Course number: 80240743

Deep Learning

Xiaolin Hu (胡晓林) & Jun Zhu (朱军)
Dept. of Computer Science and Technology
Tsinghua University

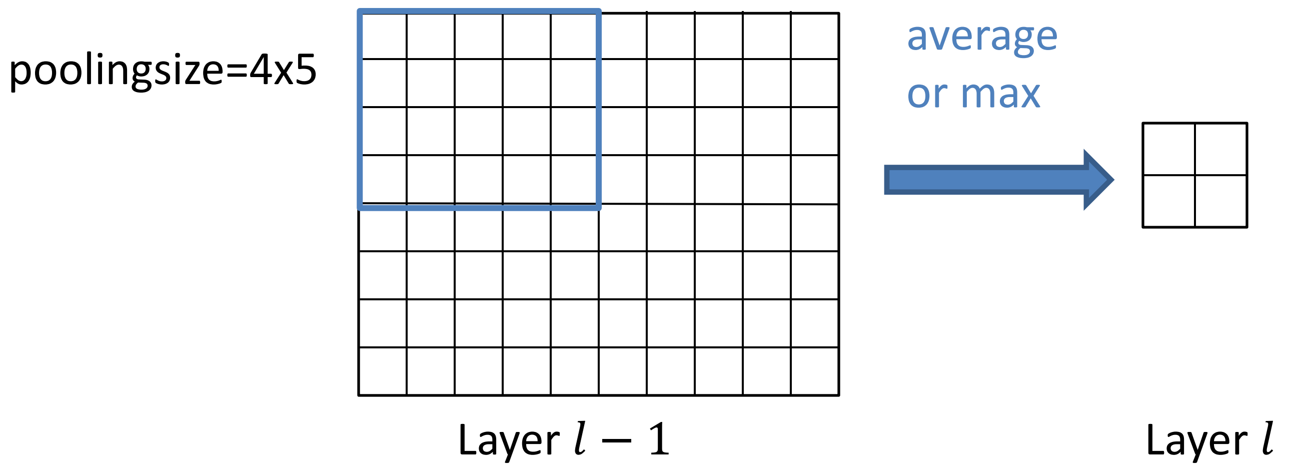
Topic 4: Convolutional Neural Networks-II

Xiaolin Hu
Dept. of Computer Science and Technology
Tsinghua University

Outline

- Pooling
- Standard CNN
- Typical CNNs
- Advanced techniques

Pooling in local regions

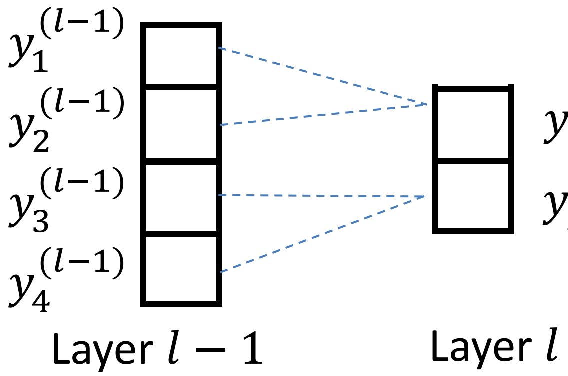


$$\boldsymbol{y}^{(l)} = \frac{1}{poolingsize}\mathrm{downsample}(\boldsymbol{y}^{(l-1)})$$

- Divide the convolved features into *disjoint* $m \times n$ regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled features
- Similar operations on 1D input
- How about 3D input? Channel-wise pooling

Average pooling layer

If layer l is an average pooling layer. Consider one single feature map



$$y_1^{(l)} = (y_1^{(l-1)} + y_2^{(l-1)})/2$$
$$y_2^{(l)} = (y_3^{(l-1)} + y_4^{(l-1)})/2$$

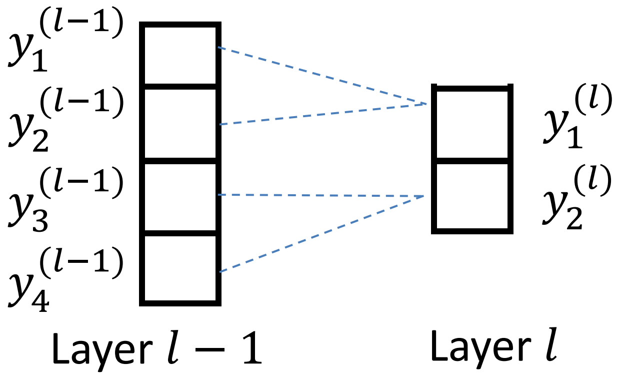
- Local sensitivity in the scalar form

$$\delta_1^{(l-1)} = \frac{\partial E^{(n)}}{\partial y_1^{(l-1)}} = \frac{\partial E^{(n)}}{\partial y_1^{(l)}} \frac{\partial y_1^{(l)}}{\partial y_1^{(l-1)}} = \frac{1}{2}\delta_1^{(l)}$$

$$\delta_2^{(l-1)} = \frac{\partial E^{(n)}}{\partial y_2^{(l-1)}} = \frac{\partial E^{(n)}}{\partial y_1^{(l)}} \frac{\partial y_1^{(l)}}{\partial y_2^{(l-1)}} = \frac{1}{2}\delta_1^{(l)}$$

Similarly we can obtain $\delta_3^{(l-1)} = \frac{1}{2}\delta_2^{(l)}, \quad \delta_4^{(l-1)} = \frac{1}{2}\delta_2^{(l)}$

Average pooling layer



$$\delta_1^{(l-1)} = \delta_1^{(l)}/2$$
$$\delta_2^{(l-1)} = \delta_1^{(l)}/2$$
$$\delta_3^{(l-1)} = \delta_2^{(l)}/2$$
$$\delta_4^{(l-1)} = \delta_2^{(l)}/2$$

- In general, local sensitivity in the vector form

$$\boldsymbol{\delta}^{(l-1)} = \frac{1}{poolingsize}\text{upsample}(\boldsymbol{\delta}^{(l)})$$

$$\text{upsample}(\boldsymbol{a}) \triangleq \begin{pmatrix} \left.\begin{matrix} a_1 \\ a_1 \end{matrix}\right\} \textit{Poolingsize} \\ \hline \vdots \\ \hline \left.\begin{matrix} a_n \\ a_n \end{matrix}\right\} \textit{Poolingsize} \end{pmatrix} \qquad \text{where} \quad \boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

- Can you derive the recursive rule for local sensitivity in the max pooling layer?
- Hint: list different conditions

Outline

- Pooling
- Standard CNN
- Typical CNNs
- Advanced techniques

Construction of CNN

- The convolutional layers and pooling layers can be combined freely with other layers that we have discussed
  - Fully connected layer
  - Sigmoid layer, ReLU layer or other activation layers
  - Euclidean loss layer
  - Cross-entropy loss layer

  as well as other layers that we haven't discussed, e.g.,
  - Local response normalization layer (Krizhevsky et al. 2012)
  - Dropout layer (Srivastava et al., 2014)
  - Batch normalization layer (Ioffe and Szegedy, 2015)

CNN Implementation

- Implement each *type* of layer as a class and provide functions for forward calculation and backward calculation, respectively
- Design different CNN structures by specifying layer modules in a main file
- Run forward process
  - Calculate the output $\boldsymbol{y}^{(l)}$ for $l = 1,2,\ldots,L$
- Run backward process
  - Calculate $\partial E/\partial \boldsymbol{W}^{(l)}$ and $\partial E/\partial \boldsymbol{b}^{(l)}$ if any, and $\boldsymbol{\delta}^{(l)}$ for $l = L, L-1,\ldots,1$
- Update $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ for $l = 1,2,\ldots,L$

Toolboxes

- Theano (outdated)
- Caffe
- Tensorflow
- Torch
- Pytorch

Demo: MNIST classification

https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

MNIST

- 60,000 training images and 10,000 test images
- 28x28 black and white images

Network setting

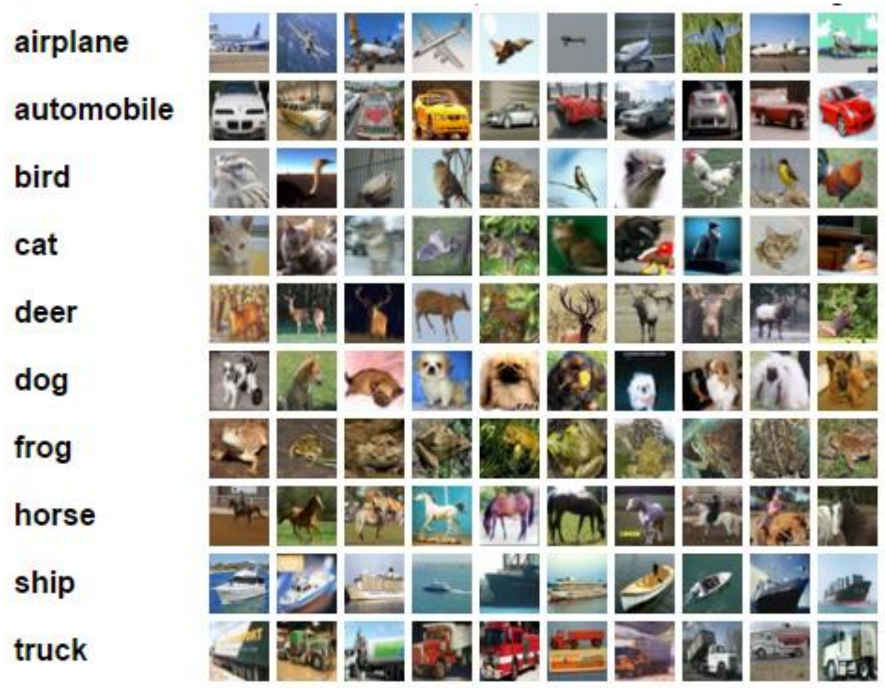
```
layer_defs = [];
layer_defs.push({type:'input', out_sx:24,
out_sy:24, out_depth:1});
layer_defs.push({type:'conv', sx:5, filters:8,
stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:16,
stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:3, stride:3});
layer_defs.push({type:'softmax',
num_classes:10});
```

Demo: CIFAR-10 classification

https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

CIFAR-10 & CIFAR100

- 50,000 training images and 10,000 test images
- 32x32 colour images



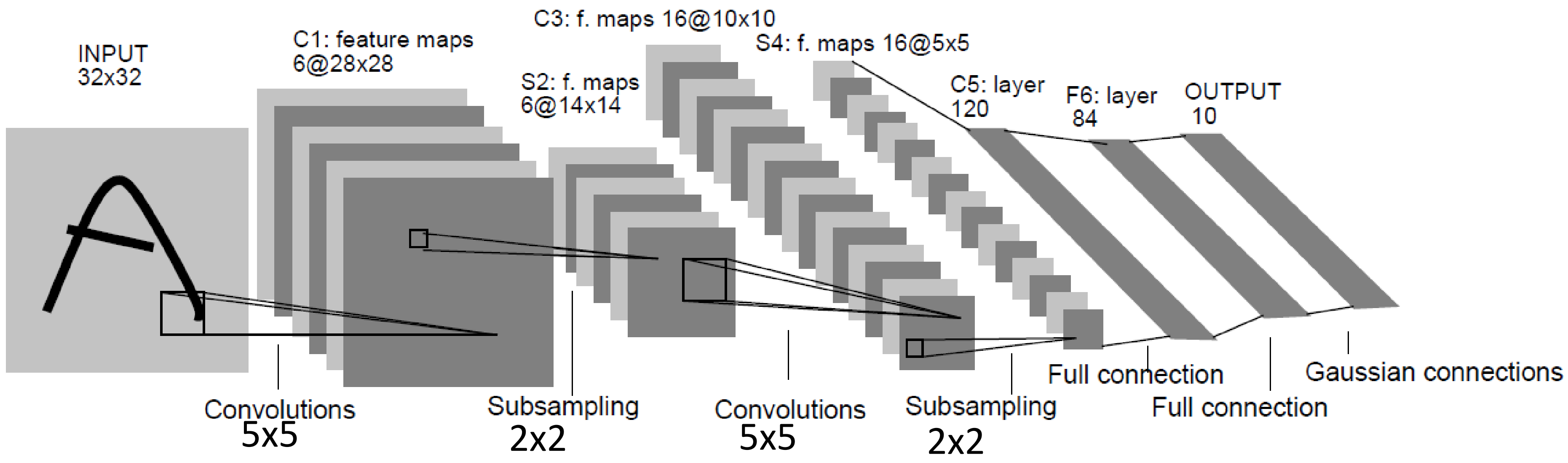
```
layer_defs = [];
layer_defs.push({type:'input', out_sx:32,
out_sy:32, out_depth:3});
layer_defs.push({type:'conv', sx:5,
filters:16, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2,
stride:2});
layer_defs.push({type:'conv', sx:5,
filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2,
stride:2});
layer_defs.push({type:'conv', sx:5,
filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2,
stride:2});
layer_defs.push({type:'softmax',
num_classes:10});
```

Outline

- Pooling
- Standard CNN
- Typical CNNs
- Advanced techniques

LeNet-5

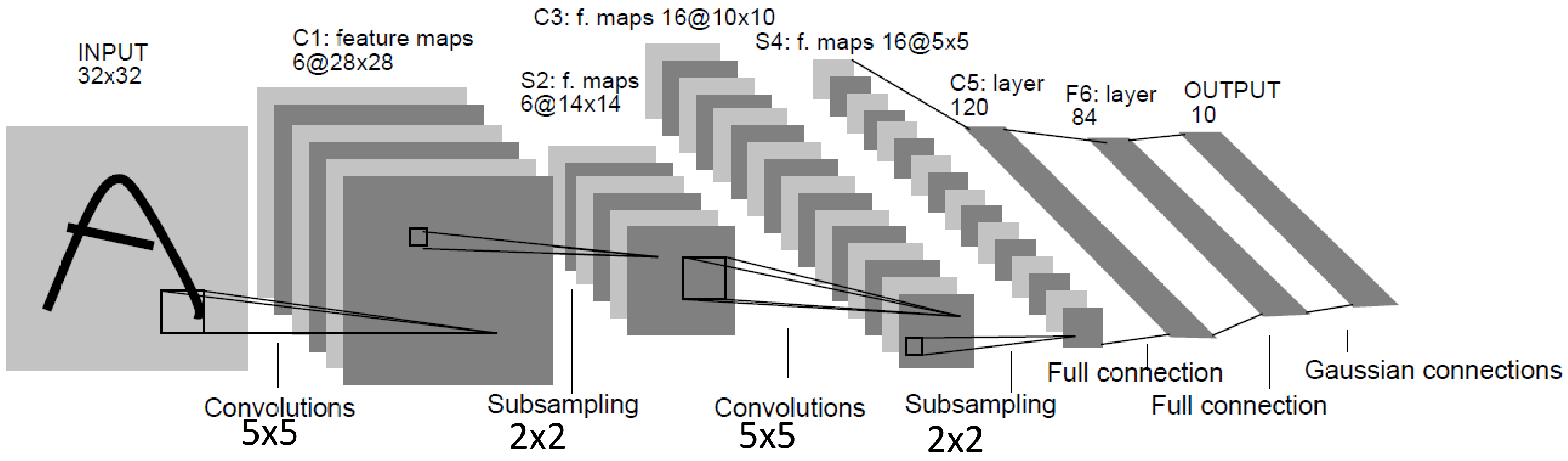
LeCun, Bottou, Bengio, Haffner, 1998



- Local connections and weight sharing
- C layers: convolution
  - Output $y_i = f(\sum_\Omega w_j x_j + b)$ where Ω is the patch size, $f(\cdot)$ is the sigmoid function, *w* and *b* are parameters
- S layers: subsampling (avg pooling)
  - Output $y_i = f(w \sum_\Omega x_j + b)$ where Ω is the pooling size

LeNet-5

LeCun, Bottou, Bengio, Haffner, 1998



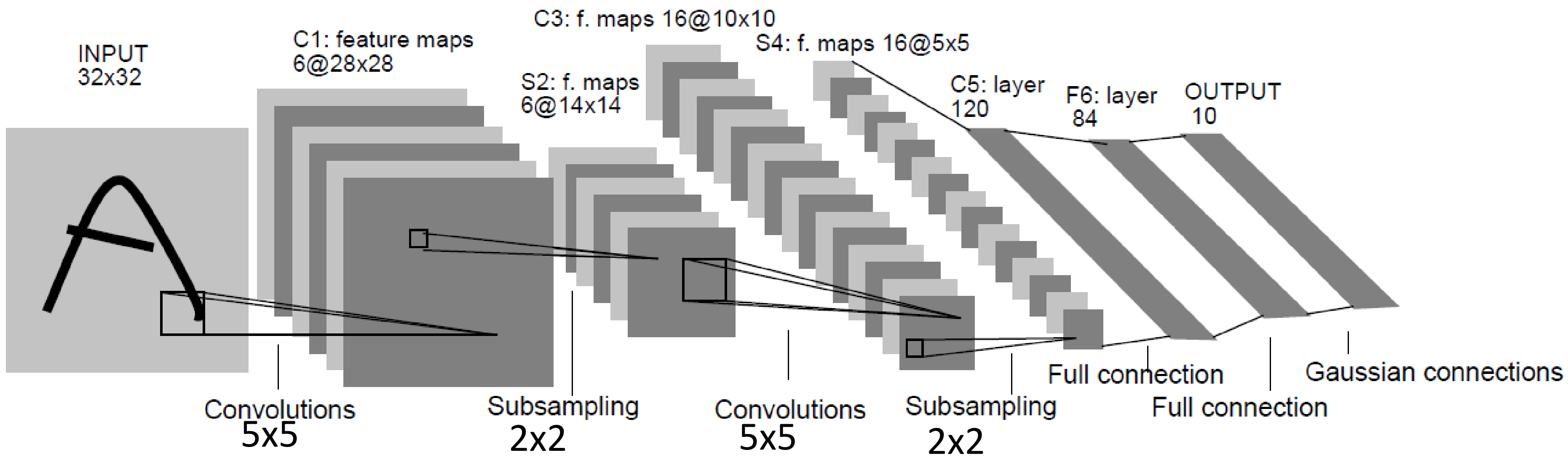
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

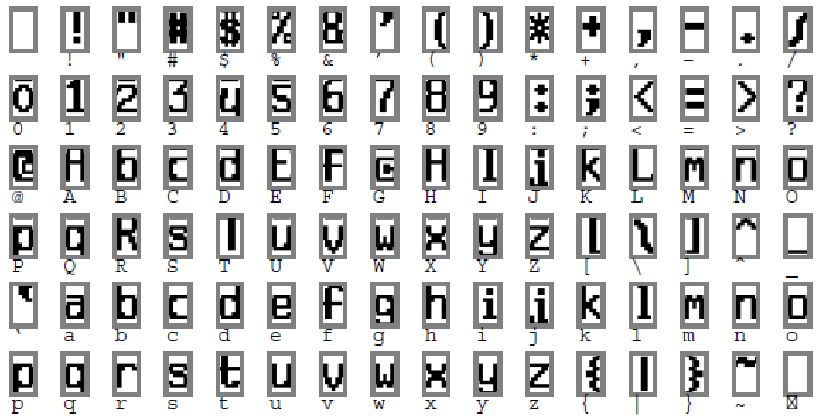
Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3.

LeNet-5

LeCun, Bottou, Bengio, Haffner, 1998



- Output layer: RBF units
- Loss function:
  - Maximum likelihood with modifications
- Train with back-prop



RBF centers

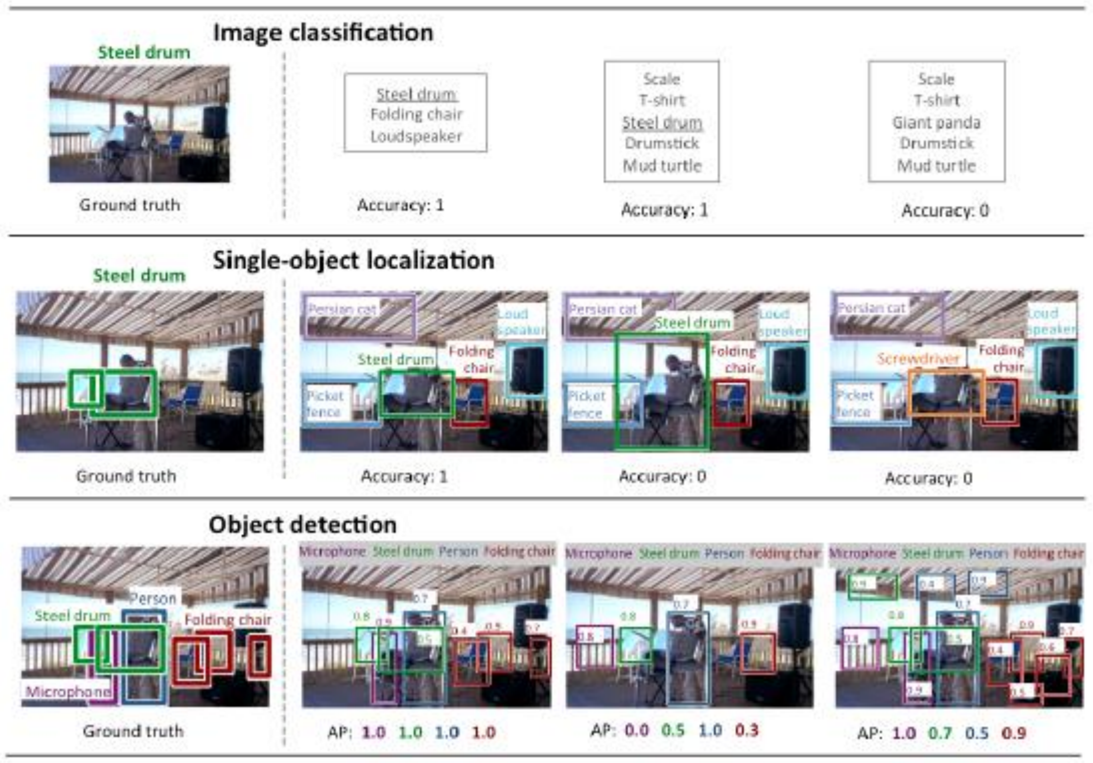
ImageNet comptition (ILSVRC)

Tasks

2010-

2011-

2013-



Top-1
Top-5 (preferred)

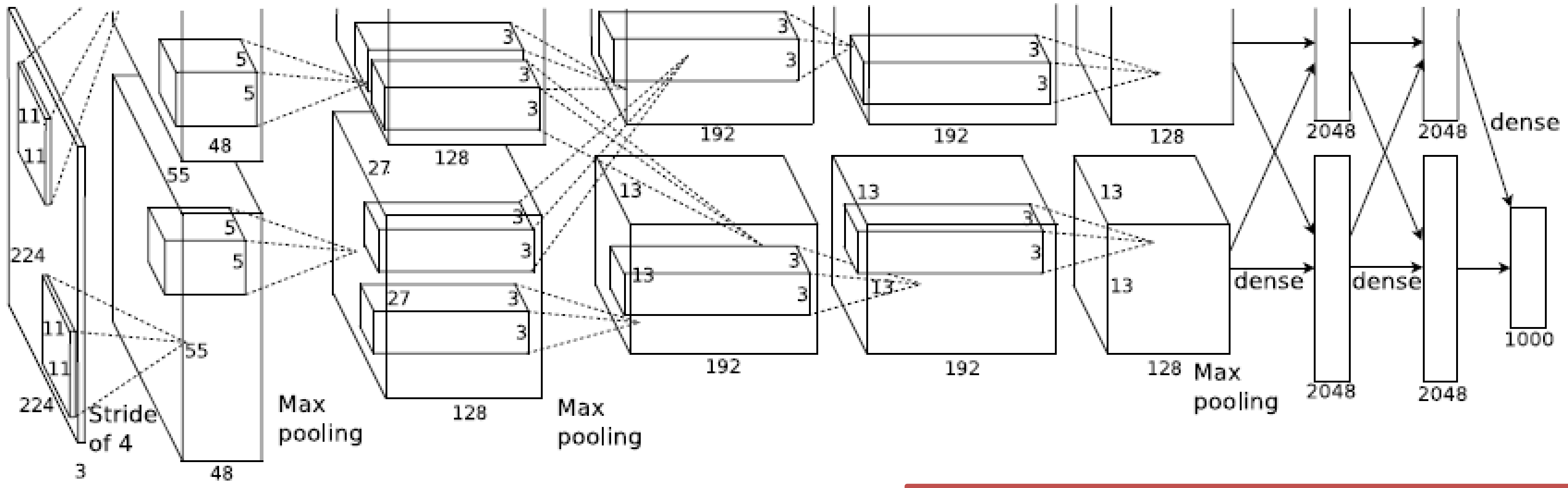
Two human expert: 5.1%, 12%

The first column shows the ground truth labeling on an example image, and the next three show three sample outputs with the corresponding evaluation score.

Russakovsky, et al., 2014

AlexNet

Krizhevsky, Sutskever and Hinton, NIPS, 2012



- Classification: 1000 classes, 1.2 million training images
- In total: 60 million parameters

Model	Top-1	Top-5
Sparse coding [2]	*47.1%*	*28.2%*
SIFT + FVs [24]	*45.7%*	*25.7%*
CNN	**37.5%**	**17.0%**

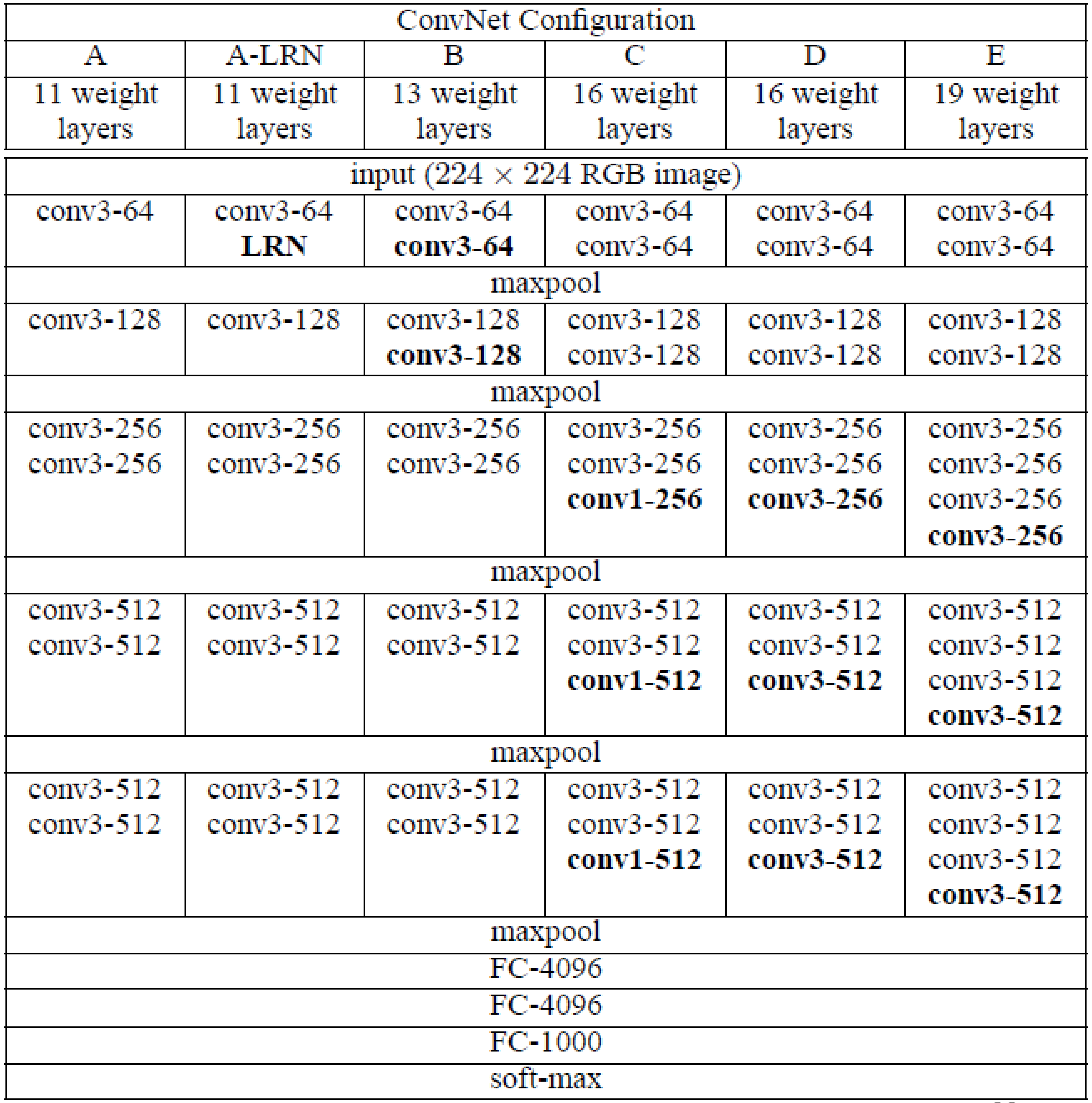
Since then, CNN dominates computer vision society

- In 2013, the vast majority of teams used CNN.
- In 2014 & 2015, almost all teams used convolutional neural networks.

VGG net

Simonyan, Zisserman, 2015

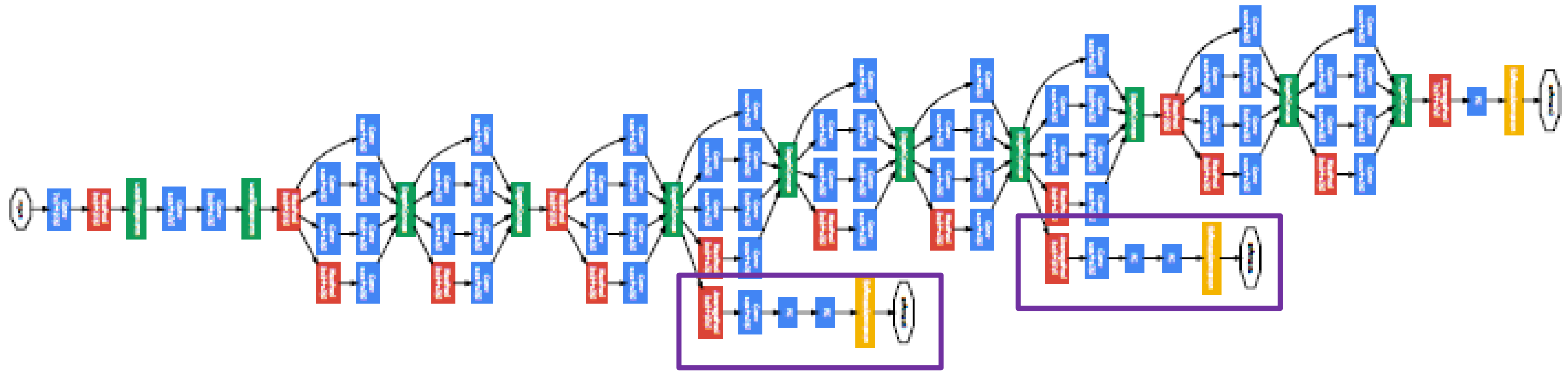
- 3*3 filters are extensively used
- GPU implementation

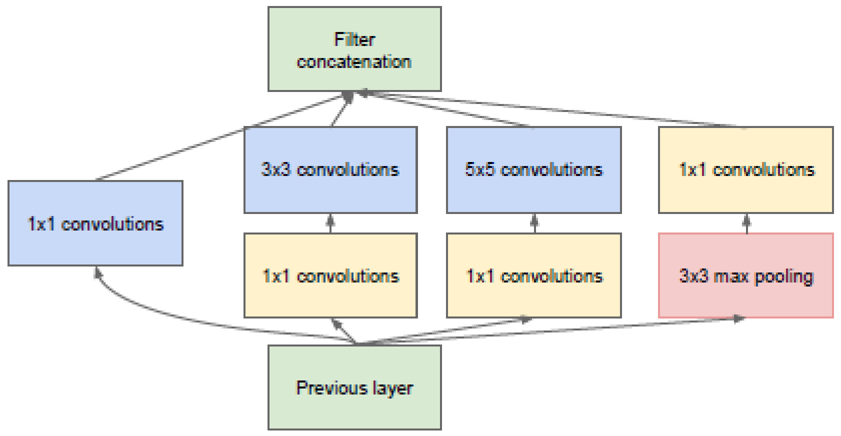


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	LRN	**conv3-64**	conv3-64	conv3-64	conv3-64
maxpool					
conv3-128	conv3-128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

GoogLeNet (Inception-v1)

Szegedy, et al., 2014





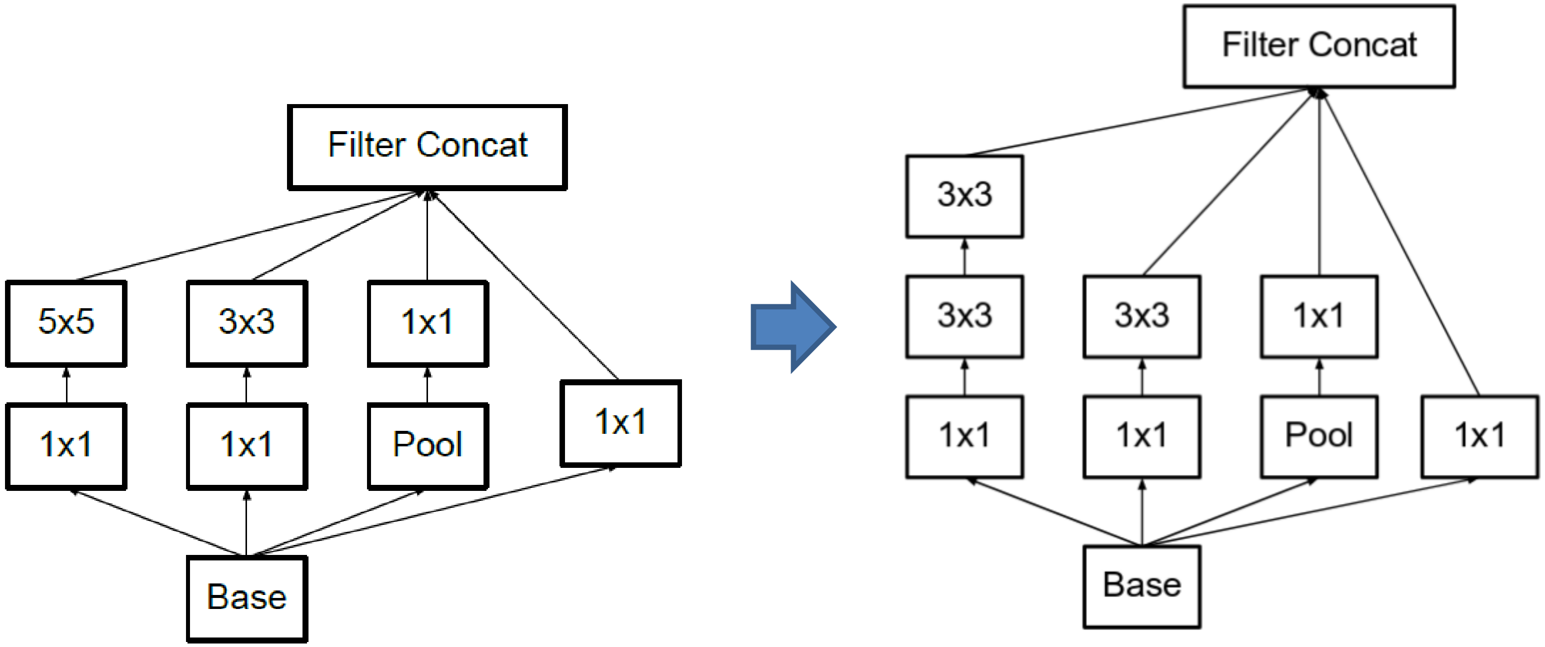
- 22 weight layers
- Small filters (1x1, 3x3, 5x5)
- Two auxiliary classifiers connected to intermediate layers are used to increase the gradient signal for BP algorithm
- A cpu-based implementation on distributed system

- Multiple sizes in the same layer
- 1×1 conv are used to reduce the number of channels

Inception-v2

Szegedy, et al., 2016

- Factorize 5x5 convolution to two 3x3 convolution operations to improve computational speed
  - Computational cost: 25/(9+9)=1.38

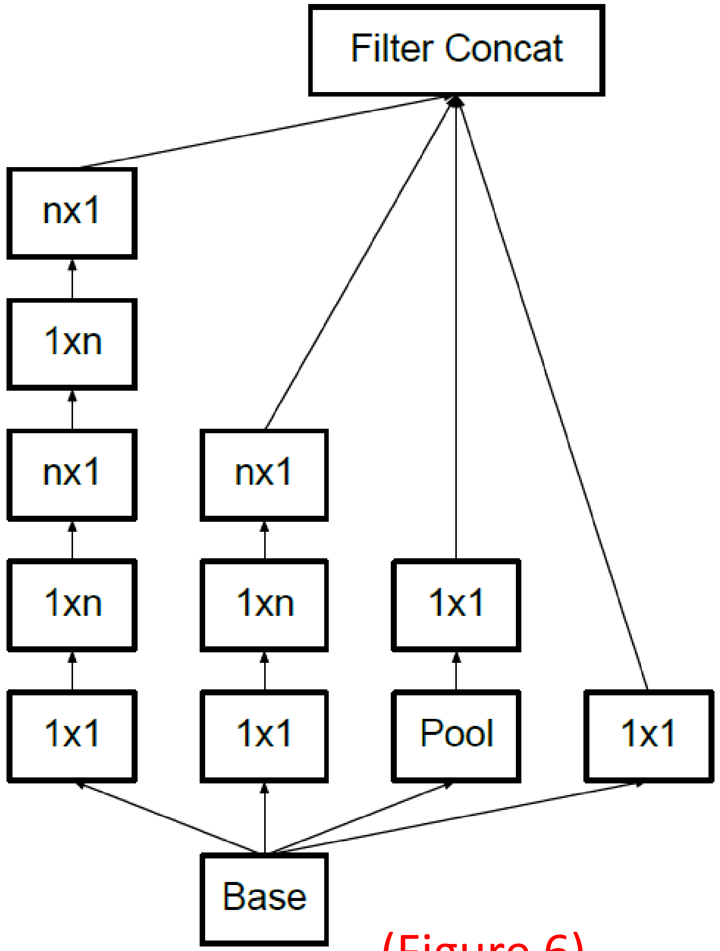


(Figure 5)

Inception-v2

Szegedy, et al., 2016

- Factorize convolutions of filter size nxn to a combination of 1xn and nx1 convolutions
  - Let n=3. They found this method to be 33% more cheaper than the single 3x3 convolution

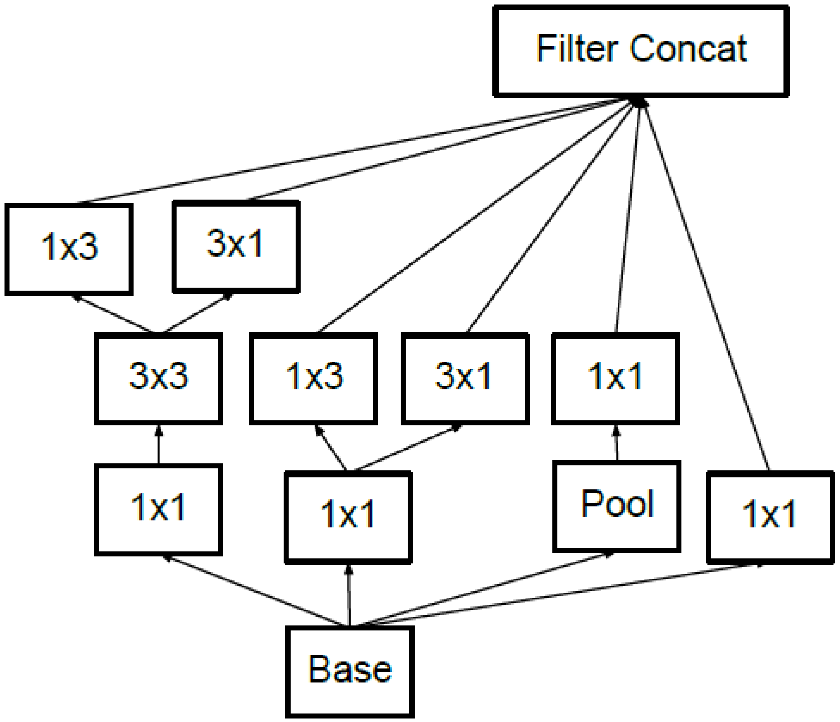


(Figure 6)

Inception-v2

Szegedy, et al., 2016

- The filter banks in the module were expanded (made wider instead of deeper) to remove the representational bottleneck



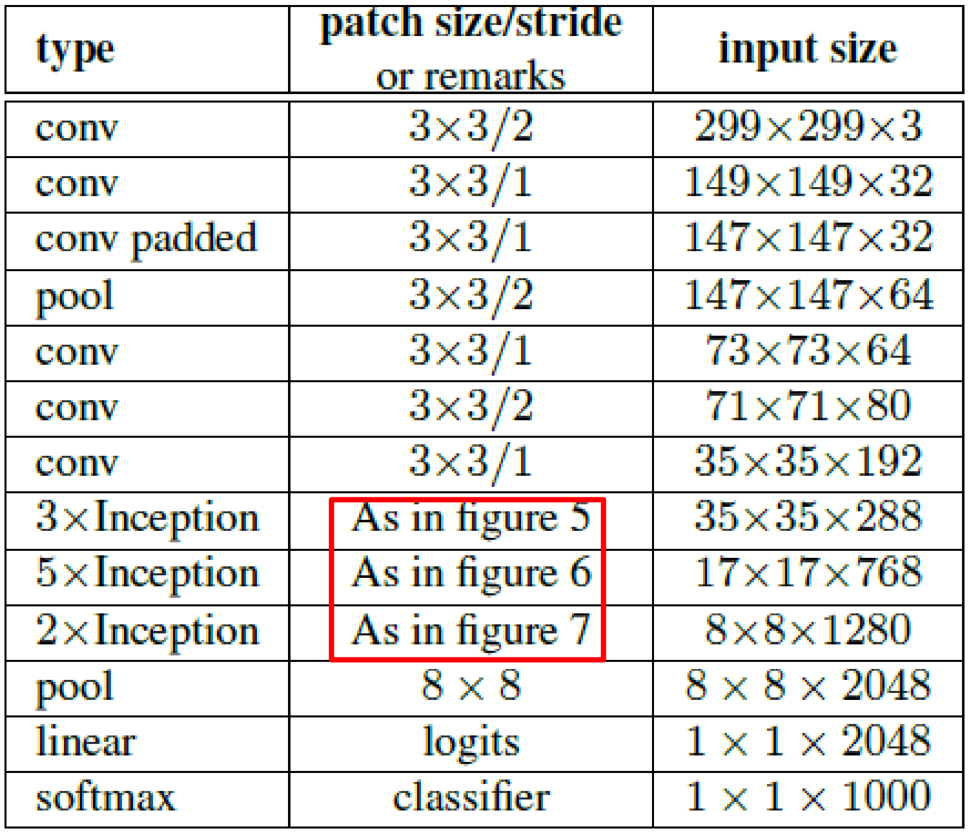
(Figure 7)

This architecture is used on the coarsest (8X8) grids to promote high dimensional representations

Inception-v2

Szegedy, et al., 2016

Entire architecture



type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Inception-v3

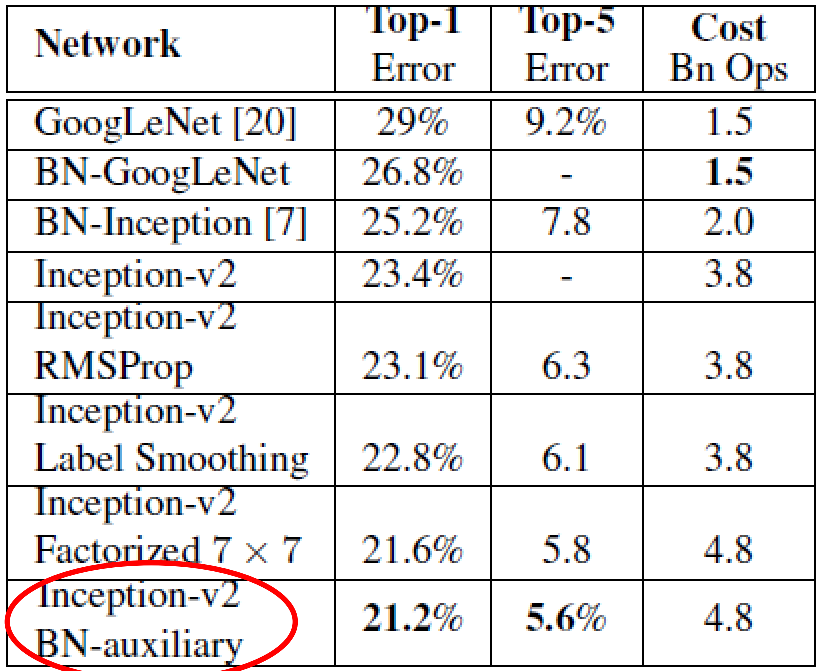
Szegedy, et al., 2016

- Add additional techniques to Inception-v2:
  - RMSProp Optimizer.
  - Factorized 7x7 convolutions.
  - BatchNorm in the Auxillary Classifiers.
  - Label Smoothing (A type of regularizing component to prevent overfitting).

Results on ImageNet

Szegedy, et al., 2016

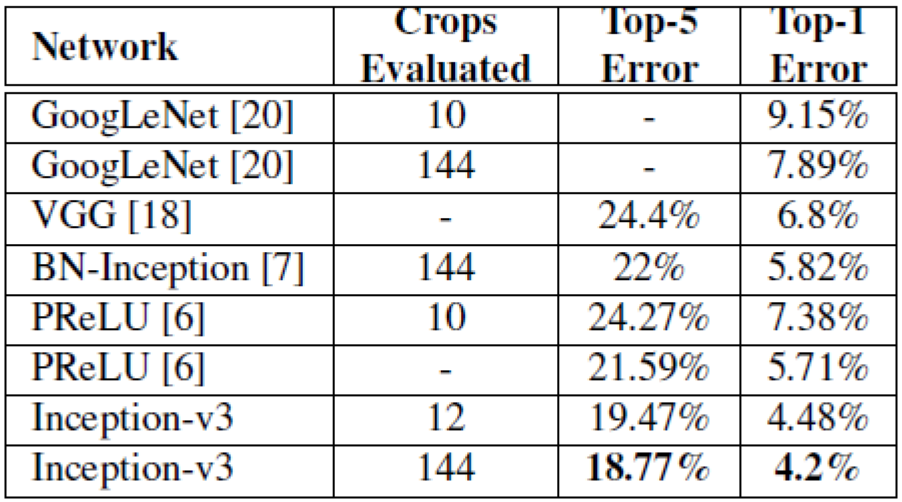
Single-crop results



Network	**Top-1 Error**	**Top-5 Error**	**Cost Bn Ops**
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	**1.5**
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7×7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	**21.2%**	**5.6%**	4.8

↑ Inception v3

Multi-crop results

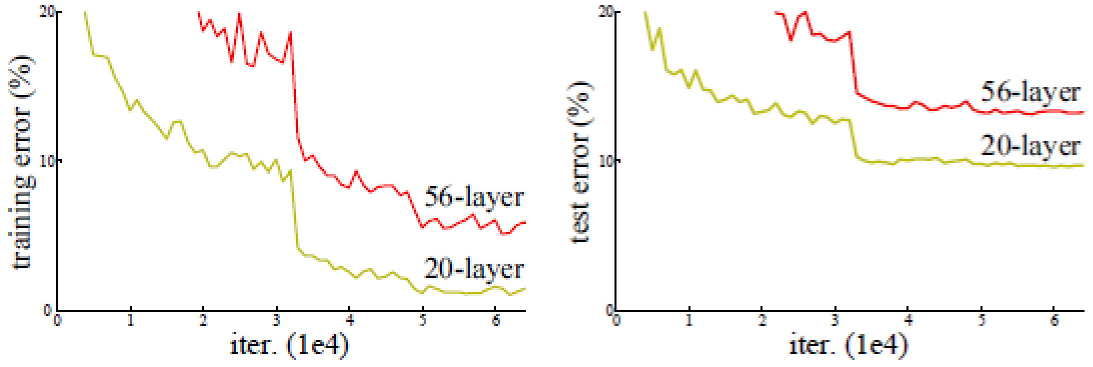


Network	**Crops Evaluated**	**Top-5 Error**	**Top-1 Error**
GoogLeNet [20]	10	-	9.15%
GoogLeNet [20]	144	-	7.89%
VGG [18]	-	24.4%	6.8%
BN-Inception [7]	144	22%	5.82%
PReLU [6]	10	24.27%	7.38%
PReLU [6]	-	21.59%	5.71%
Inception-v3	12	19.47%	4.48%
Inception-v3	144	**18.77%**	**4.2%**

Deep residual network (ResNet)

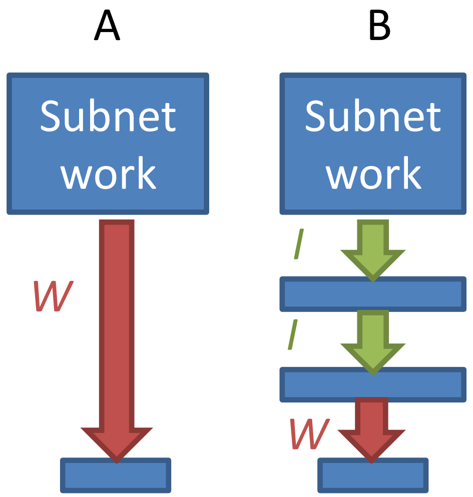
He et al., 2016

- Empirical observations: with the network depth increasing, accuracy gets saturated and then degrades rapidly



- Is it caused by over-fitting?
- This is counterintuitive!
  - Error of B should not be larger than that of A

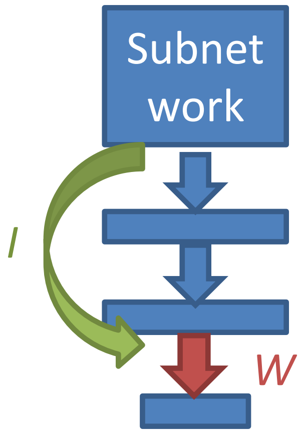
Hypo: it's difficult for nonlinear layers to approx. the identity mapping

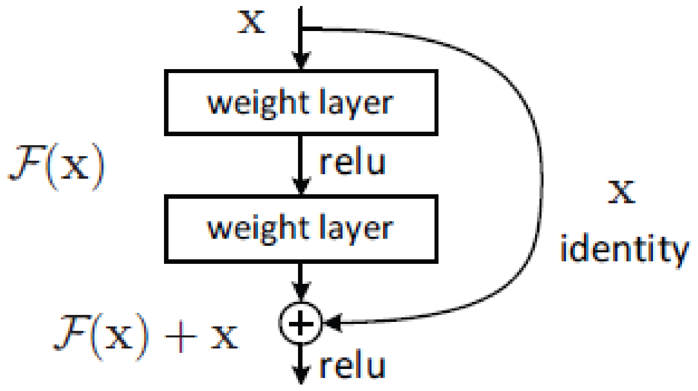


Deep residual network (ResNet)

He et al., 2016

- If this hypothesis is valid, let's explicitly use the identity mapping




x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

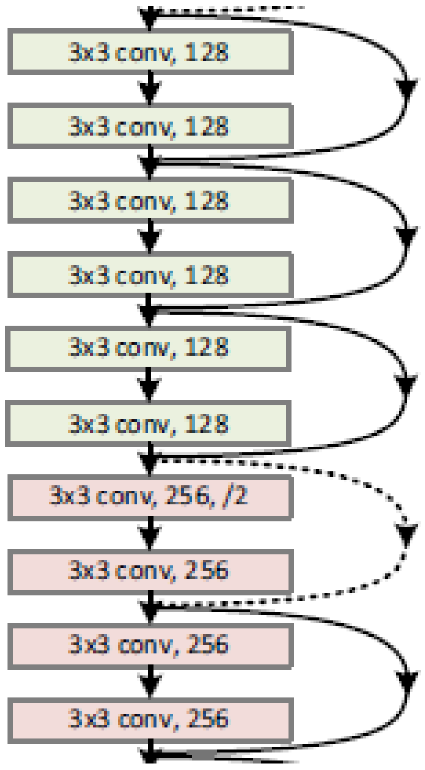
- The nonlinear mapping from input to output $H(x)$ has two parts

$$H(x) = F(x) + x$$

- Then the (two) weight layers are learning $F(x)$, that is the residual $H(x) - x$

Deep residual network (ResNet)

He et al., 2016

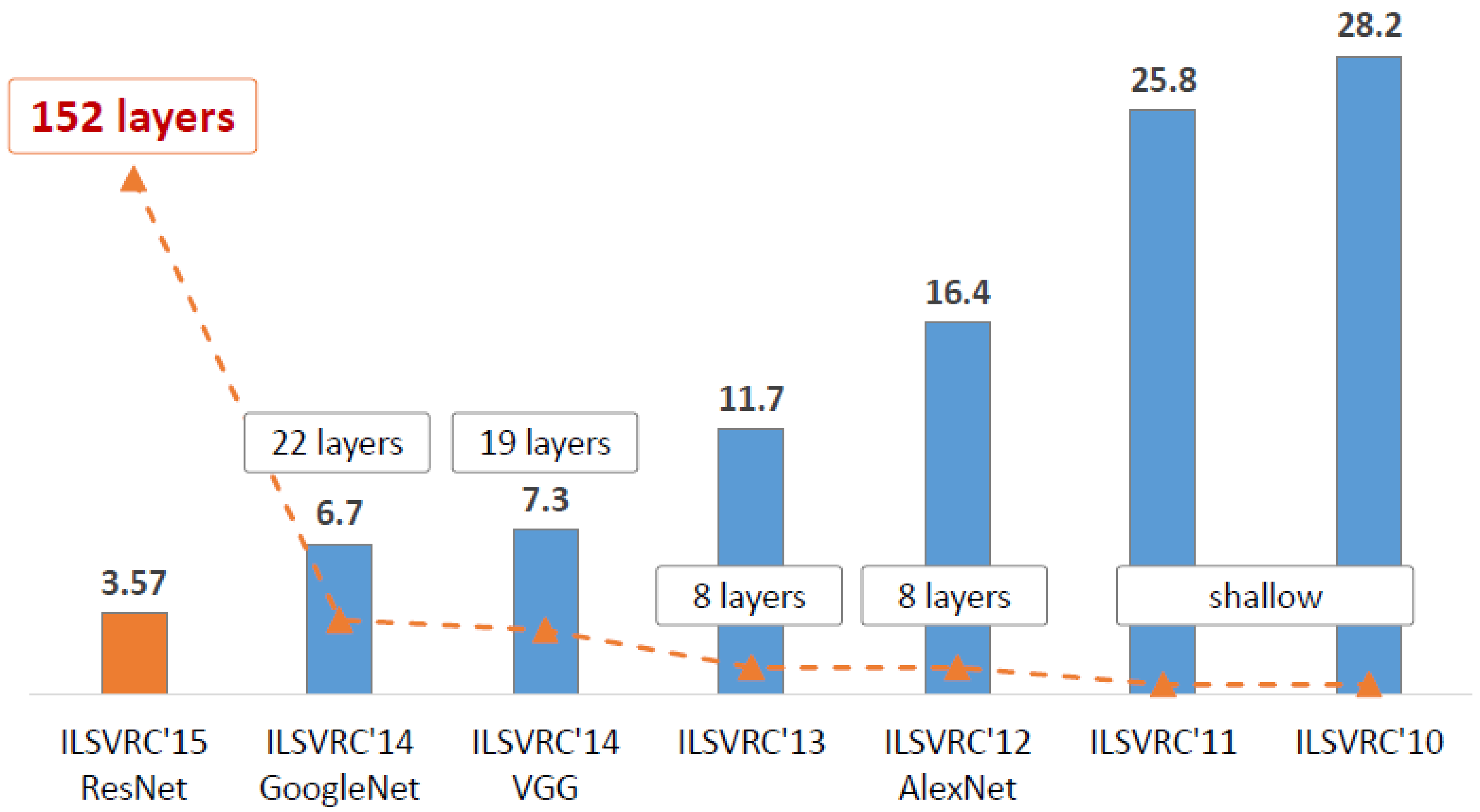


A 152-layer network achieves 3.57% error rate

method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

1000-layer model has also been tested on CIFAR-10

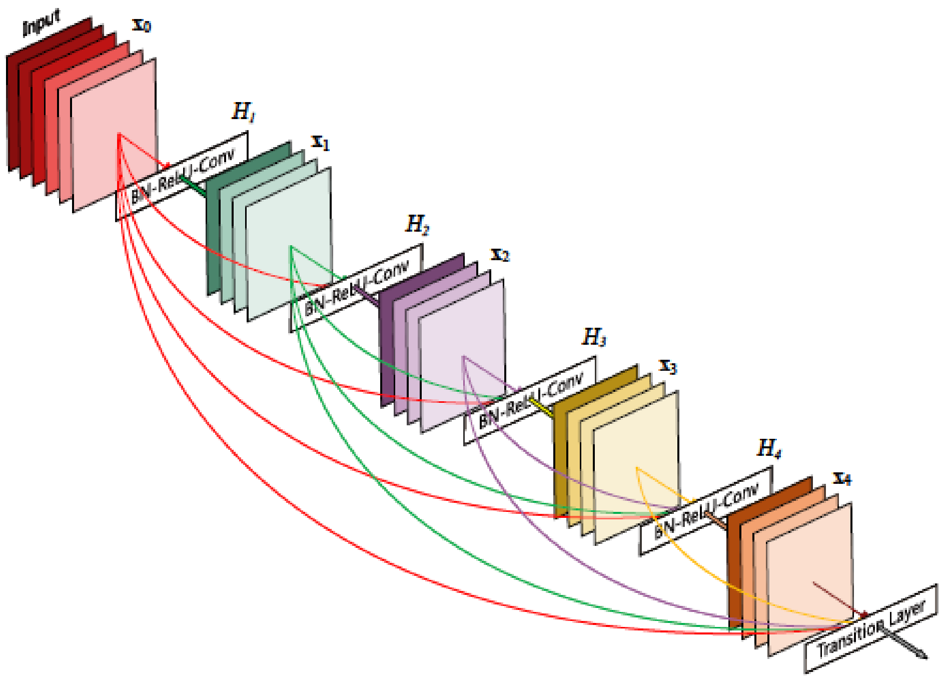
Revolution of depth



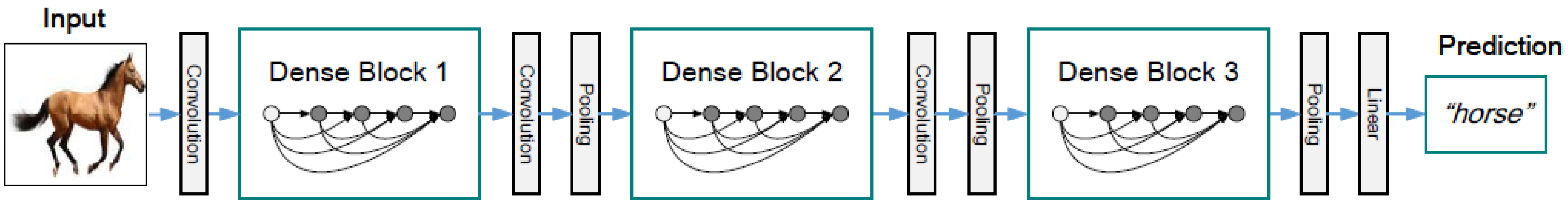
Slide credit: Kaming He

DenseNet

Huang et al., CVPR 2017

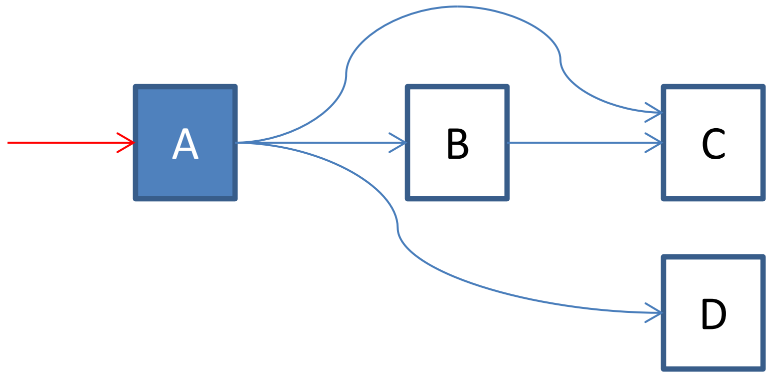


- Each layer takes all preceding feature-maps as input, which are *concatenated* together
- An L-layer net has $\frac{L(L+1)}{2}$ connections
- Each layer outputs k feature maps and k is small (e.g., 12)



Branching layer

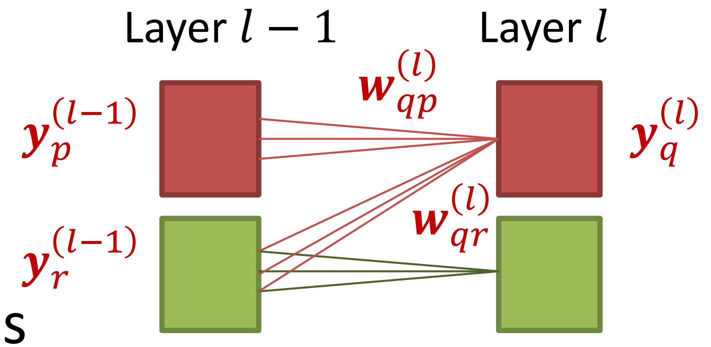
- We have seen many models having branching layers



- Feedforward calculation is straightforward
- During backward pass
  – How to calculate the gradient of parameters indicated by the red arrow?
  – How to calculate the local sensitivity in block A?

Recap: 2D convolution with feature combination

- Suppose that the l-th layer is a convolutional layer



Layer $l-1$
Layer l
$\boldsymbol{w}_{qp}^{(l)}$
$\boldsymbol{y}_p^{(l-1)}$
$\boldsymbol{y}_q^{(l)}$
$\boldsymbol{w}_{qr}^{(l)}$
$\boldsymbol{y}_r^{(l-1)}$

- Forward pass

$$\boldsymbol{y}_q^{(l)} = \sum_{p \in \mathcal{M}_q} \boldsymbol{y}_p^{(l-1)} *_{\text{valid}} \text{rot180}(\boldsymbol{w}_{qp}^{(l)}) + b_q^{(l)}$$

- Backward pass
  - Gradient:

$$\frac{\partial E^{(n)}}{\partial \boldsymbol{w}_{qp}^{(l)}} = \boldsymbol{y}_p^{(l-1)} *_{\text{valid}} \text{rot180}(\boldsymbol{\delta}_q^{(l)}), \quad \frac{\partial E^{(n)}}{\partial b_q^{(l)}} = \sum_i (\boldsymbol{\delta}_q^{(l)})_{ij}$$

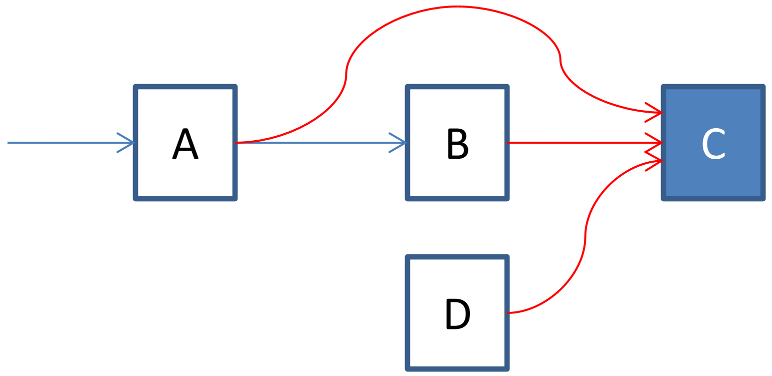
  - Local sensitivity:

$$\boldsymbol{\delta}_p^{(l-1)} = \sum_{q \in \tilde{\mathcal{M}}_p} \boldsymbol{\delta}_q^{(l)} *_{\text{full}} \boldsymbol{w}_{qp}^{(l)}$$

($\mathcal{M}_q$ and $\widetilde{\mathcal{M}}_p$ are defined before)

Combination layer

- We have seen many models having combination layers



- Feedforward calculation is straightforward
- During backward pass
  - How to calculate the gradient of parameters indicated by red arrows?
  - How to calculate the local sensitivity in block C?
- What if the combination is not summation but "concatenation"?

Outline

- Pooling
- Standard CNN
- Typical CNNs
- Advanced techniques
  - Optimizers
  - Dropout
  - Batch normalization

Advanced techniques

- Optimizers
- Dropout
- Batch normalization

Recap: SGD and momentum

- SGD optimizes over individual minibatches $\left(\boldsymbol{X}^{(i)}, \boldsymbol{t}^{(i)}\right)$ at each iteration

$$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)}\right)$$
$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{g}$$

- The momentum update is given by,

$$\boldsymbol{v} = \gamma \boldsymbol{v} - \eta \boldsymbol{g}$$
$$\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{v}$$

- We need to adjust learning rates η during training
  - Recall different strategies
  - Tuning the learning rates is expensive!
- Are there any methods that can adaptively tune the learning rates?

Per-parameter adaptive learning rate methods

- The previous method manipulates the learning rate globally and equally for all parameters
- It is possible to adaptively tune the learning rates for individual parameters
- Many of these methods may still require other hyperparameter settings, but the argument is that they are well-behaved for a broader range of hyperparameter values than the raw learning rate

Adagrad

- An adaptive learning rate method (Duchi et al. 2011)
- Denote the gradient by $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)}\right)$
- The updating rule

$$\left.\begin{aligned} \boldsymbol{c} &= \boldsymbol{c} + \boldsymbol{g}^2 \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \eta \frac{\boldsymbol{g}}{\sqrt{\boldsymbol{c} + \epsilon}} \end{aligned}\right\} \text{Elementwise operations}$$

  where ϵ is usually set between 10^{-4} and 10^{-8}
- c is used to normalize the parameter update step, elementwise
  - Parameters received small updates will have larger effective learning rates
- Any problem with this method?
  - The effective learning rates are monotonically decreasing, which may leads to early stopping

RMSProp

- RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate
- In practice

$$\boldsymbol{c} = \gamma \boldsymbol{c} + (1 - \gamma)\boldsymbol{g}^2$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \frac{\boldsymbol{g}}{\sqrt{\boldsymbol{c} + \epsilon}}$$

- Typical values for γ are 0.9, 0.99, 0.999
- RMSProp still modulates the learning rate of each parameter based on the magnitudes of its gradients, but unlike Adagrad the updates do not get monotonically smaller.

Adam

- This method is proposed by Kingma and Ba (2015). Default algorithm!
- The simplified version

$$\boldsymbol{m} = \beta_1 \boldsymbol{m} + (1 - \beta_1)\boldsymbol{g}$$
$$\boldsymbol{c} = \beta_2 \boldsymbol{c} + (1 - \beta_2)\boldsymbol{g}^2$$
$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \frac{\boldsymbol{m}}{\sqrt{\boldsymbol{c} + \epsilon}}$$

Recommended values: $\epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

  - Compared with RMSProp, it uses the "smooth" version of the gradient $\boldsymbol{m}$. This is like a momentum.
- The full version ("warm up" version)

$$\boldsymbol{m} = \beta_1 \boldsymbol{m} + (1 - \beta_1)\boldsymbol{g}, \qquad \boldsymbol{m}_t = \boldsymbol{m}/(\boldsymbol{1} - \beta_1^t)$$
$$\boldsymbol{c} = \beta_2 \boldsymbol{c} + (1 - \beta_2)\boldsymbol{g}^2, \qquad \boldsymbol{c}_t = \boldsymbol{c}/(\boldsymbol{1} - \beta_2^t)$$
$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \frac{\boldsymbol{m}_t}{\sqrt{\boldsymbol{c}_t + \epsilon}}$$

where t denotes the iteration

Advanced techniques

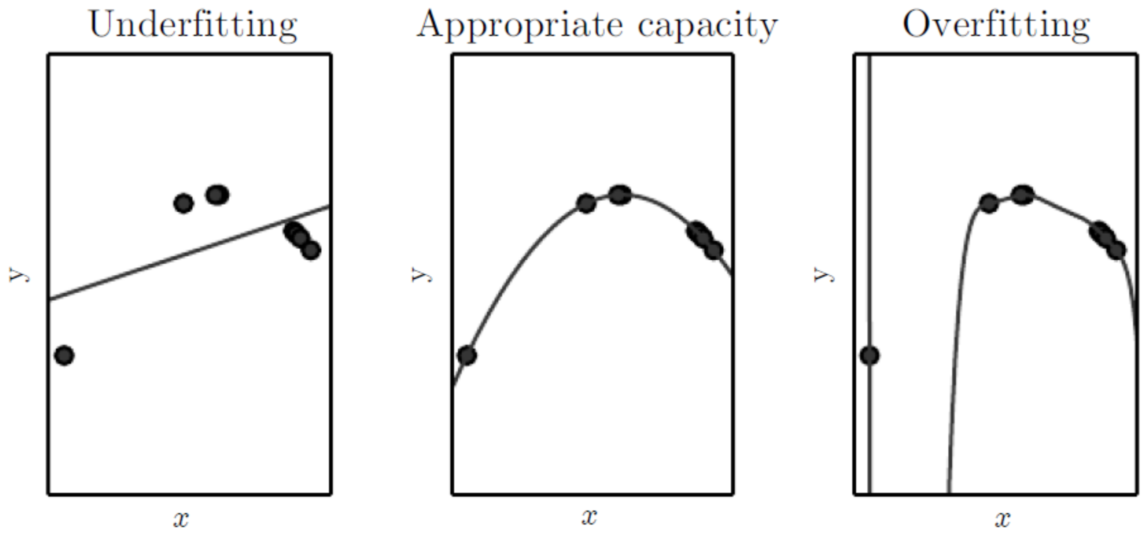
- Optimizers
- Dropout
- Batch normalization

Recall: polynomial regression example

- Consider a regression problem in which the input x and output y are both scalars. Find a function $f: \mathbb{R} \rightarrow \mathbb{R}$ to fit the data
  - $f(x) = b + wx$
  - $f(x) = b + w_1 x + w_2 x^2$
  - $f(x) = b + \sum_{i=1}^{9} w_i x^i$

MSE training:

$$\min_w \frac{1}{N} \sum_{n=1}^{N} \left\| f\left(x^{(n)}\right) - y^{(n)} \right\|_2^2$$

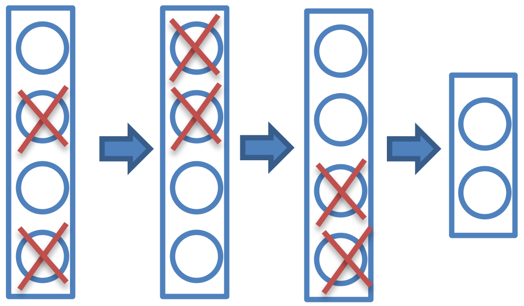


Overfitting

- A neural network (as well as other machine learning models) typically contains many parameters to learn (e.g., millions to billions) which tends to overfit the training data
- What's overfitting?
  - Fits the training data well but performs poorly on held-out test data
- Weight regularization is one method to handle this
- Dropout is another method, which was proposed by Hinton et al. (2012)

Dropout

- On each presentation of each training case, each hidden unit is randomly omitted (zero its output) from the network with a probability p (e.g., 0.5)

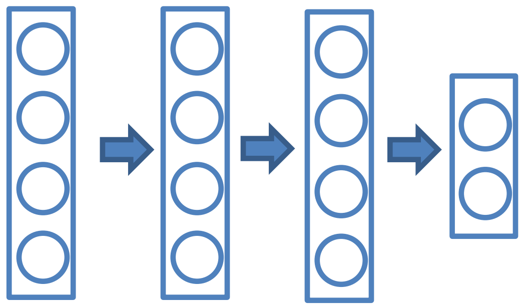


These zero values are used in the backward pass propagating the derivatives to the parameters

- Advantage
  - A hidden unit cannot rely on other hidden units being present, therefore we prevent complex co-adaptations of the neurons on the training data
  - It trains a huge number of different networks in a reasonable time, then average their predictions

Testing phase

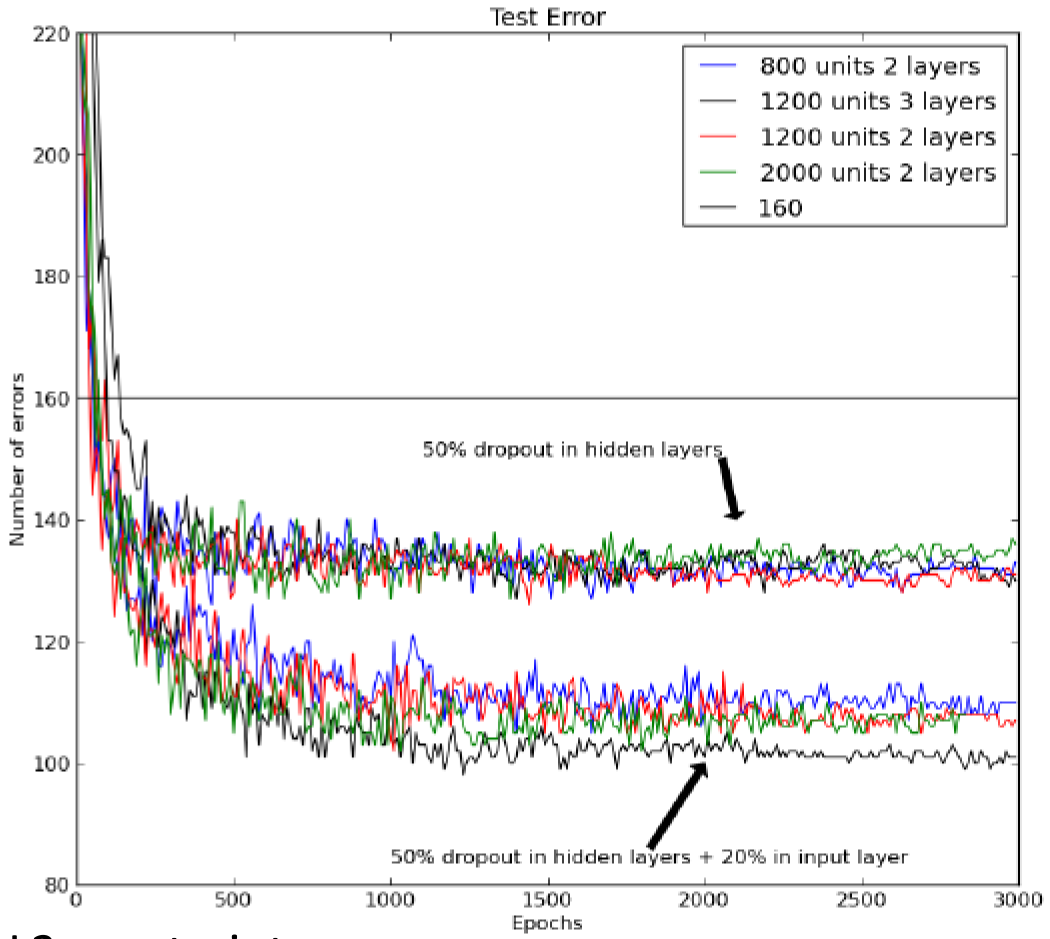
- Use the “mean network” that contains all of the hidden units



- But we need to adjust the outgoing weights of neurons to compensate for the fact that in training only a portion of them are active
  - If $p = 0.5$, we halve the weights
- In practice, this gives very similar performance to averaging over a large number of dropout networks.

Results on MNIST

- Standard MLP without the tricks
  - Enhancing training data with transformed images
  - Generative pre-training
- In this setting without dropout the best results is 160 errors
- Dropout can significantly reduce errors



Another trick is used: separate L2 constraints on the incoming weights of each hidden unit

Hinton et al., 2012

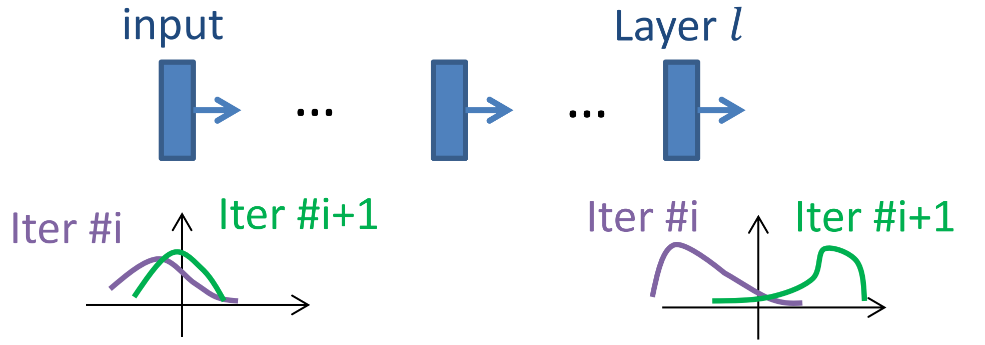
Advanced techniques

- Optimizers
- Dropout
- Batch normalization

"Internal covariate shift"

- Since we use SGD, the input mini-batches to the neural network at different iterations are different
- This *may* cause the distributions of the output of a layer to be different at different iterations

Distribution of a neuron (or some neurons) over a mini-batch



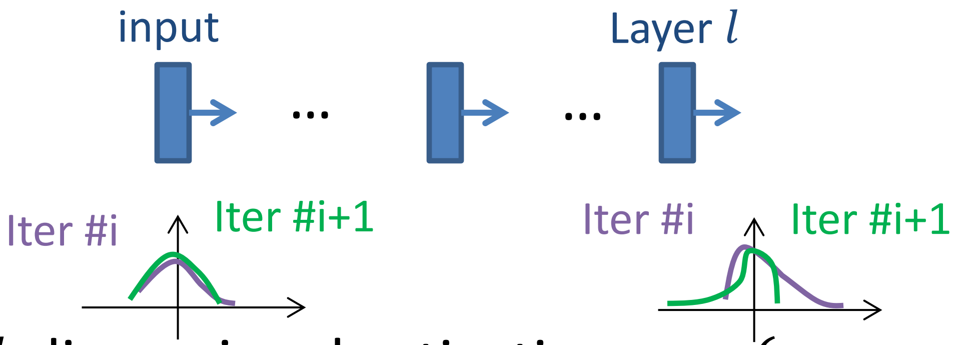
Internal Covariate Shift (ICS): The chage in the distributions of internal nodes of a deep network, in the course of training

It *may* cause difficulty in optimization

Reduce ICS by normalization

Ioffe, Szegedy, 2015

- We normalize each scalar feature independently, by making it have the mean of zero and the variance of 1



- Denote a d-dimensional activation $\boldsymbol{x} = (x_1, \ldots, x_d)$, do normalization

$$\hat{x}_i = \frac{x_i - \mathrm{E}[x_i]}{\sqrt{\mathrm{Var}[x_i]}}$$

- Keep the representation ability of the layer

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

If $\gamma_i = \sqrt{\mathrm{Var}[x_i]}$ and $\beta_i = \mathrm{E}[x_i]$, then we recover the original activations

Batch normalization

- Construct a new layer:

$$\boldsymbol{y}^{(n)} = BN_{\boldsymbol{\gamma},\boldsymbol{\beta}}\left(\boldsymbol{x}^{(n)}\right)$$

- Where $\boldsymbol{\gamma}, \boldsymbol{\beta}, \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)} \in R^d$
- Forward pass
  - $\boldsymbol{\mu}_B = \frac{1}{m}\sum_{n=1}^{M} \boldsymbol{x}^{(n)}$
  - $\boldsymbol{\sigma}_B^2 = \frac{1}{m}\sum_{n=1}^{M}\left(\boldsymbol{x}^{(n)} - \boldsymbol{\mu}_B\right)^2$
  - $\widehat{\boldsymbol{x}}^{(n)} = \frac{\boldsymbol{x}^{(n)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$
  - $\boldsymbol{y}^{(n)} = \boldsymbol{\gamma}\widehat{\boldsymbol{x}}^{(n)} + \boldsymbol{\beta}$

The arithmetic operations are elementwise

- What are needed to be computed in the backward pass?

During inference

- The normalization of activations that depends on the mini-batch allows efficient training, but is neither necessary nor desirable during inference
- Once the network has been trained, we normalize

$$\widehat{\boldsymbol{x}} = \frac{\boldsymbol{x} - \mathrm{E}[\boldsymbol{x}]}{\sqrt{\mathrm{Var}[\boldsymbol{x}] + \epsilon}}$$

  where $\mathrm{E}[\boldsymbol{x}]$ and $\mathrm{Var}[\boldsymbol{x}]$ are measured over the entire training set

  - $\mathrm{E}[\boldsymbol{x}] = \mathrm{E}_B[\boldsymbol{\mu}_B]$
  - $\mathrm{Var}[\boldsymbol{x}] = \frac{m}{m-1}\mathrm{E}_B[\boldsymbol{\sigma}_B^2]$ where m is the number of mini-batches

- If you want to track the accuracy of a model as it trains, you can use the moving averages instead

Location in a network

- Often applied before the non-linearity of the previous layer
  - After nonlinearity, the shape of the activation distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift
- The previous layer is always a linear transformation layer (fully-connected layer or convolutional layer)

$$\boldsymbol{y}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}$$

  - The bias term can be ignored because BN has a shift term that have the same effect. Therefore

$$\boldsymbol{y}^{(l)} = \mathrm{BN}\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)}\right)$$

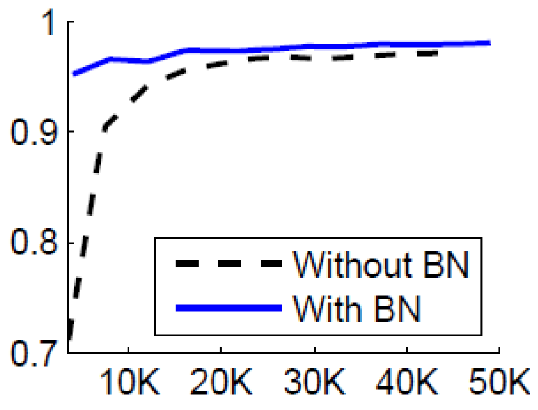
BN in CNN

- As said before, BN is applied after the convolutional layer
- Require different elements of the same feature map, at different locations are normalized in the same way
- Normalize all activations in a mini-batch, over all locations
  - Suppose the mini-batch size is M, and the feature map size is $P \times Q$, then the mean and variance are calculated across $M \cdot P \cdot Q$ elements
  - Learn a pair of parameters γ_i and β_i per feature map, rather than per activation
- The inference procedure is modified similarly, so that during inference BN applies the same linear transformation to each activation in a given feature map

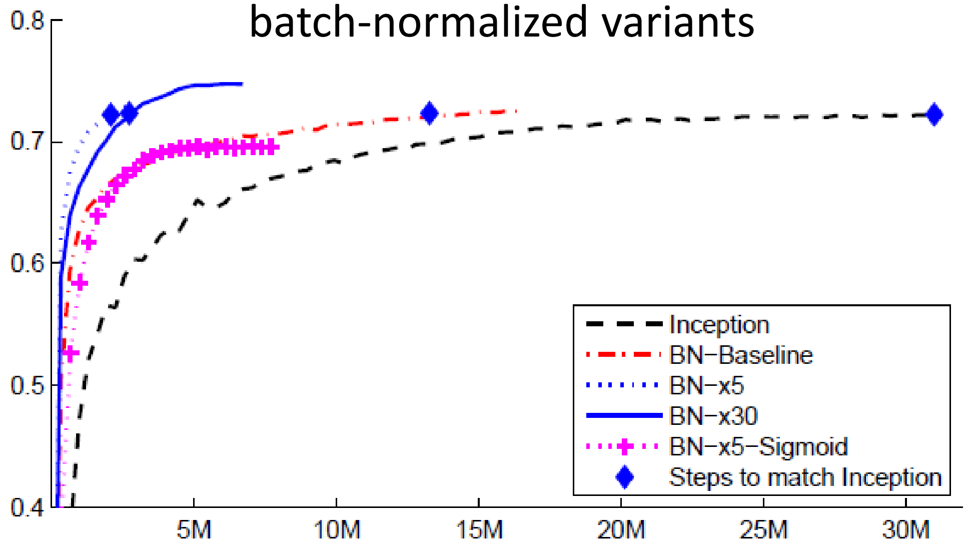
Advantages

- BN enables higher learning rates
- BN regularizes the model
  - The training network no longer produce deterministic values for a given training example
  - Dropout may not be needed

Test accuracy on the MNIST



Single crop validation accuracy of Inception and its batch-normalized variants



X-axis: The number of training steps

Summary

- Standard CNN
  - Convolution and pooling
- Typical CNNs
  - AlexNet
  - VggNet
  - GoogLeNet, Inception v2, v3
  - ResNet
  - DenseNet
- Optimizers
  - Adagrad, RMSProp, Adam
- Dropout
  - A regularizer
- Batch normalization
  - Good performance in practice but the reason is controversial

Further reading

- http://cs231n.github.io/neural-networks-3/#update