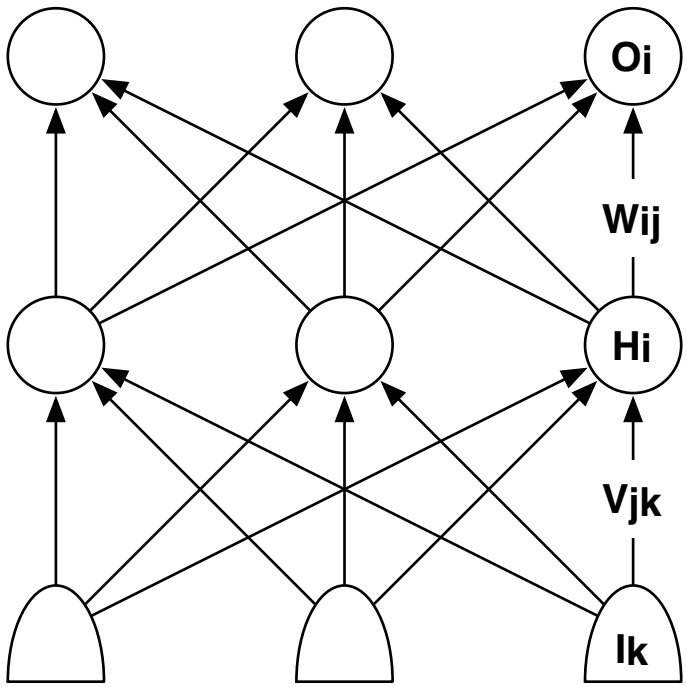
Traditional Backpropagation (Two Layers)



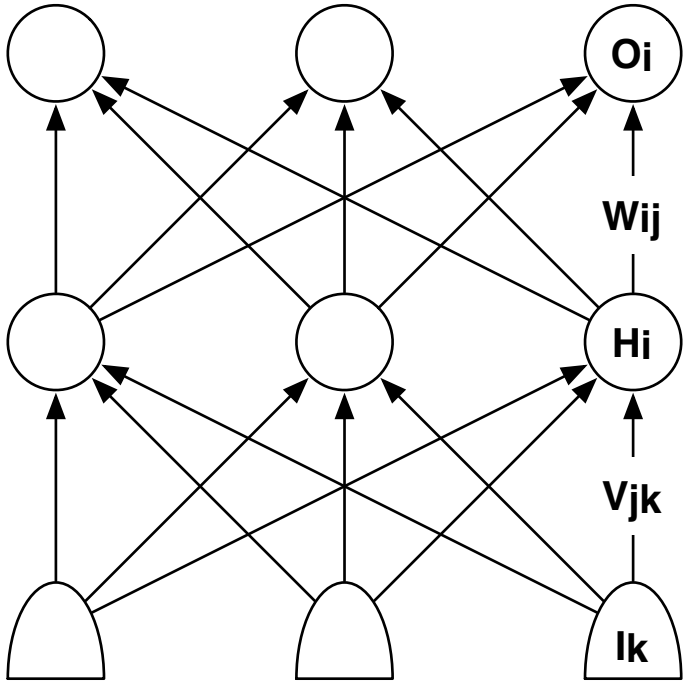
$$\Delta W_{ij} = -\alpha \frac{\partial E}{\partial W_{ij}}$$

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial O_i} \frac{\partial O_i}{\partial W_{ij}}$$

$$\Delta V_{jk} = -\alpha \frac{\partial E}{\partial V_{jk}}$$

$$\frac{\partial E}{\partial V_{jk}} = \sum_i \frac{\partial E}{\partial O_i} \frac{\partial O_i}{\partial H_j} \frac{\partial H_j}{\partial V_{jk}}$$

Traditional Backpropagation (Two Layers)



$$\Delta W_{ij} = -\alpha \frac{\partial E}{\partial W_{ij}}$$

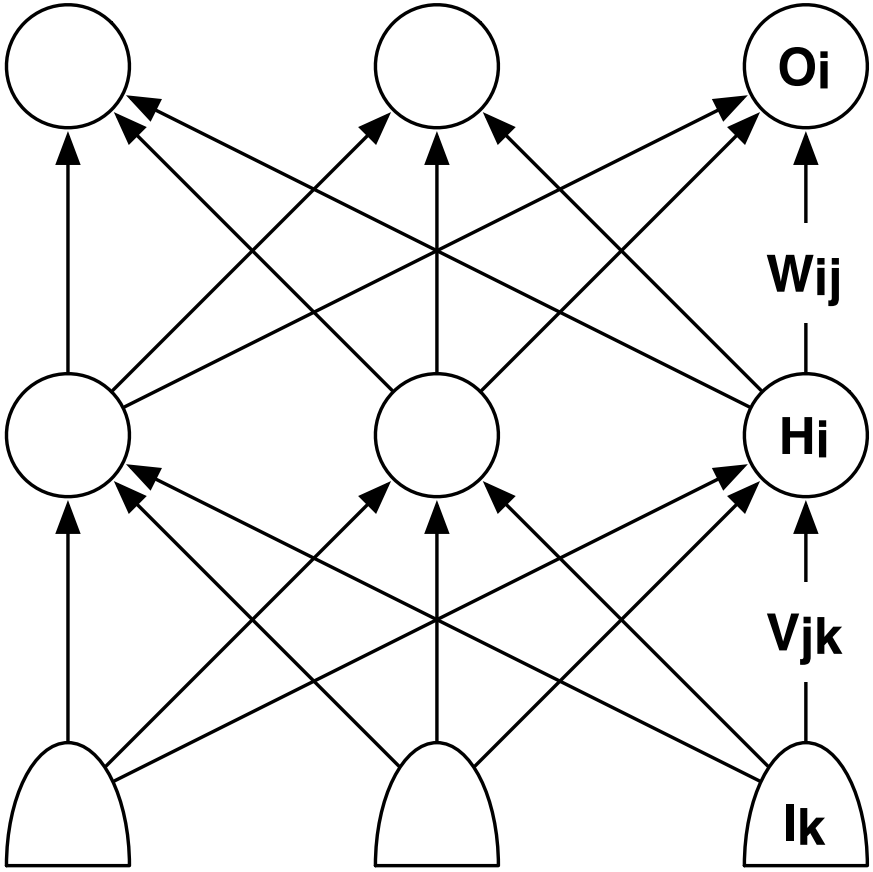
$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial O_i} \frac{\partial O_i}{\partial W_{ij}}$$

$$\Delta V_{jk} = -\alpha \frac{\partial E}{\partial V_{jk}}$$

$$\frac{\partial E}{\partial V_{jk}} = \frac{\partial E}{\partial H_j} \frac{\partial H_j}{\partial V_{jk}}$$

$$\frac{\partial E}{\partial H_j} = \sum_i \frac{\partial E}{\partial O_i} \frac{\partial O_i}{\partial H_j}$$

Modularizing Backpropagation



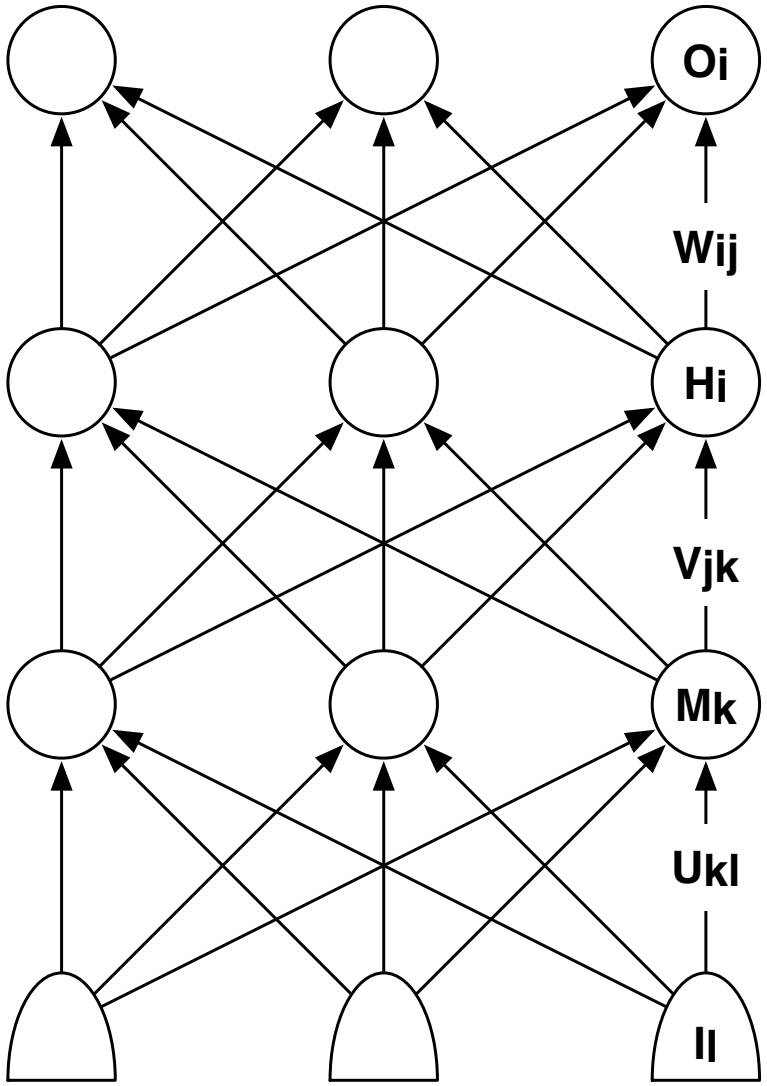
$$\frac{\partial E}{\partial O_i}$$

$$\frac{\partial E}{\partial W_{ij}} \qquad \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial O_i}\frac{\partial O_i}{\partial W_{ij}}$$

$$\frac{\partial E}{\partial H_j} \qquad \frac{\partial E}{\partial H_j} = \sum_i \frac{\partial E}{\partial O_i}\frac{\partial O_i}{\partial H_j}$$

$$\frac{\partial E}{\partial V_{jk}} \qquad \frac{\partial E}{\partial V_{jk}} = \frac{\partial E}{\partial H_j}\frac{\partial H_j}{\partial V_{jk}}$$

Modularizing Backpropagation



$$\frac{\partial E}{\partial O_i}$$

$$\frac{\partial E}{\partial W_{ij}} \qquad \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial O_i}\frac{\partial O_i}{\partial W_{ij}}$$

$$\frac{\partial E}{\partial H_j} \qquad \frac{\partial E}{\partial H_j} = \sum_i \frac{\partial E}{\partial O_i}\frac{\partial O_i}{\partial H_j}$$

$$\frac{\partial E}{\partial V_{jk}} \qquad \frac{\partial E}{\partial V_{jk}} = \frac{\partial E}{\partial H_j}\frac{\partial H_j}{\partial V_{jk}}$$

$$\frac{\partial E}{\partial M_k} \qquad \frac{\partial E}{\partial M_k} = \sum_j \frac{\partial E}{\partial H_j}\frac{\partial H_j}{\partial M_k}$$

$$\frac{\partial E}{\partial U_{kl}} \qquad \frac{\partial E}{\partial U_{kl}} = \frac{\partial E}{\partial M_k}\frac{\partial M_k}{\partial U_{kl}}$$

Why do multiple hidden layers fail?

- A single hidden layer can, in theory, represent any differentiable function. Multiple layers don't add much to that.
- Multiple layers compound the **vanishing gradient problem.**
  - Weights are updated proportionally to the size of the error gradient.
  - Sigmoid's gradient is in the range [0,1).
  - In two layers, you're multiplying two small [0,1) gradients.
  - By three layers, you're multiplying three small [0,1) gradients. Not much updating going on there.

Why do multiple hidden layers succeed?

- A single hidden layer can, in theory, represent any differentiable function. But in reality it's hard to learn a lot of functions, particularly ones with a lot of modular, repeated elements (like images).

- Multiple layers can often learn such things with many fewer edges.

- The **vanishing gradient problem** goes away with smarter nonlinear functions than sigmoid.

Rectifiers and Softplus

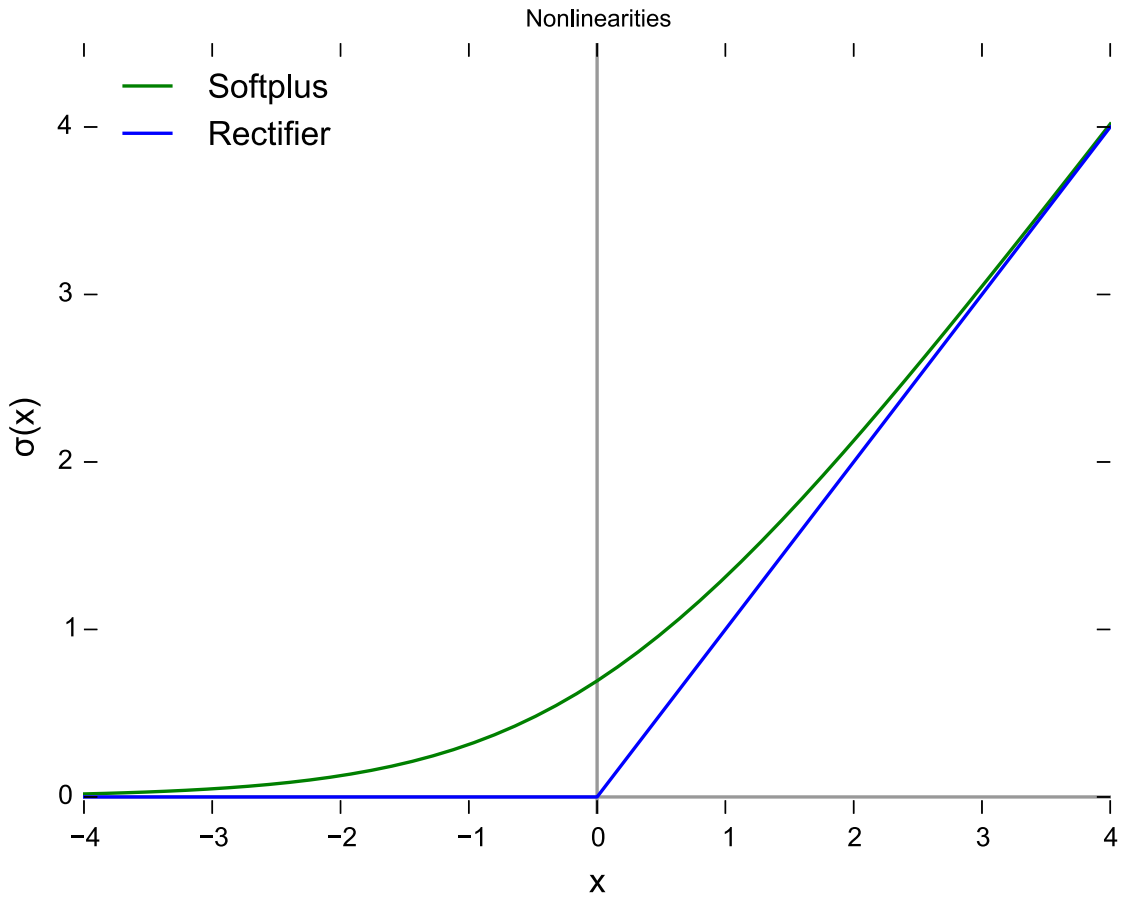
- Rectifier is just
  f(x) = max(0,x)

- Softplus is an approximation to a rectifier which has a first derivative.

  $f(x) = \ln(1 + e^x)$

- The first derivative of Softplus is sigmoid! That's convenient.

  $f'(x) = 1/(1+e^{-x})$



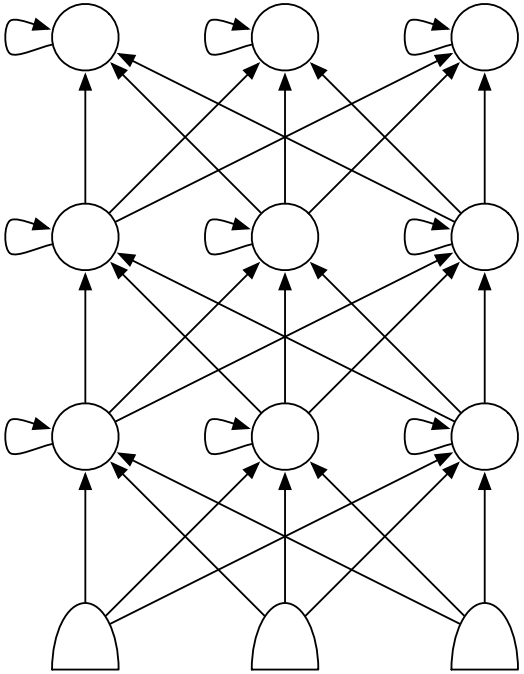
Why Use a Rectifier? (or Softplus)

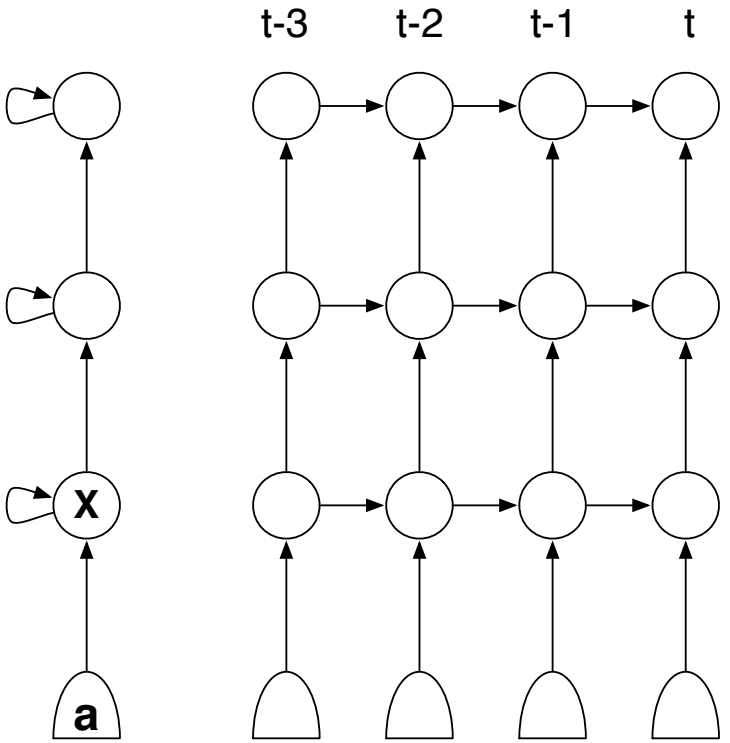
- Sigmoid has an input of (-∞,+ ∞) but its output is (0,1) and gradient is [0,1), often small.

- Rectifiers have an input of (-∞,+ ∞) and an output of (0, + ∞) with a gradient of EITHER 0 or 1. So either the input neurons are *entirely turned off* or they have a full gradient.

- This (1) helps with the vanishing gradient problem and (2) introduces ***sparsity***, a way for parts of the network to be dedicated to one task only.

Deep Recurrent Networks



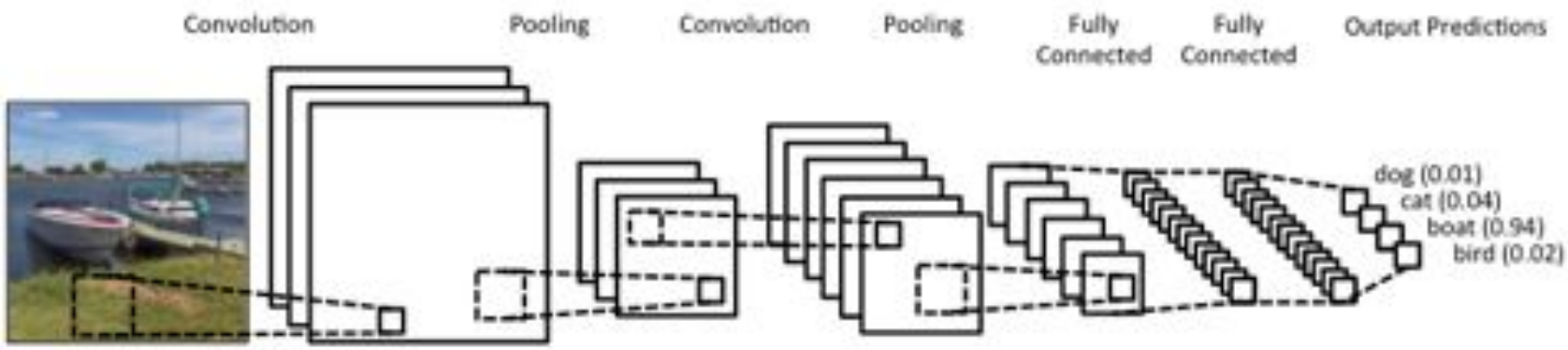


$$x^{(t)} = f(x^{(t-1)}, a^{(t)})$$

What does this equation look like?

Deep Convolutional Networks

- 3 Kinds of Layers (besides input/output):
  - Convolution Pooling (or "Subsampling") Fully Connected



Softmax

- Commonly we want our network to output a **class** (for classification). We could do this by having ***N* output neurons, one per class.** The outputted class is whichever output neuron is highest.

- Softmax is a useful function for this. It is a joint nonlinear function for all N output neurons which guarantees that their values sum to 1 (nice for probability).

$$O_i = \frac{e^{\sum_j I_j W_{ij}}}{\sum_k e^{\sum_j I_j W_{kj}}}$$

- Softmax is often used at the far output end of a deep convolutional network.

Convolution of a Kernel over an array

- Array ***M*** Original array (maybe of image values?)

Array ***P*** New array

Kernel Function ***K***

Neighborhood size/shape ***n***

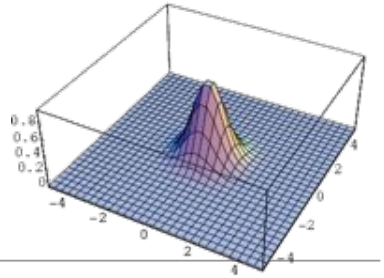
For every cell ***C*** in the original array ***M*** (or image...)

Let ***N1, ..., C, ..., Nn*** be the cells in ***M*** in the *neighborhood* of ***C***

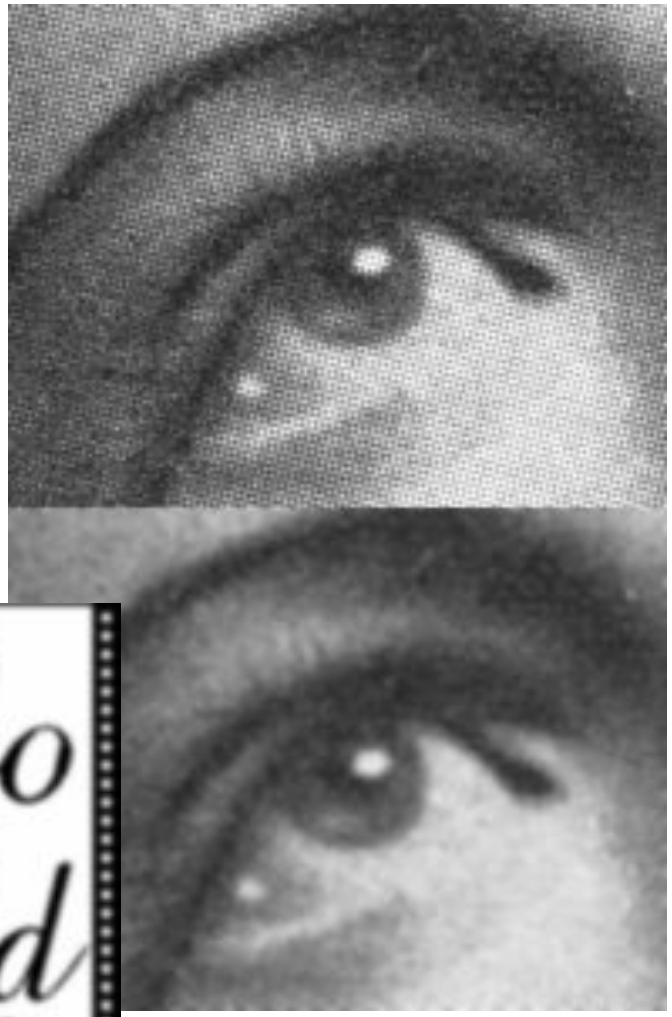
X = K(N1, ..., C, ..., Nn)

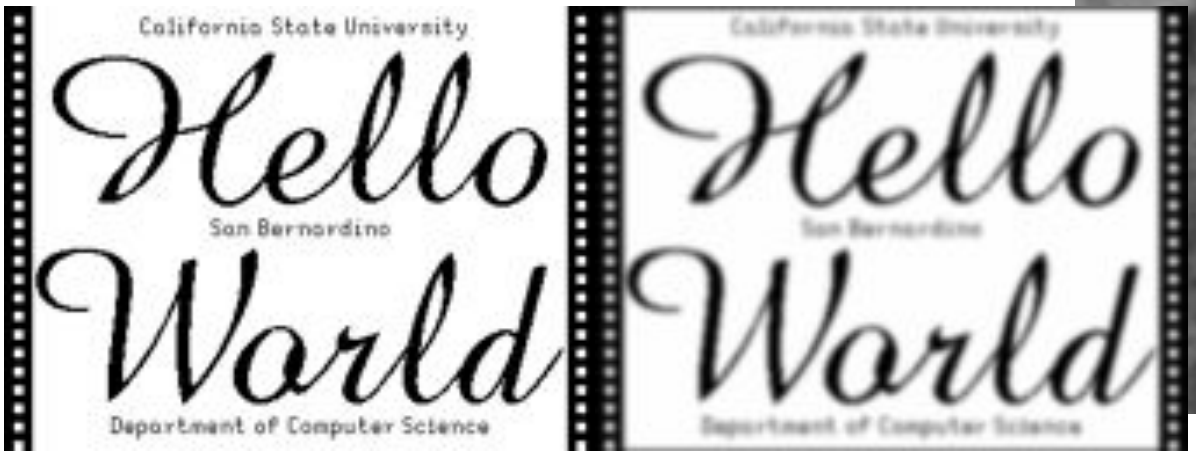
Set the equivalent cell to ***C*** in ***P*** to ***X***

Gaussian Convolution



0.00000067	0.00002292	**0.00019117**	0.00038771	**0.00019117**	0.00002292	0.00000067
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	**0.00019117**
0.00038771	0.01330373	0.11098164	**0.22508352**	0.11098164	0.01330373	0.00038771
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	**0.00019117**
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00000067	0.00002292	**0.00019117**	0.00038771	**0.00019117**	0.00002292	0.00000067





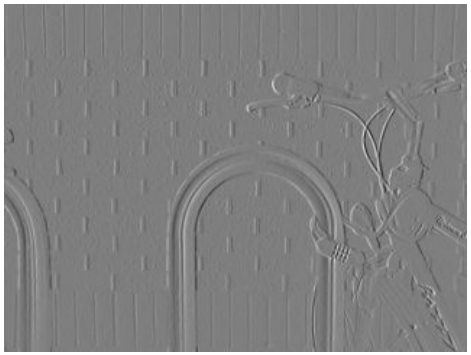
Sobel Edge Detection Via Convolution

-1	0	1
-2	0	2
-1	0	1
Gx		

-1	-2	-1
0	0	0
1	2	1
Gy		

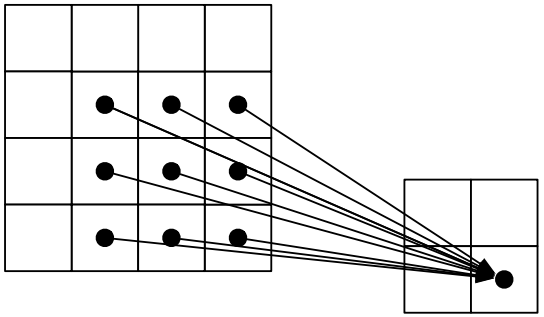
$$\theta = \arctan \frac{Gy}{Gx}$$

$$G = \sqrt{Gx^2 + Gy^2}$$

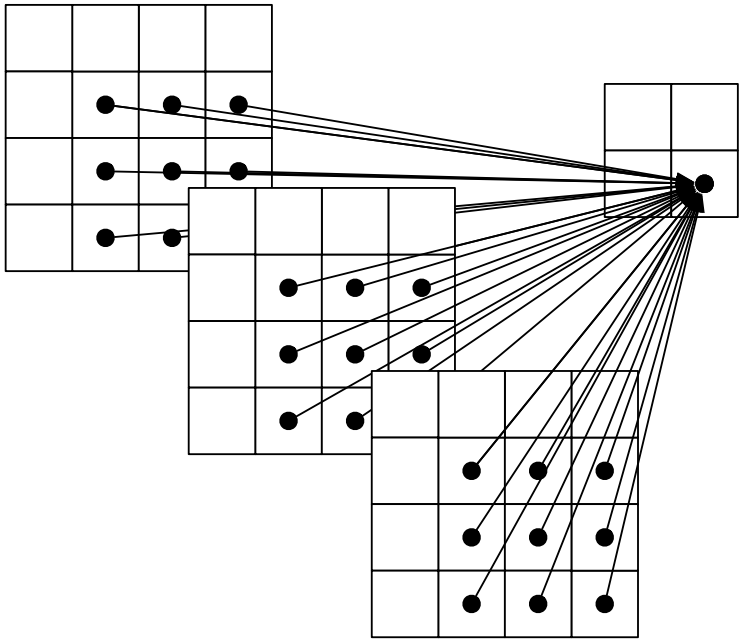


Neural Network weights as a Kernel

3x3 Kernel:
9 Edge Weights
(or *Parameters)*

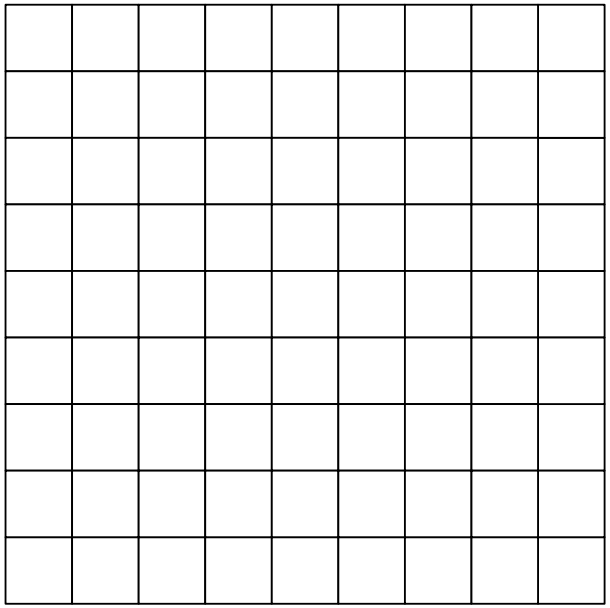


3-layer (red/green/blue?)
3x3 Kernel:
27 Edge Weights



Preparing for Convolution via a Neural Net Layer

Original Image

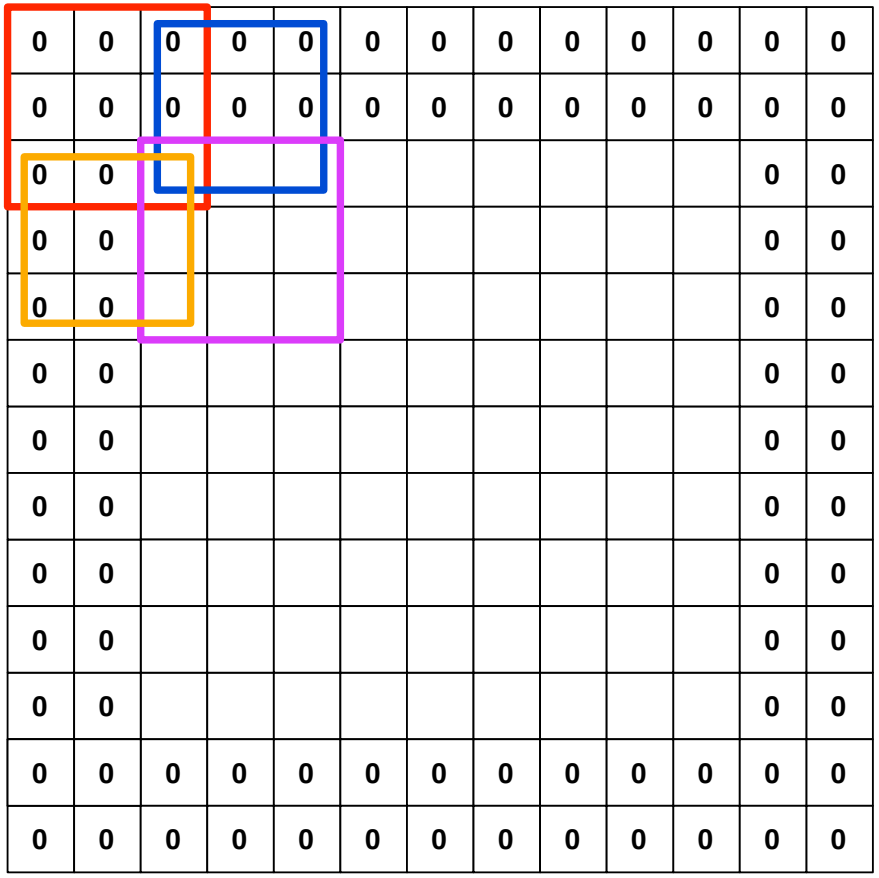


2-Cell Zero-Padded

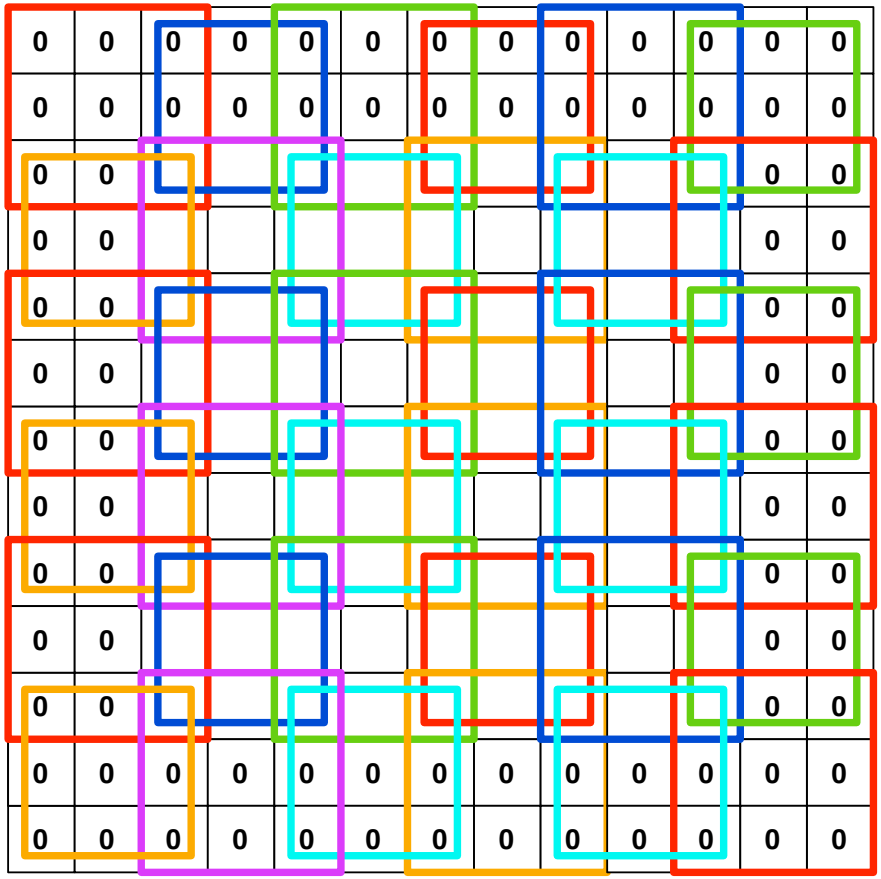
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0										0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Convolution via a Neural Net Layer

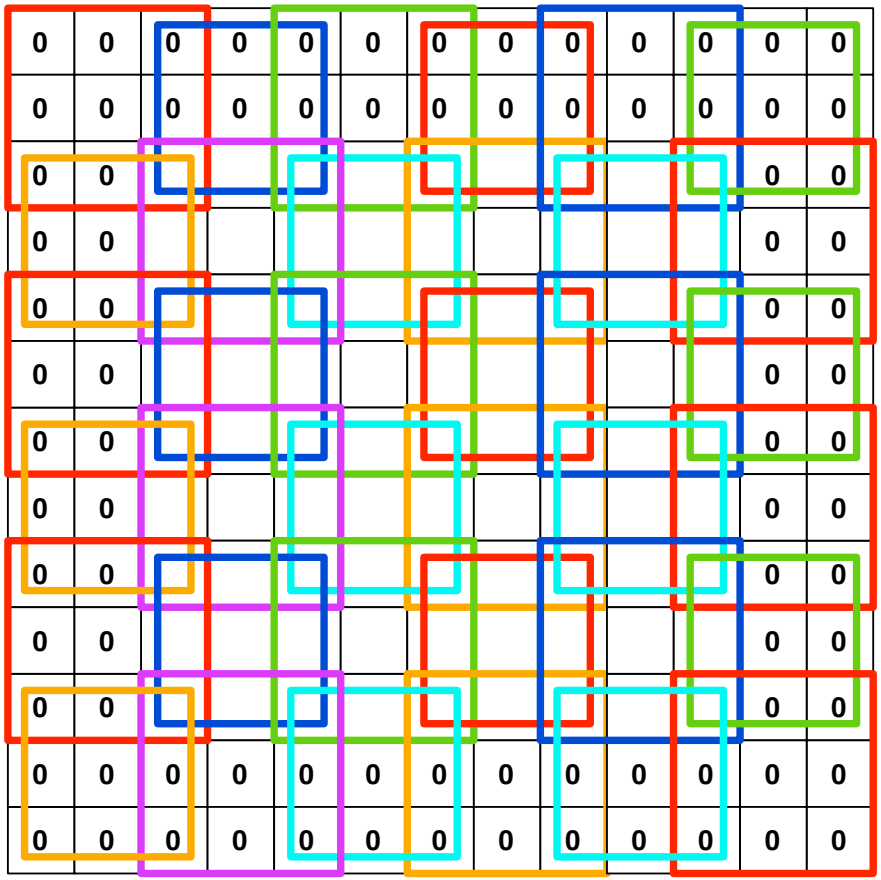
Starting Convolving with a Stride of 2



Full Convolution

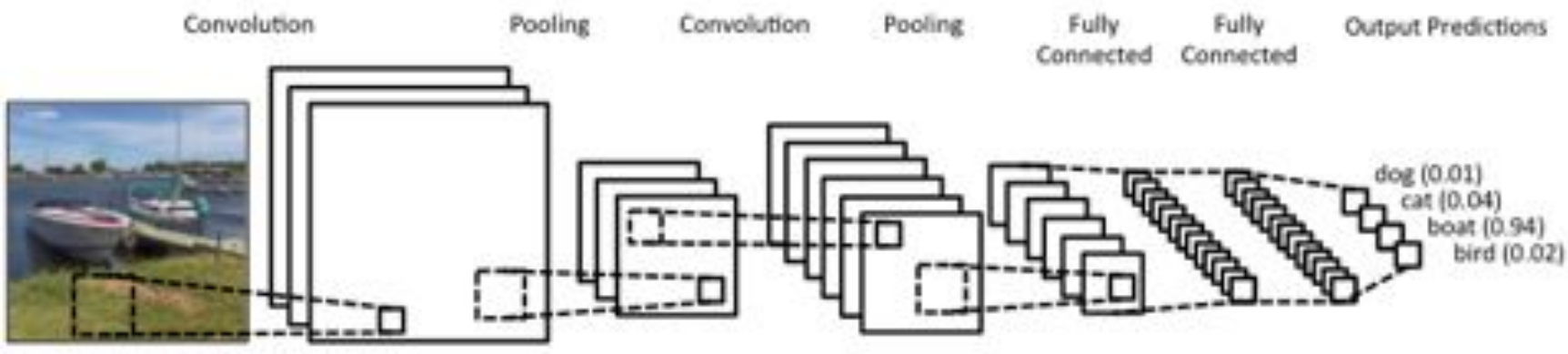


36 Convolutions



- 36 convolutions: from this 9x9, 2-padded array to a 6x6 array (called a **filter**). Each convolution involves a 3x3 single layer kernel (9 edges)
- How many edges must be learned? 3x3x36 = 324?
- No! Just **9 weights**. Because the same edges are **shared** to do all the convolutions.

Back to the Convolution Layer



3 original arrays? (r/g/b)

3 filters

If 3x3 convolution, how many total weights?

3 layers x 9 edges x 3 filters

Forward Propagation with Kernels

- It's just standard neural network propagation: multiply the inputs against the weights, sum up, then run through some nonlinear function.

- Typically you have some N input layers and some M output layers (filters). When you do a p x p convolution, for each convolution, you commonly take *all* the cells within the p x p range for all N input layers, so a total of p x p x N inputs (and weights).

- Each output layer has its own set of weights and does its own independent convolution step on the original input layers.

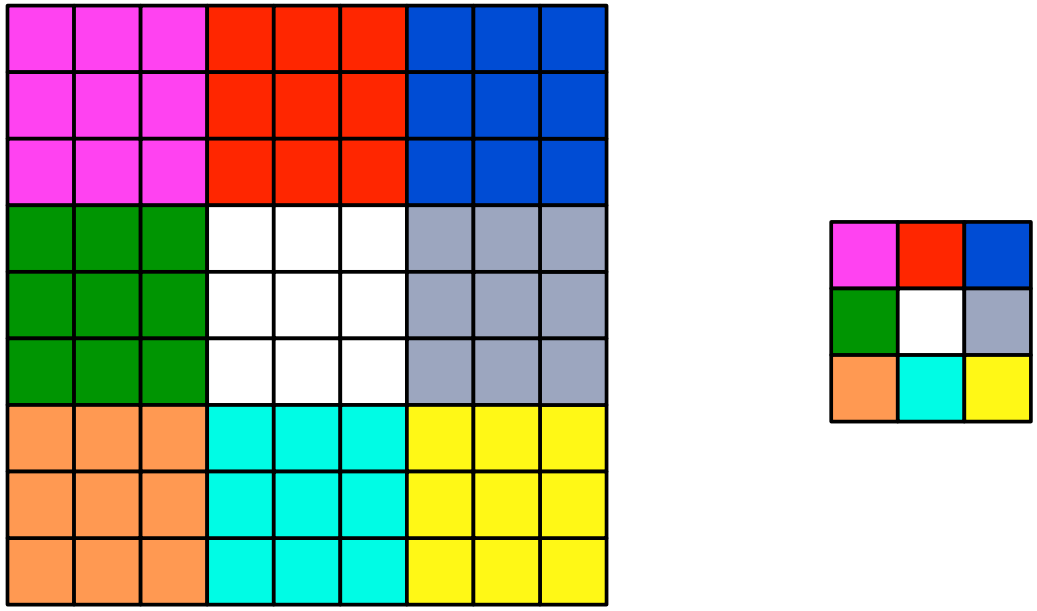
Backpropagation with Parameter Sharing

- If weights are shared, how do you do backpropagation?
- I *believe* you can do this:
  - Compute deltas as if there were unique edges for each convolution
  - Add them up into the 9 edge deltas

Pooling

- Pooling is simple: it's just subsampling
- Example. We divide our 9x9 region into 9 3x3 subregions (actually most common scenario is 2x2). We then pass all cells of a subregion through some **pooling function**, which outputs a single cell value. The result is 9 cells.
- Most common pooling function: max(cell values)

  "max pooling"

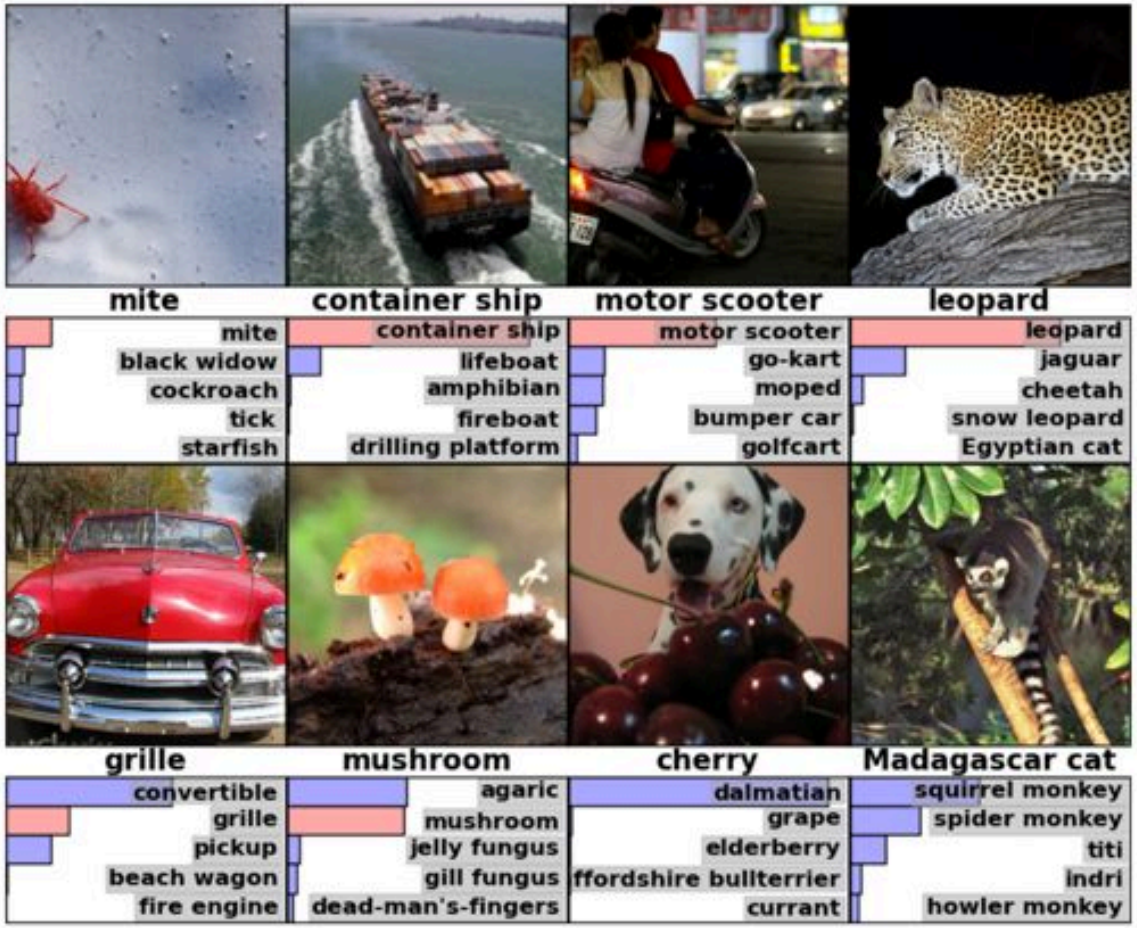


Backpropagation for Max Pooling

- Pooling has no edge weights to update.

- The error is passed back from the pooled cell to the sole cell that was the maximum value. The other cells have zero error because they made no contribution — and so you don't need to continue backpropagating them.

Results


mite
mite
black widow
cockroach
tick
starfish
container ship
container ship
lifeboat
amphibian
fireboat
drilling platform
motor scooter
motor scooter
go-kart
moped
bumper car
golfcart
leopard
leopard
jaguar
cheetah
snow leopard
Egyptian cat
grille
convertible
grille
pickup
beach wagon
fire engine
mushroom
agaric
mushroom
jelly fungus
gill fungus
dead-man's-fingers
cherry
dalmatian
grape
elderberry
ffordshire bullterrier
currant
Madagascar cat
squirrel monkey
spider monkey
titi
indri
howler monkey