MIT CSAIL

6.869: Advances in Computer Vision

Antonio Torralba, 2016

MIT COMPUTER VISION

Lecture 7

Learned feedforward visual processing

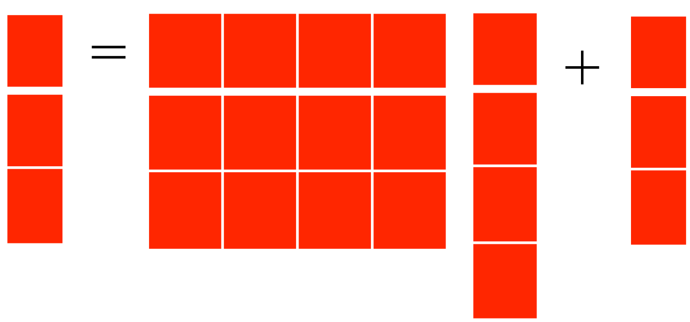
Tutorials

- Lunes: 4pm --> Torch
- Martes: 5pm --> TensorFlow
- Miércoles: 5pm--> Torch
- Jueves: 6pm ---> TensorFlow

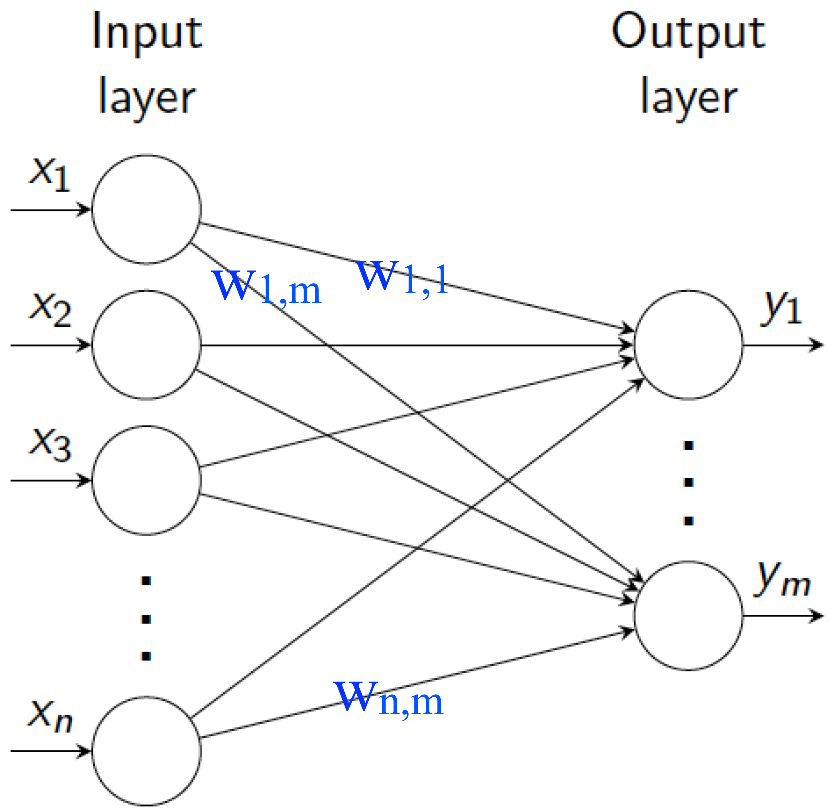
Single layer network

- Input: column vector x (size n×1)
- Output: column vector y (size m×1)
- Layer parameters:
  weight matrix W (size n×m)
  bias vector b (m×1)
- Units activation: $a = Wx + b$

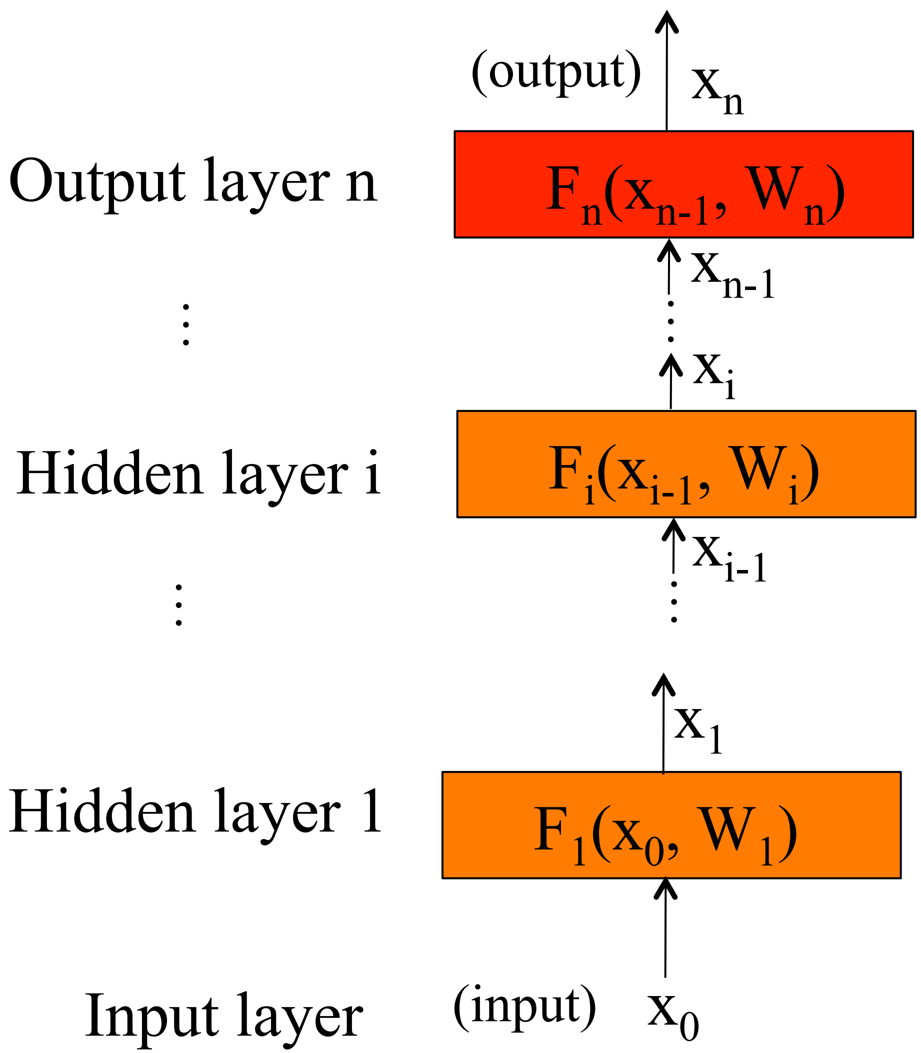
  ex. 4 inputs, 3 outputs

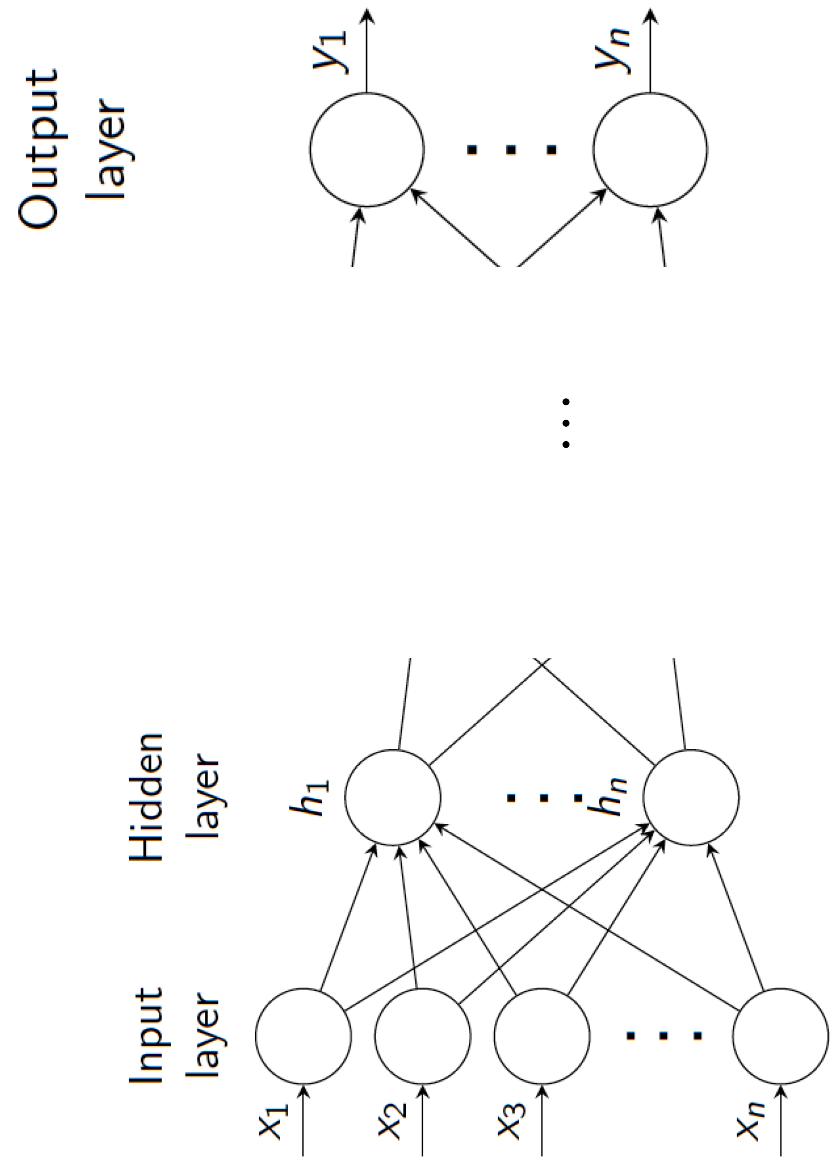


- Output: $y = f(a) = f(Wx + b)$



Multiple layers





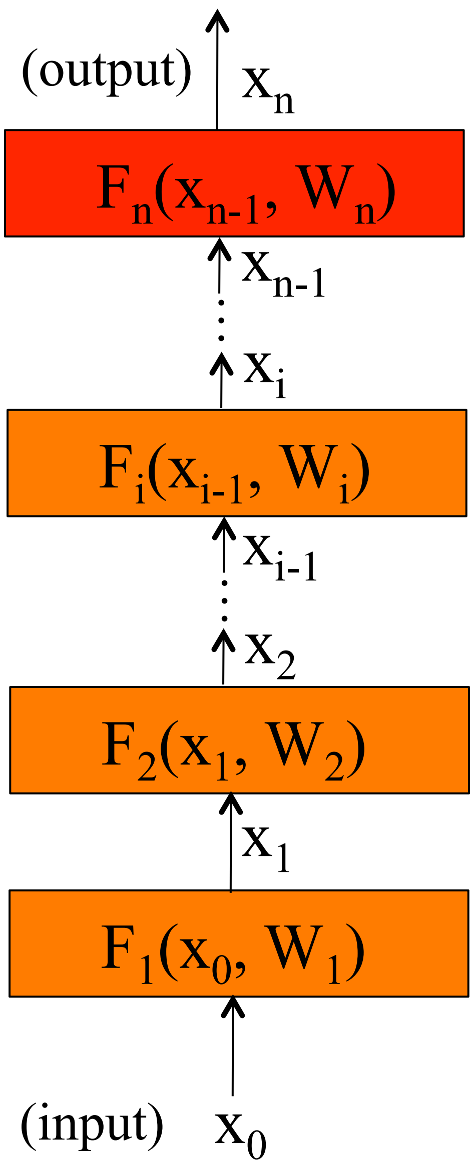
Training a model: overview

- Given a training dataset $\{x^m; y^m\}_{m=1,\ldots,M}$, pick appropriate cost function C.
- Forward-pass (f-prop) training examples through the model to get network output.
- Get error using cost function C to compare outputs to targets y^m
- Use Stochastic Gradient Descent (SGD) to update weights adjusting parameters to minimize loss/energy E (sum of the costs for each training example)

Cost function

- Consider model with *n* layers. Layer i has weights W_i.
- Forward pass: takes input x and passes it through each layer F_i:

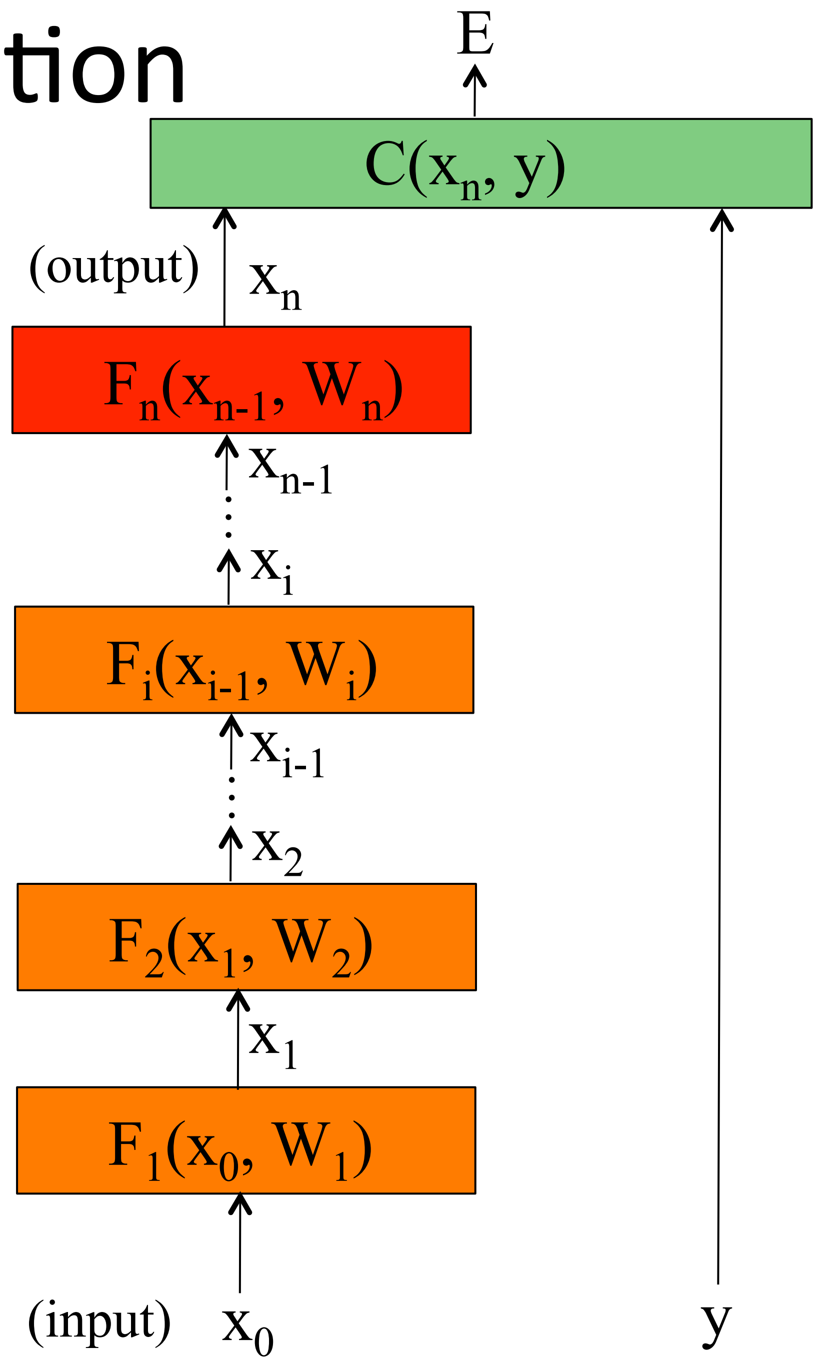
  $x_i = F_i\ (x_{i-1}, W_i)$
- Output of layer i is x_i. Network output (top layer) is x_n.



Cost function

- Consider model with *n* layers. Layer i has weights W_i.
- Forward pass: takes input x and passes it through each layer F_i:
  $x_i = F_i(x_{i-1}, W_i)$
- Output of layer i is x_i. Network output (top layer) is x_n.
- Cost function C compares x_n to y
- Overall energy is the sum of the cost over all training examples:

$$E = \sum_{m=1}^{M} C\left(x_n^m, y^m\right)$$



Stochastic gradient descend

- Want to minimize overall loss function **E.** Loss is sum of individual losses over each example.
- In gradient descent, we start with some initial set of parameters θ
- Update parameters: $\theta^{k+1} \leftarrow \theta^k + \eta \nabla \theta$

  k is iteration index, *η* is learning rate (negative scalar; set semi-manually).
- Gradients $\nabla \theta = \frac{\partial E}{\partial \theta}$ computed by *backpropagation*.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.

  If batchsize=1 then θ is updated after each example.

  If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).

Stochastic gradient descend

- We need to compute gradients of the cost with respect to model parameters w_i

- Back-propagation is essentially chain rule of derivatives back through the model.

- Each layer is differentiable with respect to parameters and input.

Computing gradients

- Training will be an iterative procedure, and at each iteration we will update the network parameters $\theta^{k+1} \leftarrow \theta^{k} + \eta \nabla \theta$

- We want to compute the gradients

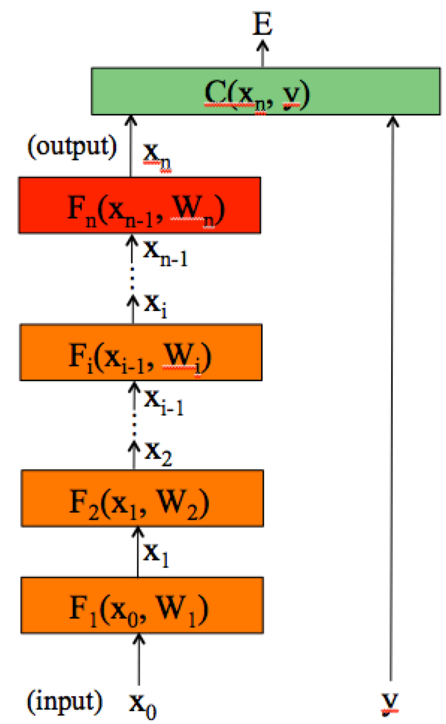
$$\nabla \theta = \frac{\partial E}{\partial \theta}$$

Where $\theta = \left\{ w_1, w_2, \ldots, w_n \right\}$

Computing gradients

To compute the gradients, we could start by wring the full energy E as a function of the network parameters.

$$E(\theta)=\sum_{m=1}^{M} C\left(F_n\left(F_{n-1}\left(F_2\left(F_1\left(x_0^m,w_1\right),w_2\right),w_{n-1}\right),w_n\right),y^m\right)$$



And then compute the partial derivatives… instead, we can use the chain rule to derive a compact algorithm: **back-propagation**

(

Matrix calculus

- x column vector of size [n×1]

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

- We now define a function on vector x: y = F(x)
- If y is a scalar, then

$$\partial y / \partial x = \begin{bmatrix} \partial y / \partial x_1 & \partial y / \partial x_2 & \cdots & \partial y / \partial x_n \end{bmatrix}$$

The derivative of y is a row vector of size [1×n]

- If y is a vector [1×m], then (*Jacobian formulation*):

$$\partial y / \partial x = \begin{bmatrix} \partial y_1 / \partial x_1 & \partial y_1 / \partial x_2 & \cdots & \partial y_1 / \partial x_n \\ \vdots & \vdots & & \vdots \\ \partial y_m / \partial x_1 & \partial y_m / \partial x_2 & \cdots & \partial y_n / \partial x_m \end{bmatrix}$$

The derivative of y is a matrix of size [m×n]

(m rows and n columns)

Matrix calculus

- If y is a scalar and x is a matrix of size [n×m], then

$$\partial y / \partial X = \begin{bmatrix} \partial y / \partial x_{11} & \partial y / \partial x_{21} & \cdots & \partial y / \partial x_{n1} \\ \vdots & \vdots & & \vdots \\ \partial y / \partial x_{1m} & \partial y / \partial x_{12} & \cdots & \partial y / \partial x_{nm} \end{bmatrix}$$

The output is a matrix of size [m×n]

Matrix calculus

- Chain rule:

For the function: z = h(x) = f (g(x))

Its derivative is: h’(x) = f’ (g(x)) g’(x)

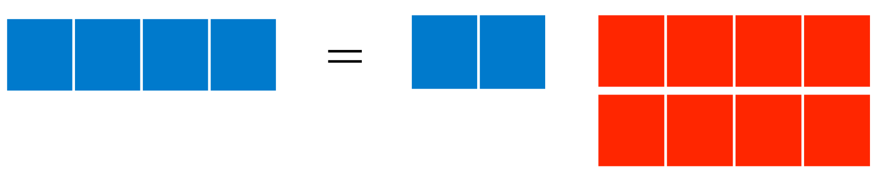
and writing z=f(u), and u=g(x):

$$\left.\frac{dz}{dx}\right|_{x=a} = \left.\frac{dz}{du}\right|_{u=g(a)} \cdot \left.\frac{du}{dx}\right|_{x=a}$$

[m×n] [m×p] [p×n]

with p = length vector u = |u|, m = |z|, and n = |x|

Example, if |z|=1, |u| = 2, |x|=4

h’(x) = 

Matrix calculus

- Chain rule:

For the function: $h(x) = f_n(f_{n-1}(\ldots(f_1(x))\,))$

With $u_1 = f_1(x)$
$u_i = f_i(u_{i-1})$
$z = u_n = f_n(u_{n-1})$

The derivative becomes a product of matrices:

$$\left.\frac{dz}{dx}\right|_{x=a} = \left.\frac{dz}{du_{n-1}}\right|_{u_{n-1}=f_{n-1}(u_{n-2})} \cdot \left.\frac{du_{n-1}}{du_{n-2}}\right|_{u_{n-2}=f_{n-2}(u_{n-3})} \cdot \ldots \cdot \left.\frac{du_2}{du_1}\right|_{u_1=f_1(a)} \cdot \left.\frac{du_1}{dx}\right|_{x=a}$$

(exercise: check that all the matrix dimensions work fine)

)

Computing gradients

The energy E is the sum of the costs associated to each training example x^m, y^m

$$E(\theta) = \sum_{m=1}^{M} C\left(x_n^m, y^m; \theta\right)$$

Its gradient with respect to the networks parameters is:

$$\frac{\partial E}{\partial \theta_i} = \sum_{m=1}^{M} \frac{C\left(x_n^m, y^m; \theta\right)}{\partial \theta_i}$$

is how much E varies when the parameter θ_i is varied.

Computing gradients

We could write the cost function to get the gradients:

$$C(x_n, y; \theta) = C(F_n(x_{n-1}, w_n), y)$$

$$\text{with} \quad \theta = [w_1, w_2, \cdots, w_n]$$

If we compute the gradient with respect to the parameters of the last layer (output layer) w_n, using the chain rule:

$$\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n(x_{n-1}, w_n)}{\partial w_n}$$

(how much the cost changes when we change w_n: is the product between how much the cost changes when we change the output of the last layer and how much the output changes when we change the layer parameters.)

Computing gradients: cost layer

If we compute the gradient with respect to the parameters of the last layer (output layer) w_n, using the chain rule:

$$\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n\left(x_{n-1}, w_n\right)}{\partial w_n}$$

Will depend on the layer structure and non-linearity.

For example, for an Euclidean loss:

$$C(x_n, y) = \frac{1}{2}\left\| x_n - y \right\|^2$$

The gradient is:

$$\frac{\partial C}{\partial x_n} = x_n - y$$

Computing gradients: layer i

We could write the full cost function to get the gradients:

$$C(x_n, y; \theta) = C\Big(F_n\Big(F_{n-1}\Big(F_2\Big(F_1\big(x_0, w_1\big), w_2\Big), w_{n-1}\Big), w_n\Big), y\Big)$$

If we compute the gradient with respect to w_i, using the chain rule:

$$\frac{\partial C}{\partial w_i} = \underbrace{\frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot \ldots \cdot \frac{\partial x_{i+1}}{\partial x_i}}_{\frac{\partial C}{\partial x_i}} \cdot \frac{\partial x_i}{\partial w_i}$$

$$\frac{\partial C}{\partial x_i}$$

And this can be computed iteratively!

$$\frac{\partial x_i}{\partial w_i} = \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

This is easy.

Backpropagation

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot \ldots \cdot \frac{\partial x_{i+1}}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i}$$

$$\frac{\partial C}{\partial x_i} \qquad \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

If we have the value of $\frac{\partial C}{\partial x_i}$ we can compute the gradient at the layer bellow as:

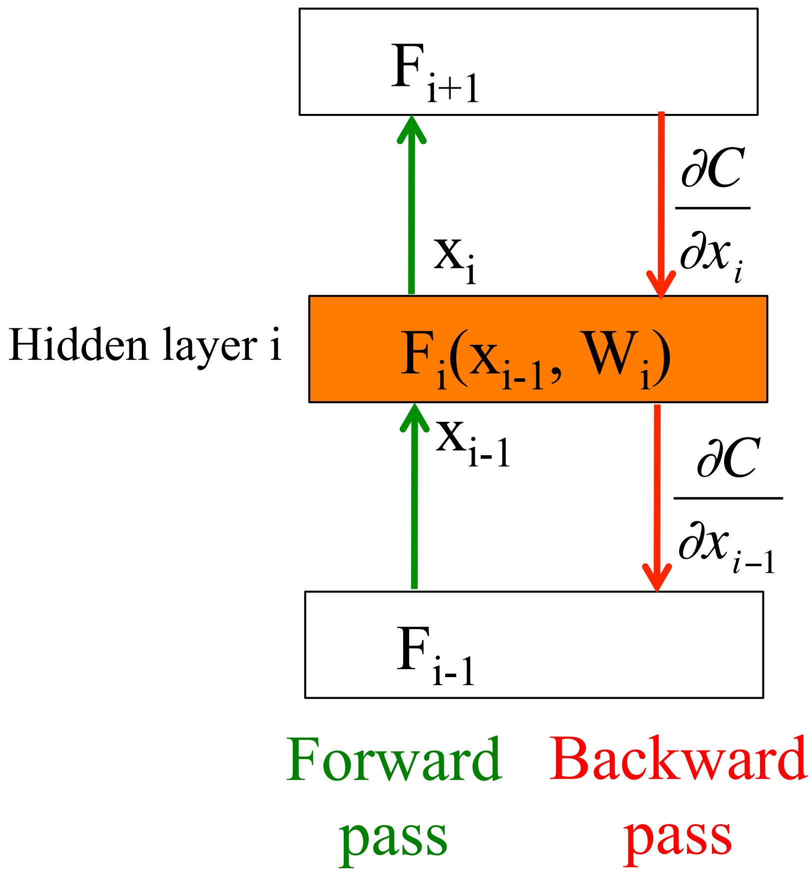
$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_{i-1}}$$

Gradient layer i-1

Gradient layer i

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

Backpropagation: layer i



- Layer i has two inputs (during training)

$$x_{i-1} \qquad \frac{\partial C}{\partial x_i}$$

- For layer i, we need the derivatives:

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}} \qquad \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

- We compute the outputs

$$x_i = F_i(x_{i-1}, w_i)$$

$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

- The weight update equation is:

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

$$w_i^{k+1} \leftarrow w_i^k + \eta_t \frac{\partial E}{\partial w_i}$$

(sum over all training examples to get E)

Backpropagation: summary

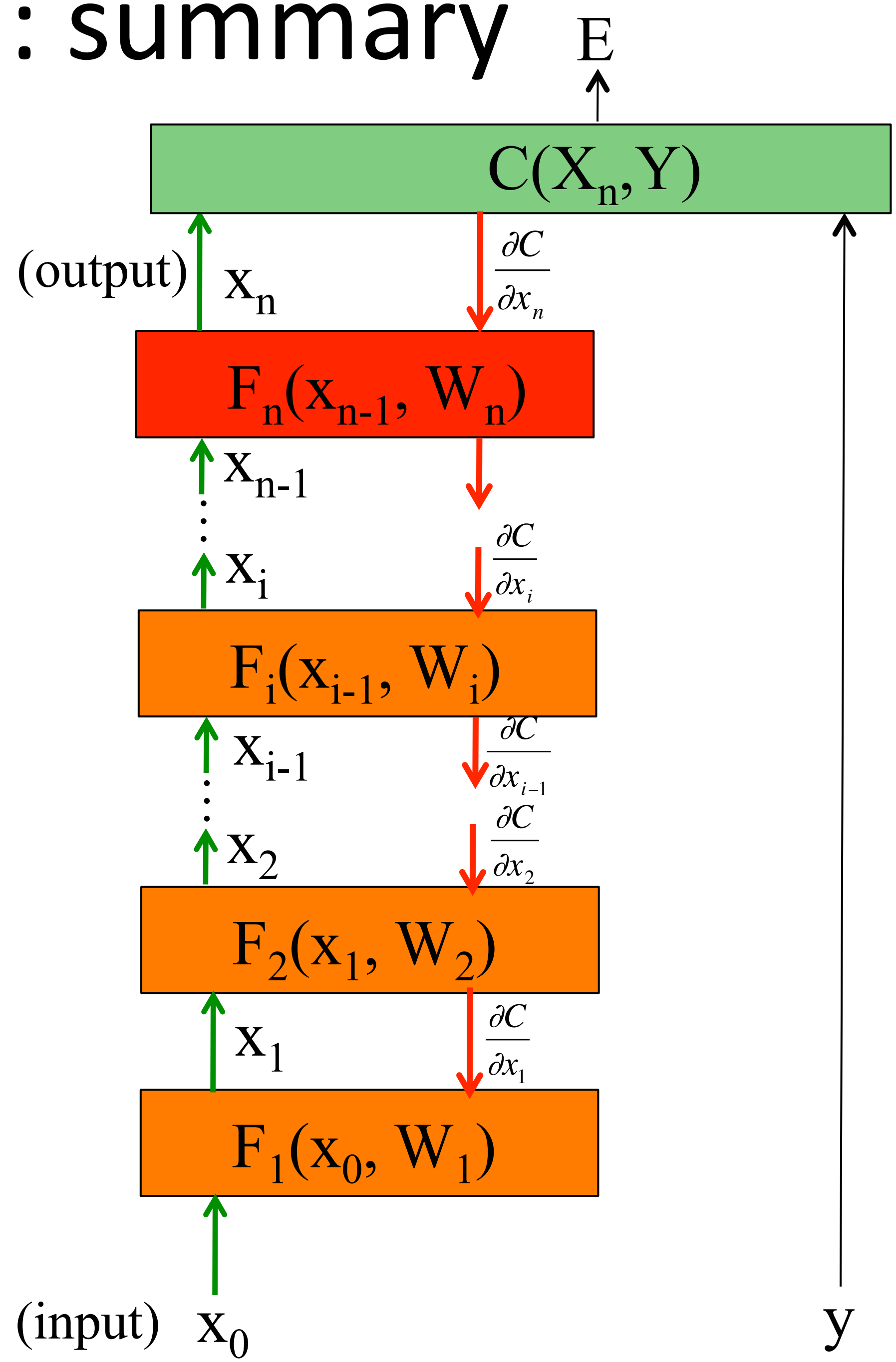
- Forward pass: for each training example. Compute the outputs for all layers

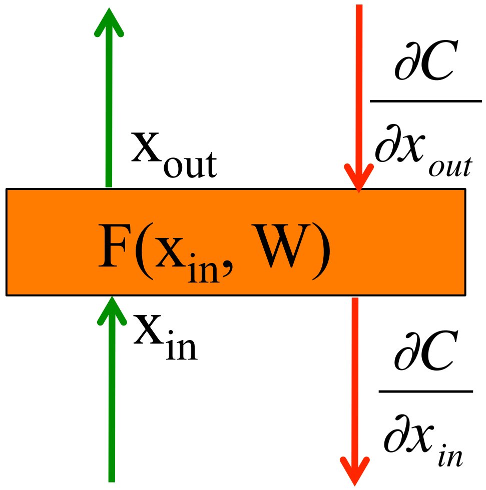
$$x_i = F_i(x_{i-1}, w_i)$$

- Backwards pass: compute cost derivatives iteratively from top to bottom:

$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

- Compute gradients and update weights.





Linear Module

- Forward propagation: $x_{out} = F(x_{in}, W) = Wx_{in}$

With W being a matrix of size $|x_{out}| \times |x_{in}|$

- Backprop to input:

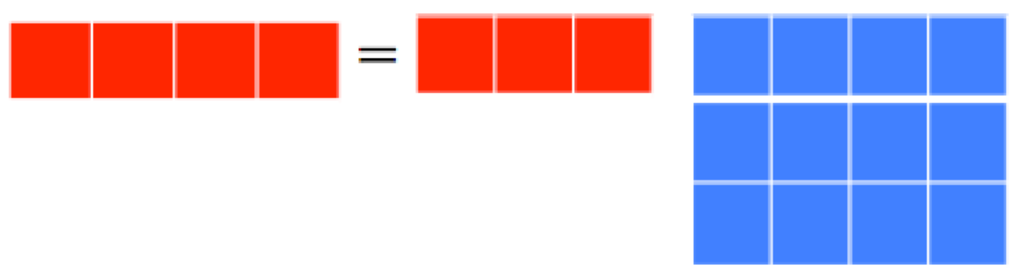
$$\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial x_{in}}$$

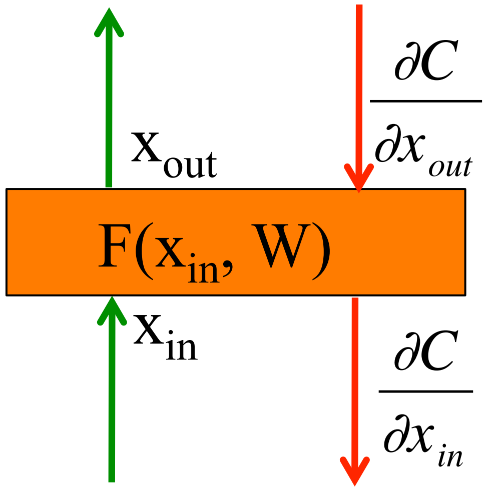
If we look at the j component of output x_{out}, with respect to the i component of the input, x_{in}:

$$\frac{\partial x_{out_i}}{\partial x_{in_j}} = W_{ij} \quad \rightarrow \quad \frac{\partial F(x_{in}, W)}{\partial x_{in}} = W$$

Therefore:

$$\boxed{\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot W}$$





Linear Module

- Forward propagation: $x_{out} = F(x_{in}, W) = W x_{in}$
- Backprop to weights:

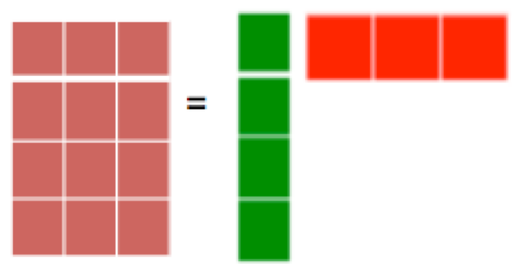
$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial W}$$

If we look at how the parameter W_{ij} changes the cost, only the i component of the output will change, therefore:

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{x_{out_i}}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot x_{in_j}$$

$$\frac{\partial x_{out_i}}{\partial W_{ij}} = x_{in_j}$$

$$\frac{\partial C}{\partial W} = x_{in} \cdot \frac{\partial C}{\partial x_{out}}$$

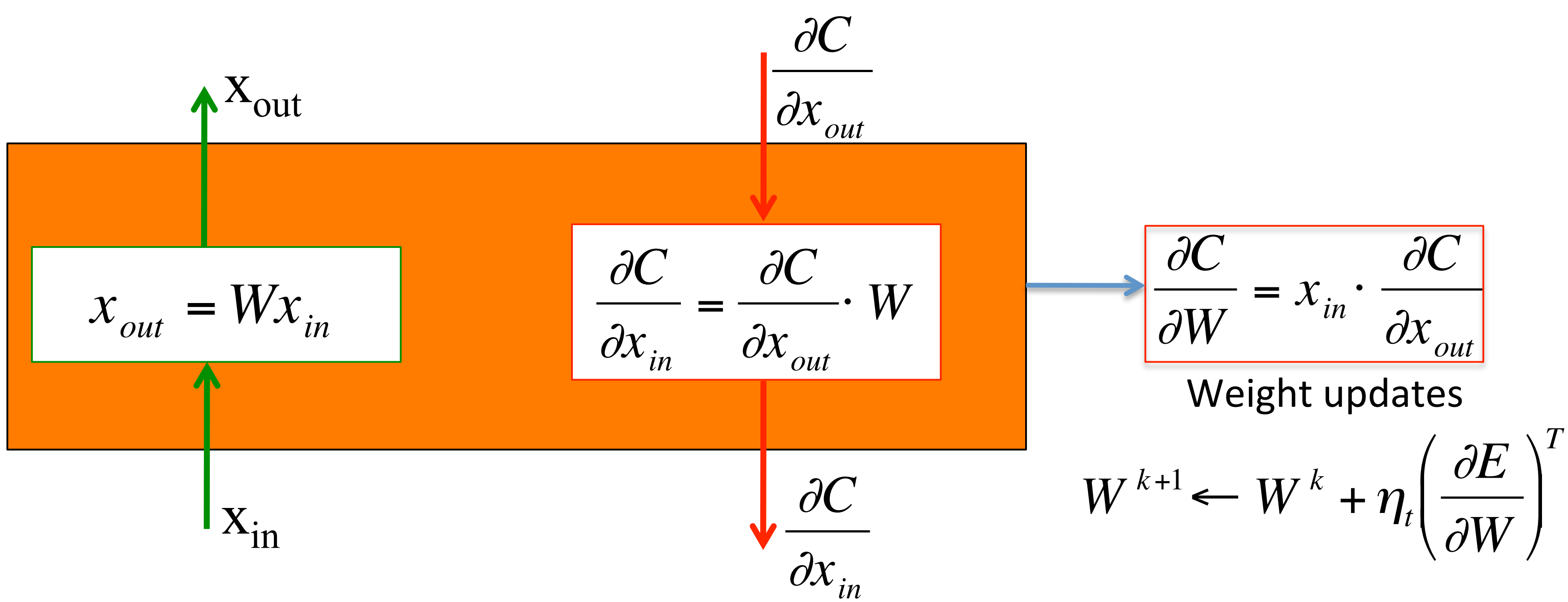


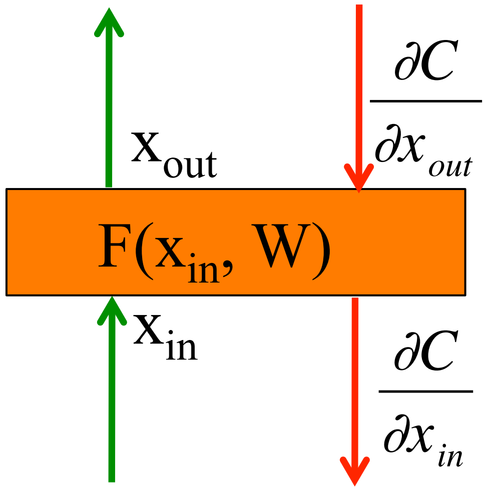
And now we can update the weights (by summing over all the training examples):

$$W_{ij}^{k+1} \leftarrow W_{ij}^{k} + \eta_t \frac{\partial E}{\partial W_{ij}}$$

(sum over all training examples to get E)

Linear Module





Pointwise function

- Forward propagation:

$$x_{out_i} = h(x_{in_i} + b_i)$$

h = an arbitrary function, b_i is a bias term.

- Backprop to input: $$\frac{\partial C}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i)$$

- Backprop to bias: $$\frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i)$$

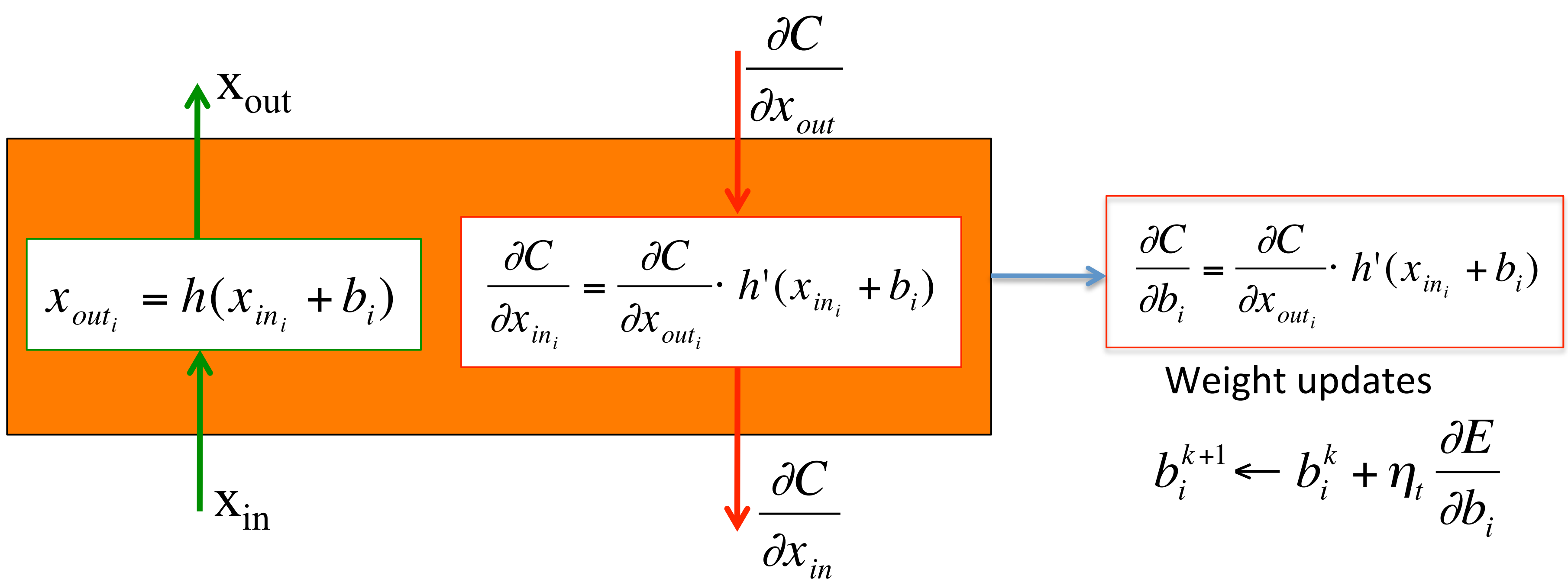
We use this last expression to update the bias.

Some useful derivatives:

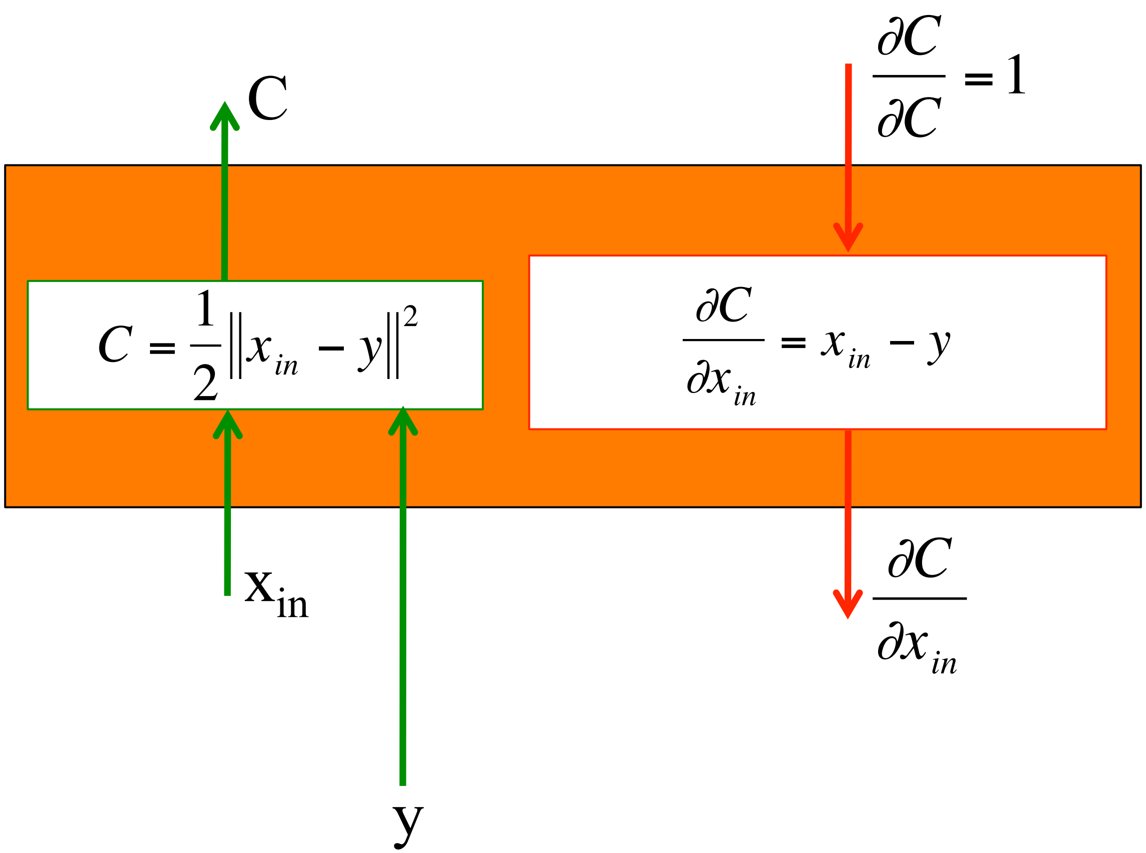
For hyperbolic tangent: $\tanh'(x) = 1 - \tanh^2(x)$

For ReLU: h(x) = max(0,x) h'(x) = 1 [x>0]

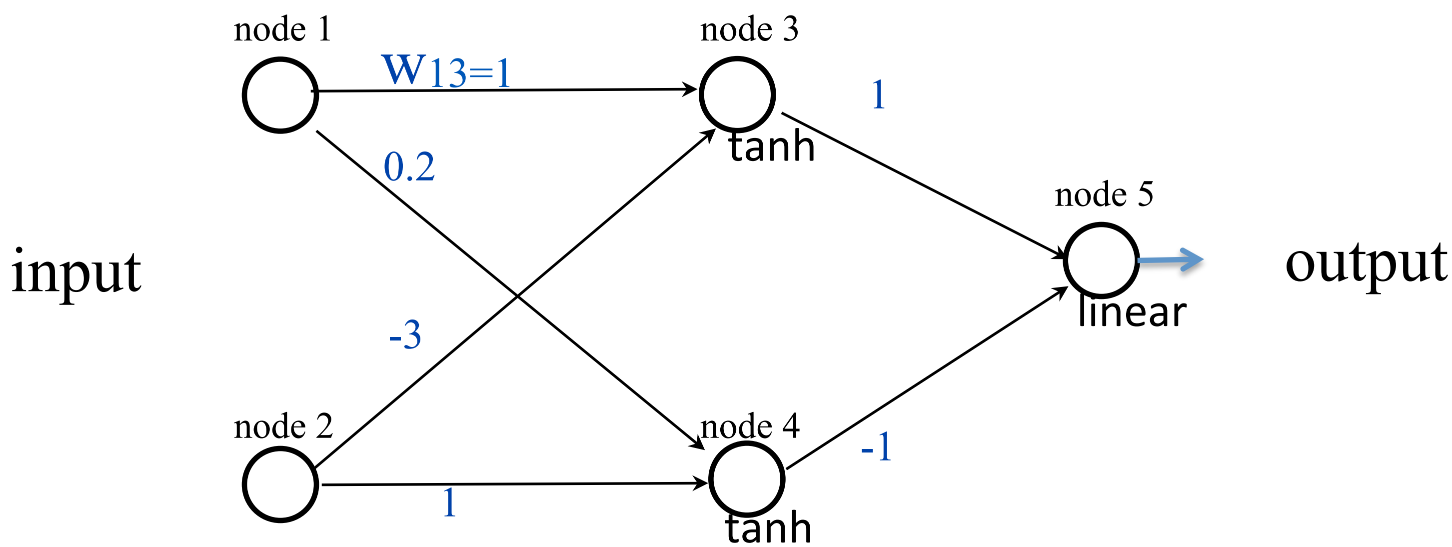
Pointwise function



Euclidean cost module



Back propagation example



Learning rate = -0.2 (because we used positive increments)

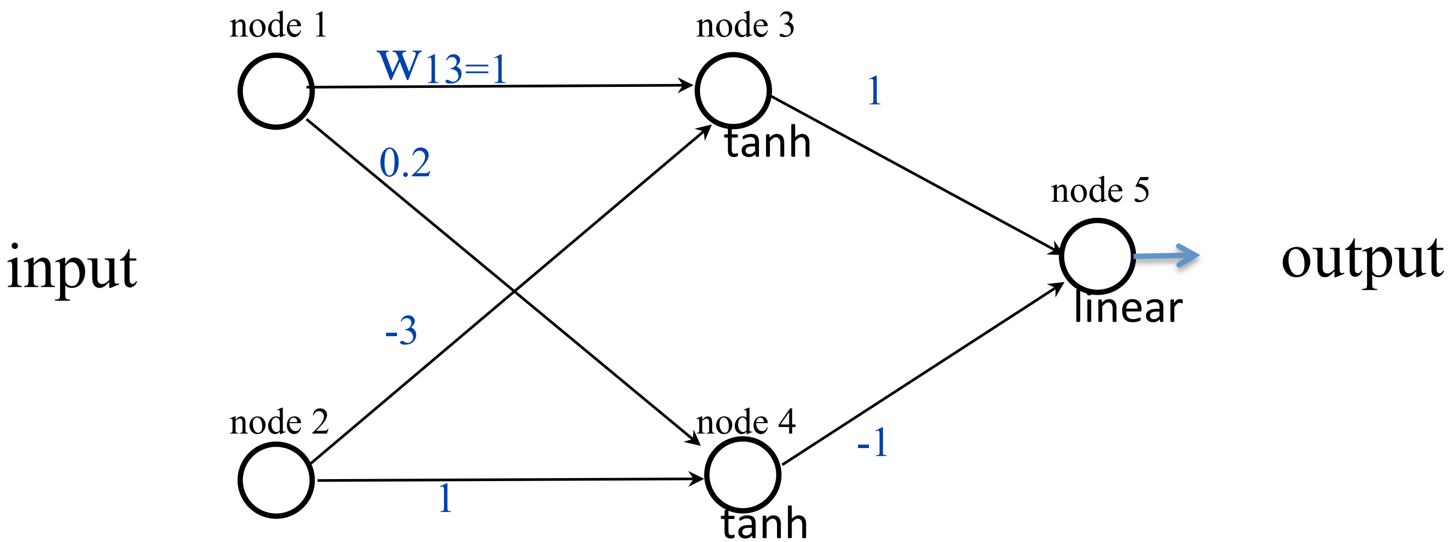
Euclidean loss

Training data:

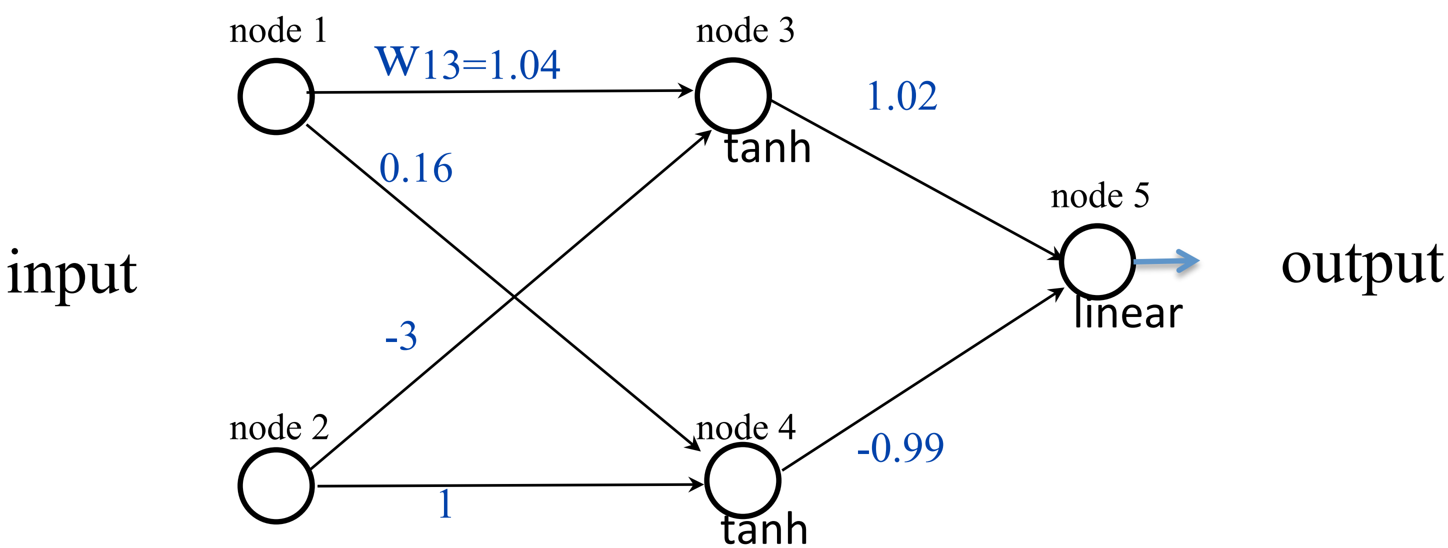
input		desired output
node 1	node 2	node 5
1.0	0.1	0.5

Exercise: run one iteration of back propagation

Back propagation example

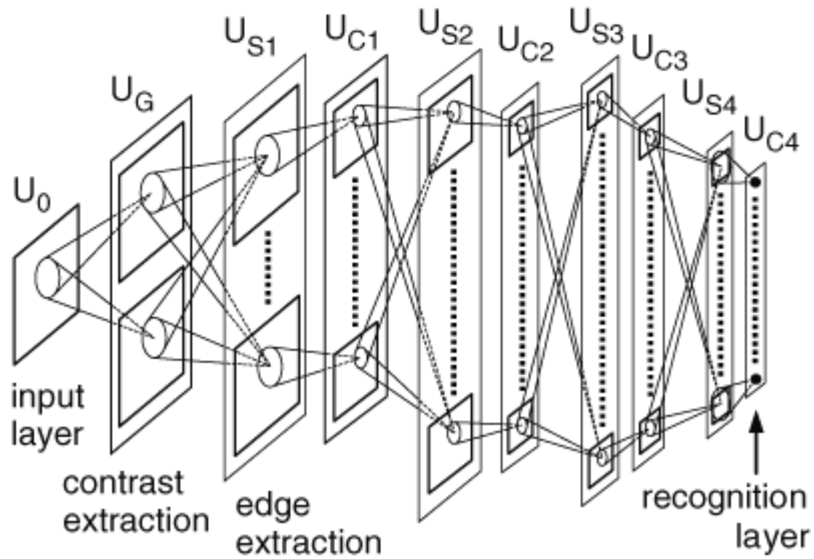


After one iteration (rounding to two digits):



Neocognitron

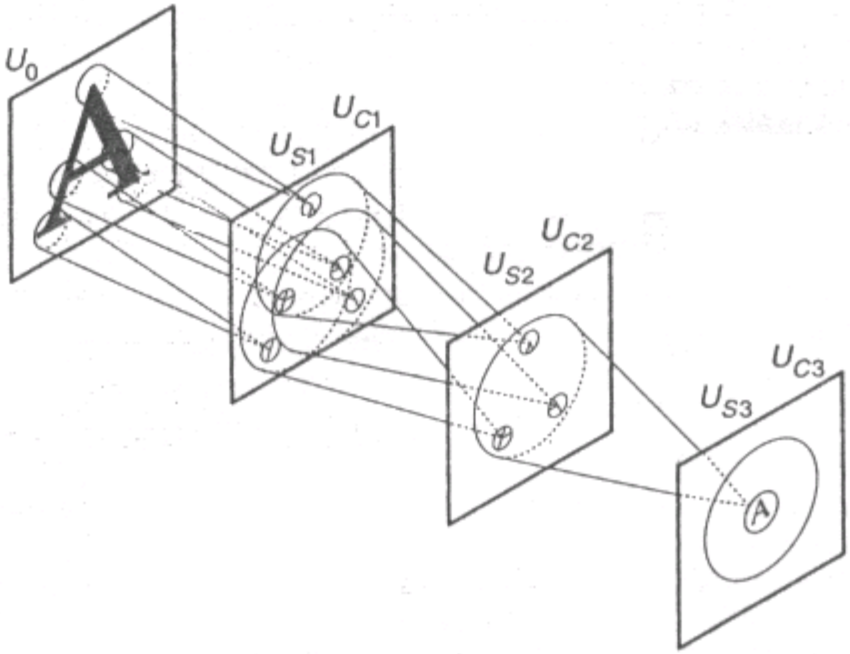
Fukushima (1980). Hierarchical multilayered neural network

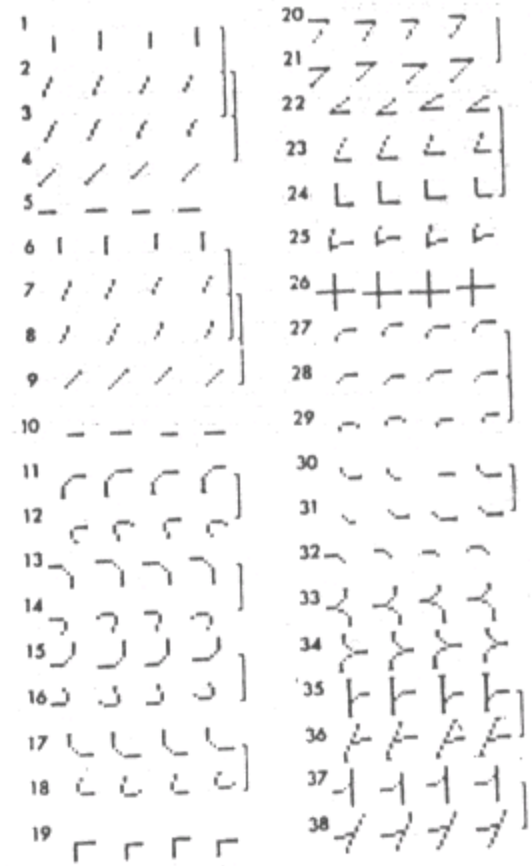


S-cells work as feature-extracting cells. They resemble simple cells of the primary visual cortex in their response.

C-cells, which resembles complex cells in the visual cortex, are inserted in the network to allow for positional errors in the features of the stimulus. The input connections of C-cells, which come from S-cells of the preceding layer, are fixed and invariable. Each C-cell receives excitatory input connections from a group of S-cells that extract the same feature, but from slightly different positions. The C-cell responds if at least one of these S-cells yield an output.

Neocognitron

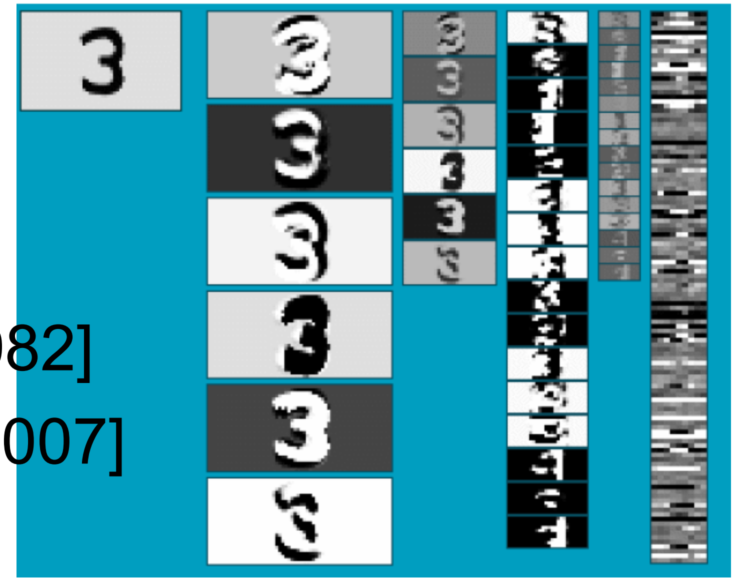




Learning is done greedily for each layer

Multistage Hubel-Wiesel Architecture

- Stack multiple stages of simple cells / complex cells layers
- Higher stages compute more global, more invariant features
- Classification layer on top



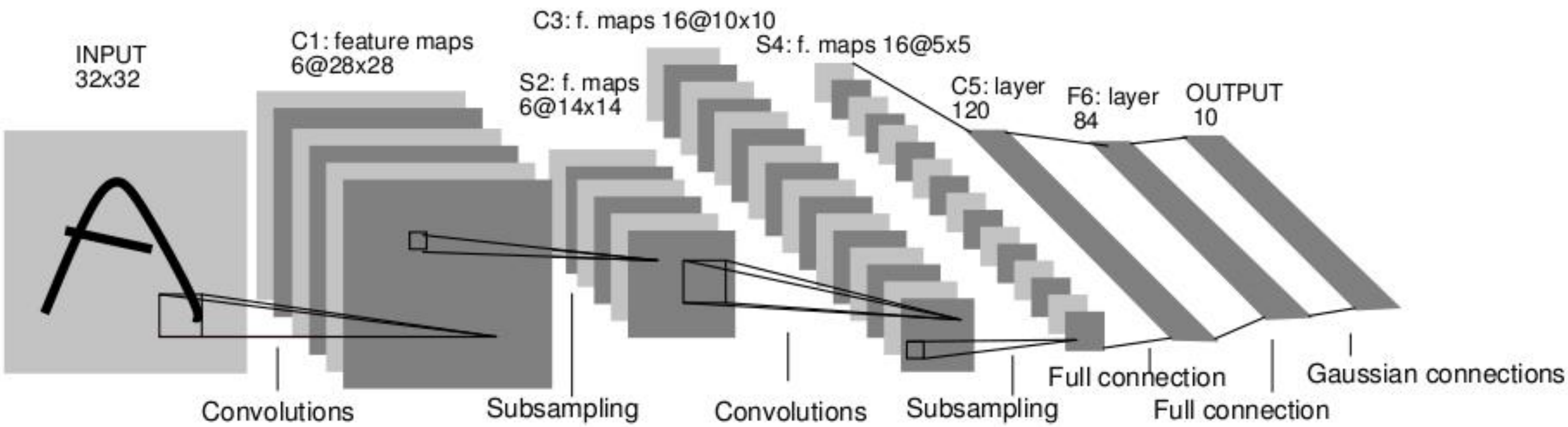
History:

- Neocognitron [Fukushima 1971-1982]
- Convolutional Nets [LeCun 1988-2007]
- HMAX [Poggio 2002-2006]
- Many others….

Slide: Y.LeCun

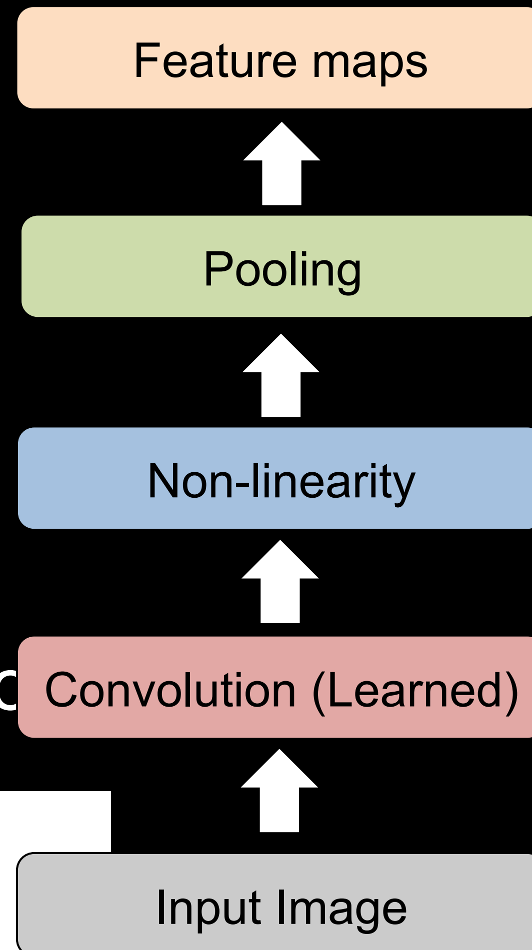
Convolutional Neural Networks

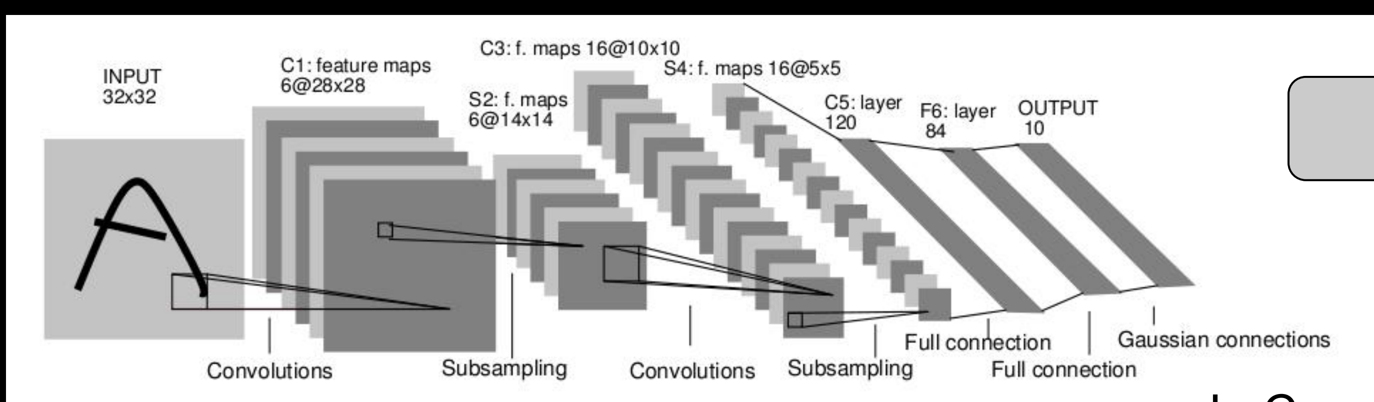
- LeCun et al. 1989
- Neural network with specialized connectivity structure



Overview of Convnets

- Feed-forward:
  - Convolve input
  - Non-linearity (rectified linear)
  - Pooling (local max)
- Supervised
- Train convolutional filters by back-propagating classification erro



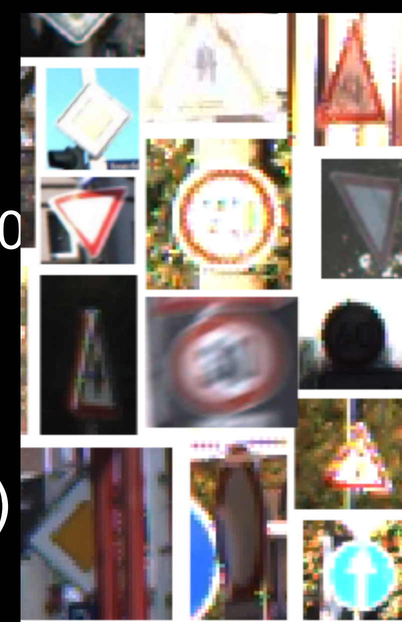


LeCun e

Convnet Successes

- Handwritten text/digits
  - MNIST (0.17% error [Ciresan et al. 2011])
  - Arabic & Chinese [Ciresan et al. 2012]

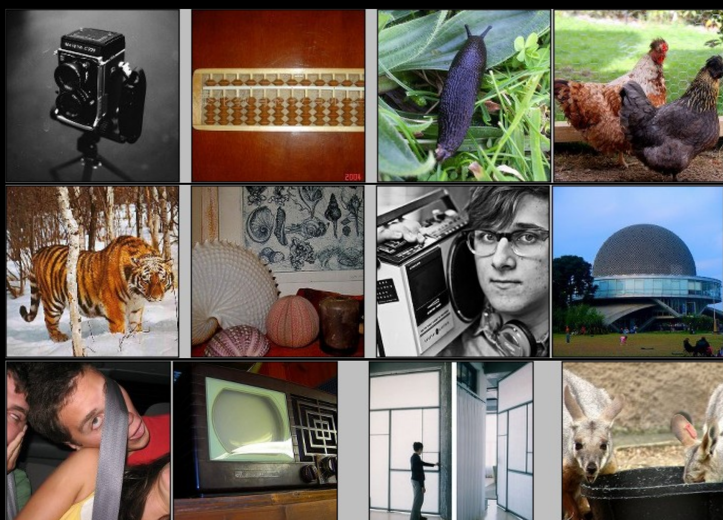
- Simpler recognition benchmarks
  - CIFAR-10 (9.3% error [Wan et al. 2013])
  - Traffic sign recognition
    - 0.56% error vs 1.16% for humans [Ciresan et al. 20



- But less good at more complex datasets
  - E.g. Caltech-101/256 (few training examples)

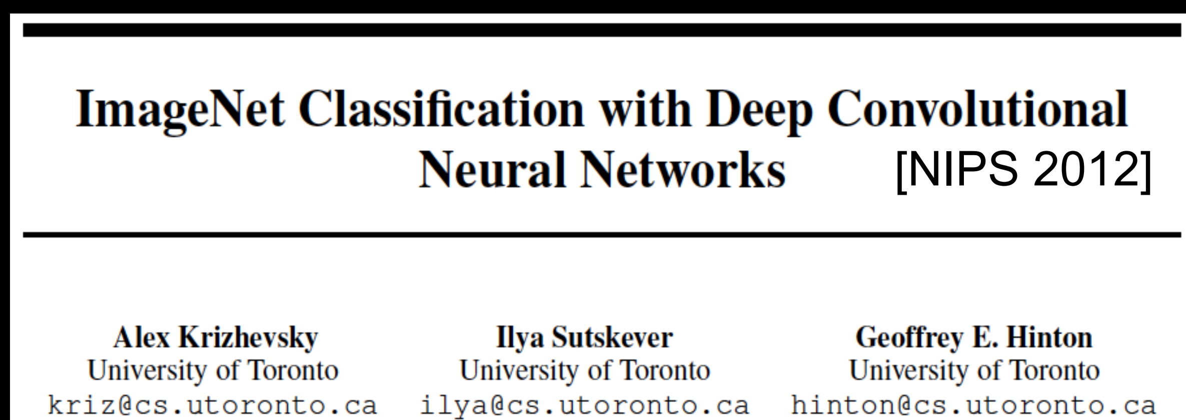
Application to ImageNet

IMAGENET



[Deng et al. CVPR 2009]

- ~14 million labeled images, 20k class
- Images gathered from Internet
- Human labels via Amazon Turk



ImageNet Classification with Deep Convolutional Neural Networks [NIPS 2012]

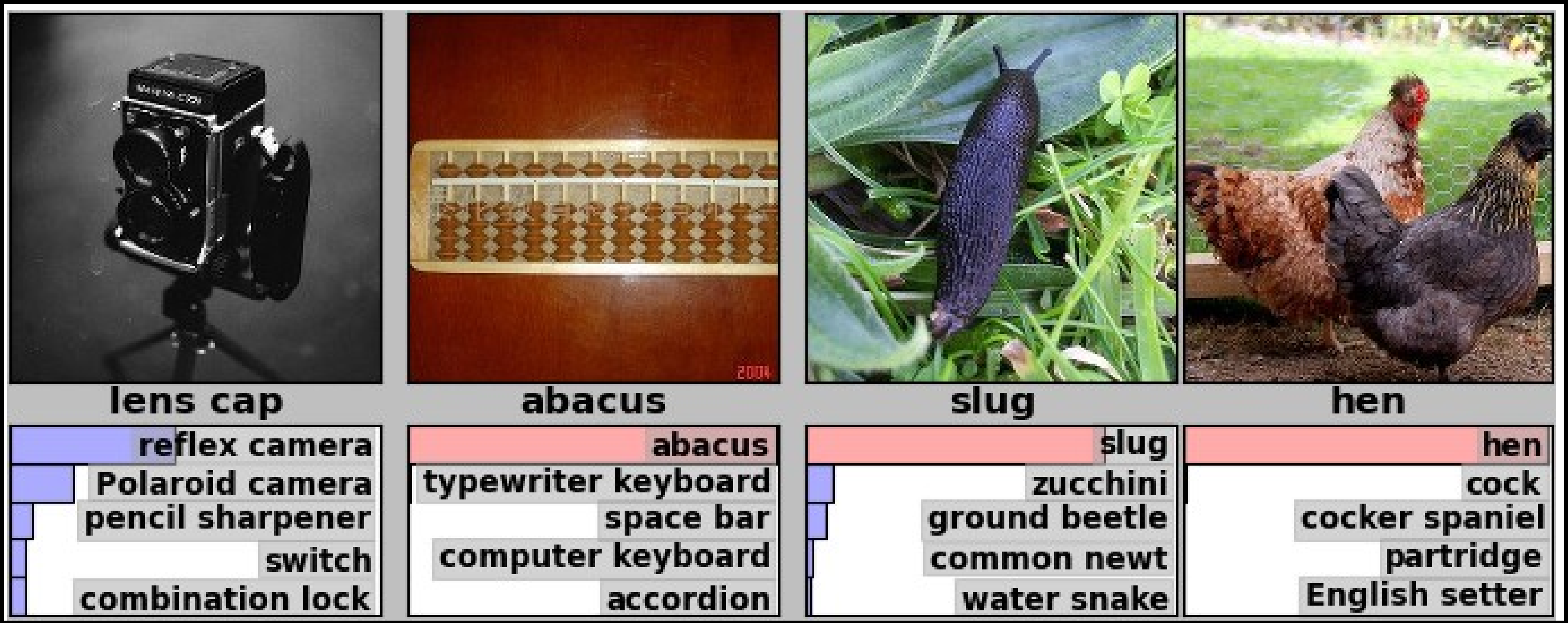
Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Goal

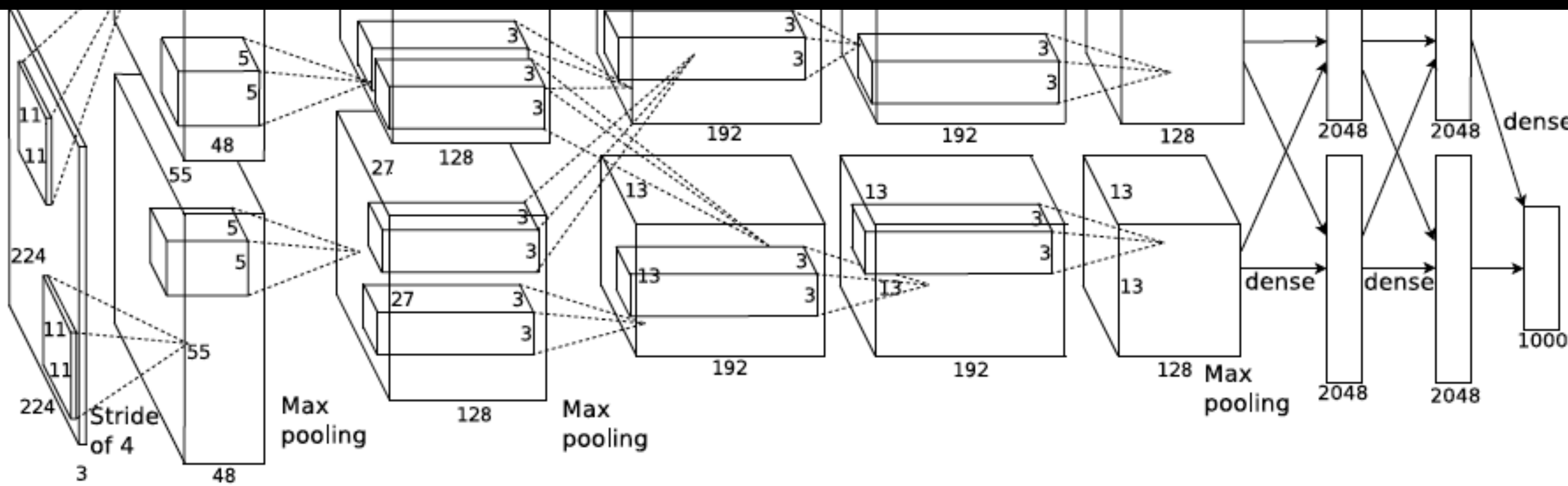
- Image Recognition
  - Pixels → Class Label



[Krizhevsky et al. NIPS 2012]

Krizhevsky et al. [NIPS2012]

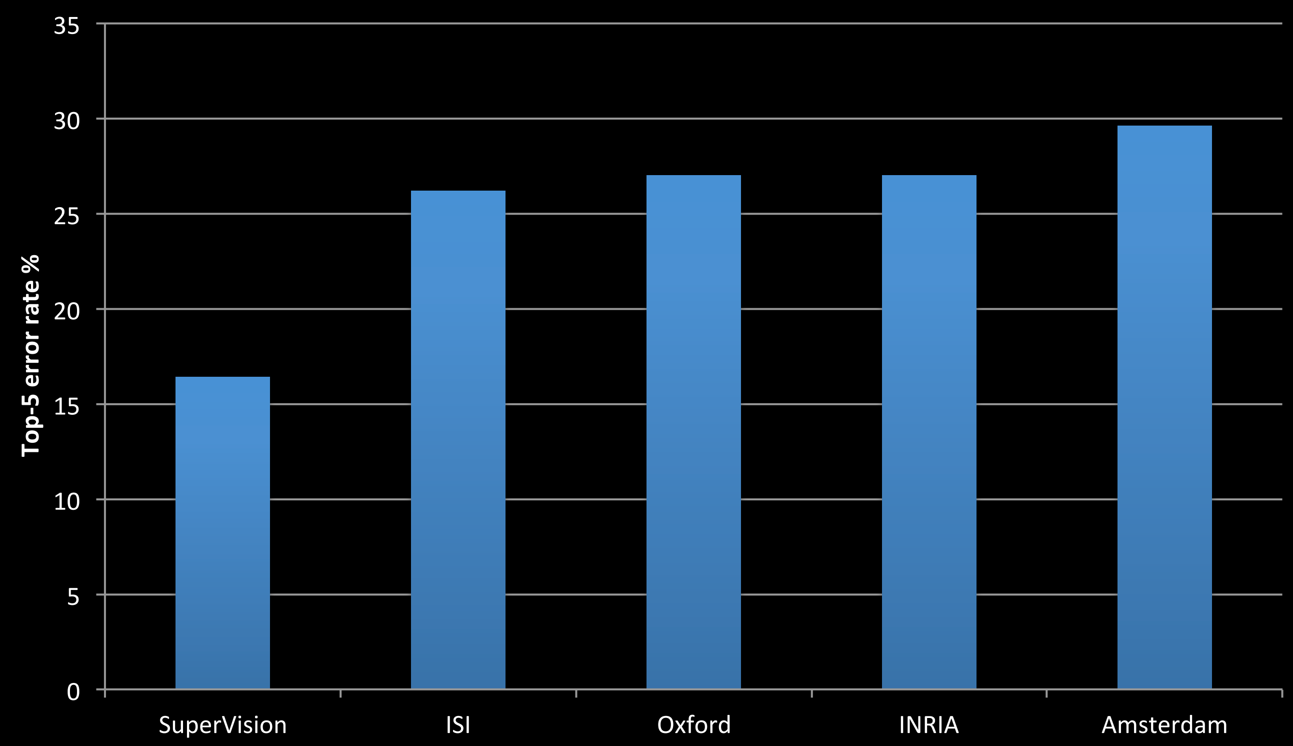
- Same model as LeCun'98 but:
  - Bigger model (8 layers)
  - More data (10^6 vs 10^3 images)
  - GPU implementation (50x speedup over CPU)
  - Better regularization (DropOut)



- 7 hidden layers, 650,000 neurons, 60,000,000 parameters
- Trained on 2 GPUs for a week

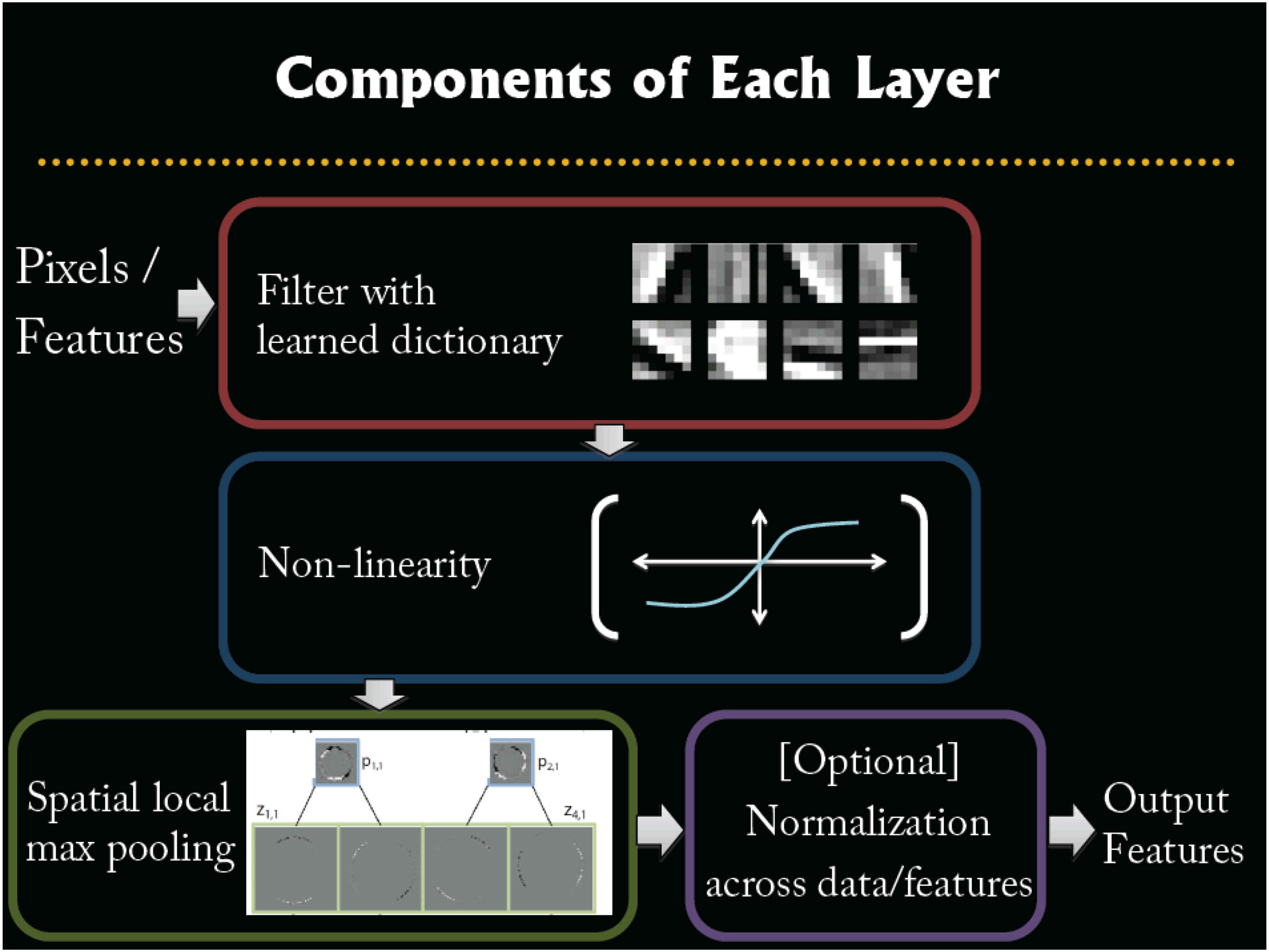
ImageNet Classification 2012

- Krizhevsky et al. -- 16.4% error (top-5)
- Next best (non-convnet) – 26.2% error



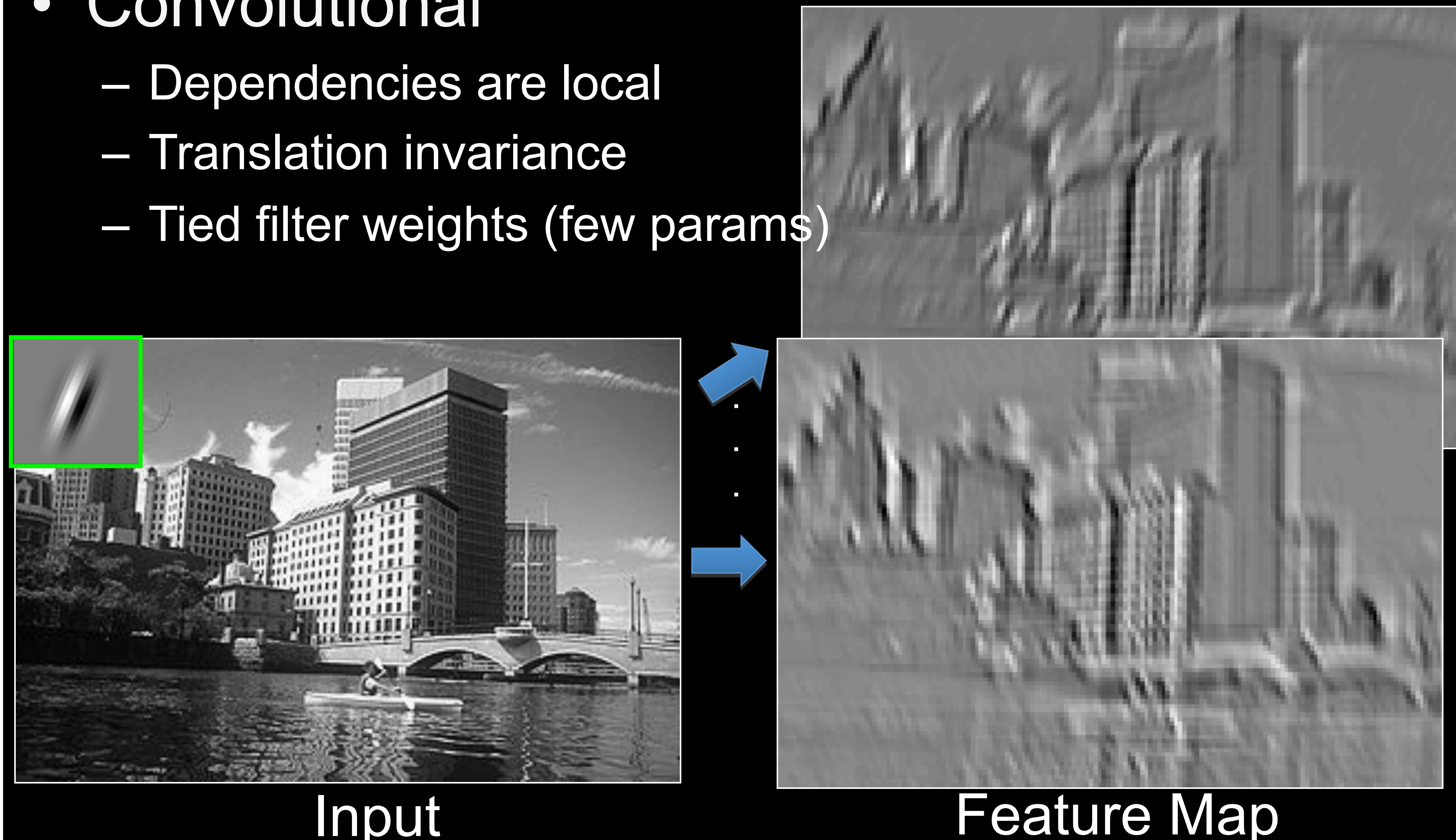
How convnets work

- Operations in each layer
- Architecture
- Training
- Results


Components of Each Layer
Pixels / Features
Filter with learned dictionary
Non-linearity
Spatial local max pooling
$p_{1,1}$
$p_{2,1}$
$z_{1,1}$
$z_{4,1}$
[Optional] Normalization across data/features
Output Features

Filtering

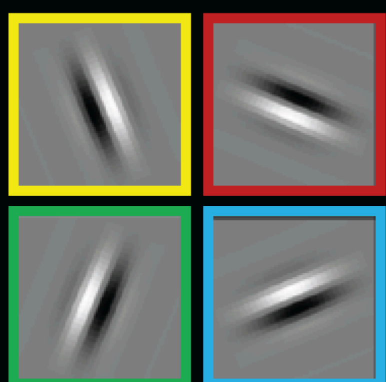
- Convolutional
  - Dependencies are local
  - Translation invariance
  - Tied filter weights (few params)



Filtering

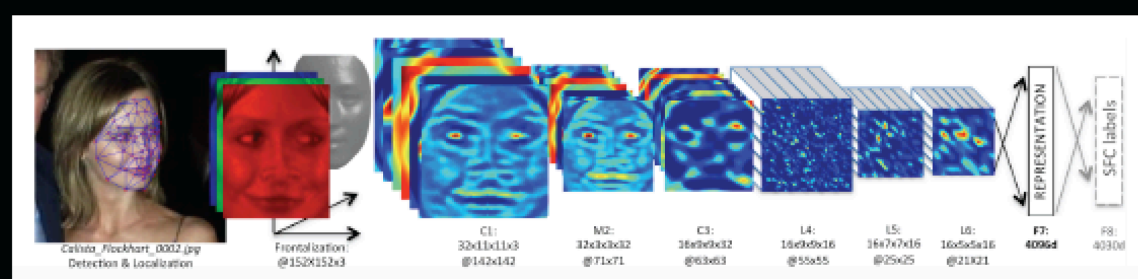
- Local
  - Each unit layer above look at local window
  - But no weight tying

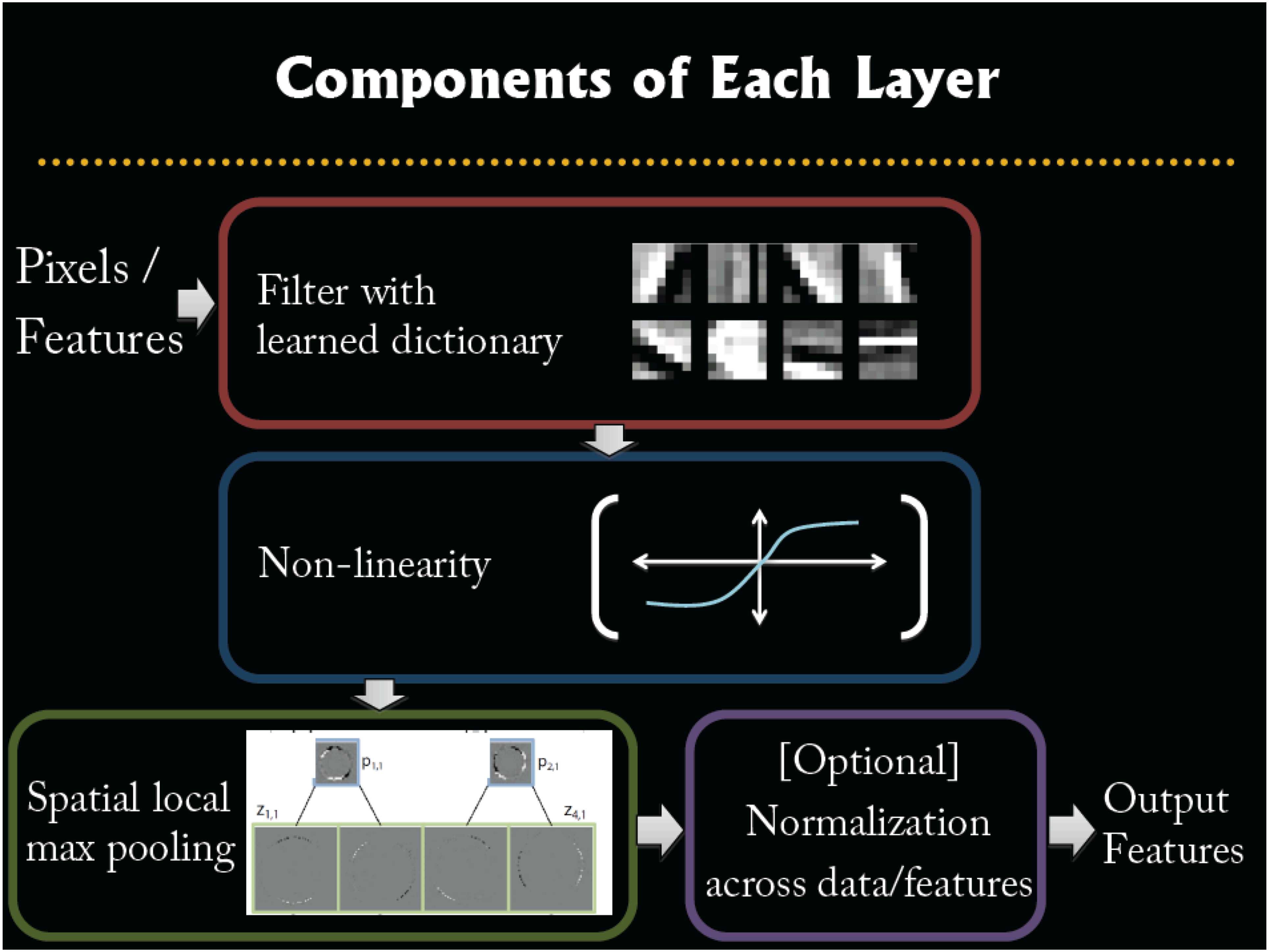
Input



Filters

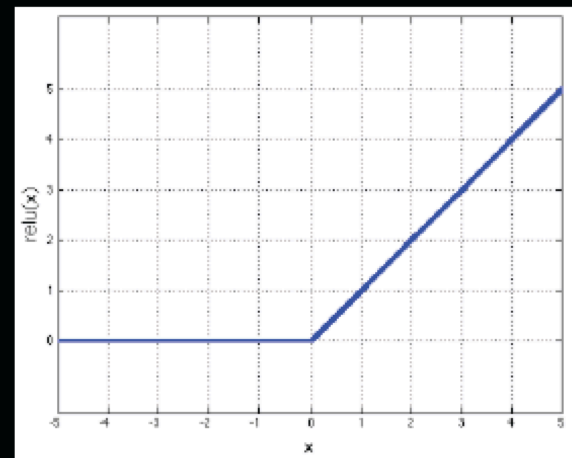
- E.g. face recognition



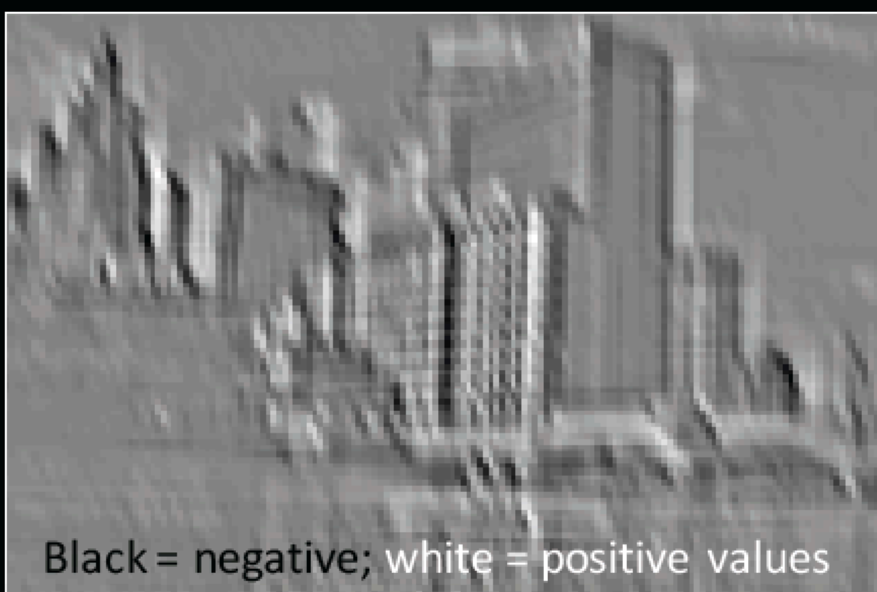

Components of Each Layer
Pixels / Features
Filter with learned dictionary
Non-linearity
Spatial local max pooling
p1,1
p2,1
z1,1
z4,1
[Optional] Normalization across data/features
Output Features

Non-Linearity

- Rectified linear function
  - Applied per-pixel
  - output = max(0,input)



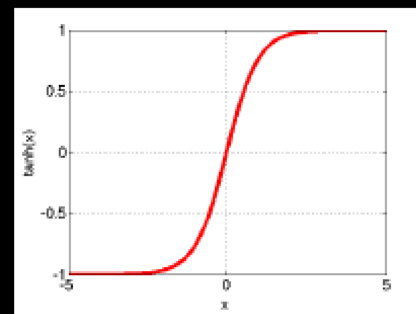
Input feature map

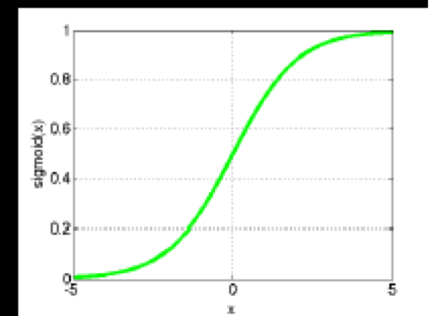


Output feature map

Non-Linearity

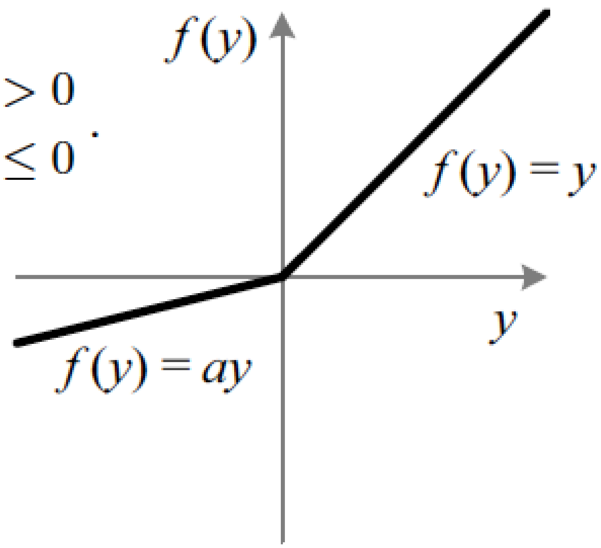
- Other choices:
  - Tanh
  - Sigmoid: 1/(1+exp(-x))
  - PReLU





[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Kaiming He et al. arXiv:1502.01852v1.pdf, Feb 2015]

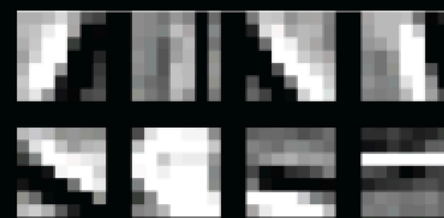
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}.$$



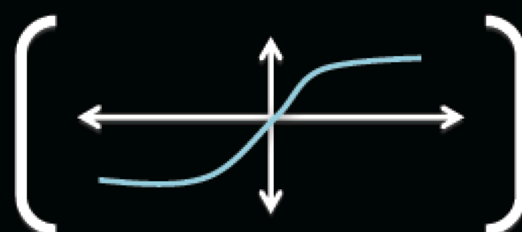
Components of Each Layer

Pixels / Features

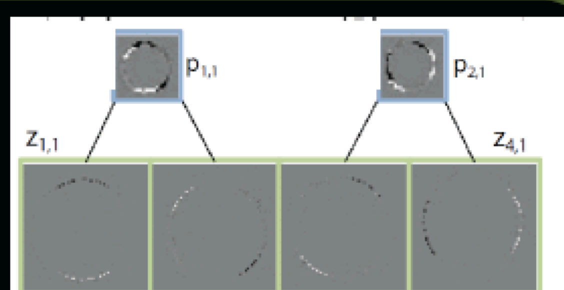
Filter with learned dictionary



Non-linearity



Spatial local max pooling

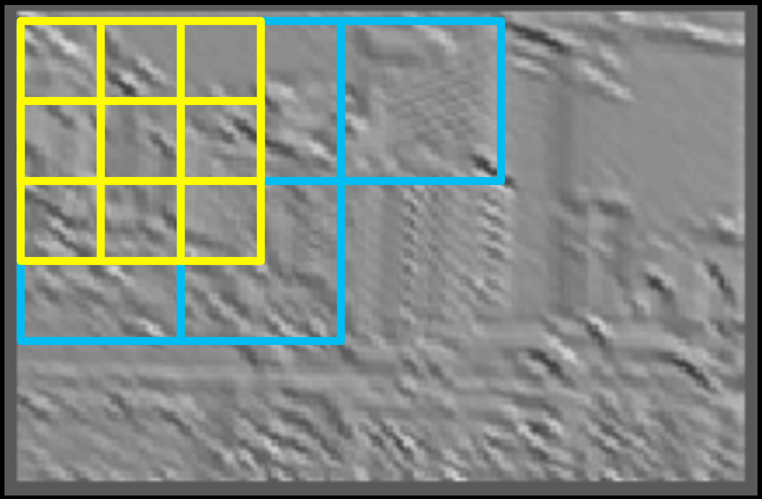


[Optional] Normalization across data/features

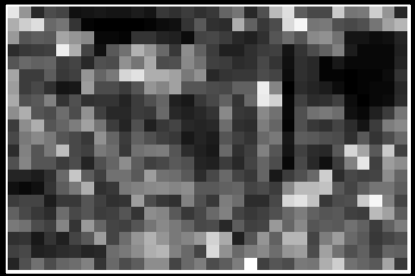
Output Features

Pooling

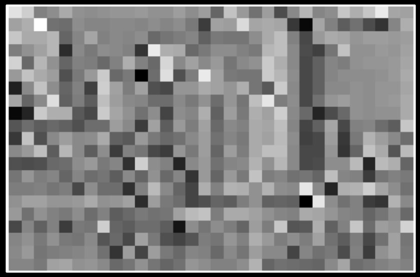
- Spatial Pooling
  - Non-overlapping / overlapping regions
  - Sum or max
  - Boureau et al. ICML'10 for theoretical analysis



Max

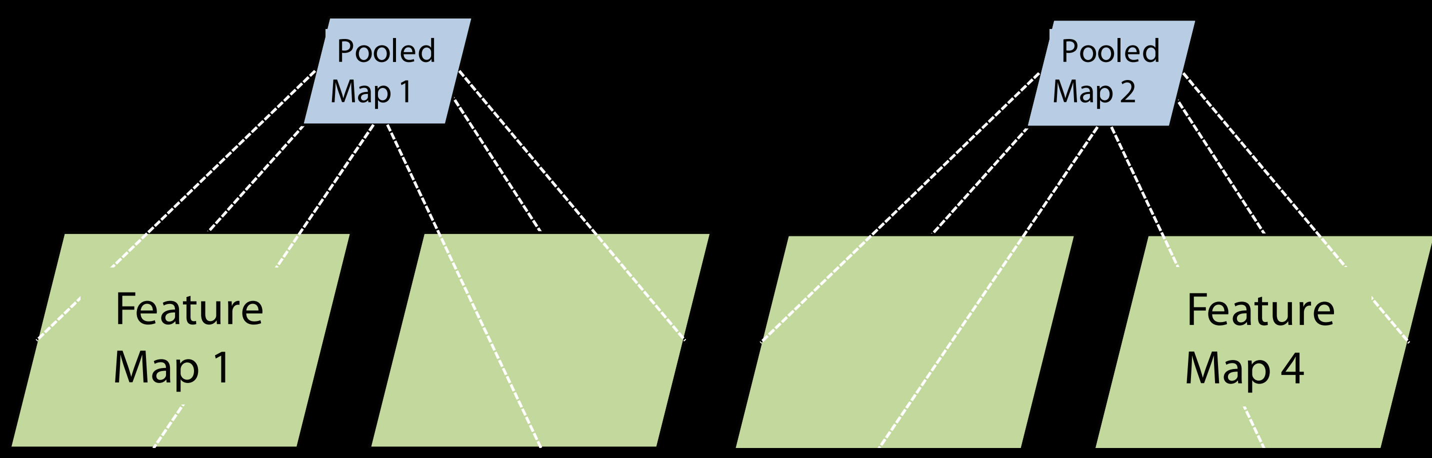


Sum



Pooling

- Pooling across feature groups
  - Additional form of inter-feature competition
  - MaxOut Networks [Goodfellow et al. ICML 2013]

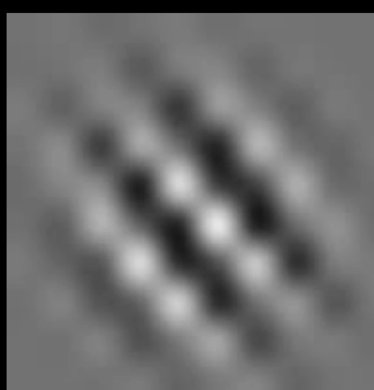
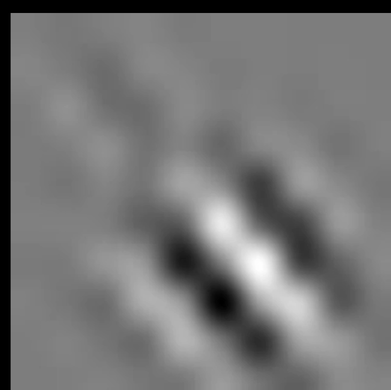


Role of Pooling

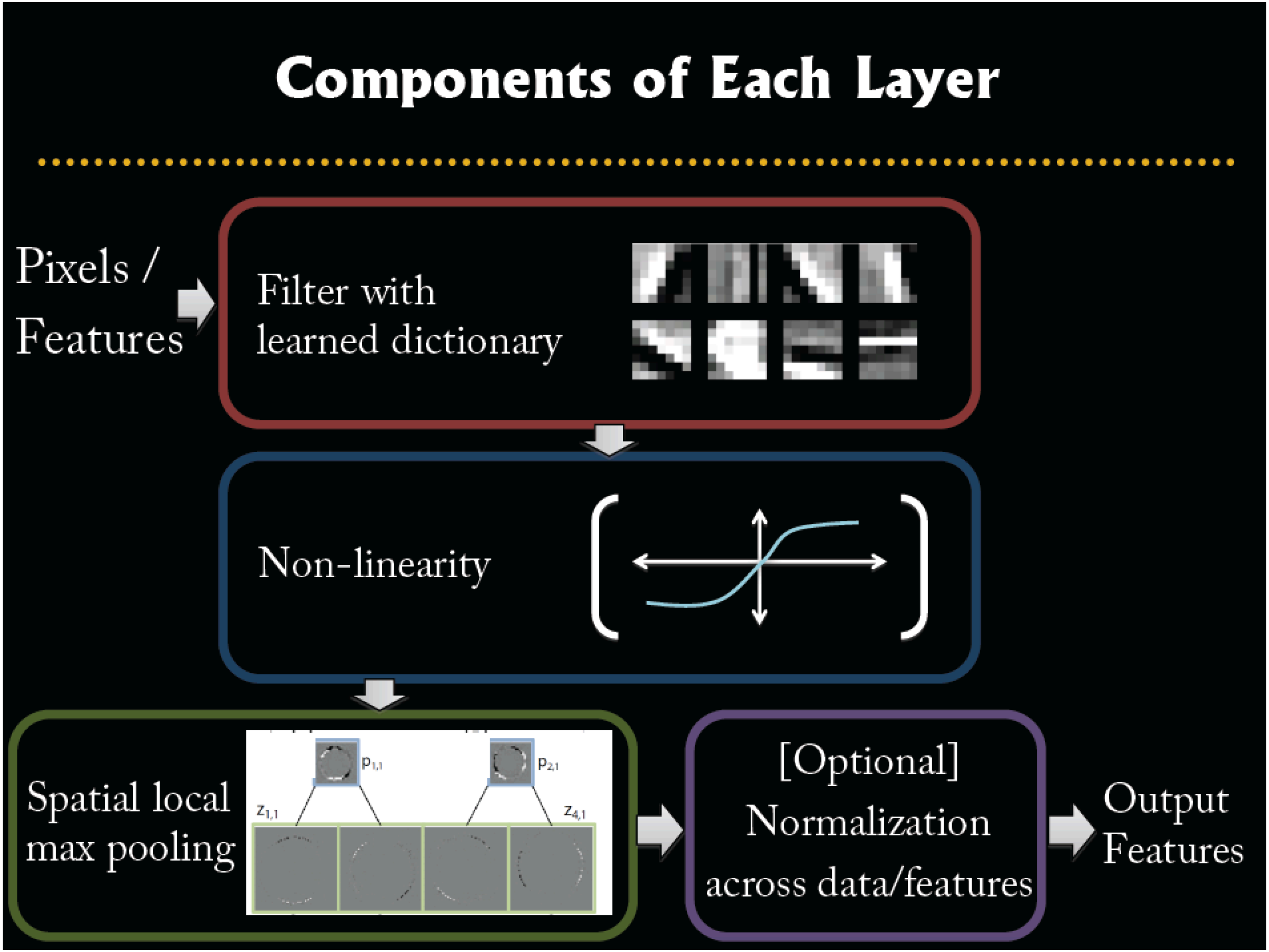
- Spatial pooling
  - Invariance to small transformations
  - Larger receptive fields (see more of input)

Zeiler, Fergus [arXiv 2013]

Visualization technique from [Le et al. NIPS'10]:

Videos from: http://ai.stanford.edu/~quocle/TCNNweb


Components of Each Layer
Pixels / Features
Filter with learned dictionary
Non-linearity
Spatial local max pooling
$p_{1,1}$
$p_{2,1}$
$z_{1,1}$
$z_{4,1}$
[Optional] Normalization across data/features
Output Features

Normalization

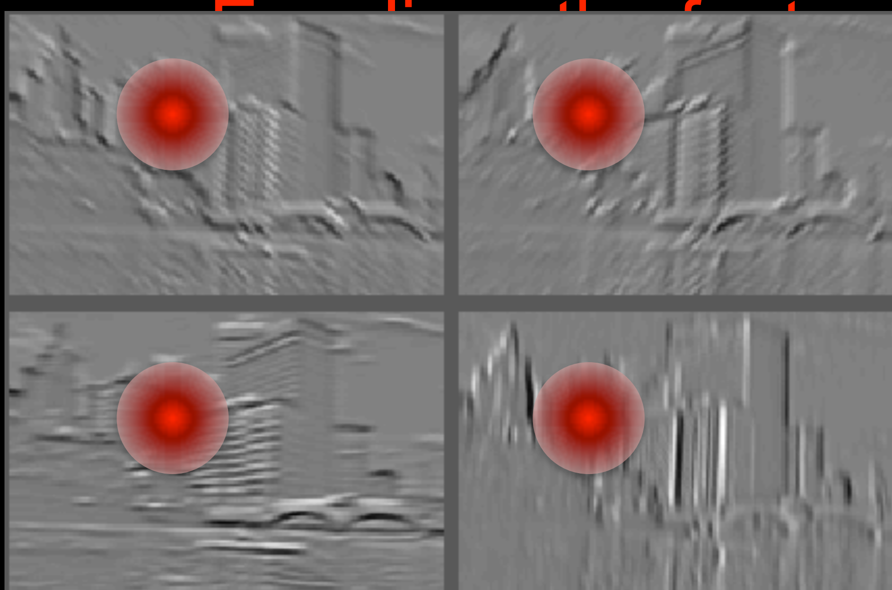
- Contrast normalization
  - See Divisive Normalization in Neuroscience

Input

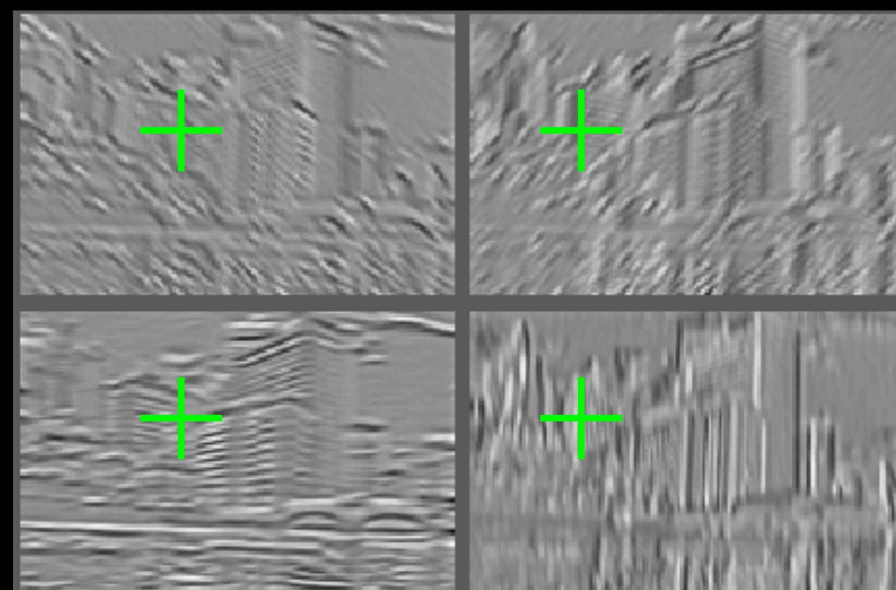
Filters

Normalization

- Contrast normalization (across feature maps)
  - Local mean = 0, local std. = 1, “Local” → 7x7 Gaussian



Feature Maps



Feature Maps
After Contrast Normalization

Role of Normalization

- Introduces local competition between features
  - "Explaining away" in graphical models
  - Just like top-down models
  - But more local mechanism
- Also helps to scale activations at each layer better for learning
  - Makes energy surface more isotropic
  - So each gradient step makes more progress
- Empirically, seems to help a bit (1-2%) on ImageNet
- Recent models do not use normalization

Normalization across Data

- Batch Normalization

[Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy, arXiv:1502.03167]

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

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

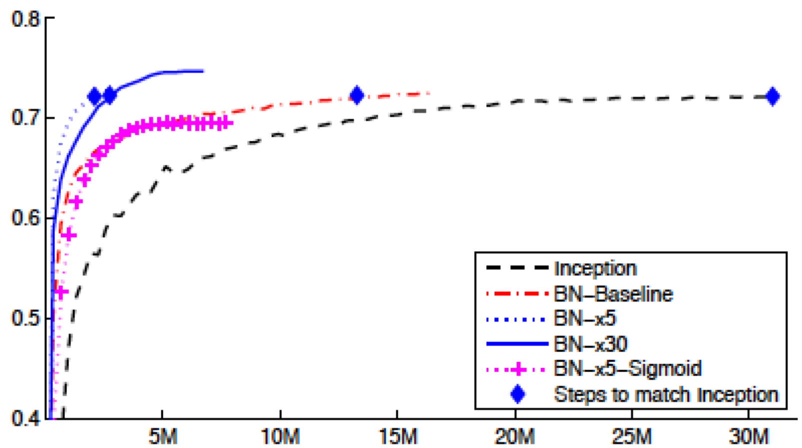
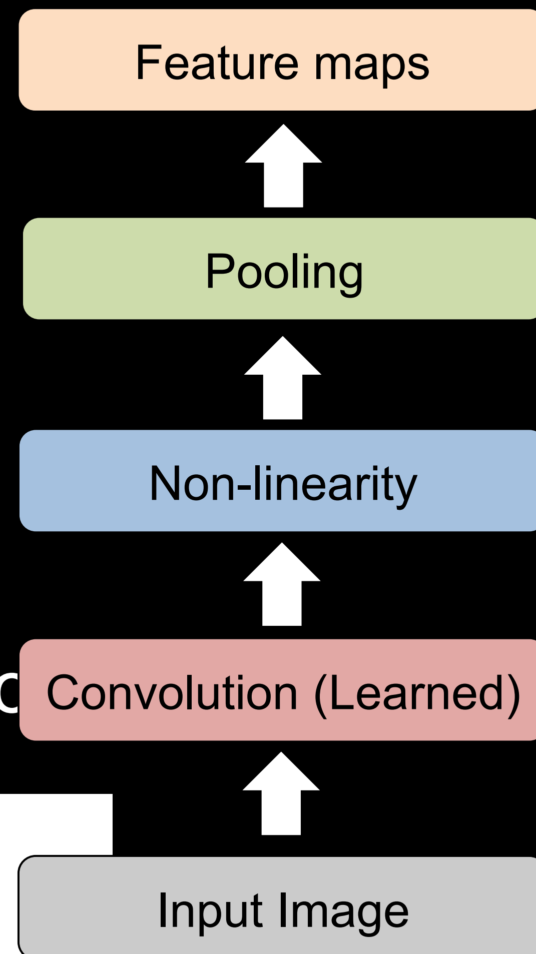
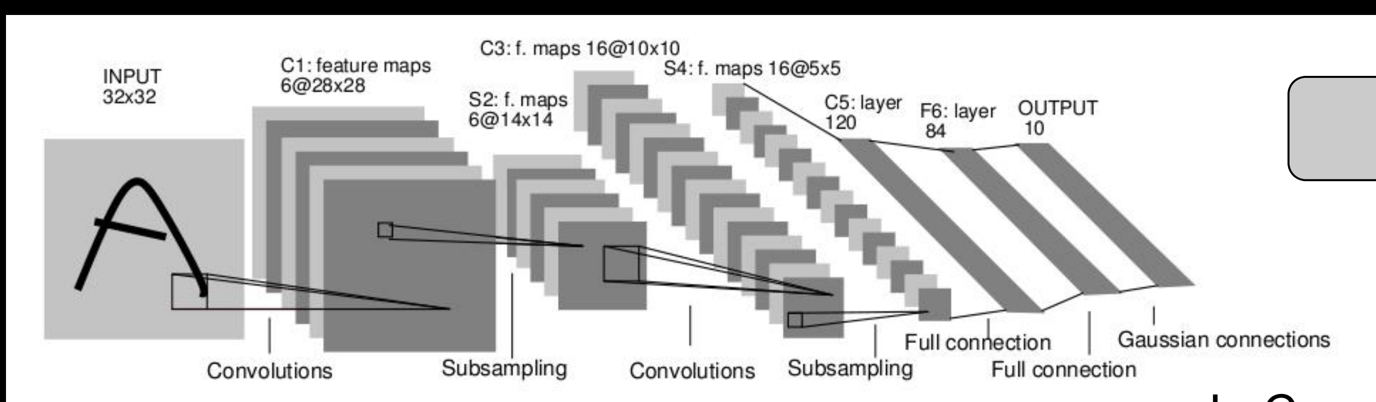


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Overview of Convnets

- Feed-forward:
  – Convolve input
  – Non-linearity (rectified linear)
  – Pooling (local max)
- Supervised
- Train convolutional filters by back-propagating classification erro





Architecture

- Big issue: how to select
  – Depth
  – Width
  – Parameter count
- Manual tuning of features has turn into manual tuning of Architectures

How we choose the architecture?

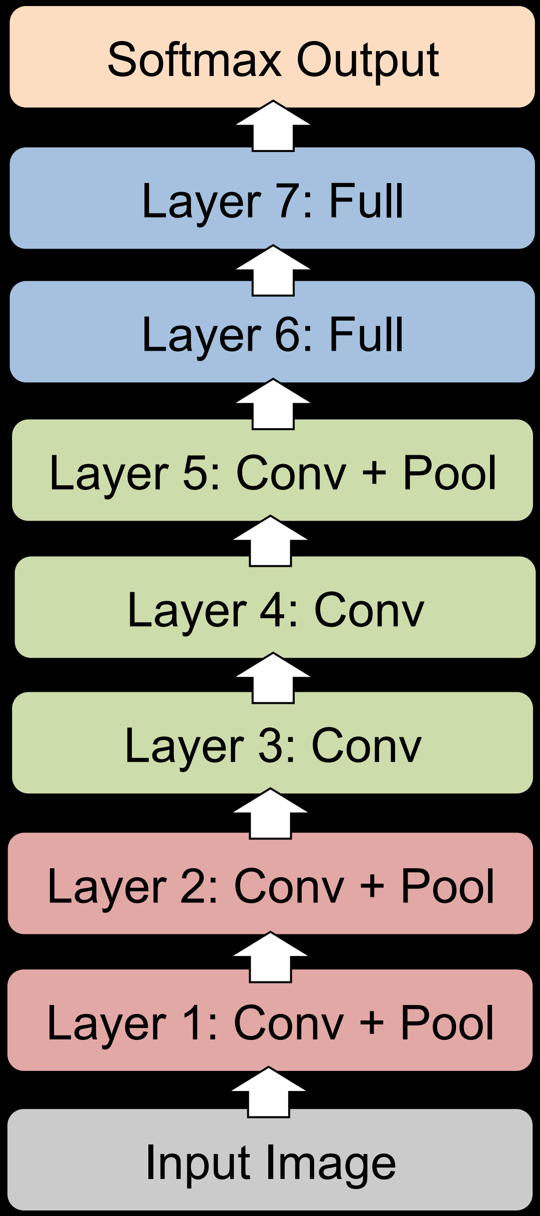
- Many hyper-parameters:
- – # layers, # feature maps
- Cross-validation
- Grid search (need lots of GPUs)
- Smarter strategies:
  - Random [Bergstra & Bengio JMLR 2012]
  - Gaussian processes [Hinton]

How important is Depth

- “Deep” in Deep Learning
- Ablation study
- Tap off features

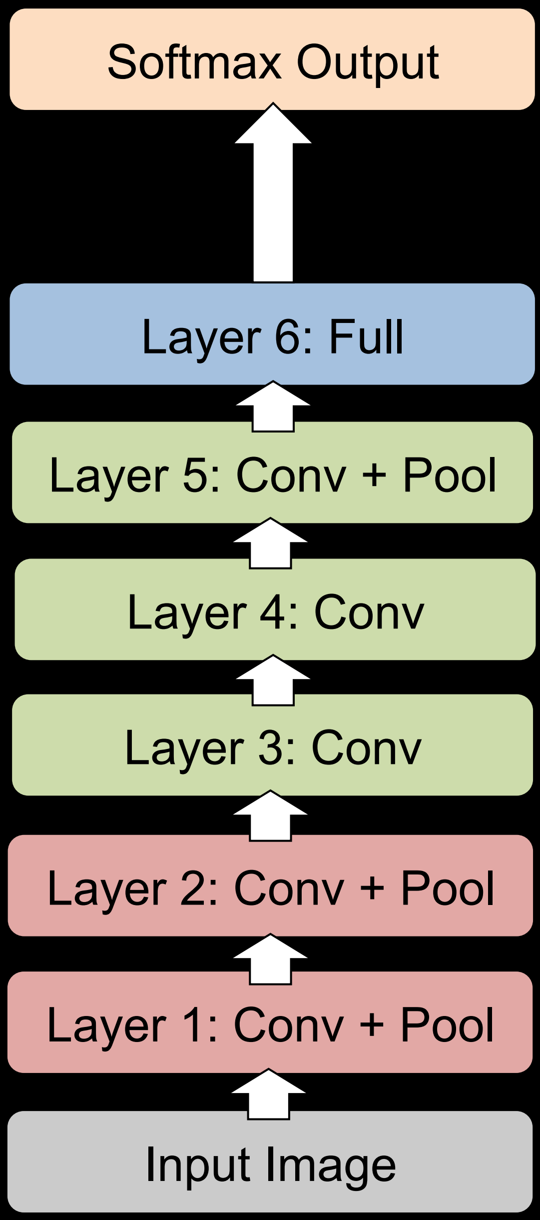
Architecture of Krizhevsky et al.

- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error
- Our reimplementation:
  18.1% top-5 error



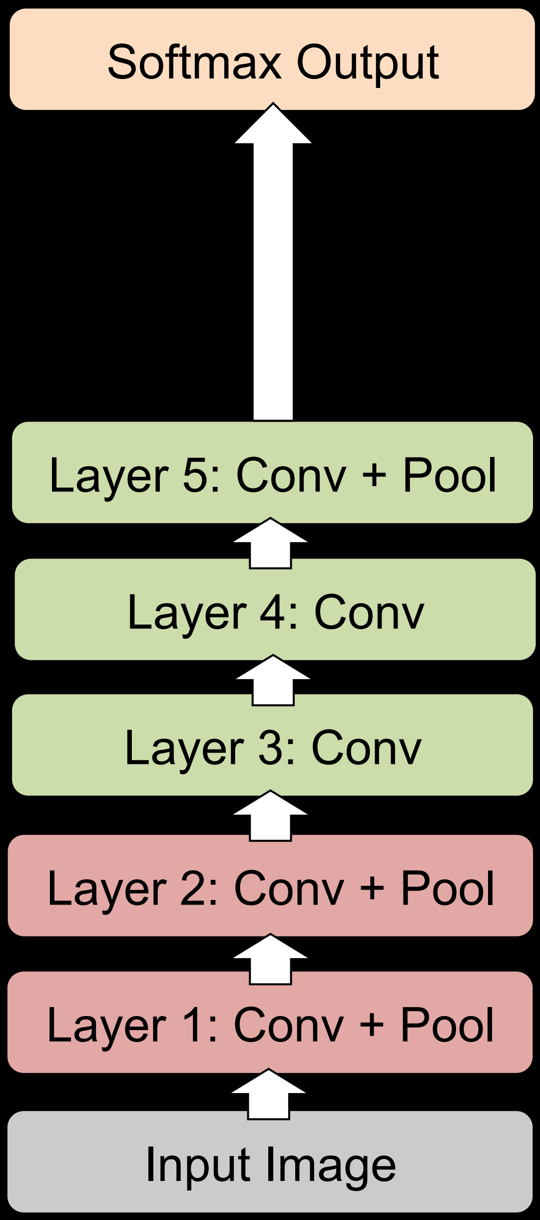
Architecture of Krizhevsky et al.

- Remove top fully connected layer
  - Layer 7
- Drop 16 million parameters
- Only 1.1% drop in performance!



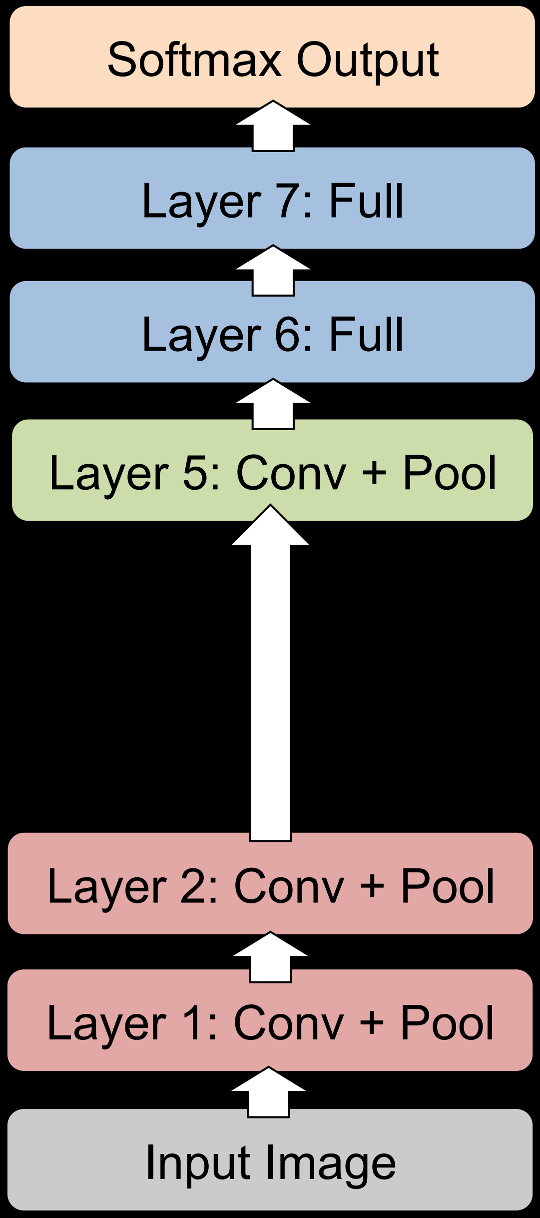
Architecture of Krizhevsky et al.

- Remove both fully connected layers
  - Layer 6 & 7
- Drop ~50 million parameters
- 5.7% drop in performance



Architecture of Krizhevsky et al.

- Now try removing upper feature extractor layers:
  - Layers 3 & 4
- Drop ~1 million parameters
- 3.0% drop in performance



Architecture of Krizhevsky et al.

- Now try removing upper feature extractor layers & fully connected:
  - Layers 3, 4, 6 ,7
- Now only 4 layers
- 33.5% drop in performance

→Depth of network is key