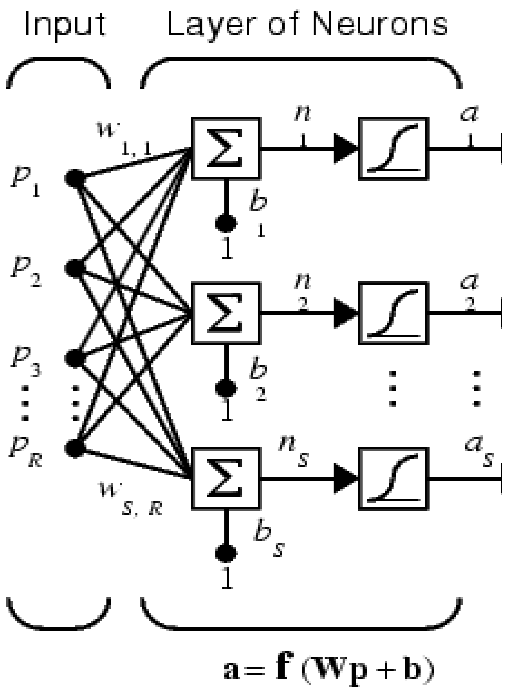
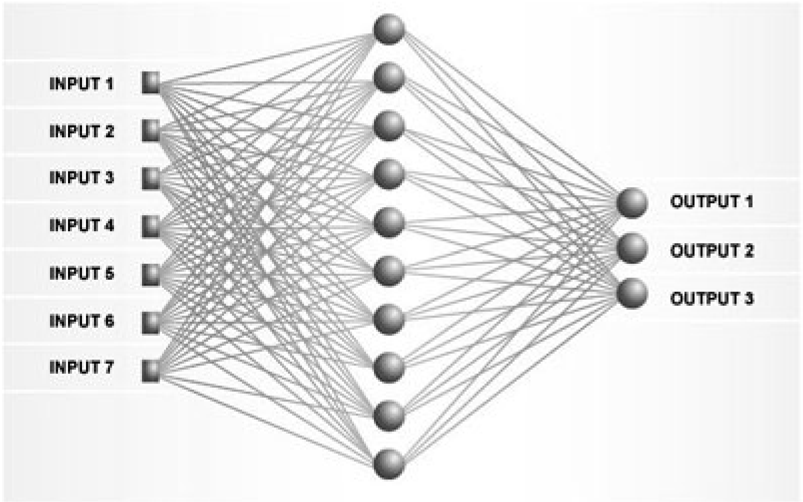
Deep Convolutional Neural Networks

Nov. 20th, 2015

Bruce Draper

Background: Fully-connected single layer neural networks





- Feed-forward classification
- Trained through back-propagation

Colorado State University

Example Computer Vision Task: label these images

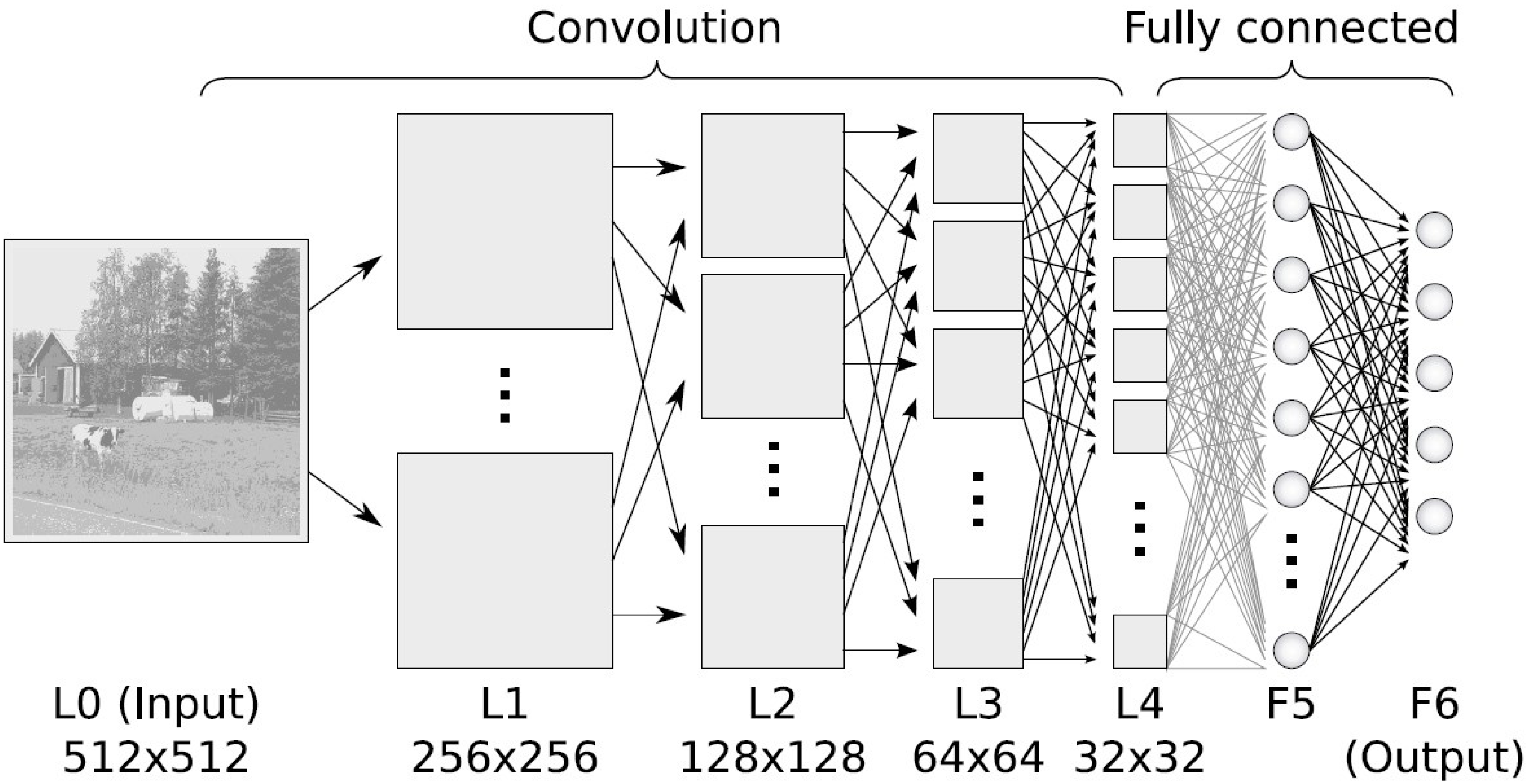
Neural Networks in Computer Vision

- Introduced in the 1980's
  - … and abandoned in the 1990's
- Images are complex signals
  - Most of any image is "background"
  - Pixel [x,y] in one images isn't the same 3D point as in another
  - Targets can be any place or scale
    - sometimes any orientation
- Fully-connected networks were too specific
  - Not translationally invariant
  - Not scale invariant
- Fully-connected networks have too many weights
  - Images have lots of pixels
- Sometimes used to classify image chips after focus of attention

So computer vision went in other directions…

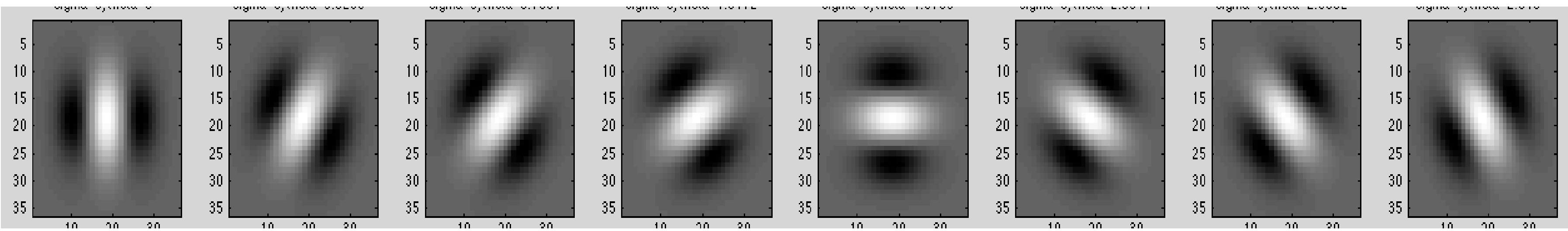
- Object recognition (1990-2010):
  - Local feature extraction
    - Edges
    - Homogeneous regions
    - High-information image patches
  - Features to descriptors
    - Histogram of Edges (e.g. HoG, SIFT)
    - Graphs of regions
    - Image patch codes (e.g. BoW)
  - Classify images based on descriptors
    - Kernelized support vector machines (SVMs)

… until deep convolutional neural networks arrived

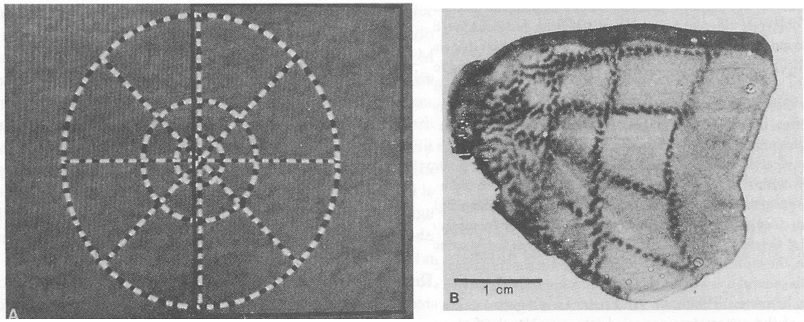


Biological motivation: human vision

- Cells in the early vision system (LGN/V1)
  - Have very small receptive fields
    - Input from a specific small piece of the visual field
  - Respond to specific local patterns



  - Spatially distributed like an image

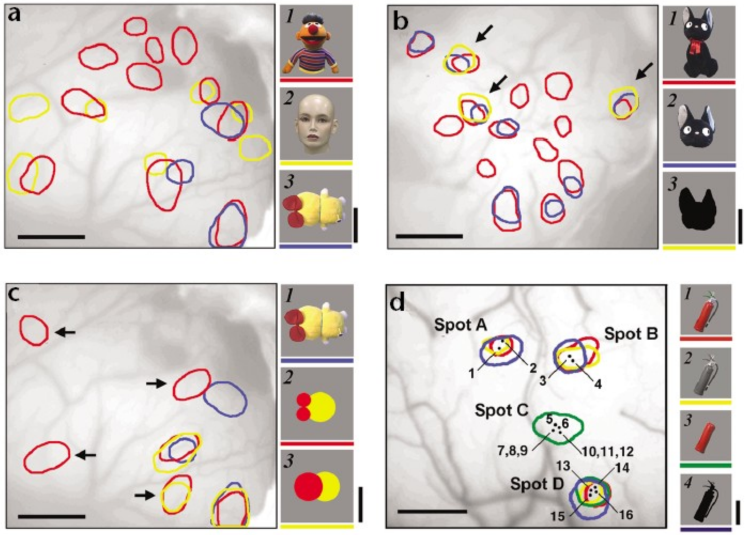


Tootell, et al. 1982.

Colorado State University

Biological motivation: human vision (II)

- Further down the visual pathway (V8/ITC)
  - Larger receptive fields
  - More complex patterns
    - Respond to wider variations
    - Some are viewpoint dependent
    - Others are viewpoint independent
  - Translationally invariant
    - Within the cells receptive field

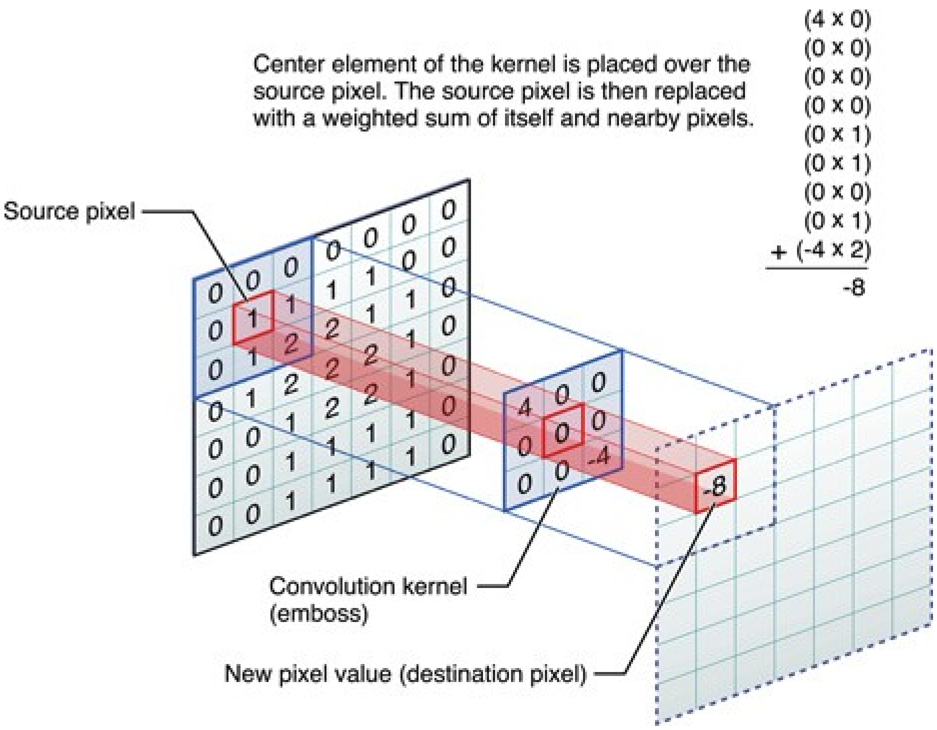


Tsunoda, et al 2001

- Inspiration for DCNNs: mimic this structure

1^{st} Convolutional Layer

- Input is an image
- For every NxN window, there is a neuron
  - N is small
- The neuron has a weight vector, a bias term and a non-linear function f
  - $o \equiv f(W \cdot I + b)$
- Here's the 1^{st} piece of magic: Every neuron has the *same* weight vector, bias term and non-linear function



Convolution

- Convolution is the process of sliding one function over another and computing the dot product at every position
- Formally, discrete 2D convolution is defined as:

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

- In practice, g[x,y] is non-zero over a small range N
- When every neuron has the same weights, the layer as a whole computes a convolution:

- In practice, g[x,y] is non-zero over a small range N
- When every neuron has the same weights, the layer as a whole computes a convolution:

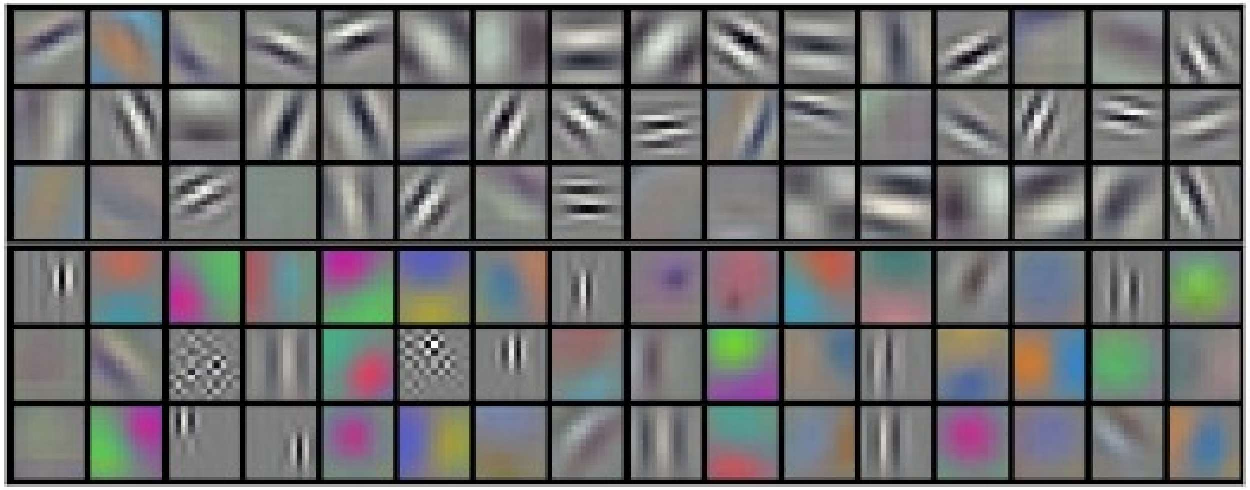
$$O \equiv f(I * W)$$

Why Convolution?

- Convolutional layers solve two problems:
  - Small numbers of parameters
    - Given a 3x3 window size, the entire layer has only 10 weights!
  - Translational invariance
    - Translating the input just translates the output
- In terms of biological motivation
  - Outputs looks like early vision features
    - Dot products with small masks
    - Same features computed everywhere
    - Output structured like an image
- But 1 early vision feature is not enough!

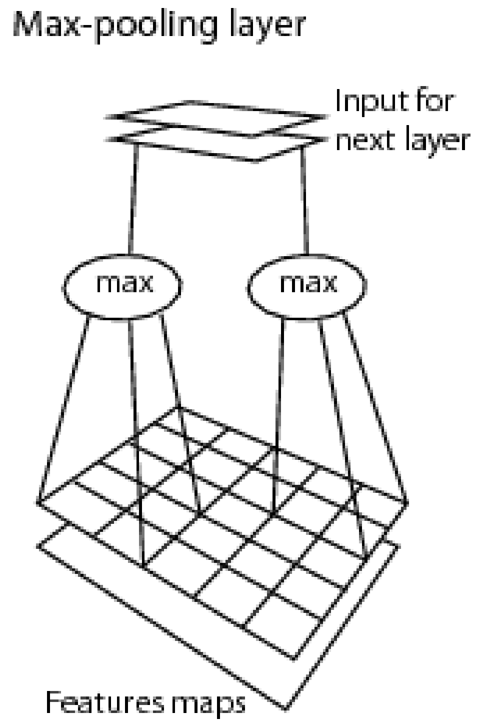
1st layer: lots of feature banks

- In practice, you don't have just 1 first layer
- You train a bank of filters (weight vectors)
- Krizehvsky et al 2012 train 96 1st level features
  - Here is an example of the trained weight vectors



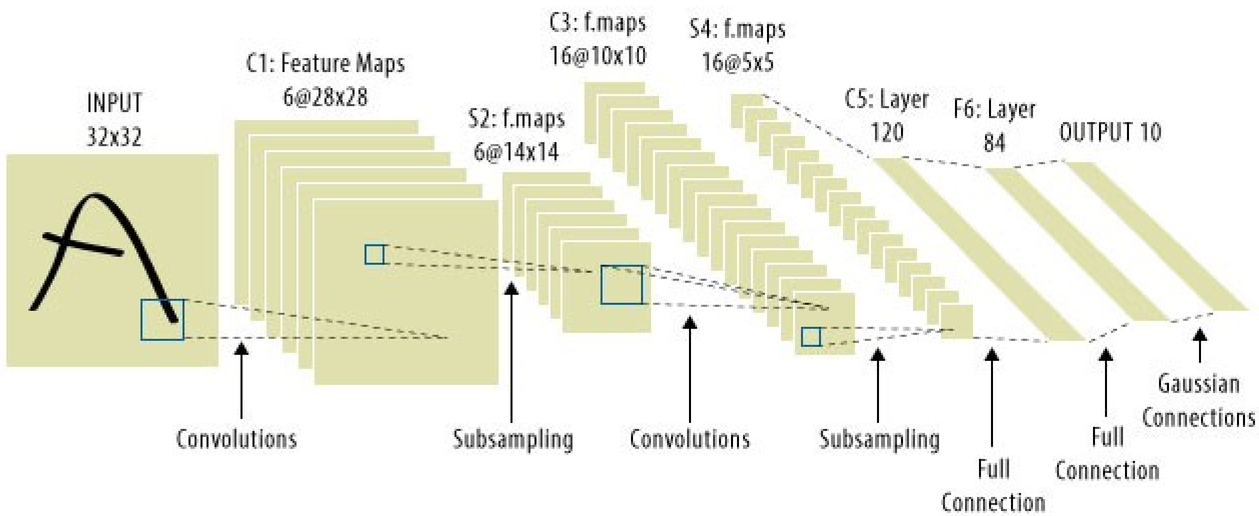
Max Pooling Layers

- Convolutional layers transform (raw) images to (feature) images
  - The data didn't get smaller or become more symbolic
  - No increase in receptive field size
- Max pooling layers reduce resolution
  - For every non-overlapping NxN window, select maximum value
  - No weights to train
  - Reduces size by N^2



Convolution & Max Pooling

- Piece of magic #2: alternate convolution and max pooling layers
  - Until the images get really small
  - Then use a fully-connected backprop net



Convolution & Max Pooling (II)

- Note that the convolution masks are 3D
  - Width x Height x Depth
  - Depth = # of features
  - 1^{st} layer: depth = 3 (color images: R, G & B)
  - Other layers: depth = # of features at previous level
- Max pooling increases the receptive field of following layers
- 1^{st} layer computes local features
  - E.g. edges, bars & impulses
- Successive layers combine previous features into larger features
  - Until image size is 1x1xD (or at least very small)
  - At which point you have 2D translationally invariant features
  - Which become the input to a fully connected single hidden layer

Training (light)

- The algorithm to train a deep convolutional network is still back propagation
  - Present image to network, calculate output
  - Compute error between actual output and target output
  - Propagate error backwards through the network
  - Update weights based on error gradients
- Last layer is a traditional output layer
  - Often one node per label
- Penultimate layer is a fully-connected hidden layer
- We need to back-propagate error gradients through:
  - Max pooling layers
  - Convolutional layers

Review: updating output layer weights

- Error function:
- Error function: $E = {}^{1}\!/_{2}\left(T_j - O_j\right)^2$
- Chain rule (no non-linear function):
- Chain rule (no non-linear function): $\frac{\delta E}{\delta W_{i,j}} = \frac{\delta E}{\delta O_j}\frac{\delta O_j}{\delta W_{i,j}}$
- $\frac{\delta E}{\delta O_j} = -\left(T_j - O_j\right)$
- $\frac{\delta O_j}{\delta W_{i,j}} = X_i$
- $\frac{\delta E}{\delta W_{i,j}} = -\left(T_j - O_j\right)X_i$

Review: Updating with non-linear functions

- Chain rule (output): $\frac{\delta E}{\delta W_{i,j}} = \frac{\delta E}{\delta F_j} \frac{\delta F_j}{\delta O_j} \frac{\delta O_j}{\delta W_{i,j}}$
- If F = Sigmoid, then $\frac{\delta S_j}{\delta O_j} = S_j(1 - S_j)$
- So $\frac{\delta E}{\delta W_{i,j}} = -(T_j - S_j)S_j(1 - S_j)X_i$
- Chain rule for hidden layer:

$$\frac{\delta E}{\delta W_{i,q}} = \left\{\sum_j \frac{\delta E}{\delta F_j} \frac{\delta F_j}{\delta O_j} \frac{\delta O_j}{\delta F_q}\right\} \frac{\delta F_q}{\delta O_q} \frac{\delta O_q}{\delta W_{i,j}}$$

- Where $\frac{\delta O_j}{\delta F_q} = w_{q,j}$

- Chain rule (output):
- If F = Sigmoid, then
- So
- Chain rule for hidden layer:
- Where

Updating max pooling layers

Q: How do we update a max pooling node?

A: We don't! It has no weights to update!

(trick question)

Updating convolutional layers

- Consider just one convolutional neuron
  - It feeds a single max pooling element
  - Which feeds elements j
  - We will use q to index both pooling & convolutional neurons
- Let $\frac{\delta E}{\delta P_q} = \left\{\sum_j \frac{\delta E}{\delta F_j} \frac{\delta F_j}{\delta O_j} \frac{\delta O_j}{\delta P_q}\right\}$ be the back-propagated gradient to the output of the pooling node P_q.
- Then $\frac{\delta E}{\delta W_{i,q}} = \left\{\sum_j \frac{\delta E}{\delta F_j} \frac{\delta F_j}{\delta O_j} \frac{\delta O_j}{\delta P_q}\right\} \frac{\delta P_q}{\delta F_q} \frac{\delta F_q}{\delta O_q} \frac{\delta O_q}{\delta W_{i,j}}$

- Let be the back-propagated gradient to the output of the pooling node P_q.
- Then

Updating convolutional layers (II)

- $\frac{\delta P_q}{\delta F_q} = \begin{cases} 1 \; if \; F_q \; is \; local \; maximum \\ 0 \; otherwise \end{cases}$
- This makes the update gradients sparse
- But every convolutional neuron has to share weights
- So average the $\Delta w_{i,j}$'s across the layer

- This makes the update gradients sparse
- But every convolutional neuron has to share weights
- So average the $\Delta w_{i,j}$'s across the layer

… one more thing

- Convolutional layers rarely use tanh or sigmoid
- More often, $F(x) = \begin{cases} x \; if \; x > 0 \\ 0 \; if \; x \leq 0 \end{cases}$
- This function:
  - Is non-linear
  - Has trivial derivatives
  - Avoids vanishing (positive) gradients
- This is the 3rd piece of magic

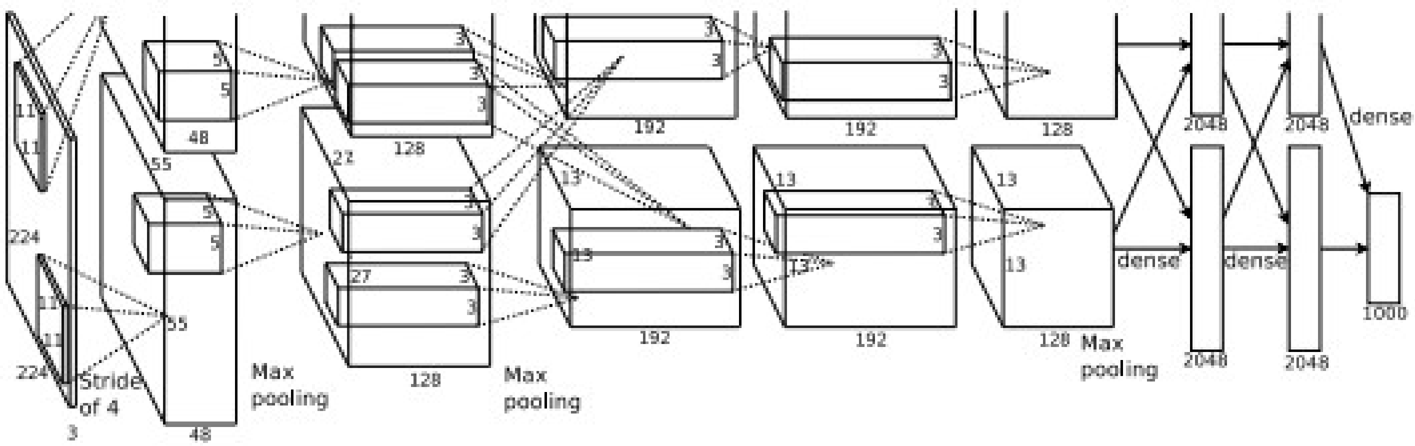
- Convolutional layers rarely use tanh or sigmoid
- More often,
- This function:
  - Is non-linear
  - Has trivial derivatives
  - Avoids vanishing (positive) gradients
- This is the 3rd piece of magic

Deep Convolutional Nets in practice

- There are three well-known, publicly available deep convolutional nets for object recognition
  - AlexNet (Krizhevsky, Sutskever & Hinton 2012)
  - GoogLeNet (Szegedy, et al 2015)
  - DeCAF (Donahue, et al 2013)

- Why just three?
  - They take many training images and a very long time to train
  - Even given lots of GPUs

AlexNet architecture



Krizhevsky, et al 2012

- Deep filter banks (max 192)
- Stride layer (instead of max pooling) at beginning
- Window sizes range from 5x5 to 15x15
- Actually, two nets that merge at the end (for efficiency)
- The majority of weights are in the last two fully connected layers

Training AlexNet

- Images resized to 256x256
- 60 million parameters
- 1,000 training classes
- 90 training cycles
- 1.2 million training images per cycle
- 6 days on two NVIDOA GTX 580 3GB GPUs