Convolutional Networks @ DL Indaba

Nando de Freitas, Martin Gorner, Karen Simonyan

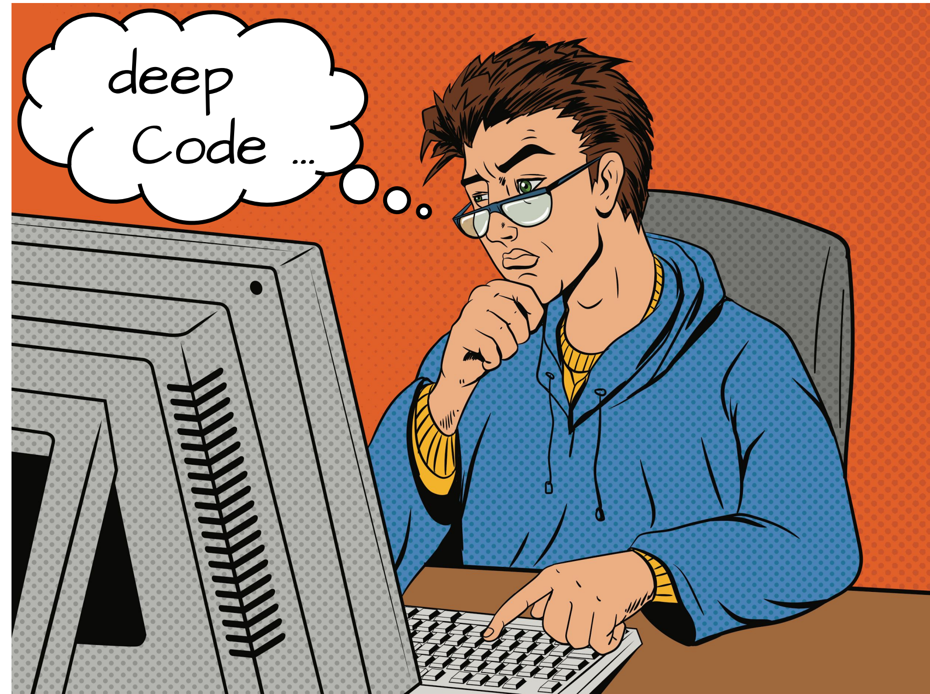
Lecture Outline

- Recap.
- Convolutional layers.
- Convolutional neural networks.
- Going deeper: the challenges and how to solve them.
- Beyond image classification.

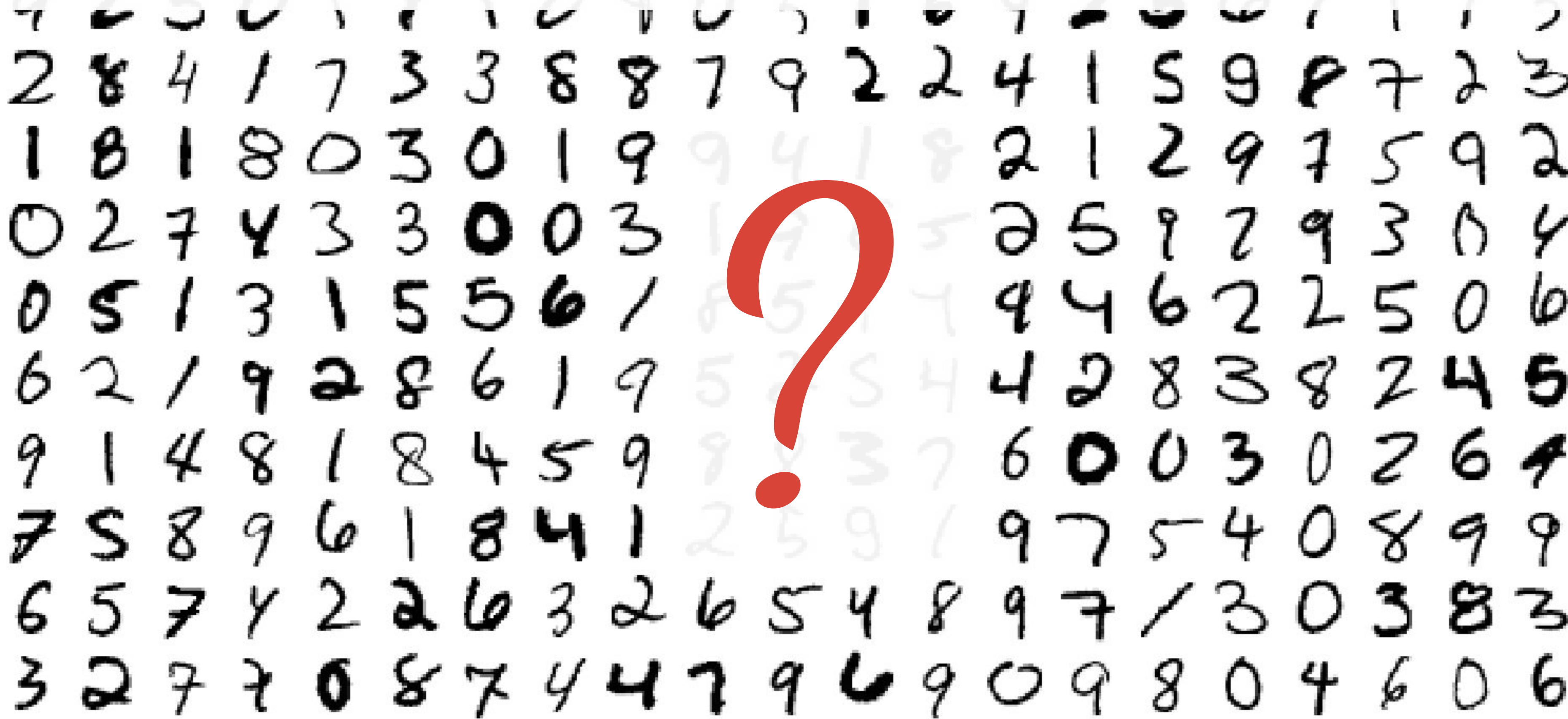
Recap

>TensorFlow and deep learning_

without a PhD

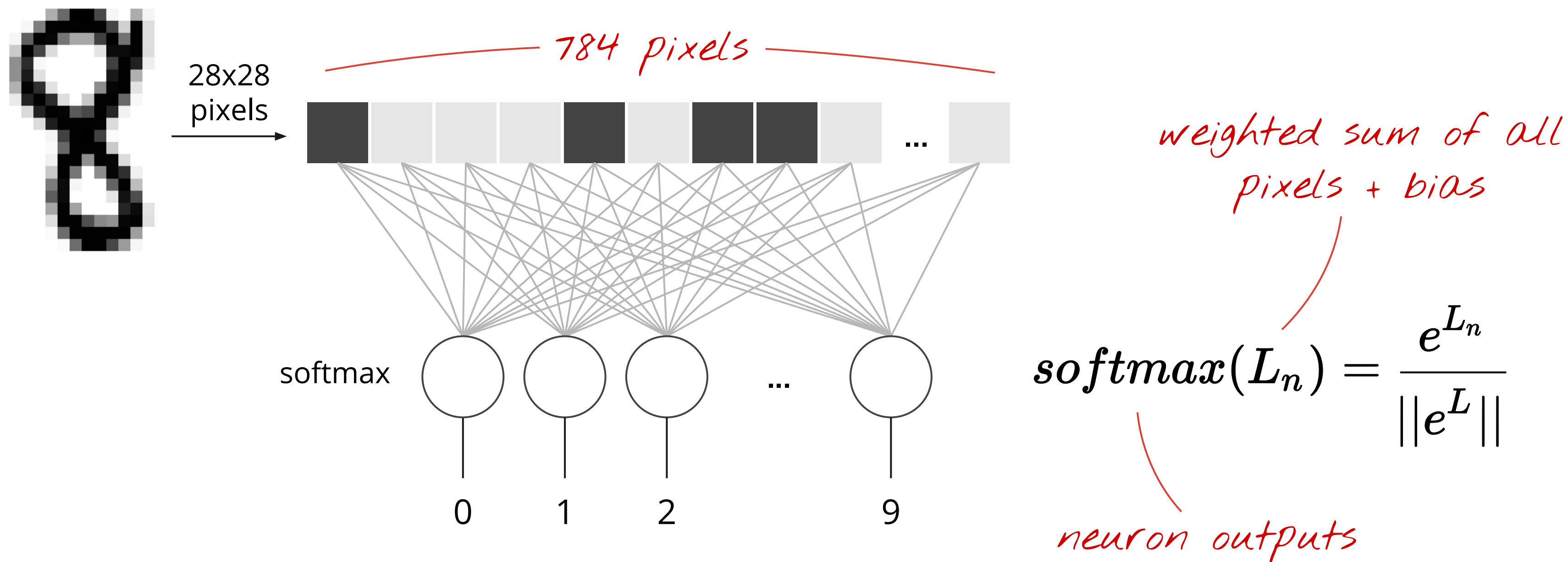


Hello World: handwritten digits classification - MNIST



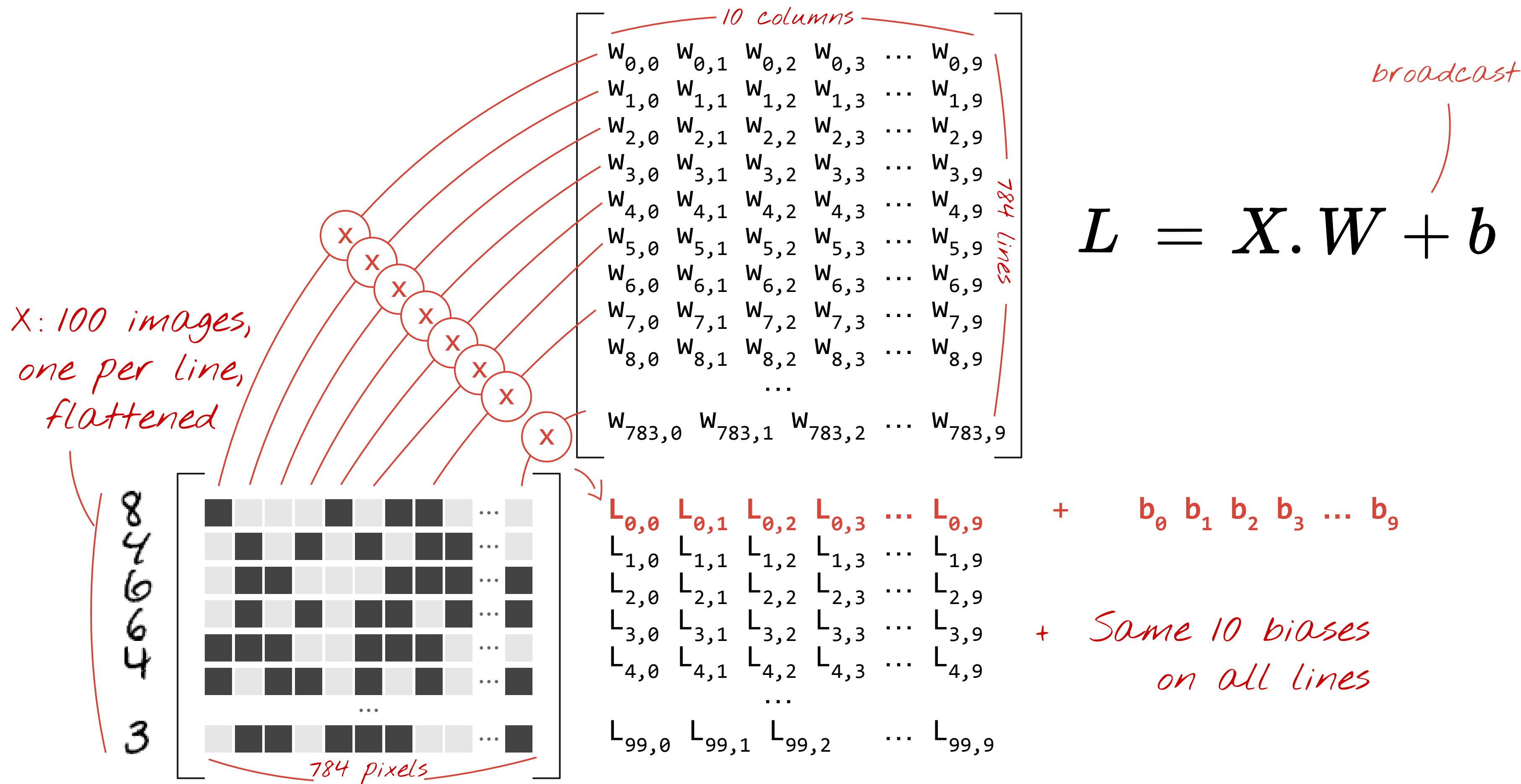
MNIST = Mixed National Institute of Standards and Technology - Download the dataset at http://yann.lecun.com/exdb/mnist/

Very simple model: softmax classification



$$softmax(L_n) = \frac{e^{L_n}}{||e^L||}$$

In matrix notation, 100 images at a time



Softmax, on a batch of images

Predictions

Y[100, 10]

Images

X[100, 784]

Weights

W[784,10]

Biases

b[10]

$$Y = softmax(X.W + b)$$

applied line by line

matrix multiply

broadcast on all lines

tensor shapes in [ ]

Now in TensorFlow (Python)

tensor shapes: X[100, 784] W[748,10] b[10]

```
Y = tf.nn.softmax(tf.matmul(X, W) + b)
```

matrix multiply

broadcast
on all lines

Success ?

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	**1**	0	0	0

actual probabilities, "one-hot" encoded

Cross entropy: $-\sum Y'_i . log(Y_i)$

computed probabilities

this is a "6"

0.1	0.2	0.1	0.3	0.2	0.1	**0.9**	0.2	0.1	0.1
0	1	2	3	4	5	6	7	8	9

TensorFlow - initialisation

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

init = tf.initialize_all_variables()
```

this will become the batch size, 100

28 x 28 grayscale images

Training = computing variables W and b

TensorFlow - success metrics

flattening images

```
# model
Y = tf.nn.softmax(tf.matmul(tf.reshape(X, [-1, 784]), W) + b)
# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])
```

"one-hot" encoded

```
# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))
```

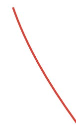
"one-hot" decoding

```
# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
```

TensorFlow - training

learning rate

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)
```



loss function

TensorFlow - run !

```
sess = tf.Session()
sess.run(init)

for i in range(1000):
    # load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)

    # success ?
    a,c = sess.run([accuracy, cross_entropy], feed_dict=train_data)

    # success on test data ?
    test_data={X: mnist.test.images, Y_: mnist.test.labels}
    a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```

running a Tensorflow computation, feeding placeholders

Tip: do this every 100 iterations

TensorFlow - full python code

initialisation

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
init = tf.initialize_all_variables()

# model
Y=tf.nn.softmax(tf.matmul(tf.reshape(X,[-1, 784]), W) + b)

# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])

# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct,tf.float32))
```

model

success metrics

training step

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)

sess = tf.Session()
sess.run(init)

for i in range(10000):
    # load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)

    # success ? add code to print it
    a,c = sess.run([accuracy, cross_entropy], feed=train_data)

    # success on test data ?
    test_data={X:mnist.test.images, Y_:mnist.test.labels}
    a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```

Run

Go deep !

Let’s try 5 fully-connected layers !

overkill

784

200

100

60

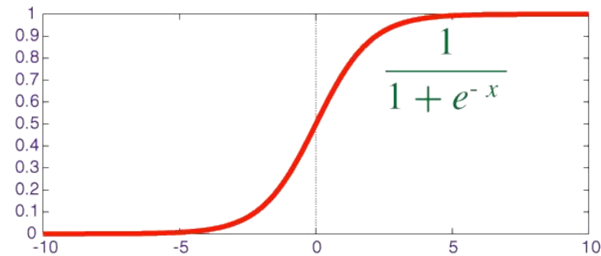
30

10

softmax

0 1 2 ... 9

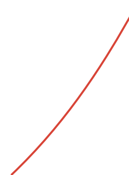
sigmoid function



TensorFlow - initialisation

```
K = 200
L = 100
M = 60
N = 30

W1 = tf.Variable(tf.truncated_normal([28*28, K] ,stddev=0.1))
B1 = tf.Variable(tf.zeros([K]))
W2 = tf.Variable(tf.truncated_normal([K, L], stddev=0.1))
B2 = tf.Variable(tf.zeros([L]))
W3 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B3 = tf.Variable(tf.zeros([M]))
W4 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B4 = tf.Variable(tf.zeros([N]))
W5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))
```

weights initialised with random values

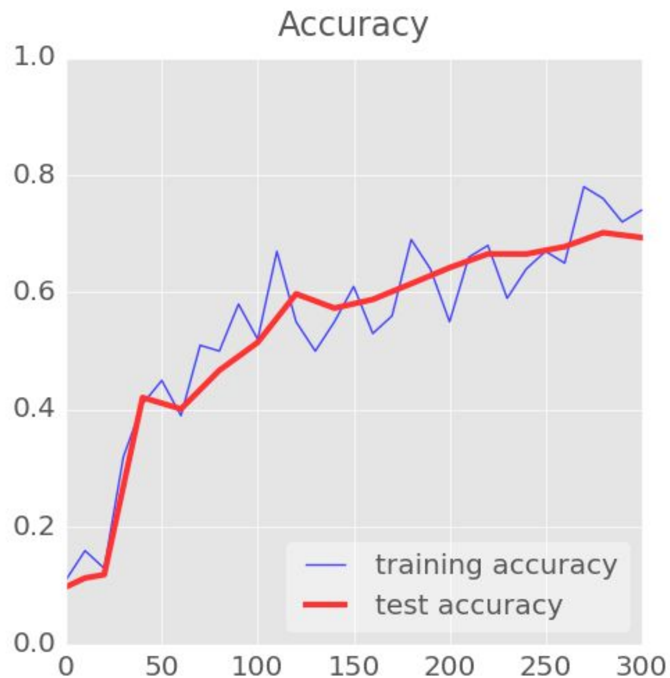


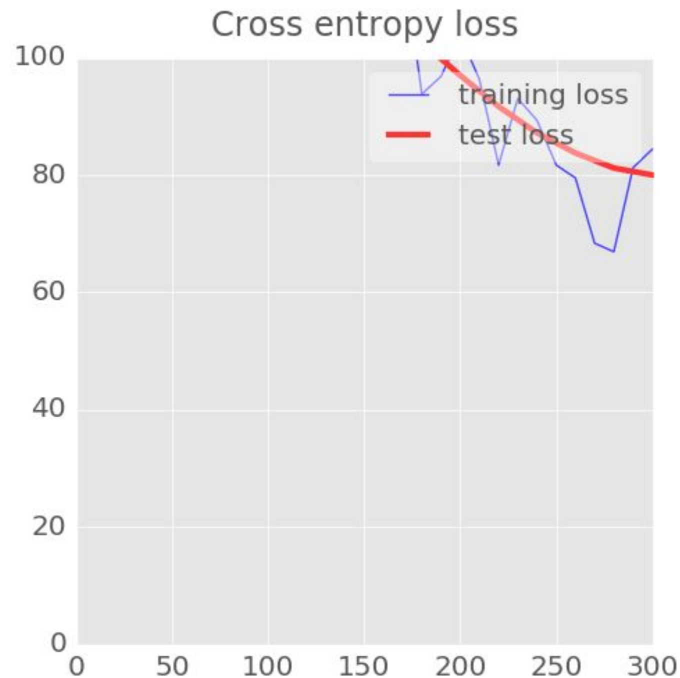
TensorFlow - the model

weights and biases

```
X = tf.reshape(X, [-1, 28*28])

Y1 = tf.nn.sigmoid(tf.matmul(X, W1) + B1)
Y2 = tf.nn.sigmoid(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.sigmoid(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.sigmoid(tf.matmul(Y3, W4) + B4)
Y = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

Demo - slow start ?

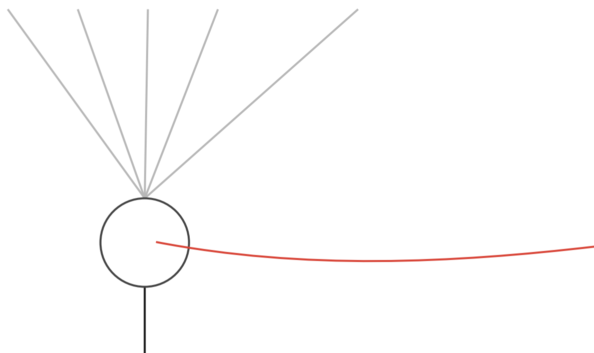


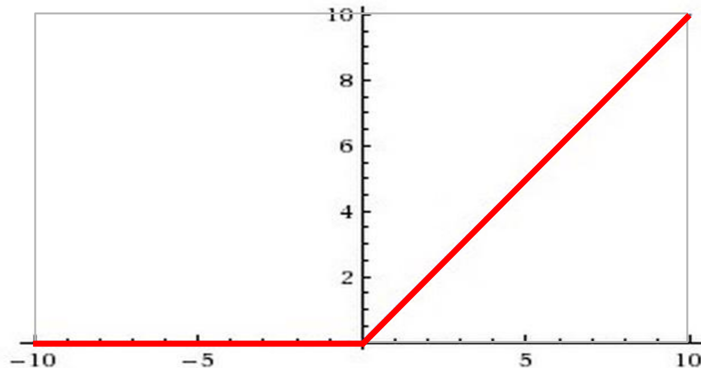


Relu !

RELU

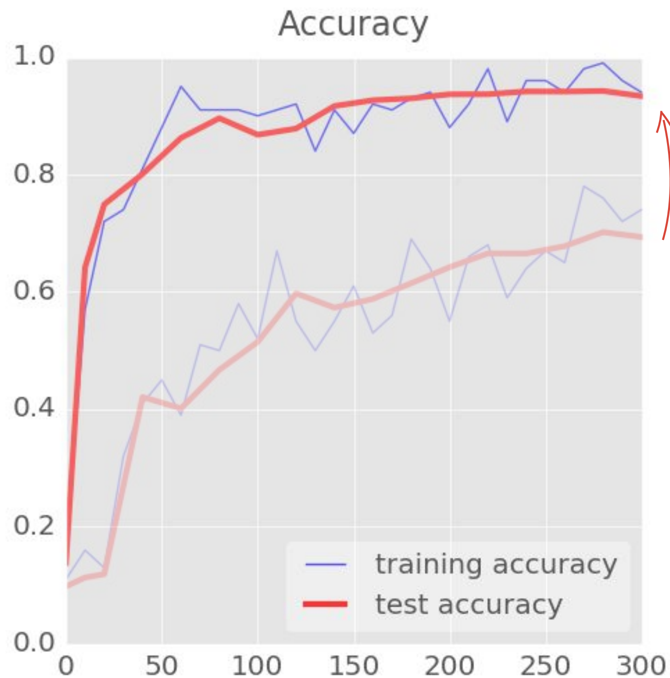
RELU = Rectified Linear Unit

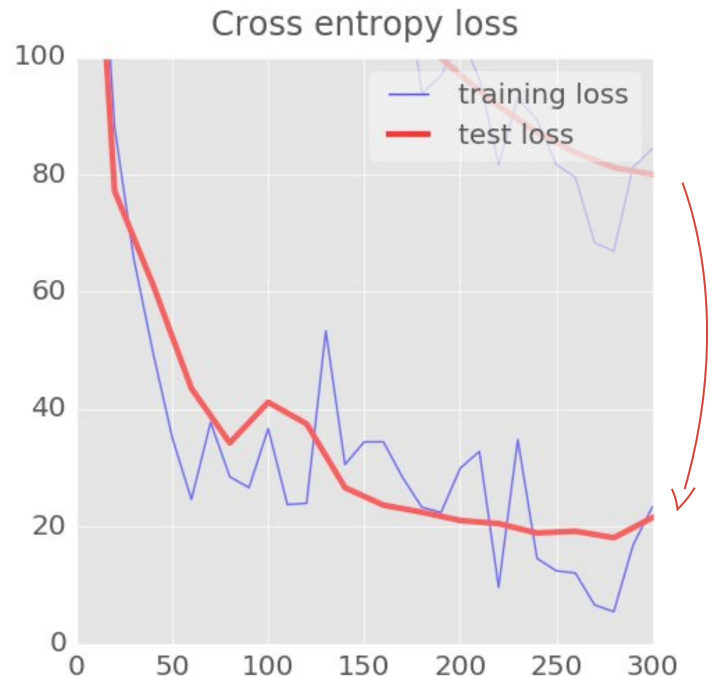




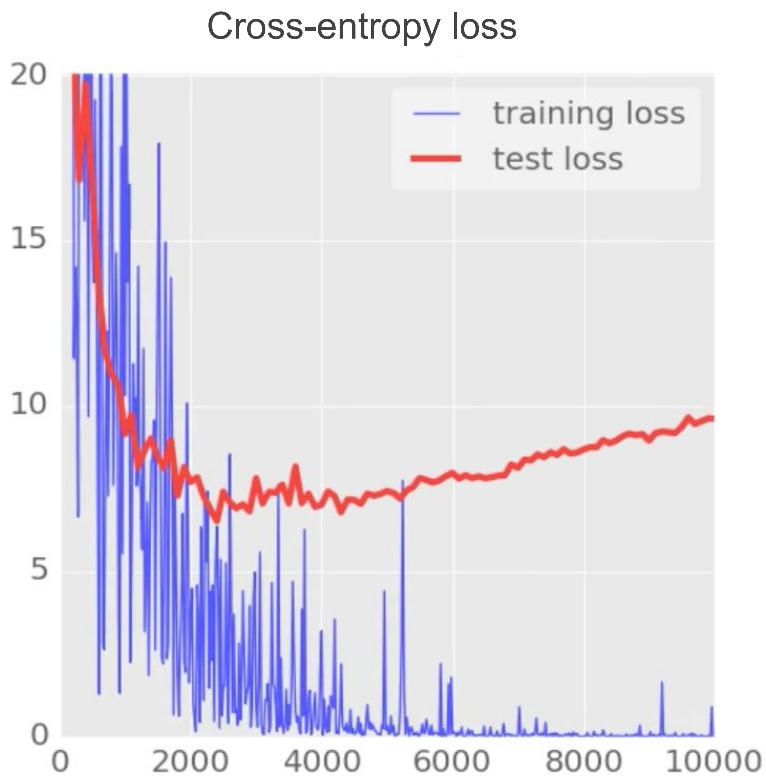

```
Y = tf.nn.relu(tf.matmul(X, W) + b)
```

RELU





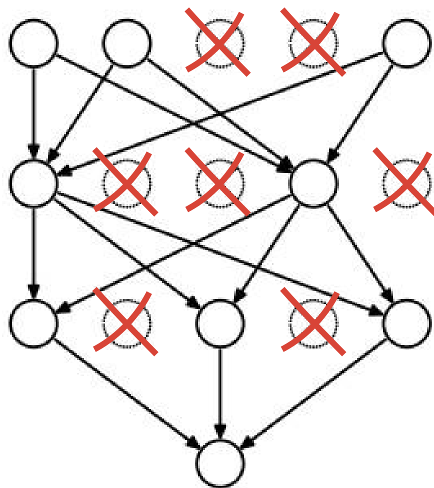
Overfitting

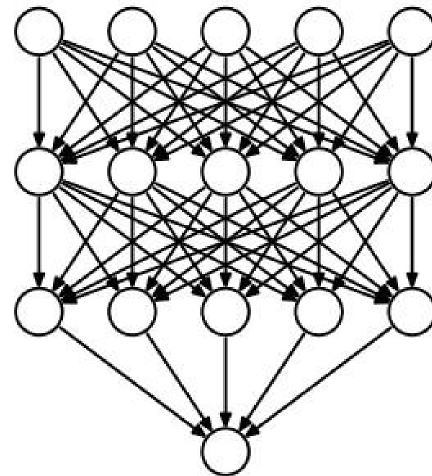


Overfitting ?

Dropout

Dropout

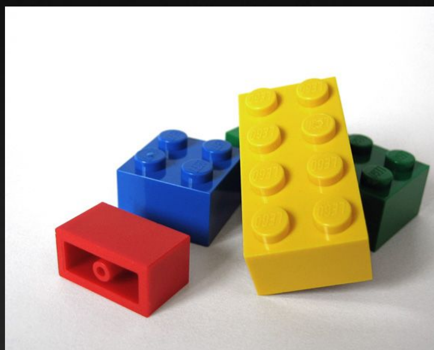




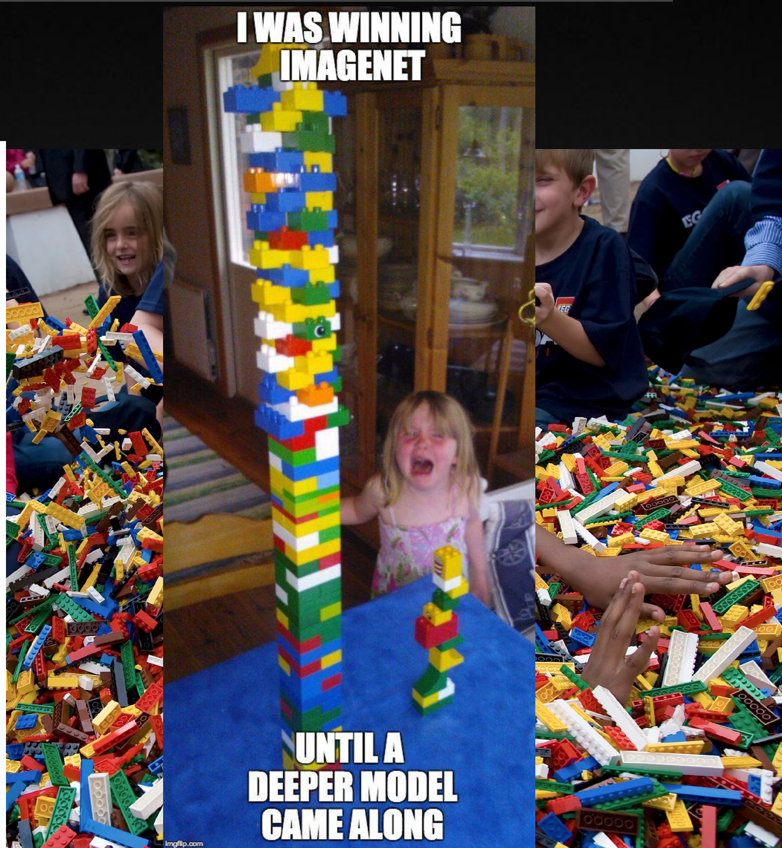

```
pkeep =
tf.placeholder(tf.float32)

Yf = tf.nn.relu(tf.matmul(X, W) + B)
Y  = tf.nn.dropout(Yf, pkeep)
```

Deep learning research is like playing with lego

Not like this



But rather like this



Combinatorial re-use is robust and amazing for creativity

Convolutional layers

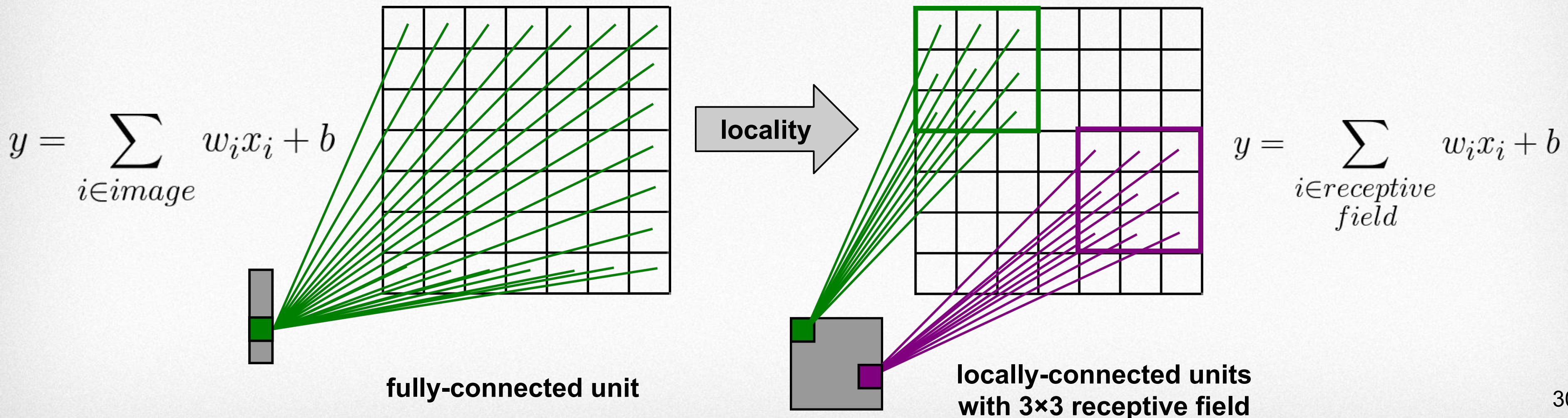
Motivation: Locality and translation invariance

- **Locality**: objects tend to have a local spatial support
- **Translation invariance**: object appearance is independent of location

The bird occupies a local area and looks the same in different parts of an image.
We should construct neural nets which exploit these properties!

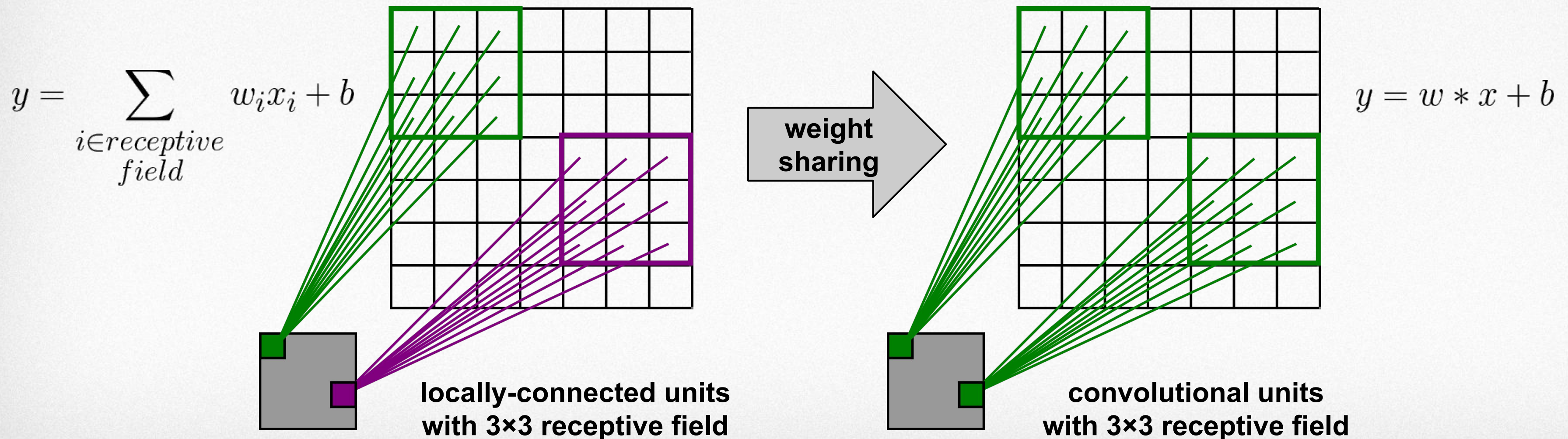
Incorporating locality assumptions

- Make **fully**-connected layer **locally**-connected
- Each unit/neuron is connected to a local rectangular area – receptive field
- Different units connected to different locations
  - output (“**feature map**”) lies on a grid itself



Incorporating invariance assumptions

- **Weight sharing**
  - units connected to different locations have the same weights
  - equivalently, each unit is applied to all locations
- *Convolutional layer – locally-connected layer with weight sharing (translation invariance)*



Correlation and convolution

$z = [\, z_1 \mid z_2 \,]$ $M_Z = 2$

$w = [\, w_1 \mid w_2 \,]$ $M_F = 2$

$x = [\, x_1 \mid x_2 \mid x_3 \,]$ $M_I = 3$

$$\begin{cases} z_1 = w_1 x_1 + w_2 x_2 \\ z_2 = w_1 x_2 + w_2 x_3 \end{cases}$$

Flip $\overrightarrow{\ }$ $\bar{w} = [\, w_2 \mid w_1 \,]$

$$z_{i'} = \sum_{i=1}^{M_F=2} w_i \, x_{i'+i-1} \quad \text{Correlation (similarity)}$$

$$\parallel$$

$$z_{i'} = \sum_{i=1}^{M_F=2} x_{i'+i-1} \, \bar{w}_{M_F-i+1} \quad \text{(convolution)}$$

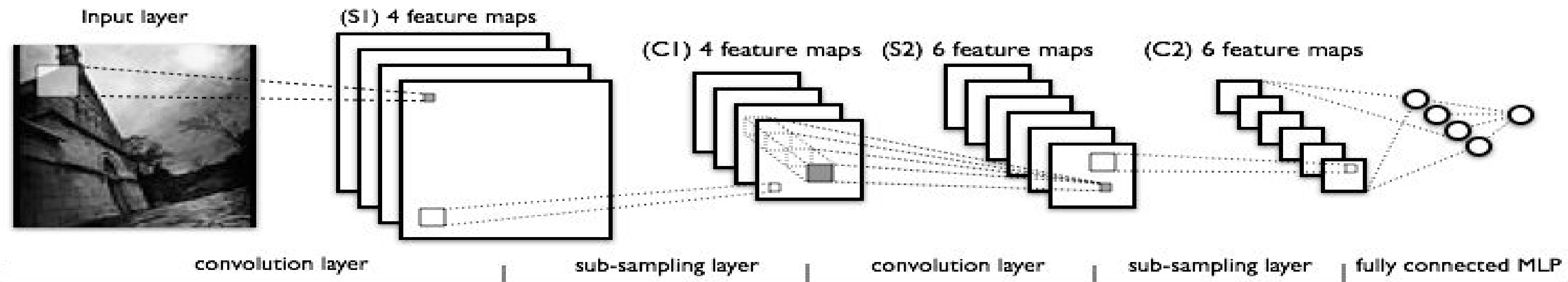
Convolution as searching for patterns

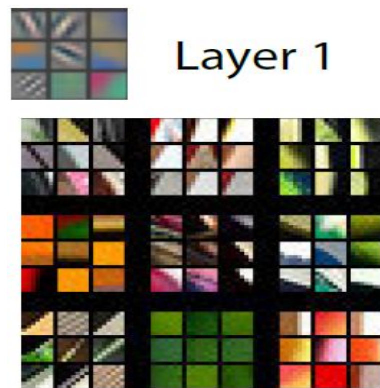
For the image, take dark pixel value = 1, light pixel value = 0.

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

Leow Wee Kheng

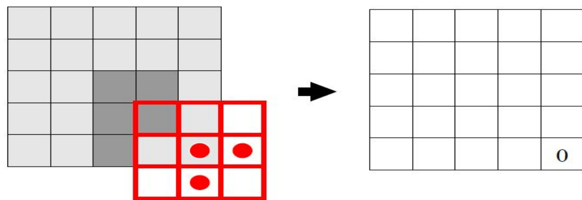
Convolutional networks



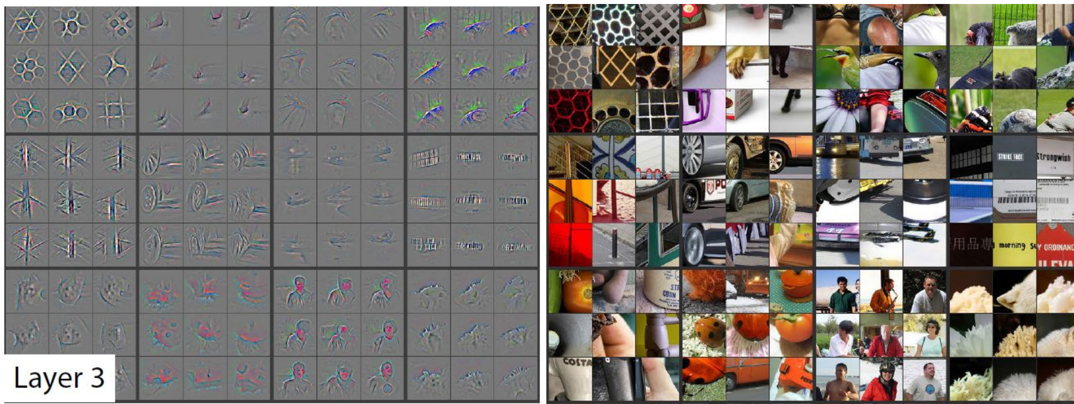


Layer 2

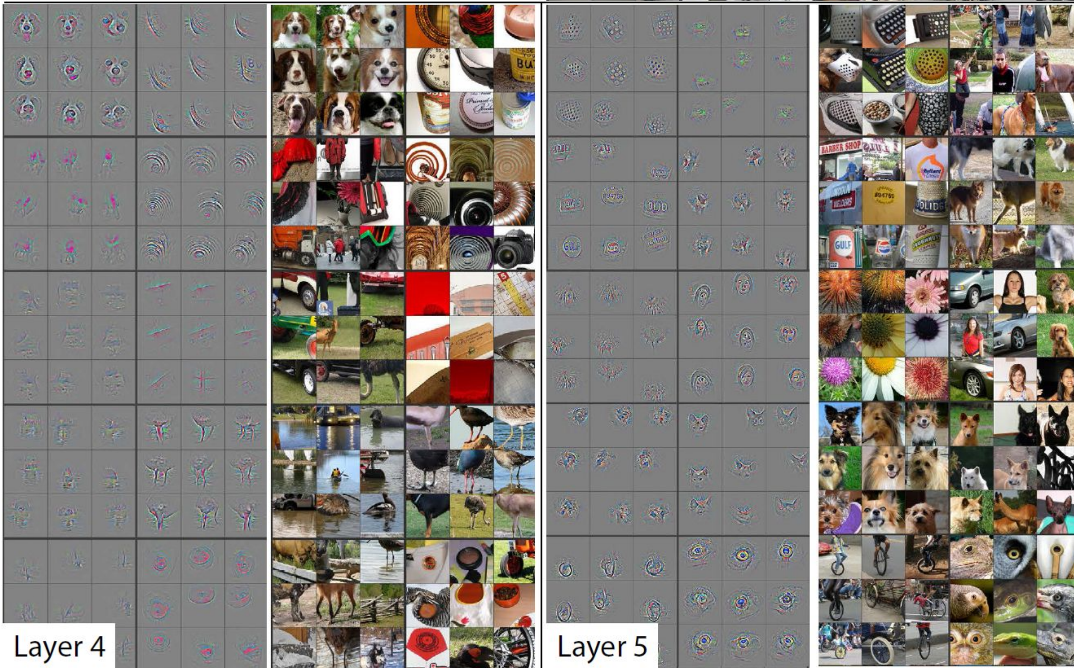
[Matthew Zeiler & Rob Fergus]



0



Layer 3



Layer 4
Layer 5

Hubel-Wiesel -> Fukushima -> Lecun and Hinton

Electrical signal from brain

Recording electrode

Visual area of brain

Stimulus

Cell response

Stimulus

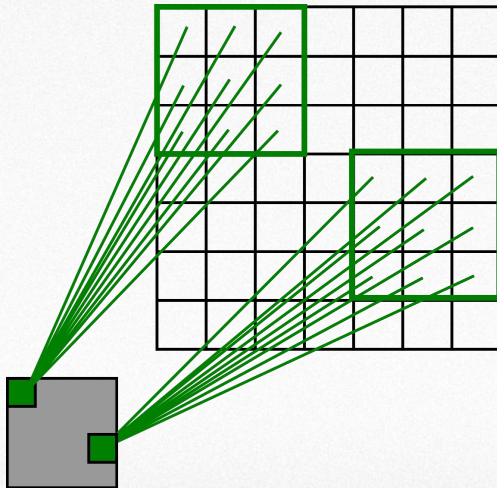
http://hubel.med.harvard.edu/book/70.jpg

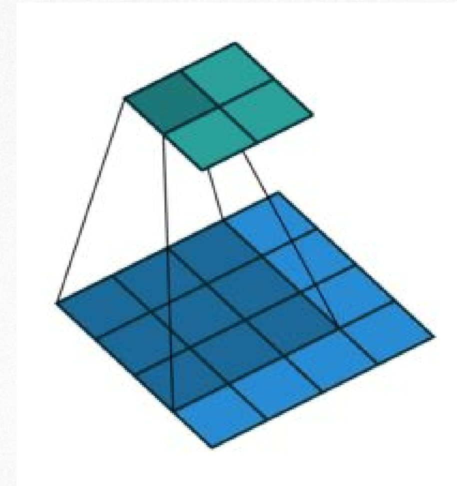
Conv Layer Mechanics

single-channel scenario

- Weight matrix of conv layer is called conv kernel (or filter)
- To compute the output feature map
  - slide the receptive field of the filter over the input and compute dot products
  - receptive field size == filter size

$y = w * x + b$





filter has
3*3=9 weights

evaluation of a single conv filter with 3×3 receptive field on 4×4 input produces 2×2 output

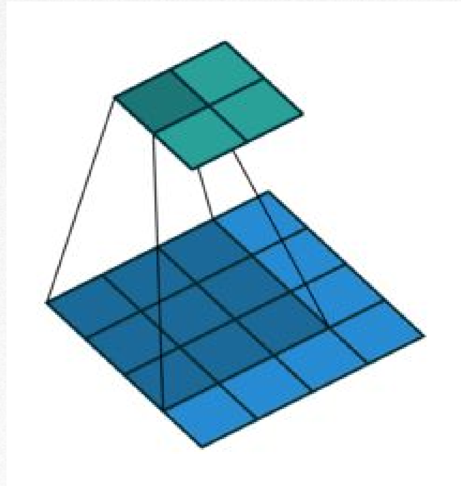
animation credit: https://github.com/vdumoulin/conv_arithmetic

Conv Layer Mechanics

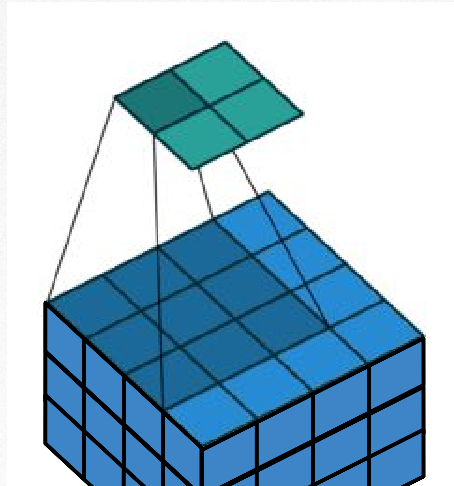
multi-channel scenario

feature maps are 3-D tensors
height × width × channels

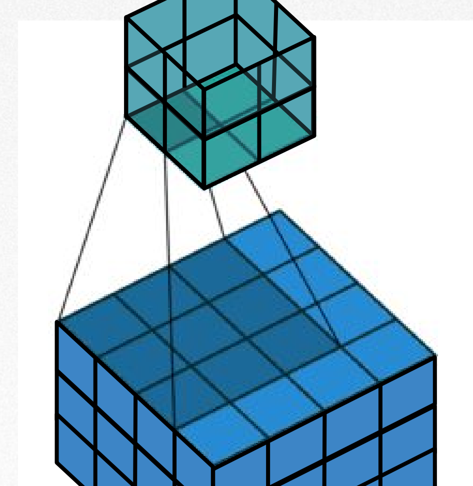
- Conv layer input and output can have multiple channels
  - e.g. 3-channel RGB image or 16-channel feature map



4×4×1 input, 2×2×1 output
3*3=9 filter weights



4×4×3 input, 2×2×1 output
3*3*3=27 filter weights

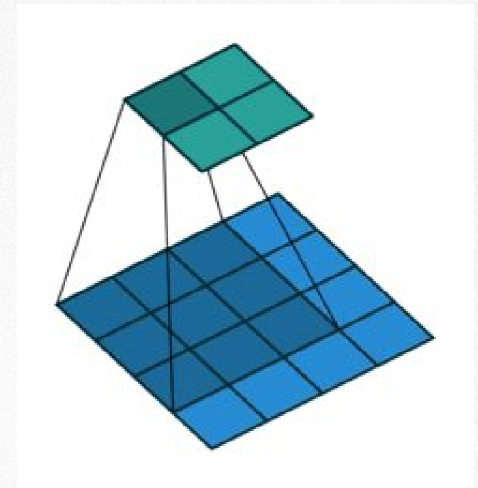


4×4×3 input, 2×2×2 output
3*3*3*2=54 filter weights

Conv Layer Mechanics

Output size

- We'll assume multi-channel input & output from now on
- For N×N input and kernel size k×k the output size is M = N - k + 1
- We consider all receptive fields lying fully within the input: known as **'VALID' convolution**

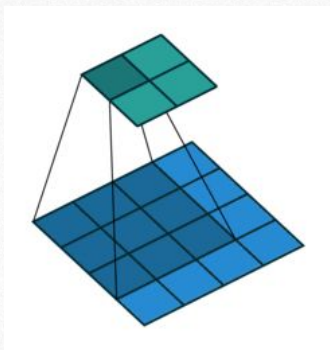


4×4×cin input
2×2×cout output

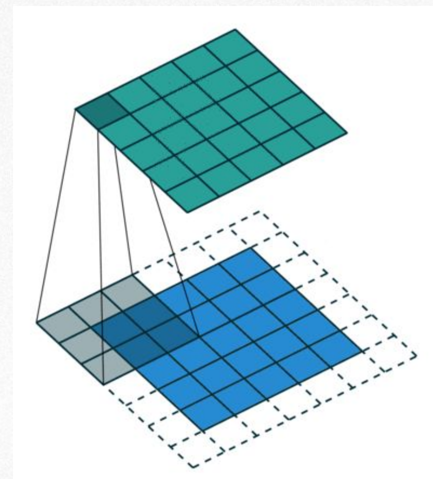
Conv Layer Variants

Padded Convolution

- Increase (pad) the input with p zeros on both sides
  - sometimes implemented as a separate padding layer
- Purpose: control output resolution (e.g. preserve resolution)
- Common settings
  - 'VALID': p=0
  - 'SAME': p = (k - 1)/2 on each side for kernel size k
    - receptive fields go beyond the original input
    - output has the same spatial size as the input



4×4×cin input, 2×2×cout output
'VALID' padding



5×5×cin input, 5×5×cout output
'SAME' padding

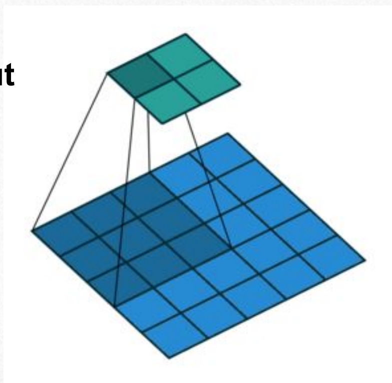
Conv Layer Variants

Strided Convolution

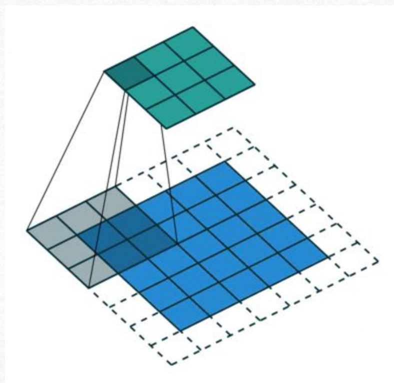
- Conv filter can be applied with a step ("**stride**") between receptive fields
- Purposes
  - reduce spatial resolution for faster processing
  - achieve invariance to local translation
- Output size: $M = \left\lfloor \frac{N + 2p - k}{s} \right\rfloor + 1$ for input size N, kernel size k, padding p, and stride s

M×M output

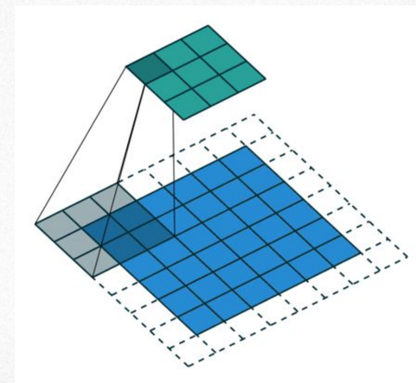
N×N input



N=5, k=3, p=0, s=2 ⇒ M=2



N=5, k=3, p=1, s=2 ⇒ M=3

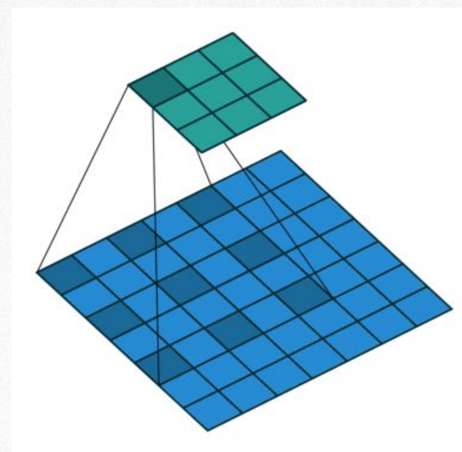


N=6, k=3, p=1, s=2 ⇒ M=3

Conv Layer Variants

Dilated Convolution

- Conv filter is applied with a step (“**dilation rate”)** between kernel elements
  - k×k kernel dilated to size k’=k+(k-1)(r-1), where r is dilation rate
- Purposes
  - large receptive field with a small kernel
  - fast alternative to large kernels
- Output size
  - computed based on the dilated kernel size k’
  - $M = \left\lfloor \frac{N + 2p - k'}{s} \right\rfloor + 1$



k=3, r=2 ⇒ k’=5
N=7, k’=5, p=0, s=1 ⇒ M=3

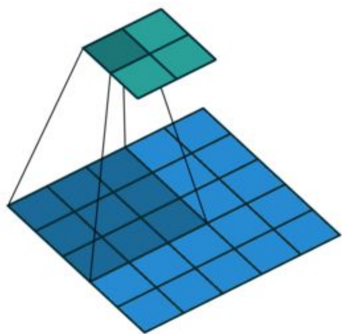
Conv Layer Variants

Transposed Convolution

- Also known as: up-convolution, de-convolution, fractionally-strided convolution
- Purpose: **increase** the resolution
- Does the opposite of strided convolution
  - implemented by swapping forward and backward operations of standard convolution
- Output size $N = s(M-1) + k - 2p$

M×M output

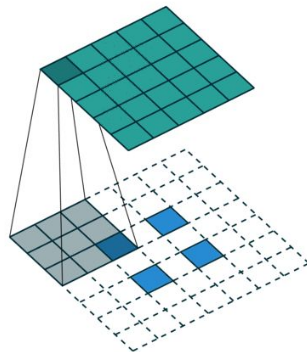
N×N input



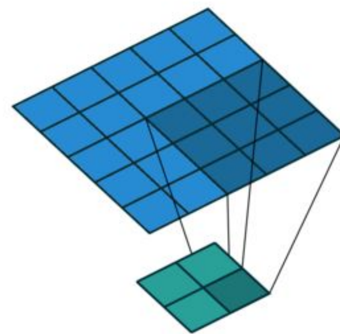
strided convolution

N×N output

M×M input

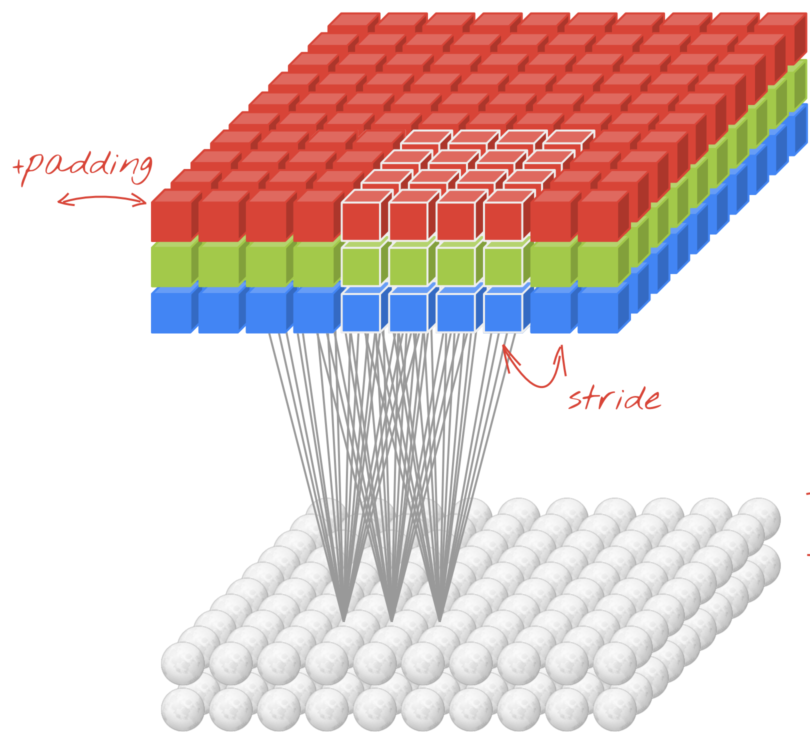


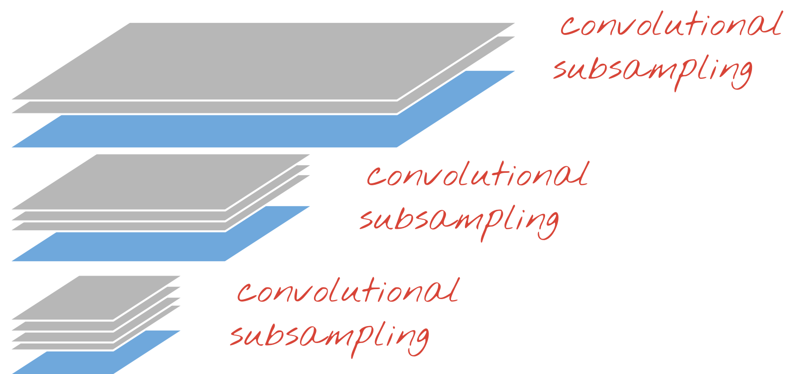
≡



transposed convolution

Convolutional layer





$W_1[4, 4, 3]$

$W_2[4, 4, 3]$

$$W[4, 4, 3, 2]$$

filter size — input channels — output channels

Pooling Layer

- Purposes (same as strided convolution)
  - reduce spatial resolution for faster processing
  - achieve invariance to local translation
- Average pooling
  - computes the average input over the receptive field
  - same as k×k strided convolution with weights fixed to 1/(k*k)
- Max pooling
  - computes the max input over the receptive field
- Global pooling
  - pooling with the whole input as the receptive field
  - gets rid of spatial dimensions, full invariance to location
  - can be average or max

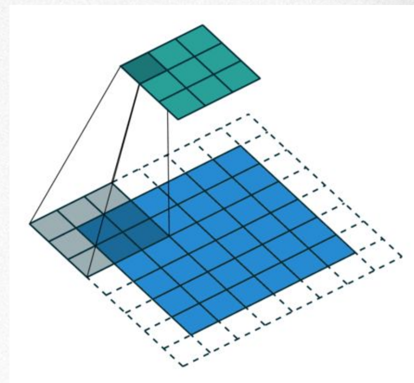
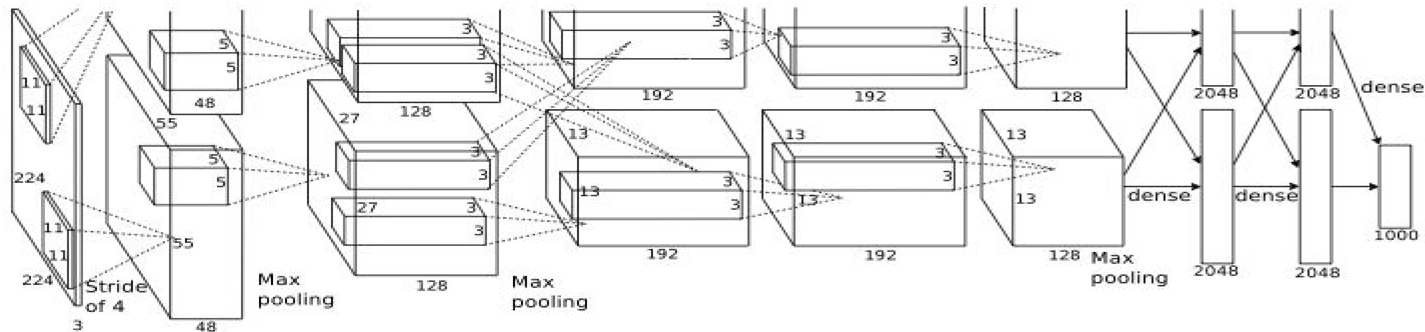
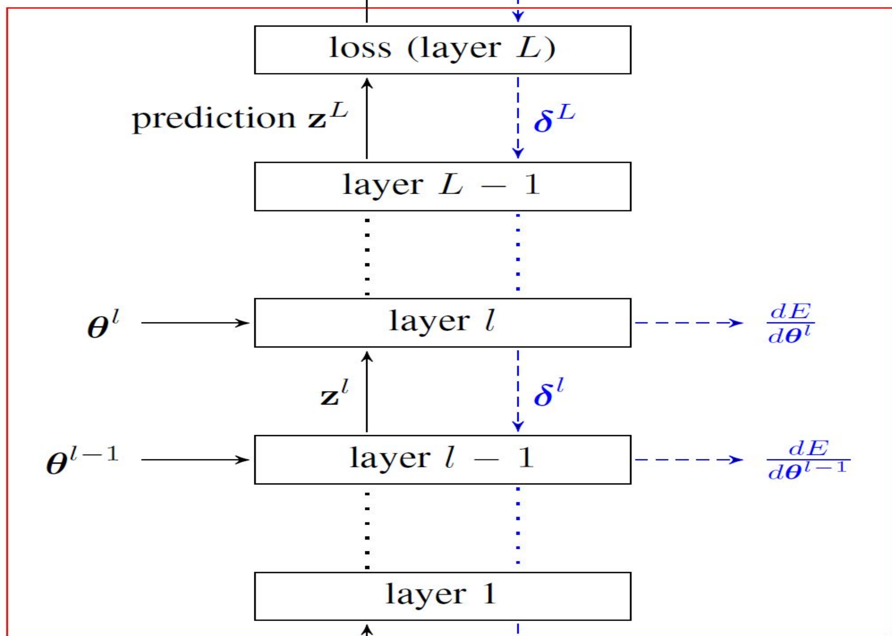
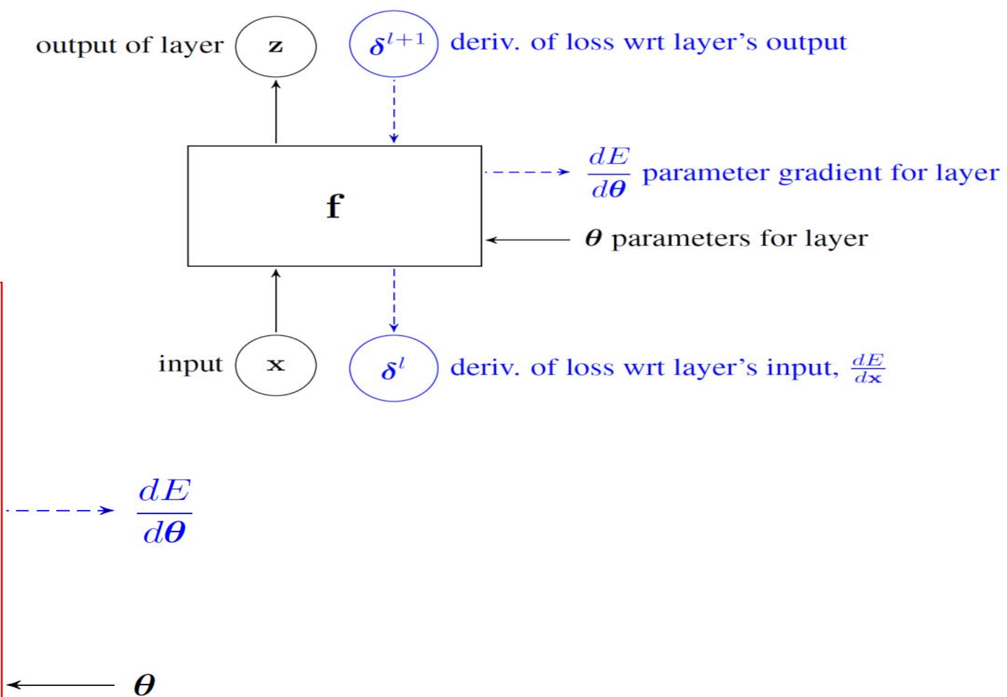


Image convolution layer

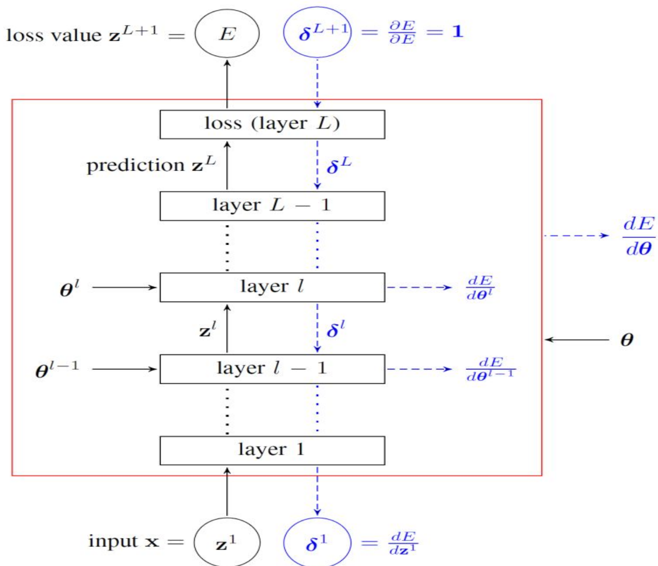


i'
j'
f'
width
height
W_F, H_F, F — # filters
z
i
j
f
X


loss value $\mathbf{z}^{L+1} =$
E
$\boldsymbol{\delta}^{L+1}$
$= \frac{\partial E}{\partial E} = \mathbf{1}$
loss (layer L)
prediction $\mathbf{z}^L$
$\boldsymbol{\delta}^L$
layer $L - 1$
$\frac{dE}{d\boldsymbol{\theta}}$
$\boldsymbol{\theta}^l$
layer l
$\frac{dE}{d\boldsymbol{\theta}^l}$
$\mathbf{z}^l$
$\boldsymbol{\delta}^l$
$\boldsymbol{\theta}$
$\boldsymbol{\theta}^{l-1}$
layer $l - 1$
$\frac{dE}{d\boldsymbol{\theta}^{l-1}}$
layer 1
input $\mathbf{x} =$
$\mathbf{z}^1$
$\boldsymbol{\delta}^1$
$= \frac{dE}{d\mathbf{z}^1}$


output of layer
$\mathbf{z}$
$\boldsymbol{\delta}^{l+1}$
deriv. of loss wrt layer's output
$\mathbf{f}$
$\frac{dE}{d\boldsymbol{\theta}}$ parameter gradient for layer
$\boldsymbol{\theta}$ parameters for layer
input
$\mathbf{x}$
$\boldsymbol{\delta}^l$
deriv. of loss wrt layer's input, $\frac{dE}{d\mathbf{x}}$

Modularity - never forget the lego image!



$$\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta}^l)$$

$$\boldsymbol{\delta}^l := \frac{\partial E}{\partial \mathbf{z}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial \mathbf{z}^l}$$

$$\delta_i^l = \sum_j \frac{\partial E}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \frac{\partial f_j^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial z_i^l}$$

$$\frac{\partial E}{\partial \boldsymbol{\theta}^l} = \frac{\partial E}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \boldsymbol{\theta}^l} = \boldsymbol{\delta}^{l+1} \frac{\partial \mathbf{f}^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial \boldsymbol{\theta}^l}$$

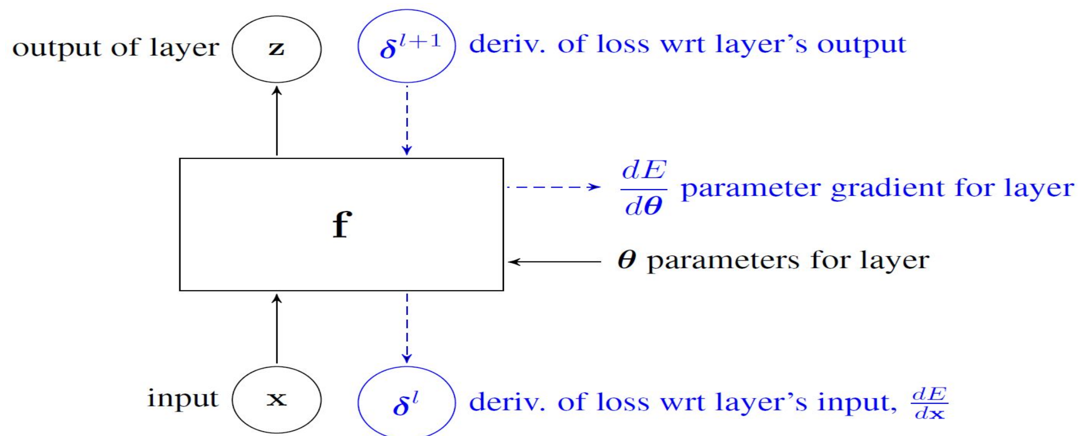
$$\frac{\partial E}{\partial \theta_i^l} = \sum_j \frac{\partial E}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial \theta_i^l} = \sum_j \delta_j^{l+1} \frac{\partial f_j^l(\mathbf{z}^l; \boldsymbol{\theta}^l)}{\partial \theta_i^l}$$

Linear layer

$$z_j = f_j(\mathbf{x}; \boldsymbol{\theta}_j) = \sum_i x_i \theta_{ij}$$

$$\delta_i^l = \sum_j \delta_j^{l+1} \frac{\partial f_j(\mathbf{x}; \boldsymbol{\theta}_j)}{\partial x_i} = \sum_j \delta_j^{l+1} \theta_{ij}$$

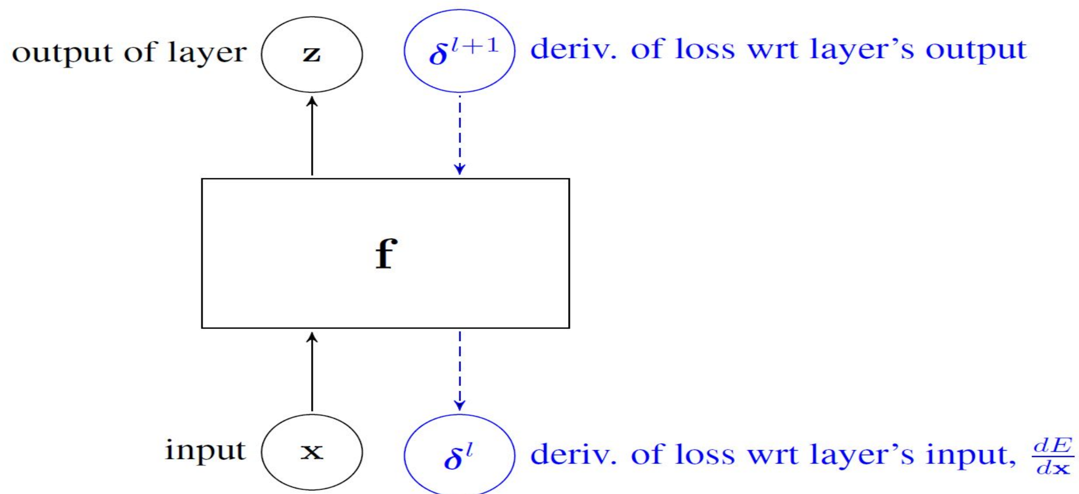
$$\frac{\partial E}{\partial \theta_{ij}} = \sum_j \delta_j^{l+1} \frac{\partial f_j(\mathbf{x}; \boldsymbol{\theta}_j)}{\partial \theta_{ij}} = \delta_j^{l+1} x_i$$



ReLU layer

$$z_j = f_j(x_j) = \max(0, x_j)$$

$$\delta_i^l = \sum_j \delta_j^{l+1} \frac{\partial f_j(x_j)}{\partial x_i} = \delta_i^{l+1} \mathbb{I}_{[x_i > 0]}$$



Conv layer

$$\mathbf{y}_{i',j',f'} = b_{f'} + \sum_{i=1}^{H_f} \sum_{j=1}^{W_f} \sum_{f=1}^{F} \mathbf{x}_{i'+i-1,j'+j-1,f} \boldsymbol{\theta}_{ijff'}$$

$$\begin{aligned}
\frac{\partial E}{\partial \theta_{ijff'}} &= \sum_{i'j'f'} \delta^{l+1}_{i'j'f'} \frac{\partial f_{i'j'f'}(\mathbf{x}; \boldsymbol{\theta}_{f'})}{\partial \theta_{ijff'}} \\
&= \sum_{i'j'} \delta^{l+1}_{i'j'f'} \mathbf{x}_{i'+i-1,j'+j-1,f}
\end{aligned}$$

Conv layer

$$\mathbf{y}_{i',j',f'} = b_{f'} + \sum_{i''=1}^{H_f} \sum_{j''=1}^{W_f} \sum_{f''=1}^{F} \mathbf{x}_{i'+i''-1,j'+j''-1,f''} \boldsymbol{\theta}_{i''j''f''f'}$$

$$i = i' + i'' - 1$$

$$i'' = i - i' + 1$$

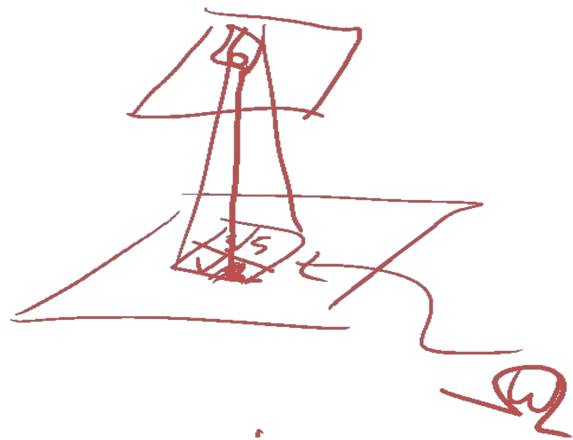
$$\delta^l_{ijf} = \sum_{i'j'f'} \delta^{l+1}_{i'j'f'} \frac{\partial f_{i'j'f'}(\mathbf{x}; \boldsymbol{\theta}_{f'})}{\partial \mathbf{x}_{ijf}}$$

$$= \sum_{i'j'f'} \delta^{l+1}_{i'j'f'} \theta_{i-i'+1,j-j'+1,f,f'}$$

Pooling layer

$$\mathbf{y}_{i',j'} = \max_{ij \in \Omega(i'j')} \mathbf{x}_{ij}$$

$$\begin{aligned}\delta^l_{ij} &= \sum_{i'j'} \delta^{l+1}_{i'j'} \frac{\partial f_{i'j'}(\mathbf{x})}{\partial \mathbf{x}_{ij}} \\ &= \delta^{l+1}_{ij} \mathbb{I}_{ij=\arg\max_{i''j'' \in \Omega(i'j')} \mathbf{x}_{i''j''}}\end{aligned}$$



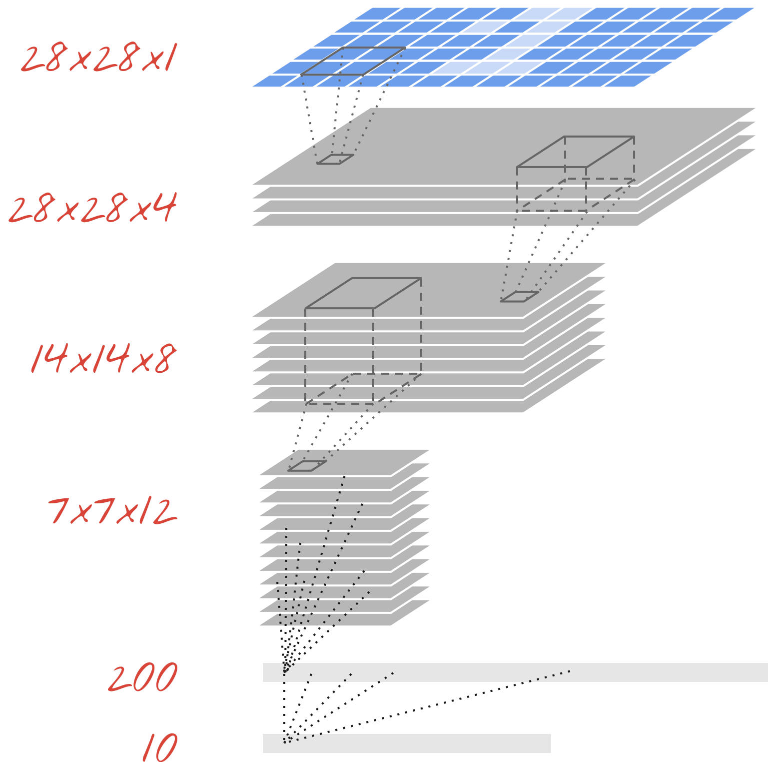
Convolutional networks

stacking the layers together

Hacker's tip

Convolutional neural network

+ biases on
all layers



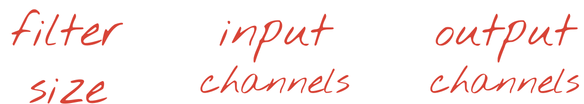
convolutional layer, 4 channels
W1[5, 5, 1, 4] stride 1

convolutional layer, 8 channels
W2[4, 4, 4, 8] stride 2

convolutional layer, 12 channels
W3[4, 4, 8, 12] stride 2

fully connected layer W4[7x7x12, 200]
softmax readout layer W5[200, 10]

Tensorflow - initialisation




```
K=4
L=8
M=12

W1 = tf.Variable(tf.truncated_normal([5, 5, 1, K] ,stddev=0.1))
B1 = tf.Variable(tf.ones([K])/10)
W2 = tf.Variable(tf.truncated_normal([5, 5, K, L] ,stddev=0.1))
B2 = tf.Variable(tf.ones([L])/10)
W3 = tf.Variable(tf.truncated_normal([4, 4, L, M] ,stddev=0.1))
B3 = tf.Variable(tf.ones([M])/10)

N=200

W4 = tf.Variable(tf.truncated_normal([7*7*M, N] ,stddev=0.1))
B4 = tf.Variable(tf.ones([N])/10)
W5 = tf.Variable(tf.truncated_normal([N, 10] ,stddev=0.1))
B5 = tf.Variable(tf.zeros([10])/10)
```

weights initialised with random values

Tensorflow - the model

input image batch
X[100, 28, 28, 1]

weights

stride

biases

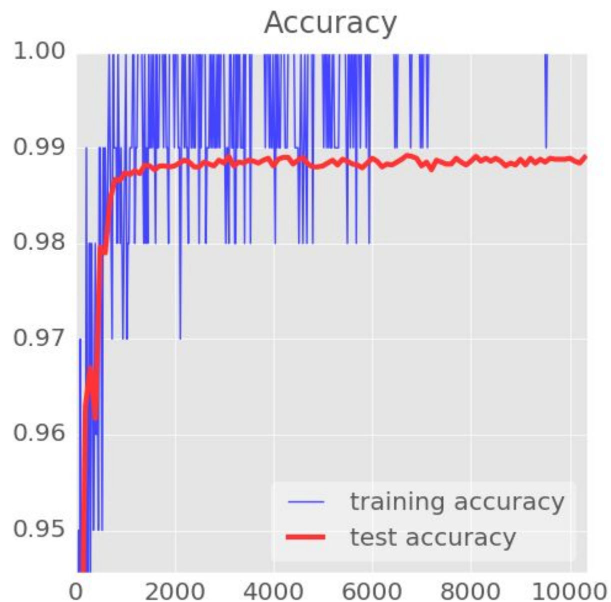
```
Y1 = tf.nn.relu(tf.nn.conv2d(X, W1, strides=[1, 1, 1, 1], padding='SAME') + B1)
Y2 = tf.nn.relu(tf.nn.conv2d(Y1, W2, strides=[1, 2, 2, 1], padding='SAME') + B2)
Y3 = tf.nn.relu(tf.nn.conv2d(Y2, W3, strides=[1, 2, 2, 1], padding='SAME') + B3)

YY = tf.reshape(Y3, shape=[-1, 7 * 7 * M])

Y4 = tf.nn.relu(tf.matmul(YY, W4) + B4)
Y  = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

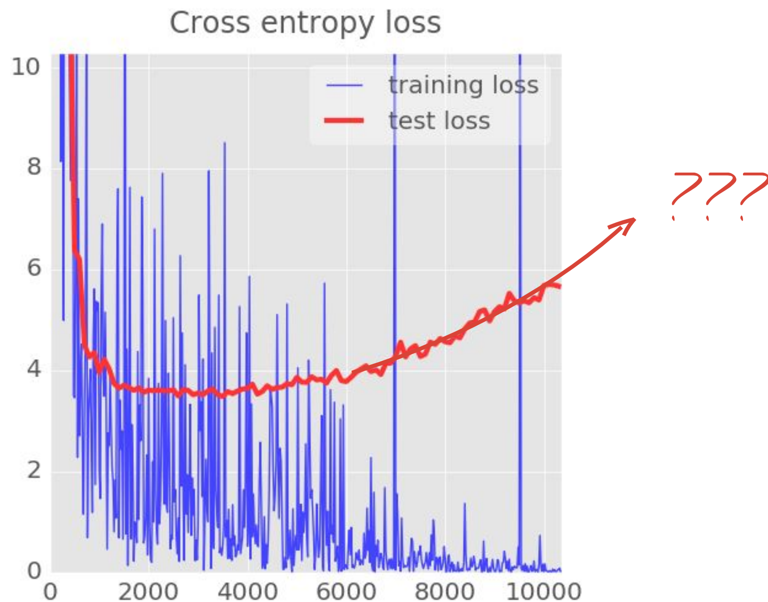
flatten all values for
fully connected layer

Y3 [100, 7, 7, 12]

YY [100, 7x7x12]

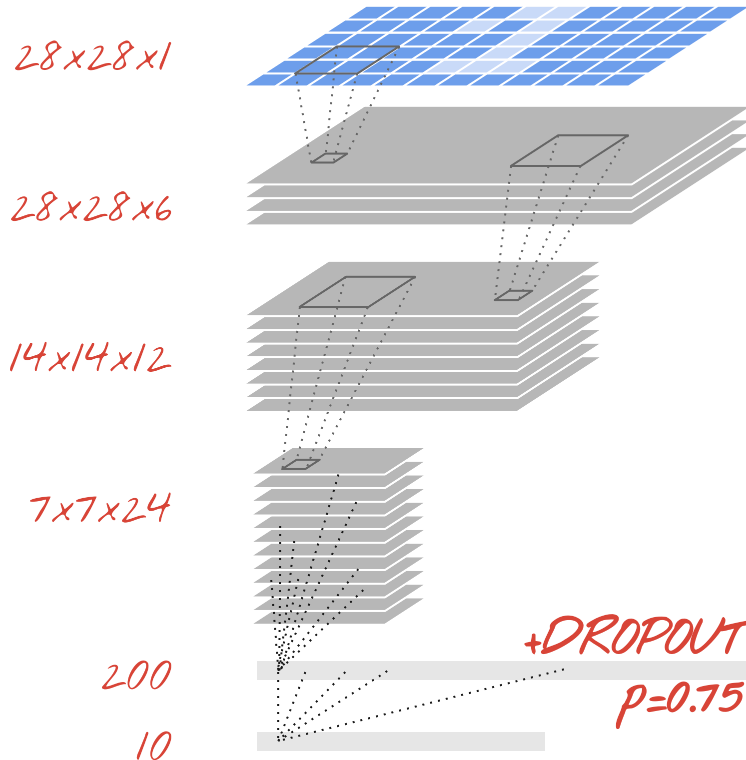
WTFH ???





Bigger convolutional network + dropout

+ biases on all layers



convolutional layer, **6 channels**

W1[**6**, **6**, 1, 6] stride 1

convolutional layer, **12 channels**

W2[**5**, **5**, 6, 12] stride 2

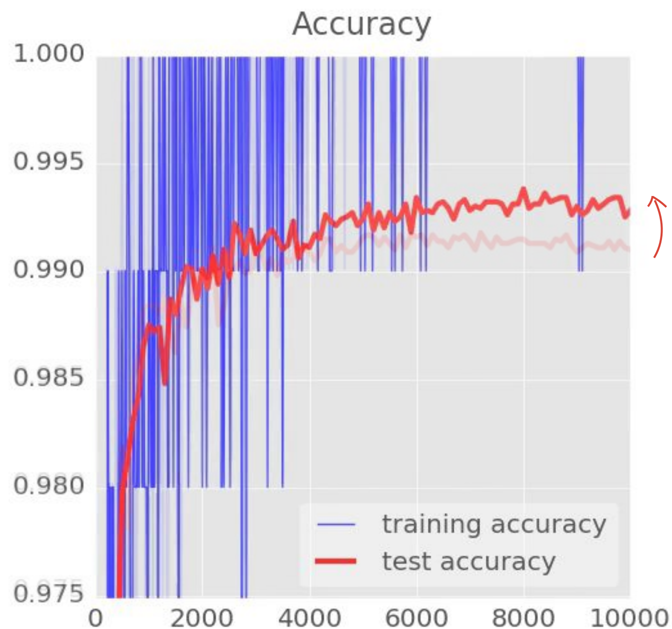
convolutional layer, **24 channels**

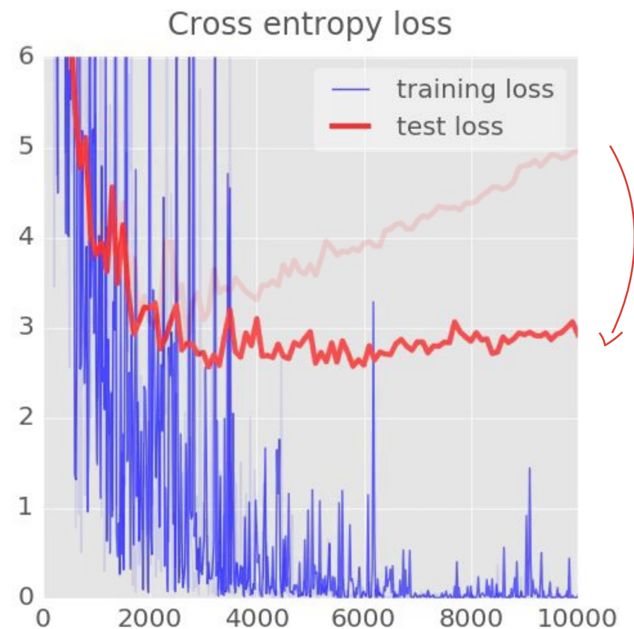
W3[**4**, **4**, 12, 24] stride 2

fully connected layer W4[7x7x24, 200]

softmax readout layer W5[200, 10]

YEAH !





with dropout

Convolutional Networks

for image classification

- Now we are ready to build an image classification network using conv layers

- Activation function: RELU(x) = max(x, 0)

- Typical structure for image classification
  - image → [[conv →] * M → pool] * N → [linear] * K → softmax

Case study 1: MNIST classification

LeNet-5 [LeCun et al., 1998]

INPUT 32x32

C1: feature maps 6@28x28

S2: f. maps 6@14x14

C3: f. maps 16@10x10

S4: f. maps 16@5x5

C5: layer 120

F6: layer 84

OUTPUT 10

Convolutions

Subsampling

Convolutions

Subsampling

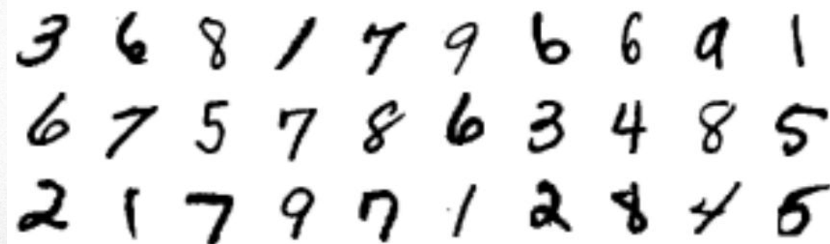
Full connection

Full connection

Gaussian connections

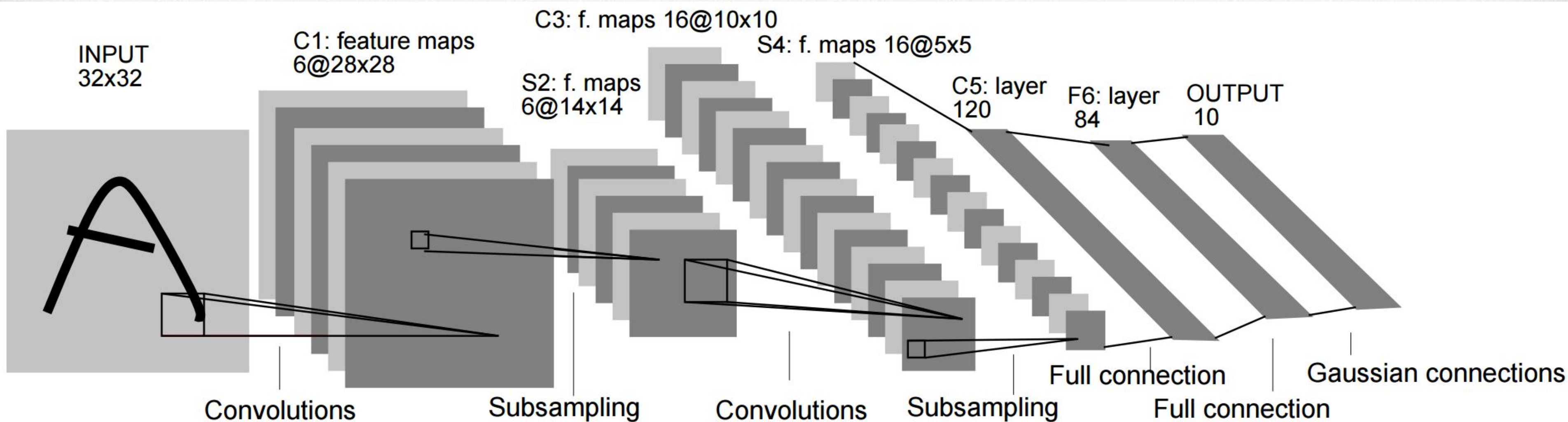
Task

- hand-written digit classification
- 10 classes



Case study 1: MNIST classification

LeNet-5 [LeCun et al., 1998]



Layer configuration:

- 5×5 conv, stride=1, ‘VALID’ padding, sigmoid activation
- 2×2 average pool, stride=2

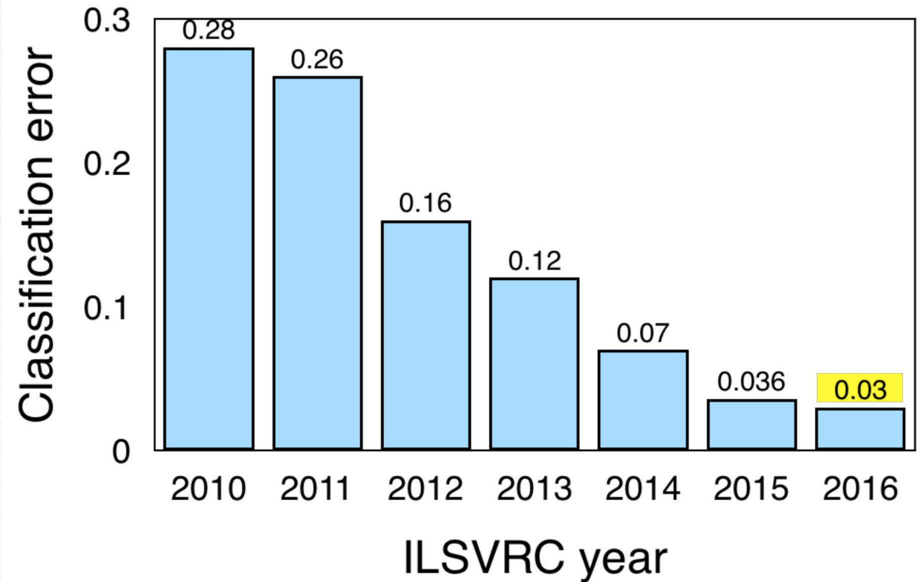
ImageNet Challenge

- Large-scale image recognition challenge
  - Major computer vision benchmark
  - Running since 2010 (Stanford, UNC)
  - http://www.image-net.org/challenges/LSVRC/
- 1.4M images, 1000 classes
- Main tasks
  - classify an image into 1 of the classes
    - top-1 error
      - predicted class should be correct
    - top-5 error
      - predict 5 classes, the correct one should be among them
  - detect all objects in an image

ImageNet Challenge

Overview of the classification task

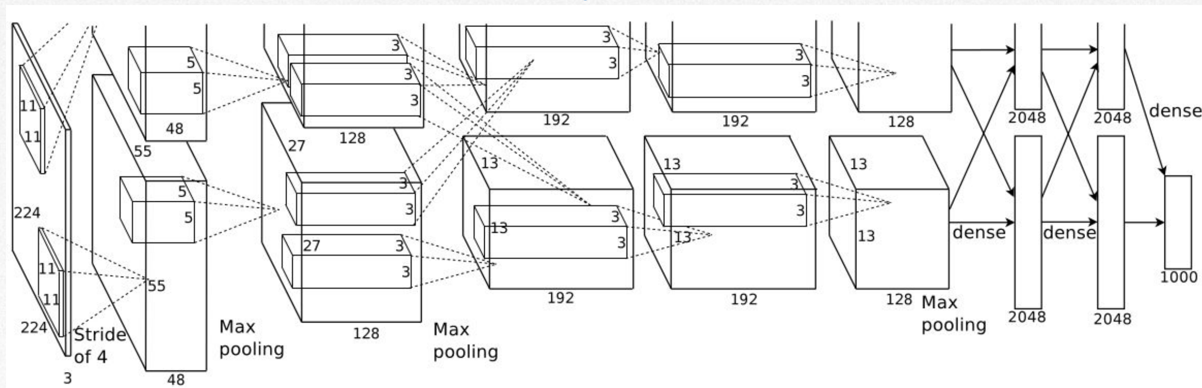
- 2010-11: hand-crafted computer vision pipelines
- 2012-2016: ConvNets
  - 2012: AlexNet
    - major deep learning success
  - 2013: ZFNet
    - improvements over AlexNet
  - 2014
    - VGGNet: deeper, simpler
    - InceptionNet: deeper, faster
  - 2015
    - ResNet: even deeper
  - 2016
    - ensembled networks, results have saturated



Case study 2: AlexNet

Krizhevsky et al., 2012

224x224x3 RGB input



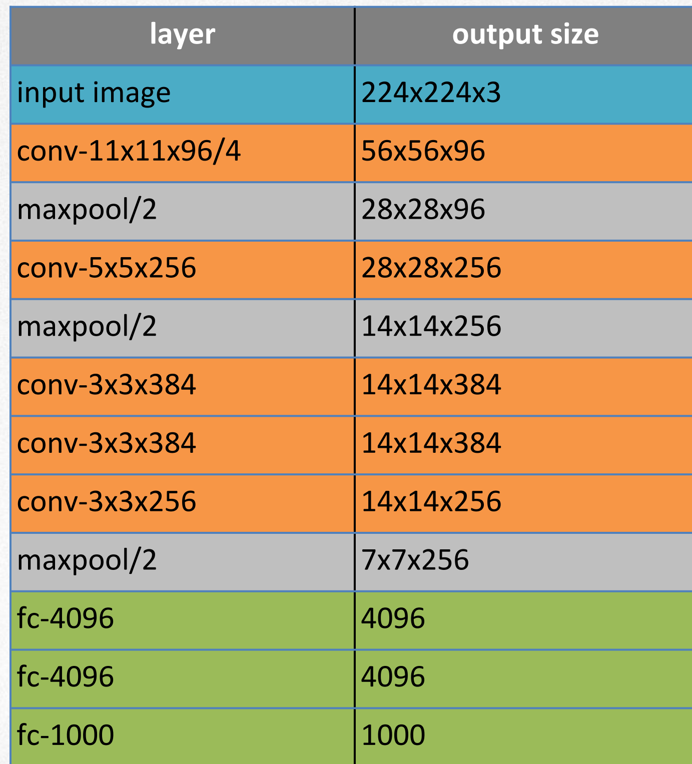
1000 class likelihoods

- 8-layer ConvNet: 5 conv layers, 3 fc layers
- Ingredients for success
  - Architecture
    - ReLU non-linearities
    - regularisation: dropout, weight decay (L_2 penalty)
  - Infrastructure
    - large dataset with random augmentation
    - two GPUs (model split across GPUs), 6 days of training

two important components of successful deep learning models: **architecture** and **infrastructure**

Case study 2: AlexNet

Krizhevsky et al., 2012



layer	output size
input image	224x224x3
conv-11x11x96/4	56x56x96
maxpool/2	28x28x96
conv-5x5x256	28x28x256
maxpool/2	14x14x256
conv-3x3x384	14x14x384
conv-3x3x384	14x14x384
conv-3x3x256	14x14x256
maxpool/2	7x7x256
fc-4096	4096
fc-4096	4096
fc-1000	1000

With depth: higher-level representations, more spatial invariance

- spatial resolution is reduced
- #channels is increased

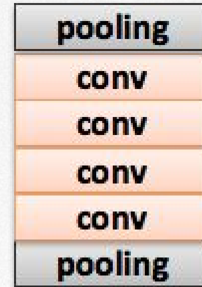
Linear layers at the bottom of AlexNet contain a lot of parameters

Deeper is Better

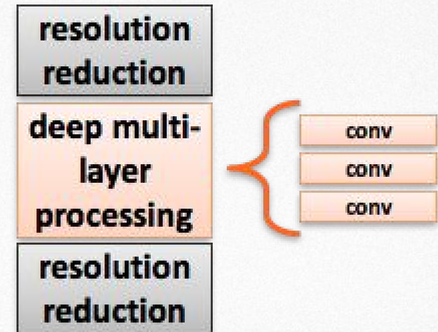
- Each weight layer performs a linear operation, followed by non-linearity
    - layer can be seen as a linear classifier itself
- More layers – more non-linearities
    - leads to a more discriminative (more powerful) model
- What limits the number of layers in ConvNets?
    - early ConvNet models used pooling after each conv. layer
        - input image resolution sets the limit: log(N) for N×N input
    - computational complexity

Building Very Deep ConvNets

- Stack several conv. layers between pooling
  - #conv. layers >> #pooling layers
  - #conv. layers will not affect resolution if each layer preserves spatial resolution
  - stride = 1 & input padding (‘SAME’ convolution)

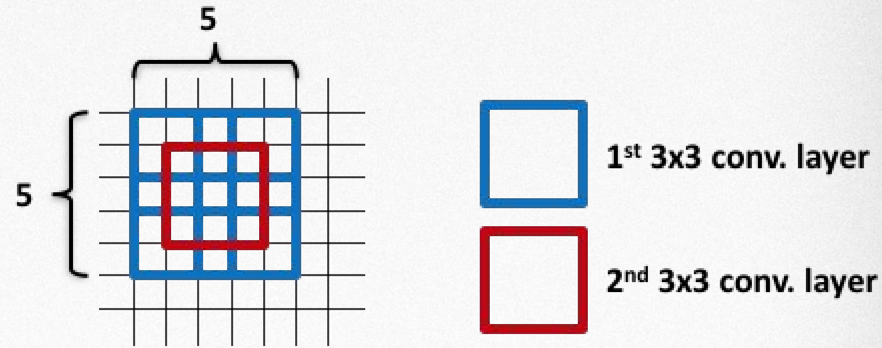


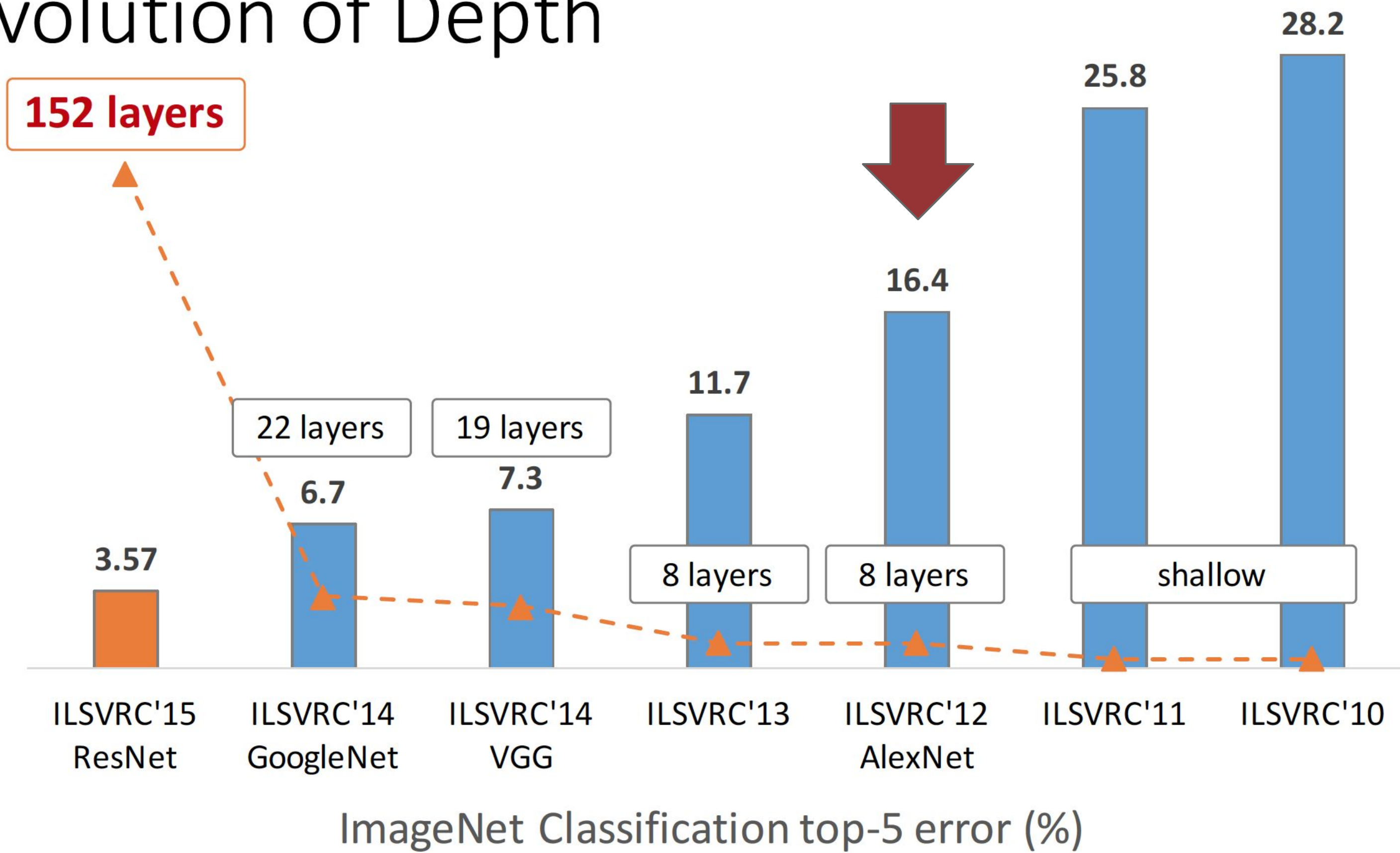
- More generally, interleave deep multi-layer blocks with resolution reduction layers
  - strided conv instead of pooling



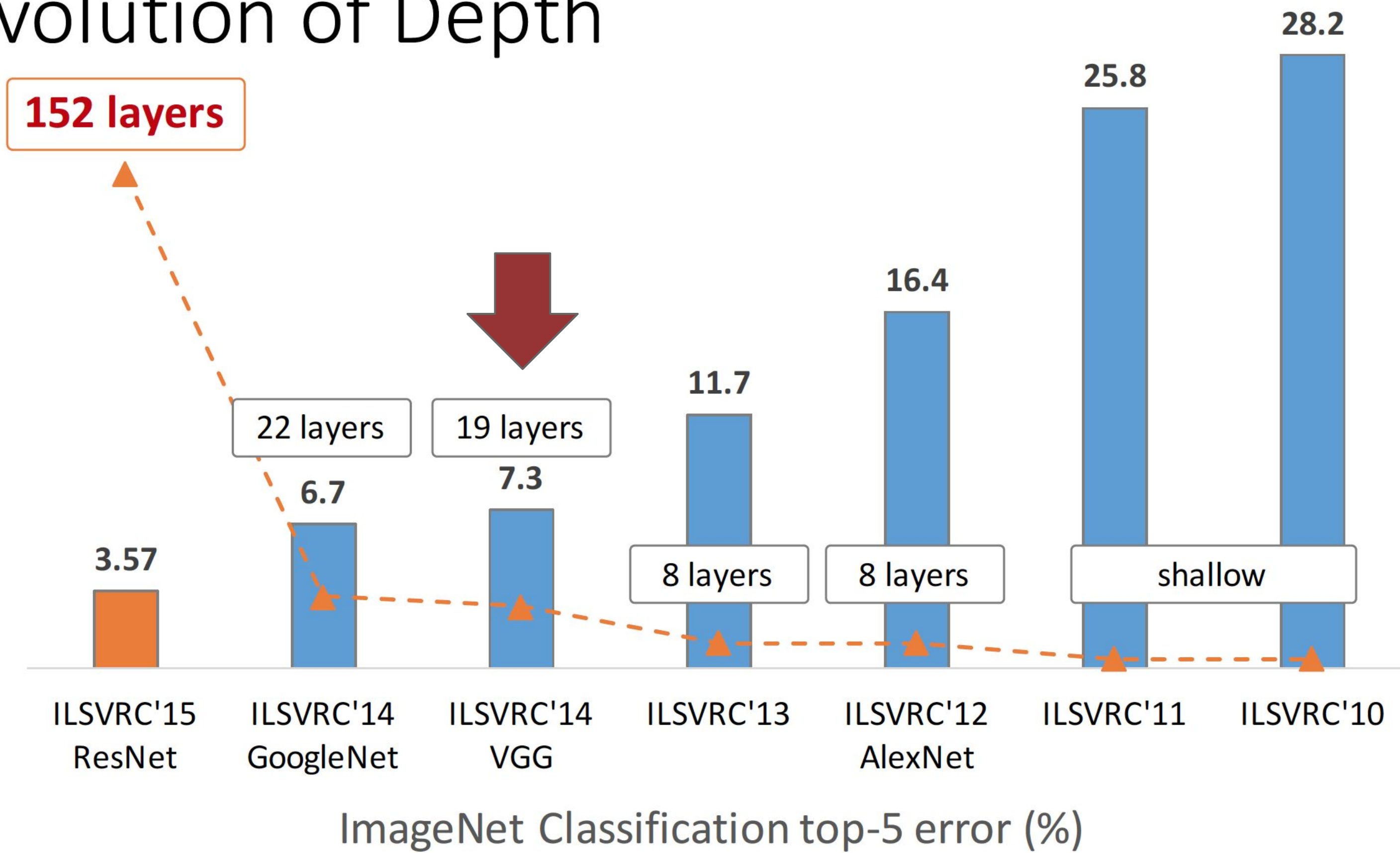
Building Very Deep ConvNets

- Use stacks of small (3×3) conv. layers
  - in most cases, the only kernel size you need
  - a cheap way of building a deep ConvNet
- Stacks have a large receptive field
  - two 3×3 layers – 5×5 field
  - three 3×3 layers – 7×7 field
- Less parameters than a single layer with a large kernel




Revolution of Depth
152 layers
22 layers
19 layers
8 layers
8 layers
shallow
3.57
6.7
7.3
11.7
16.4
25.8
28.2
ILSVRC'15
ResNet
ILSVRC'14
GoogleNet
ILSVRC'14
VGG
ILSVRC'13
ILSVRC'12
AlexNet
ILSVRC'11
ILSVRC'10
ImageNet Classification top-5 error (%)
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Revolution of Depth



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.

Case study 3: VGGNet

Simonyan & Zisserman, 2014

- Straightforward implementation of very deep nets:
  - stacks of conv. layers followed by max-pooling
  - 3x3 conv. kernels, stride=1
  - ReLU non-linearities
  - regularisation: dropout, weight decay (L2 penalty)
- A family of architectures
  - derived by injecting more conv. layers
- Infrastructure
  - trained on 4 GPUs (training data split across GPUs)
  - 2-3 weeks

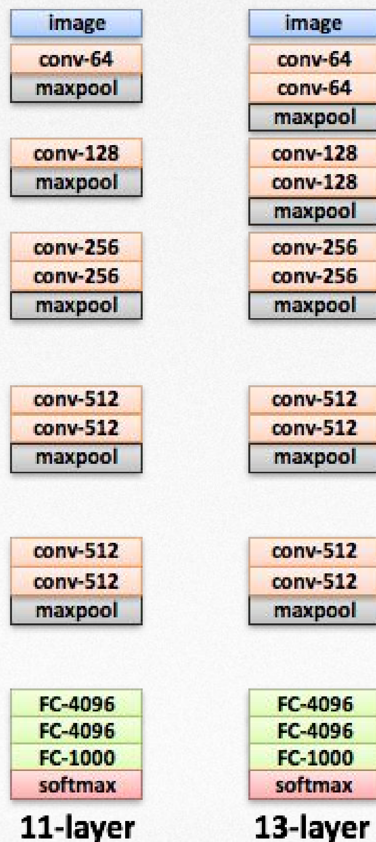
VGGNet Incarnations

- Started from 11 layers

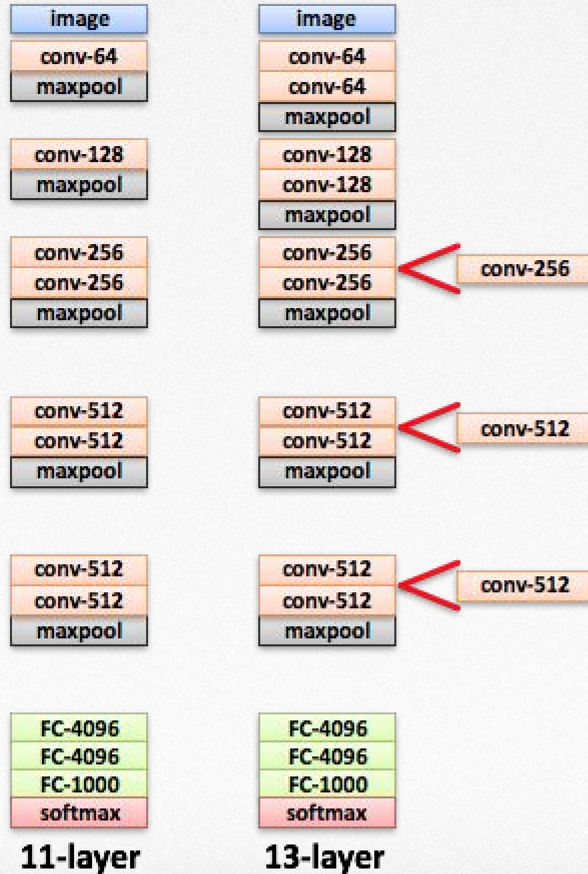
VGGNet Incarnations

- Started from 11 layers & injected more conv. layers

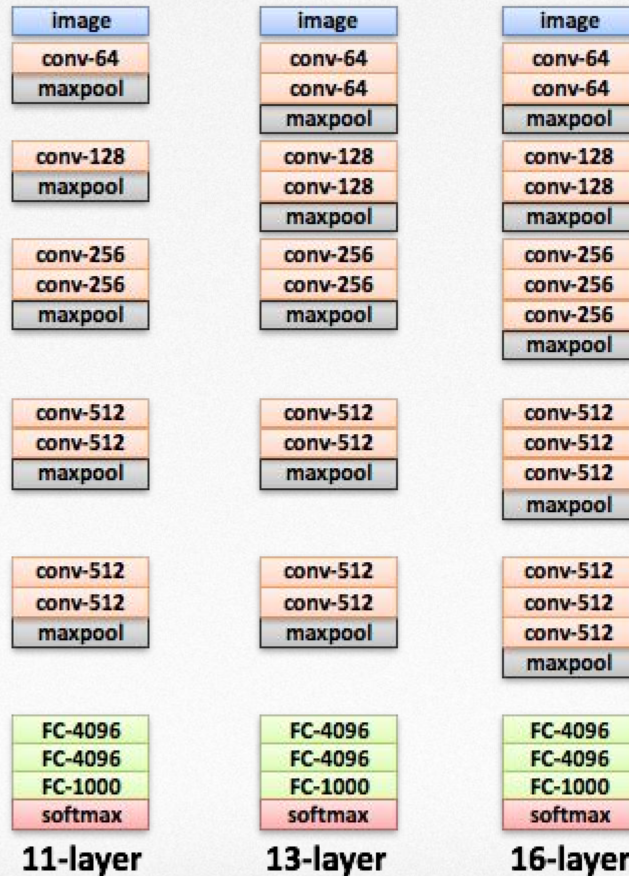
VGGNet Incarnations



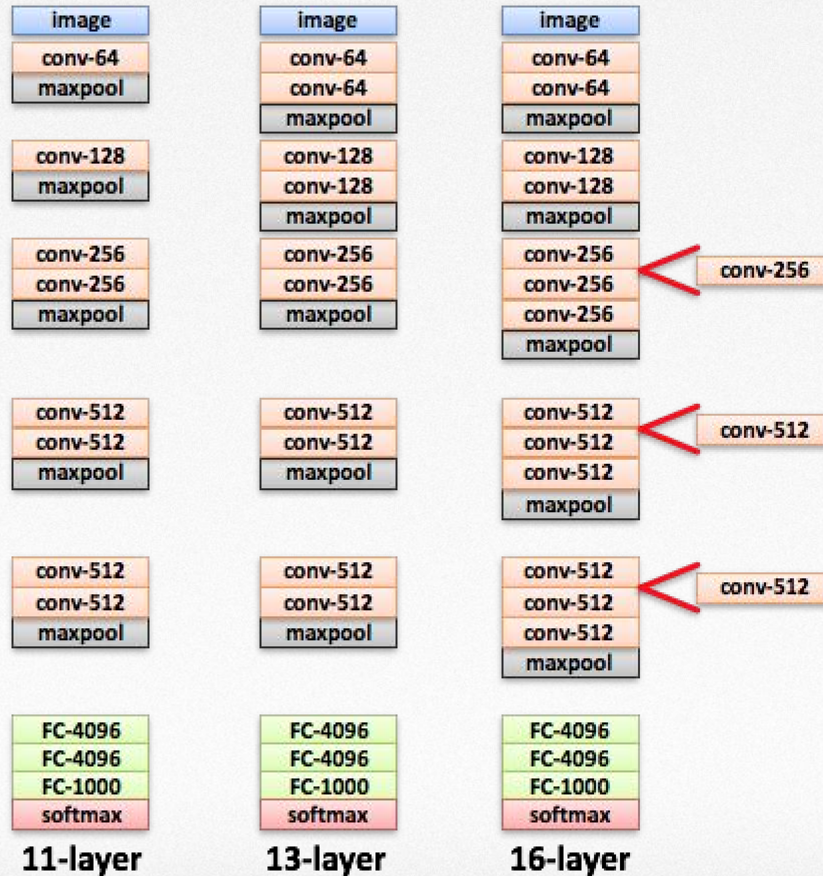
VGGNet Incarnations



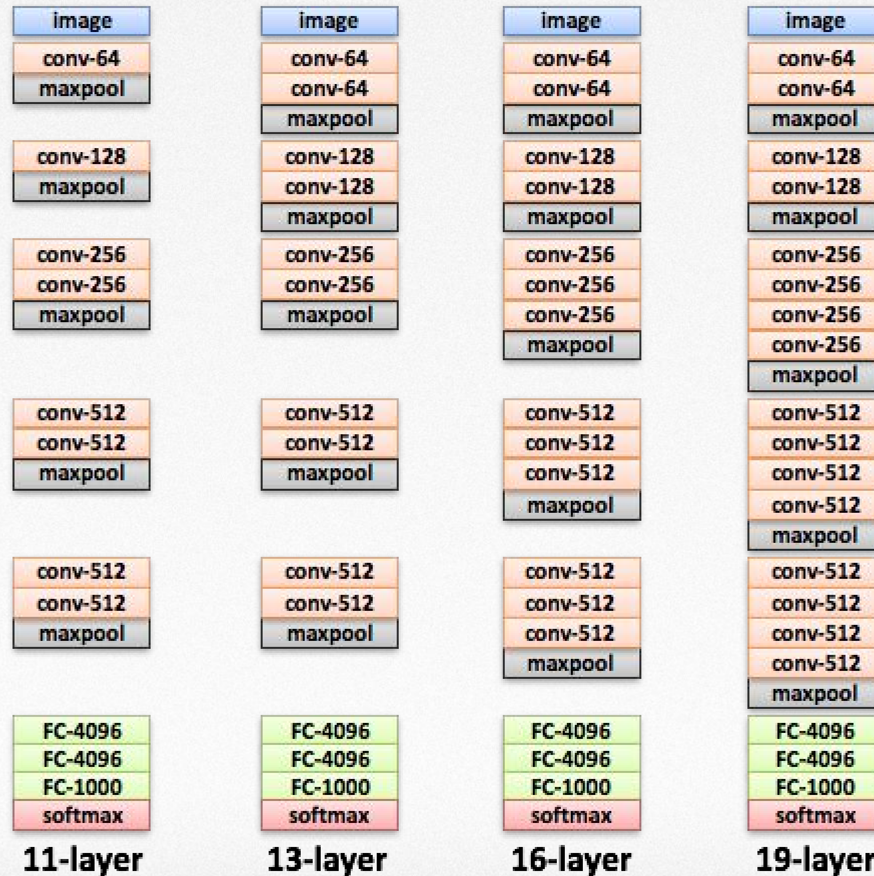
VGGNet Incarnations



VGGNet Incarnations



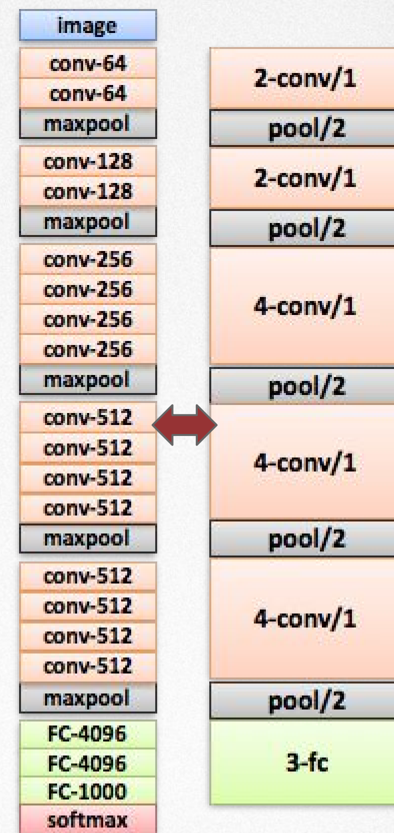
VGGNet Incarnations



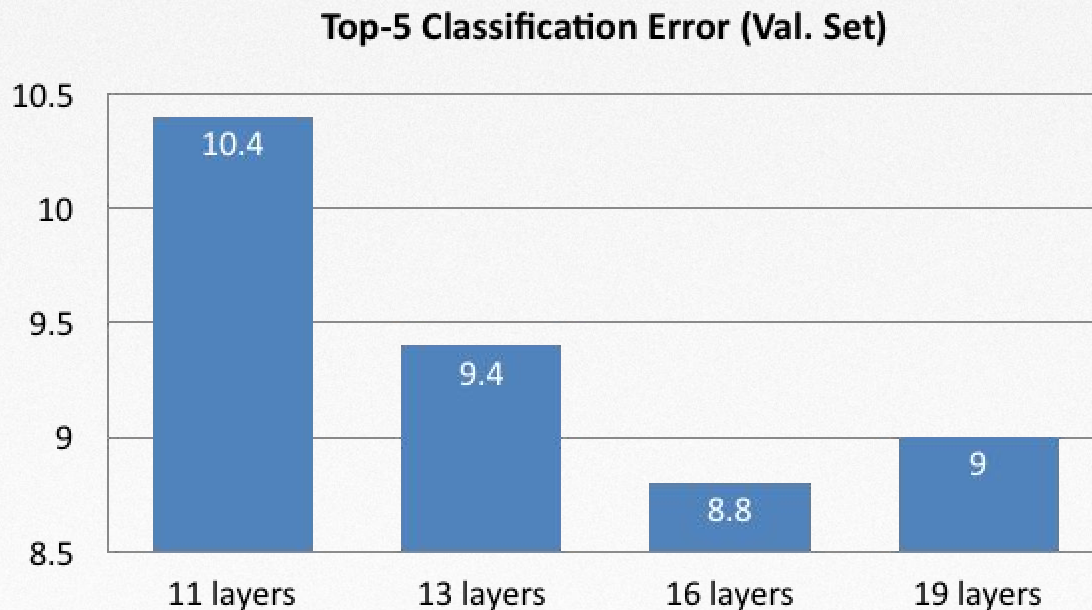
VGGNet Layer Pattern

- Multi-layer stacks (conv. layers, stride=1) interleaved with resolution reduction (max-pooling, stride=2)

- Other very deep nets (discussed later) follow a similar pattern



VGGNet Error vs Depth



- Error reduces with depth
- Plateaus after 16 layers
  - we'll discuss how to fix that

Going Deeper

challenges of training very deep ConvNets and how to solve them

Challenges of training very deep ConvNets

- We have seen that depth is important
- Why not to keep adding layers to VGGNet?

Two main reasons:

- computational complexity
  - ConvNet will be too slow to train and evaluate
- optimisation
  - we won’t be able to train such nets

Optimisation

- Model optimisation is important
  - some architectures are hard to train – in particular very deep nets
- A plethora of gradient-based optimisation methods
  - weight update rules are different: SGD, rms-prop, Adam, etc.
  - SGD with momentum – typical choice for ConvNets
- Major problem: gradient instability
  - when we backprop through many layers, compute a product of weights
  - if the weights are small, the gradients vanish (get too small)
  - if the weights are large, the gradients explode (get too large)

The superpower: batch normalisation

Batch Normalisation

Ioffe and Szegedy, 2015

- Motivation: the distribution of activations changes during training, making it harder
- Batchnorm layer normalises the input to zero mean and unit variance
- Can be placed anywhere in the network
    - typically after each conv layer before activation

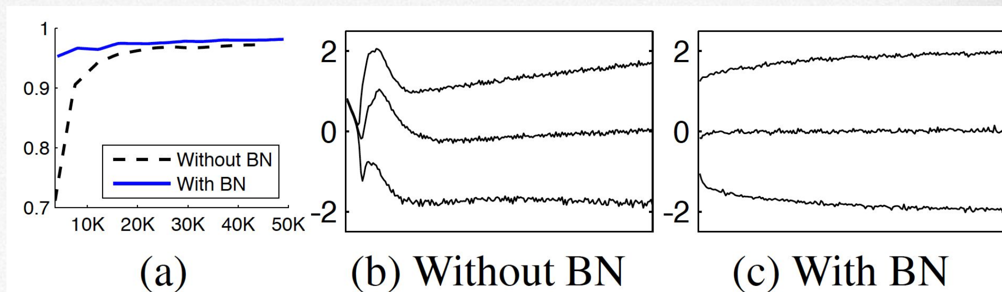


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Batch Normalisation (2)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

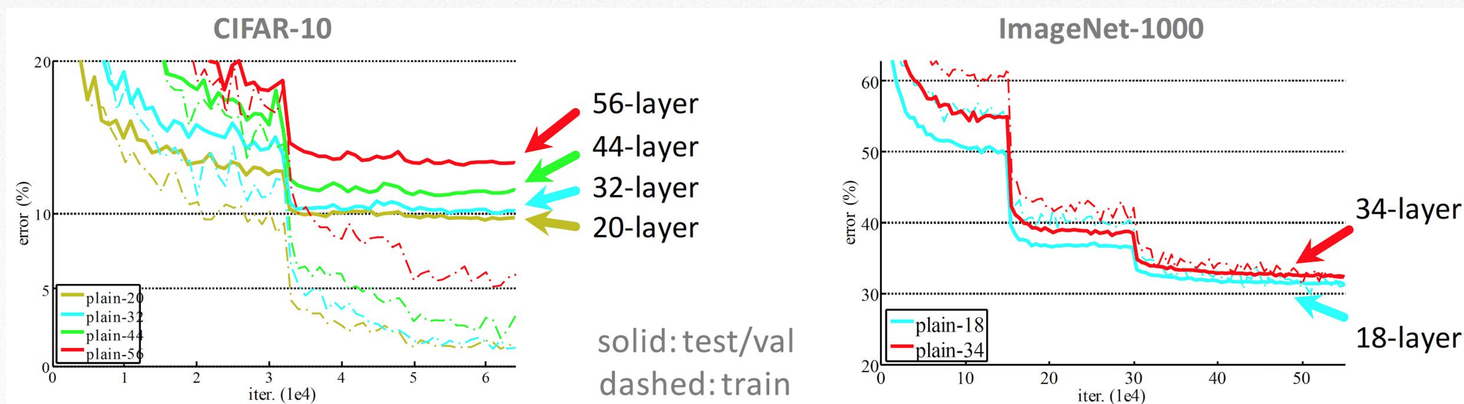
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Requires batched training
- Batchnorm is differentiable
- Means and variances are (slightly) different for different batches
  - adds randomness, which is a good regulariser
  - nets with batchnorm need less regularisation, dropout is rarely needed
- Less sensitive to initialisation, can use N(0, 0.01)

Residual Connections

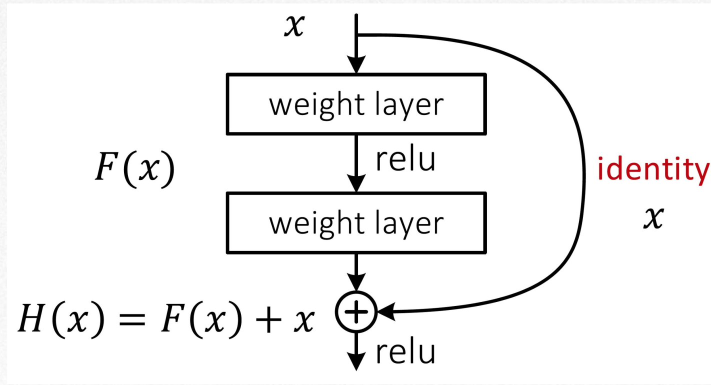
Motivation

- Construction to facilitate training of ultra deep nets (100-1000 layers)
  - complementary to batchnorm
- Motivation: after certain depth, deeper nets have higher **training** error



error curves for VGG-like nets (3×3 conv throughout) with batchnorm

Residual Connections



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}\left(\frac{\partial F}{\partial x} + 1\right)$$

- Identity connection which **skips** a few layers
- We only need to learn the **residual**
- Becomes easier to learn identity, if need to
  - just set the weights to 0
- Backprop perspective
  - gradient skips weight layers – no vanishing
  - improves gradient flow through layers

ResNets

He at al., 2015-16

- A family of models
  - won the classification task of ImageNet-2015
- Simple network design
  - inspired by VGGNet, but **10x deeper**
  - **residual connections** & batchnorm

Revolution of Depth

152 layers
3.57
22 layers
6.7
19 layers
7.3
11.7
8 layers
16.4
8 layers
25.8
28.2
shallow

ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11
ILSVRC'10

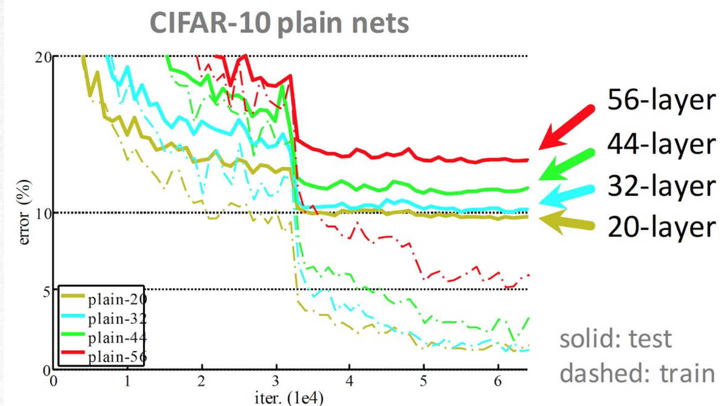
ImageNet Classification top-5 error (%)

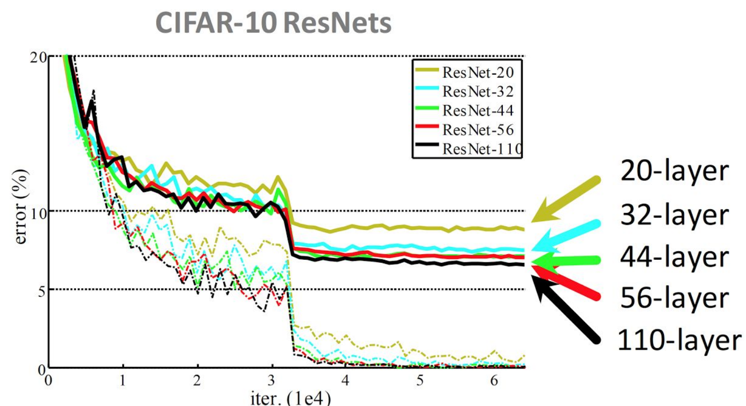
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

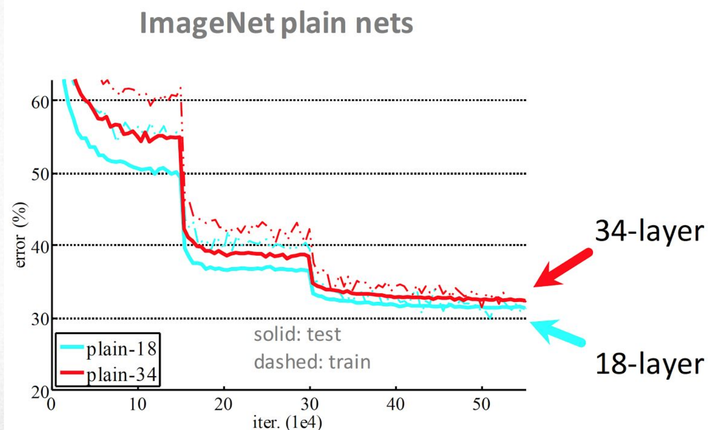
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000

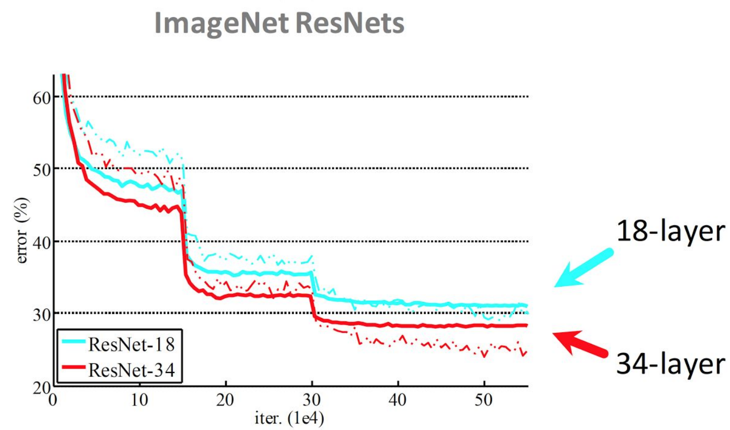
ResNets

Deeper ResNets have lower training and test errors

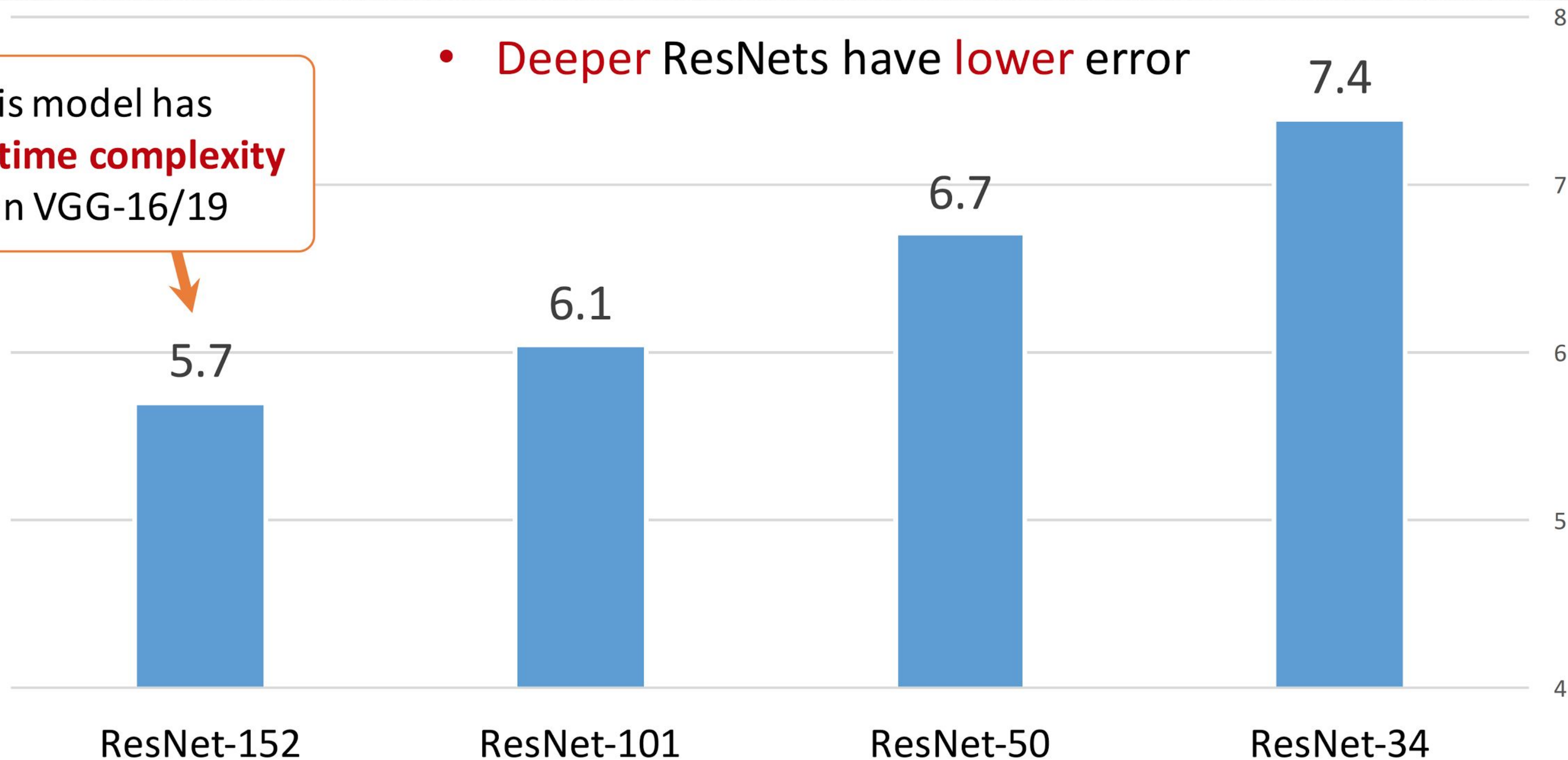








ResNet ImageNet Results

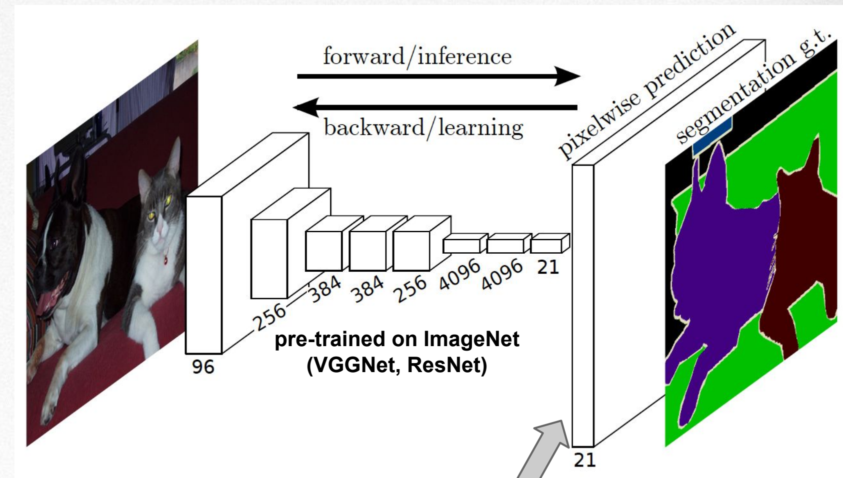


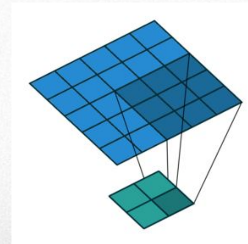
Beyond ImageNet Classification

Fully Convolutional Networks

Shelhamer et al., 2014

- ConvNet w/o linear layers (“fully convolutional”)
  - pre-trained on ImageNet classification
- Penultimate conv layer has 21 channel
  - 20 classes & background
- ConvNet contains pooling layers
  - which reduce resolution
  - compensated by transposed conv in the end

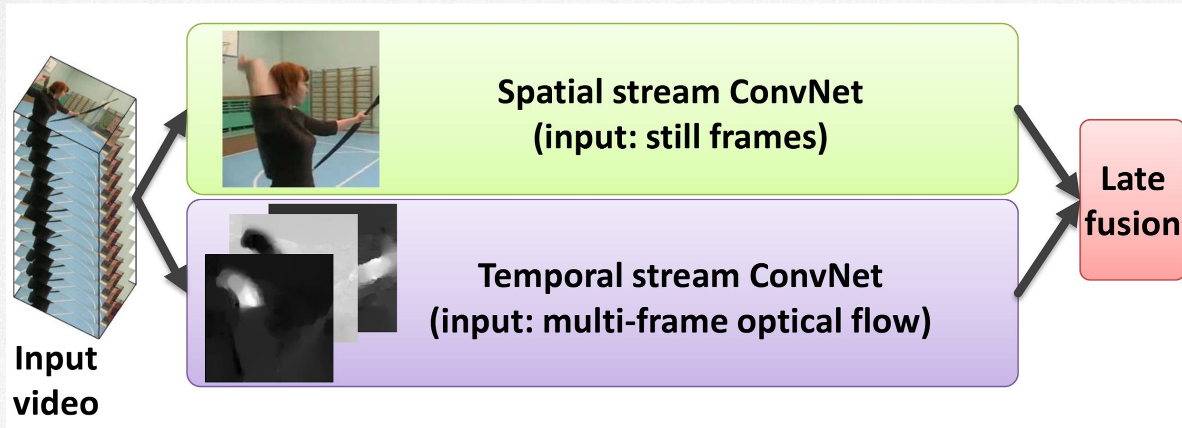


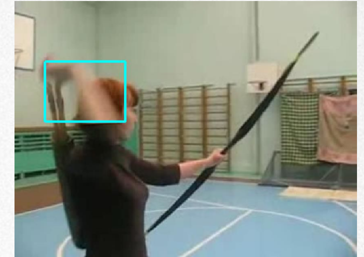


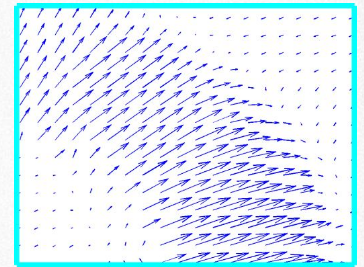
transposed conv increases spatial resolution

Two-Stream ConvNet for Video

Simonyan & Zisserman, 2014







- Appearance and motion are processed separately
- Spatial stream ConvNet
  - input: RGB frame
- Temporal stream ConvNet
  - input: motion vector field between several frames
- Each ConvNet can be pre-trained (again!)

Lipreading (convnets for video)

time

t frames

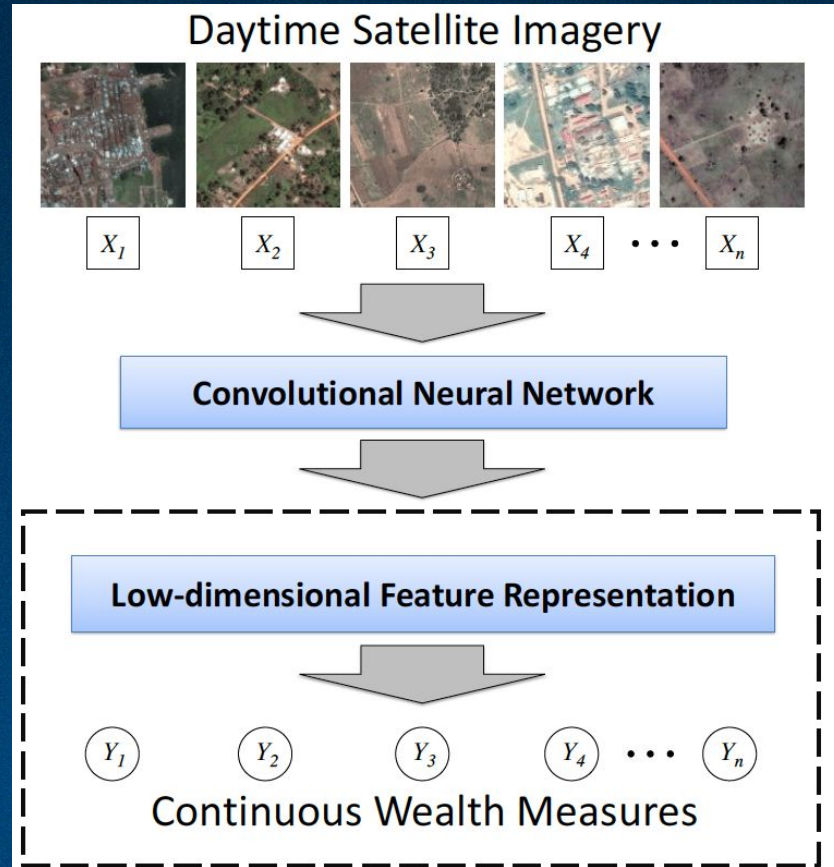
STCNN + Spatial Pooling (x3)

Bi-GRU (x2)

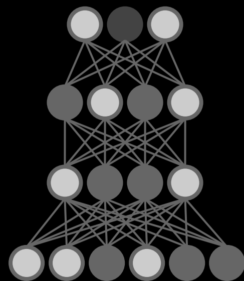
Linear

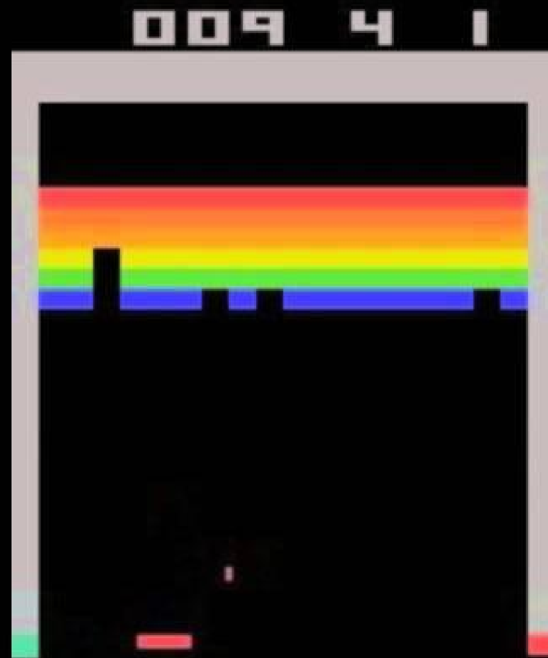
CTC loss

Finding poverty in satellite images (Stanford)

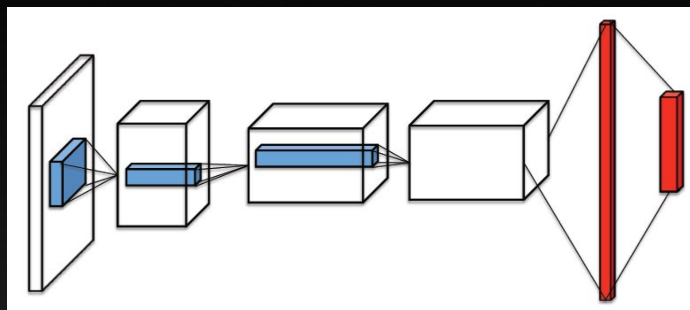


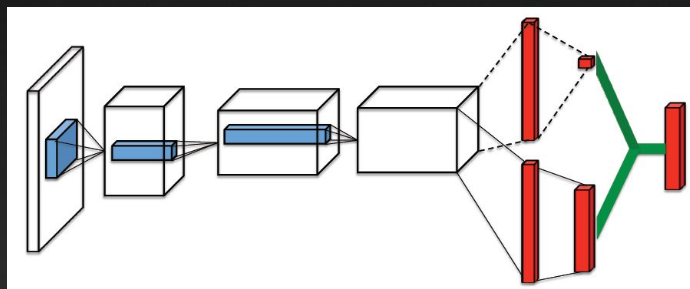
Atari with deep RL

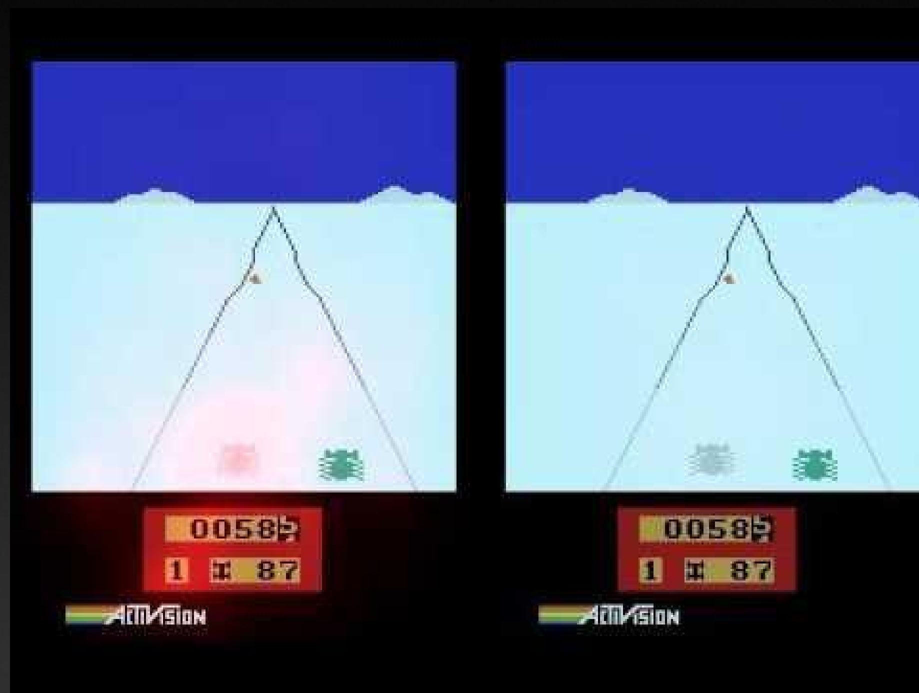




Visualizing what nets attend to







Wang et al (2016)

AlphaGo

What is Go?

One of the four arts to be mastered by a true scholar (Confucius)

40 million players, 2000 pros: Go schools in Japan, China and S. Korea

Simple rules leading to profound complexity

10^170 possible board configurations > no. of atoms in the universe!

Why is it hard for computers to play?

Sheer complexity of the game means that exhaustive search intractable

Branching factor is 200 in Go compared to 20 in Chess

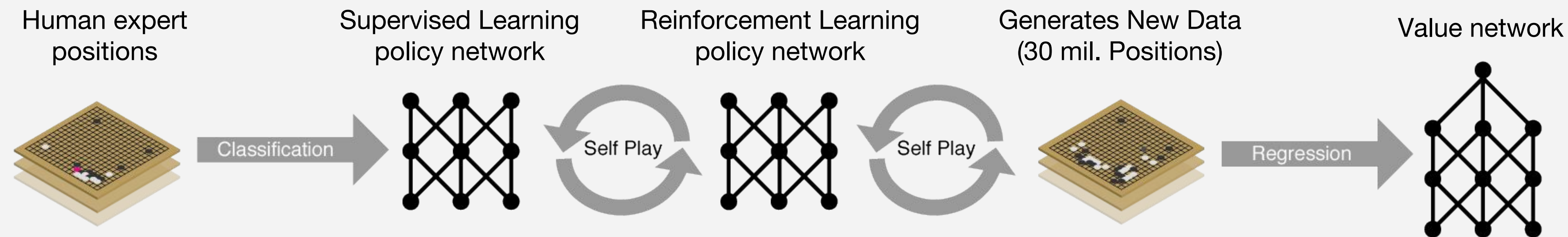
Primarily a game about intuition rather than brute calculation

Writing evaluation fn to determine who is winning, thought impossible

Combines pattern recognition with search and planning

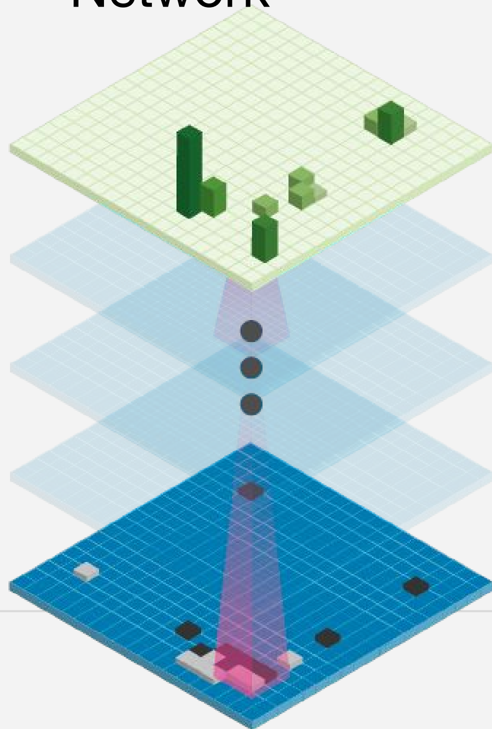
“Beating a professional Go player” a long-standing grand challenge of AI

Training the deep neural networks

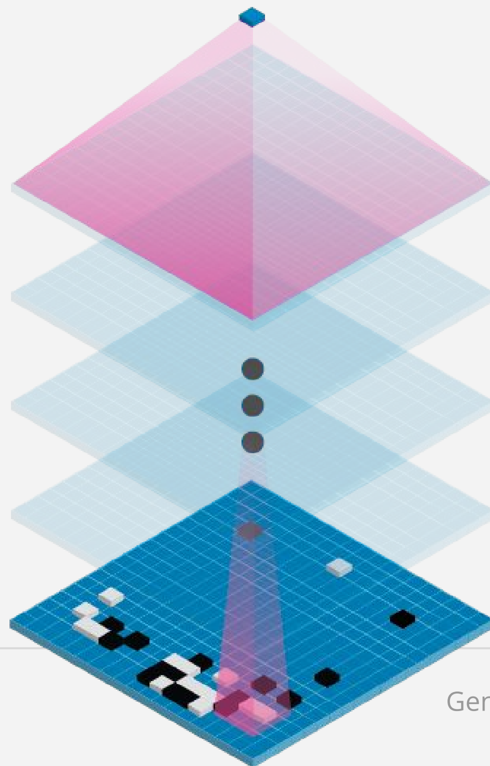


Two networks: Policy and Value Nets

Policy Network



Value Network



Internal Testing	Calibration	External Testing

AlphaGo (May 2017)

Wins 3/3 Matches

Ke Jie (9p)
World number 1

AlphaGo (Mar 2016)

Wins ⅘ Matches

Lee Sedol (9p)
Top player of past decade

AlphaGo (Oct 2015)

Wins 5/5 Matches

Fan Hui (2p)
3-times reigning Euro Champion

Thank you!

DeepMind