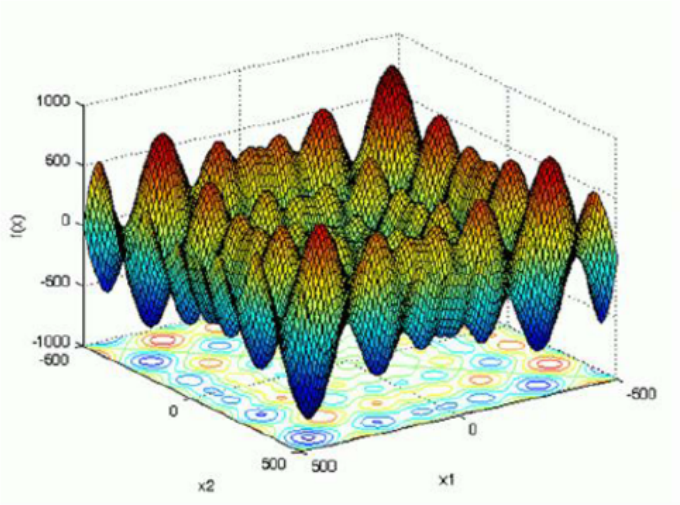
In classical machine learning we have distribution-free sample complexity results…



Optimization Error: Fundamental Questions

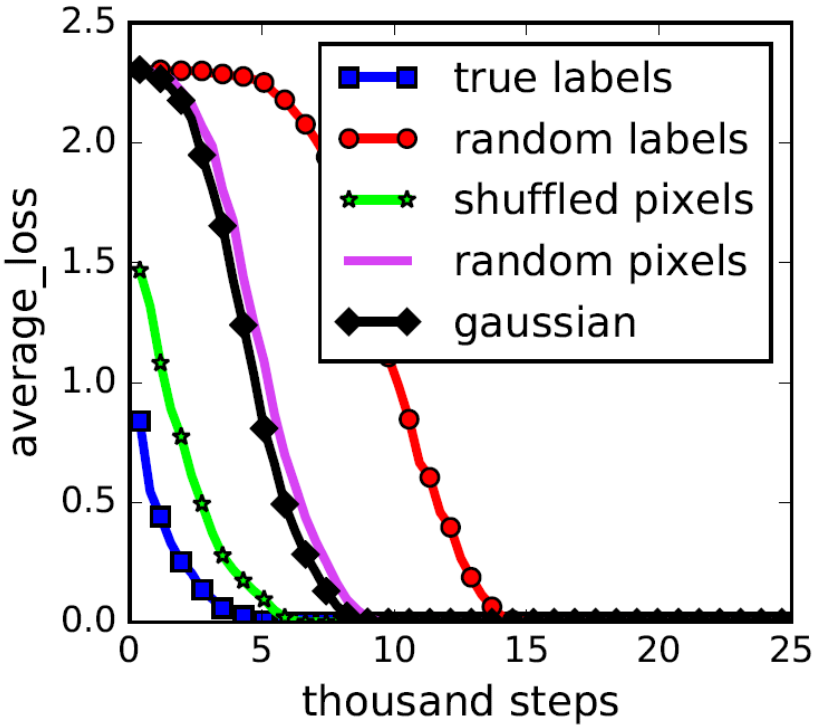
How does SGD optimize a non-convex objective?

In classical machine learning objectives
are in many cases convex…



Optimization Error: Fundamental Questions

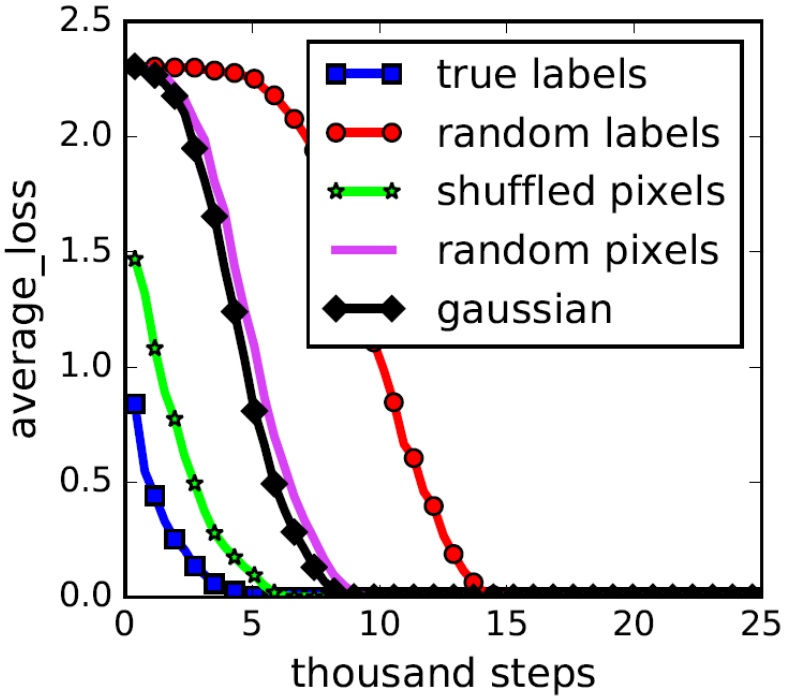
Does convergence depend on data distribution?



Optimization Might Be Connected to Generalization

How the optimization affects the generalization?

- Some evidence that faster convergence to generalization
- Theory suggests SGD is biased to ”simpler” solutions



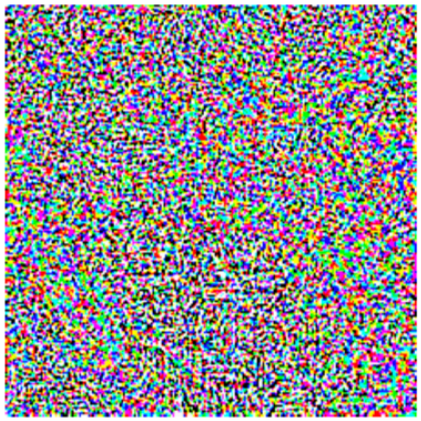
Adversarial Examples

ConvNets could be easily fooled:

$\boldsymbol{x}$

"panda"
57.7% confidence

$+ .007 \times$



$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

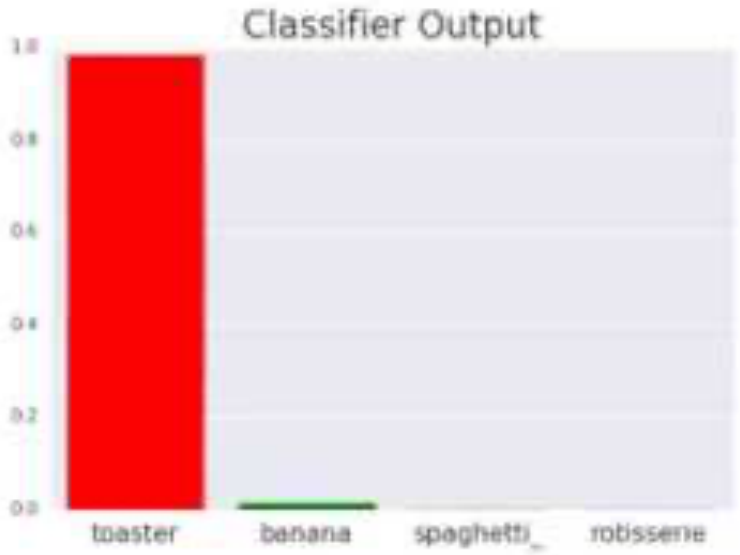
"nematode"
8.2% confidence

$=$

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
99.3 % confidence

Adversarial Examples (cont')



End-to-End Learning

End-to-End Learning Principle

(dog, center, jumping)

(frisbee, top-left, flying)

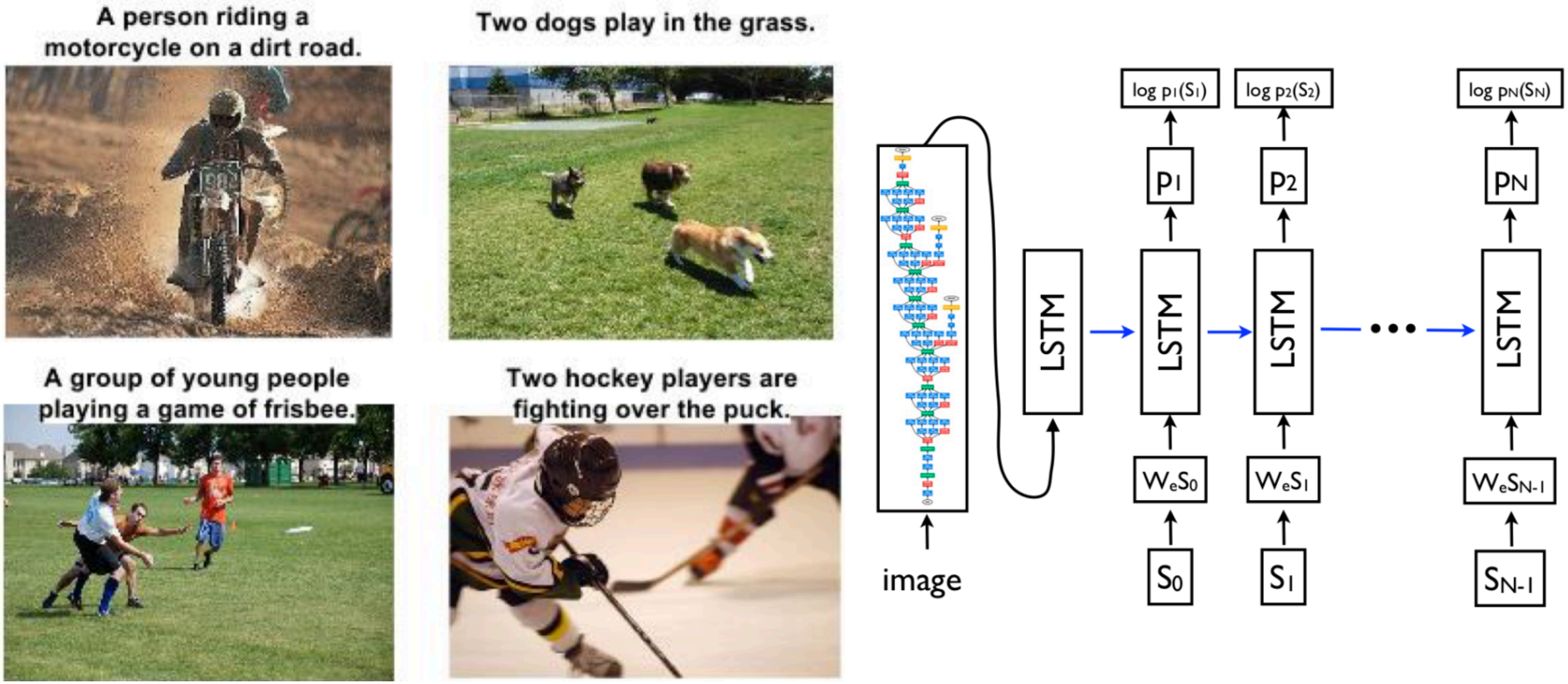
Complex Task

map input directly to output

Complex Task

Domain Translation

Image Captioning: (Image => Text)



(Vinyals et al., 2014)

Domain Translation (cont')

- Lip Reading: (Video => Voice)

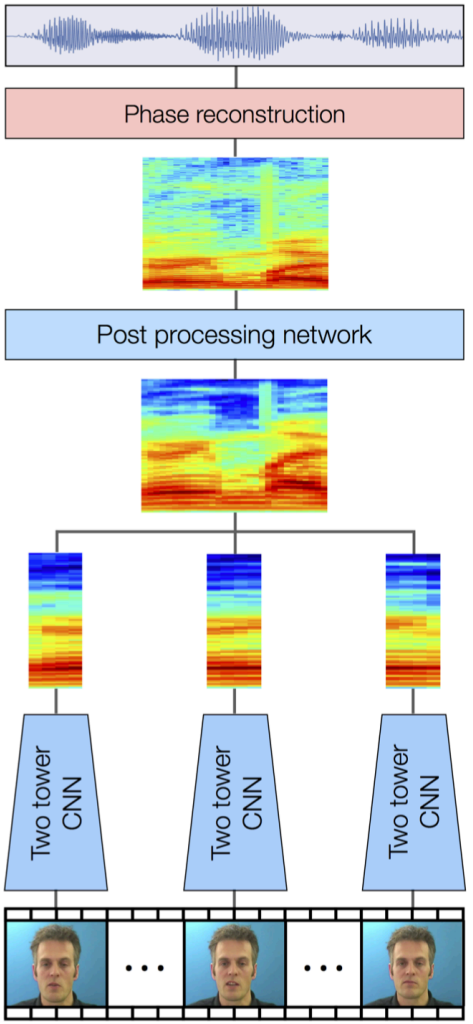
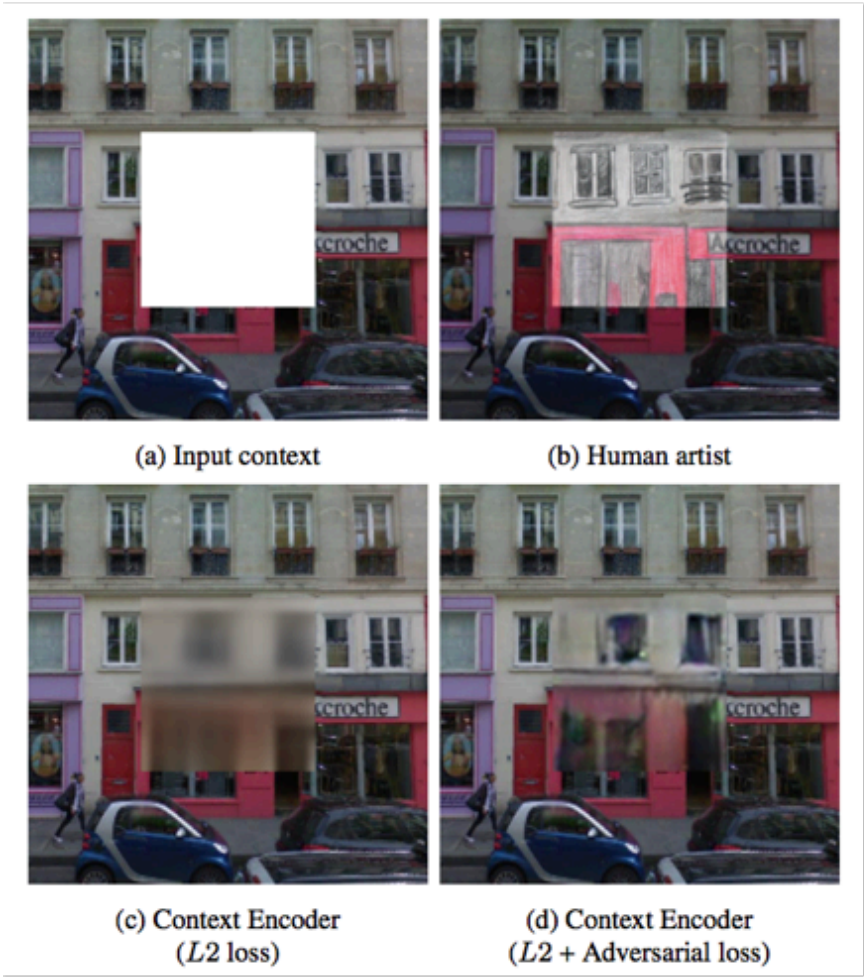


Image Restoration

Image Inpainting



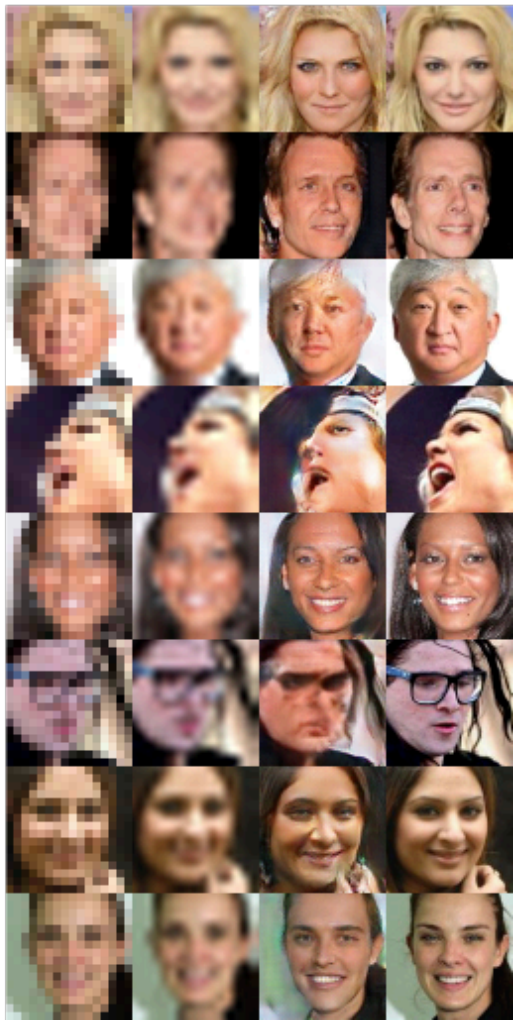
(a) Input context

(b) Human artist

(c) Context Encoder ($L2$ loss)

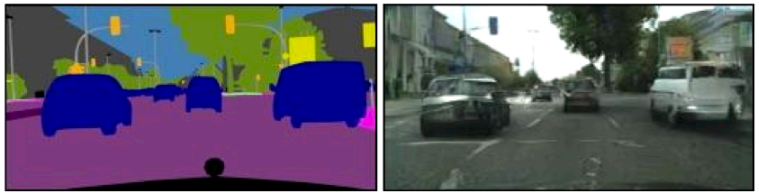
(d) Context Encoder ($L2$ + Adversarial loss)

Super Resolution



… and many more

Labels to Street Scene

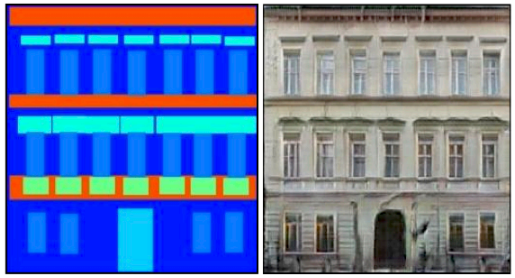


input output

Aerial to Map

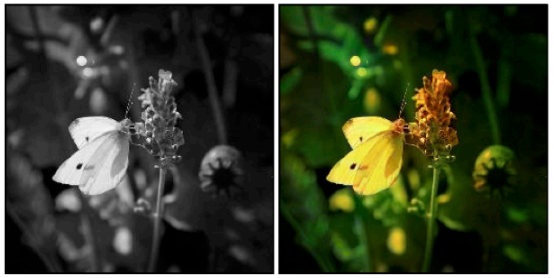
input output

Labels to Facade



input output

BW to Color



input output

Day to Night

input output

Edges to Photo

input output

ConvNets as Generative Models

Generative Models: What & Why

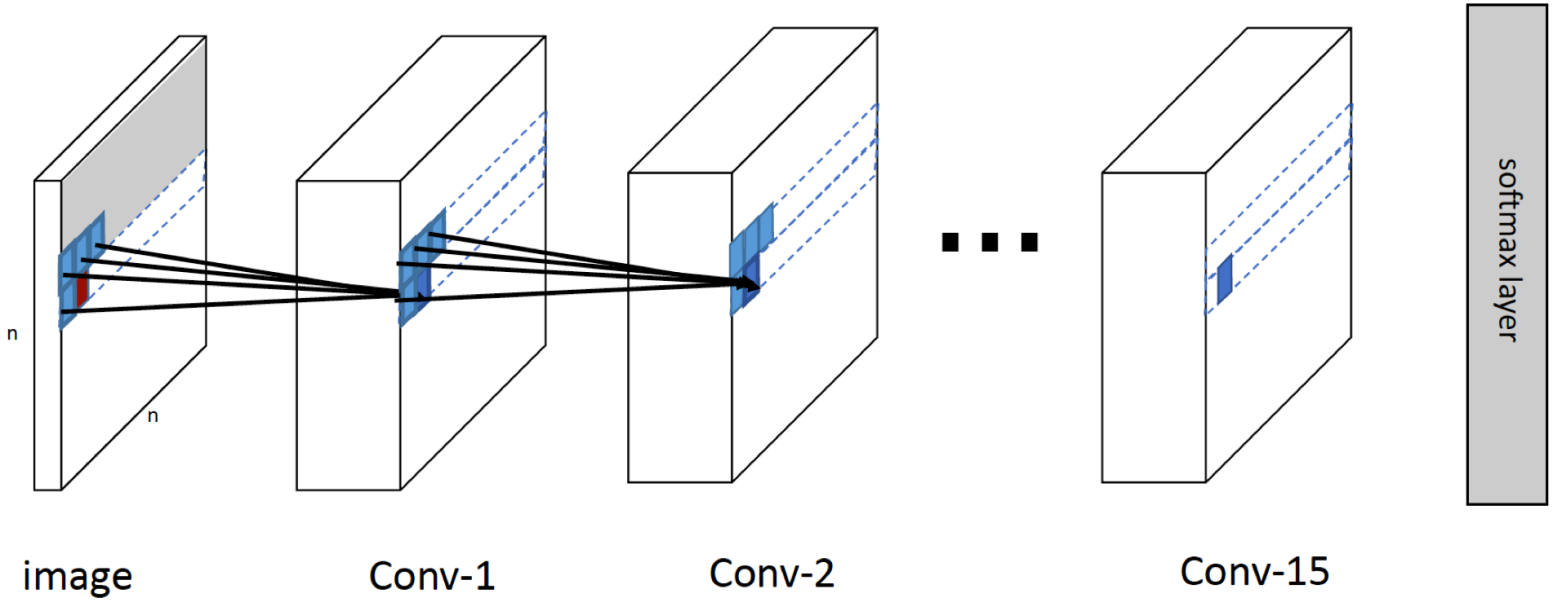
- Represents a distribution over data P(x).
- Doesn’t require labels for training (unsupervised learning).
- Can be used as a representation for other tasks.
- Can be used to detect “outliers” (rare events, bad inputs).
- Can be used to complete missing inputs, or generate more data.

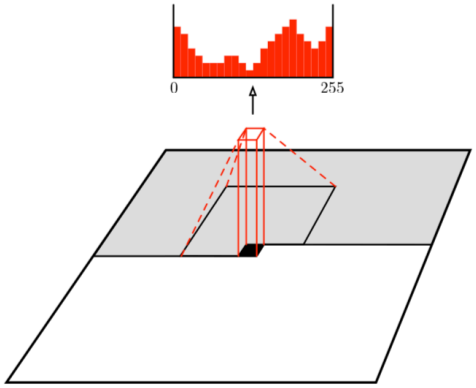
Neural Autoregressive Models

- Any distribution can be written as $P(x_1, \ldots, x_n) = \prod_i P(x_i | x_{i-1}, \ldots, x_1)$ where $x_i \in \{1, \ldots, d\}$
- Represent $P(x_i | x_{<i})$ as a network.
- NN has $n \times d$ outputs, denoted by $f_{ij}(x_{<i})$

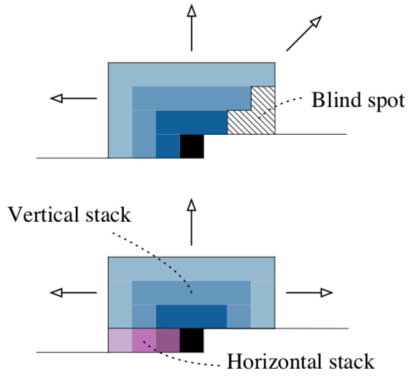
$$P(x_i = j | x_{<i}) = \frac{\exp(f_{ij}(x_{<i}))}{\sum_k \exp(f_{ik}(x_{<i}))}$$

PixelCNN

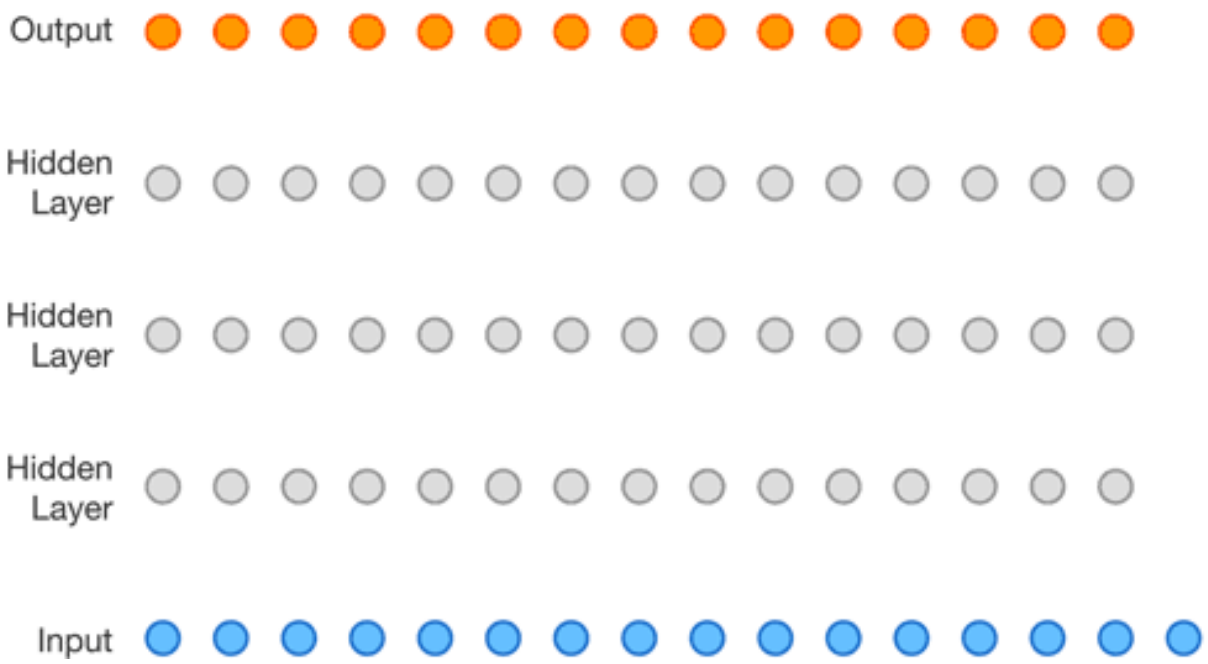




1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0



WaveNet



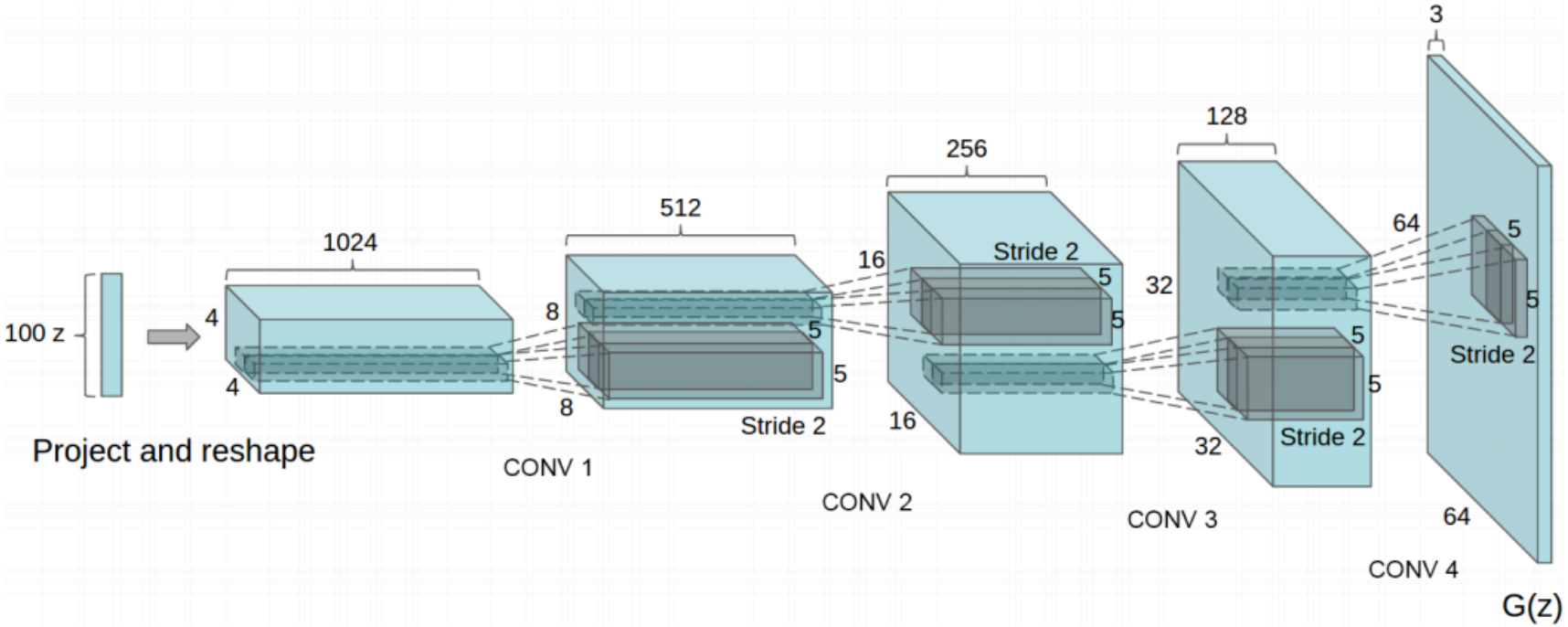
See examples at:
https://deepmind.com/blog/wavenet-generative-model-raw-audio/

Generative Adversarial Networks

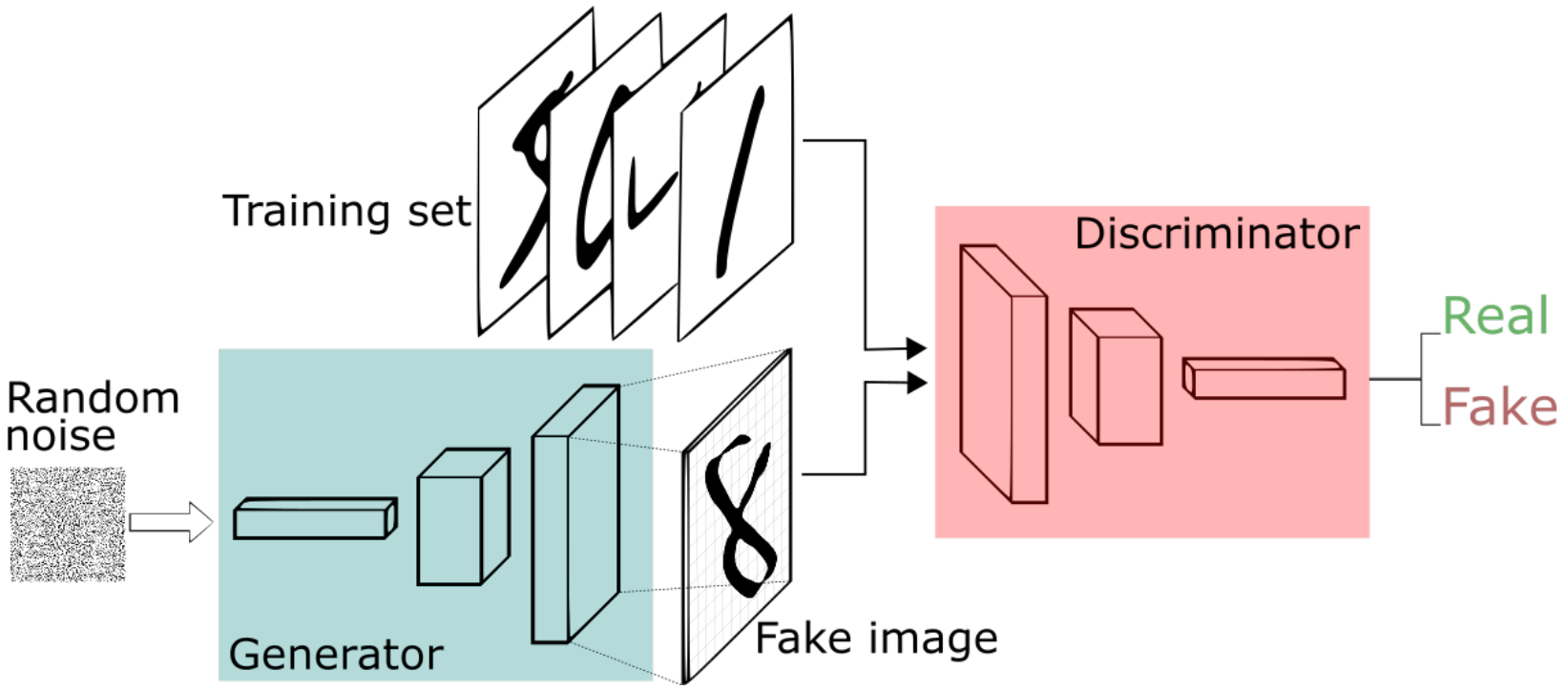
- Most distributions cannot be represent through a tractable likelihood function!
- Assume we have a differentiable generative process, then how could we train it without a likelihood function?

Generator as a Network

Sample a random vector $z \in \mathbb{R}^d$ and then run it through a network:



Training as a Game



Training as a Game (cont')

- Let $G_\theta(z)$ be a generator.
- Train a network $D_\omega(x)$ to discriminate between real images and "fake" images coming from G.
- Train G to "fool" D.

Optimization Problem:

$$\min_{\Theta} \max_{\Phi} V(\Theta, \Phi) = \underbrace{\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D_\Phi(\mathbf{x})]}_{\text{Classify as "real"}} + \underbrace{\mathbb{E}_{\mathbf{z} \sim p_z}[\log(1 - D_\Phi(G_\Theta(\mathbf{z}))]}_{\text{Classify as "fake"}}$$

Shockingly “Real” Results

Summary

Summary

- Baseline ConvNet architecture:

  $[(\text{conv-ReLU}) \times \text{N} - \text{pool}] \times \text{M} \rightarrow (\text{dense-ReLU}) \times \text{K}$

- Recent state-of-the-art architectures (e.g. GoogLeNet, ResNet) challenge this paradigm
- Ongoing trend: “the deeper the better”
- Training involves many “bells and whistles”
- Fundamental theoretical questions remain open