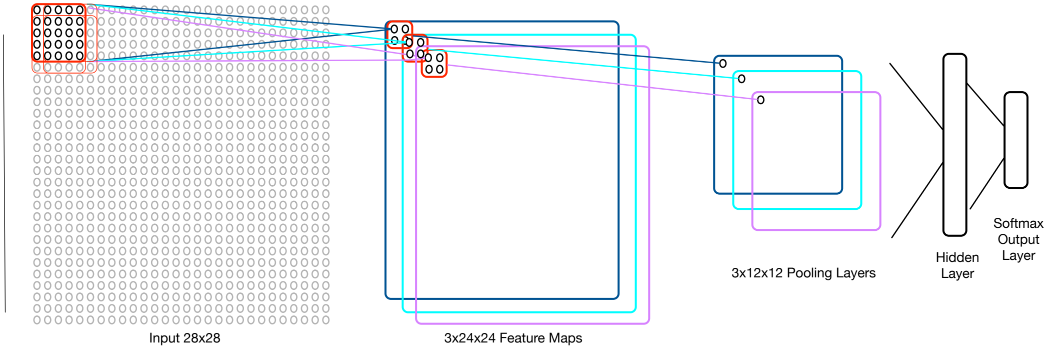
Convolutional Networks 2: Training, deep convolutional networks

Steve Renals

Machine Learning Practical — MLP Lecture 8
8 November 2017 / 13 November 2017

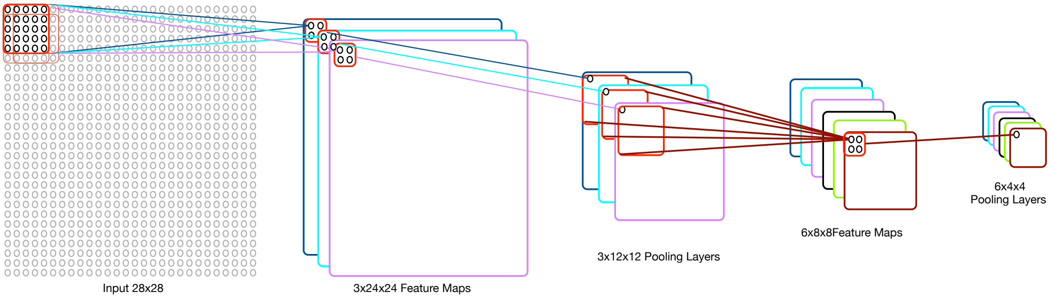
Recap: Convolutional Network



Simple ConvNet:

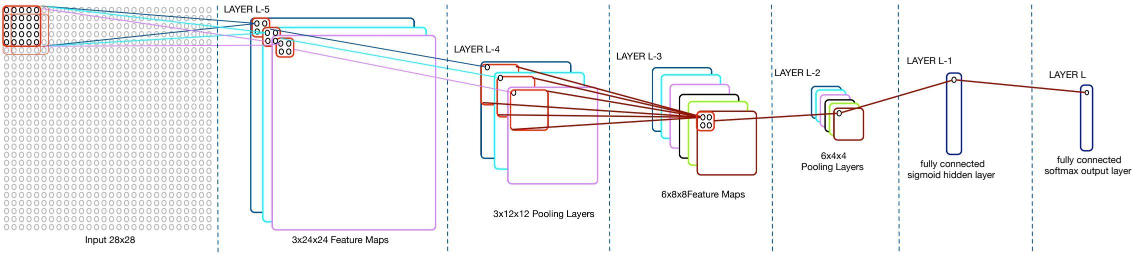
- One convolutional layer with max-pooling
- Final fully connected hidden layer (no sharing weight)
- Softmax output layer

Recap: Stacking convolutional layers



- Local receptive fields
- Weight sharing
- Pooling/subsampling

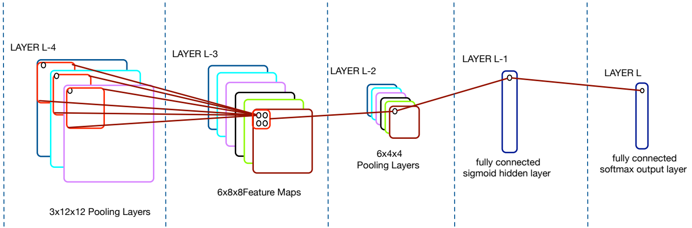
Training Convolutional Networks



BACKPROP GRADIENTS		?	$\mathbf{g}^{L-3} = \mathbf{g}^{L-2}\mathbf{G}^{L-2\mathsf{T}} \circ f'(\mathbf{a}^{L-3})$	$\mathbf{g}^{L-2} = \mathbf{g}^{L-1}\mathbf{W}^{L-1\mathsf{T}}$	$\mathbf{g}^{L-1} = \mathbf{g}^{L}\mathbf{W}^{L\mathsf{T}} \circ f'(\mathbf{a}^{L-1})$	$\mathbf{g}^{L} = \mathbf{h}^{L} - \mathbf{t}^{L}$
WEIGHT GRADIENTS			?	–	$\frac{\partial E}{\partial \mathbf{W}^{L-1}} = \mathbf{h}^{L-2\mathsf{T}}\mathbf{g}^{L-1}$	$\frac{\partial E}{\partial \mathbf{W}^{L}} = \mathbf{h}^{L-1\mathsf{T}}\mathbf{g}^{L}$

Remember: $g_j^\ell = \partial E / \partial a_j^\ell$, is the error signal for unit j in layer ℓ

Training Convolutional Networks – Pooling Layer



	LAYER L-4	LAYER L-3	LAYER L-2	LAYER L-1	LAYER L
BACKPROP GRADIENTS	?	$\mathbf{g}^{L-3} = \mathbf{g}^{L-2}\mathbf{G}^{L-2^{\mathsf{T}}} \circ f'(\mathbf{a}^{L-3})$	$\mathbf{g}^{L-2} = \mathbf{g}^{L-1}\mathbf{W}^{L-1^{\mathsf{T}}}$	$\mathbf{g}^{L-1} = \mathbf{g}^{L}\mathbf{W}^{L^{\mathsf{T}}} \circ f'(\mathbf{a}^{L-1})$	$\mathbf{g}^{L} = \mathbf{h}^{L} - \mathbf{t}^{L}$
WEIGHT GRADIENTS		?	–	$\frac{\partial E}{\partial \mathbf{W}^{L-1}} = \mathbf{h}^{L-2^{\mathsf{T}}}\mathbf{g}^{L-1}$	$\frac{\partial E}{\partial \mathbf{W}^{L}} = \mathbf{h}^{L-1^{\mathsf{T}}}\mathbf{g}^{L}$

G is a "pseudo-weight matrix" for max-pooling which is set during the forward propagation:

$G_{ba} = 1$ if feature map unit b is contained in max-pool a and is the maximum value for the current input.

Note that **G** is different for each item in the mini-batch.

Training Convolutional Networks – Convolutional Layer Weight Update

To update the shared weights of the convolutional layer, we take account of all units to which a shared weight is connected, by summing over the convolutional units:

$$\frac{\partial E}{\partial w_{k,\ell}^{L-3}} = \sum_{i=0}^{D-1} \sum_{j=0}^{D-1} \frac{\partial E}{\partial a_{i,j}^{L-3}} \frac{\partial a_{i,j}^{L-3}}{\partial w_{k,\ell}^{L-3}}$$

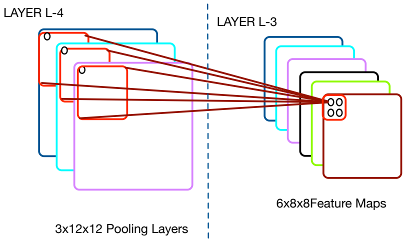
where the kernel has dimension $m \times m$, the feature map has dimension $D \times D$ and $a_{i,j}$ is the activation of the (i,j)th unit in the feature map (see slide 16 in lecture 7):

$$a_{i,j}^{L-3} = \sum_{r=0}^{m-1} \sum_{s=0}^{m-1} w_{r,s}^{L-3} h_{i+r,j+s}^{L-4} + b^{L-3}$$

Recalling that $g_{i,j}^{L-3} = \partial E / \partial a_{i,j}^{L-3}$, then we have:

$$\frac{\partial E}{\partial w_{k,\ell}^{L-3}} = \sum_{i=0}^{D-1} \sum_{j=0}^{D-1} g_{i,j}^{L-3} h_{i+k,j+\ell}^{L-4} = \boldsymbol{g}^{L-3} \otimes \boldsymbol{h}^{L-4}(k,\ell)$$

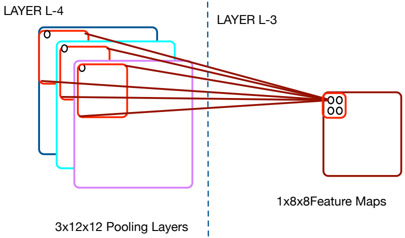
Training Convolutional Networks – Convolutional Layer



	LAYER L-4	LAYER L-3
BACKPROP GRADIENTS	?	$\mathbf{g}^{L-3} = \mathbf{g}^{L-2}\mathbf{G}^{L-2\mathsf{T}} \circ f'(\mathbf{a}^{L-3})$
WEIGHT GRADIENTS		$\frac{\partial E}{\partial \mathbf{W}^{L-3}} = \mathbf{g}^{L-3} \otimes \mathbf{h}^{L-4}$

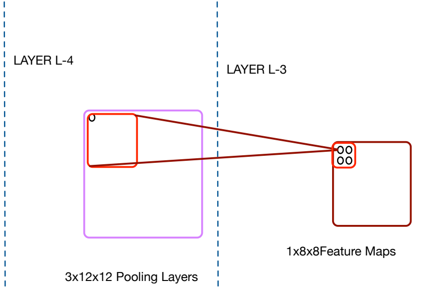
Training the convolutional layer is more complicated

Training Convolutional Networks – Convolutional Layer



BACKPROP GRADIENTS	?	$\mathbf{g}^{L-3} = \mathbf{g}^{L-2}\mathbf{G}^{L-2\mathsf{T}} \circ f'(\mathbf{a}^{L-3})$
WEIGHT GRADIENTS		$\frac{\partial E}{\partial \mathbf{W}^{L-3}} = \mathbf{g}^{L-3} \otimes \mathbf{h}^{L-4}$

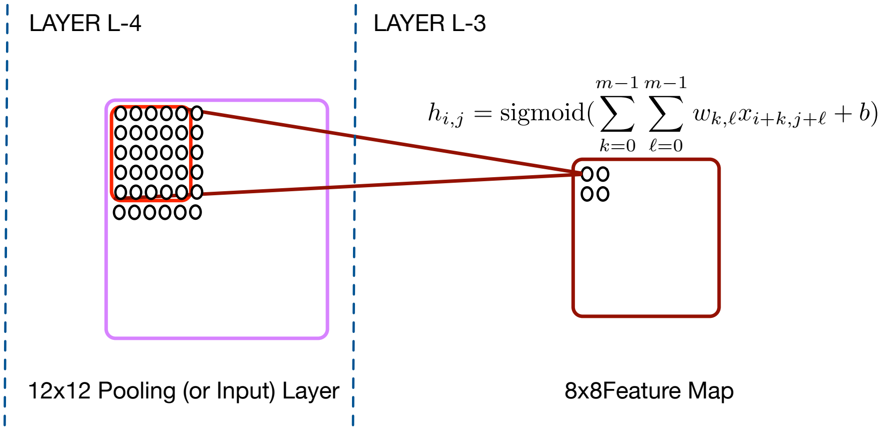
Only need to consider one pooling layer



BACKPROP GRADIENTS	?	$\mathbf{g}^{L-3} = \mathbf{g}^{L-2}\mathbf{G}^{L-2^\mathsf{T}} \circ f'(\mathbf{a}^{L-3})$
WEIGHT GRADIENTS		$\frac{\partial E}{\partial \mathbf{W}^{L-3}} = \mathbf{g}^{L-3} \otimes \mathbf{h}^{L-4}$

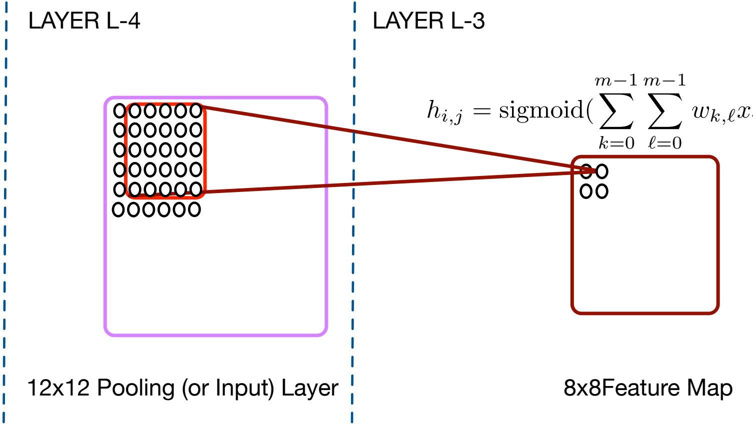
Simplify by only considering one feature map

Convolutional Layer – Forward Pass



Forward pass: each hidden unit connected to a region of input units (receptive field)

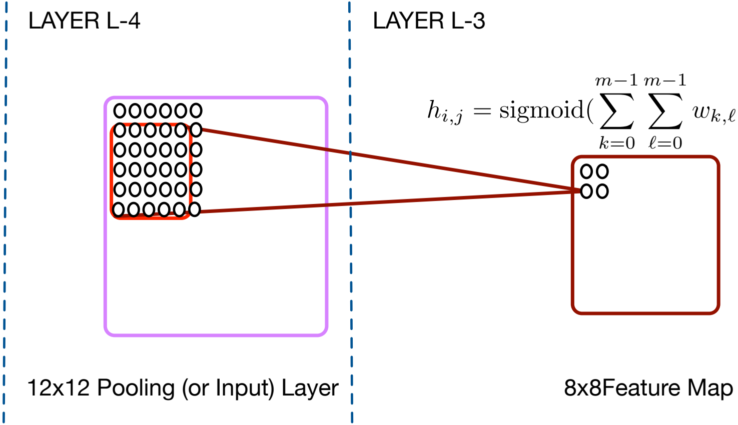
Convolutional Layer – Forward Pass



$$h_{i,j} = \text{sigmoid}\left(\sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} w_{k,\ell} x_{i+k,j+\ell} + b\right)$$

Forward pass: each hidden unit connected to a region of input units (receptive field)

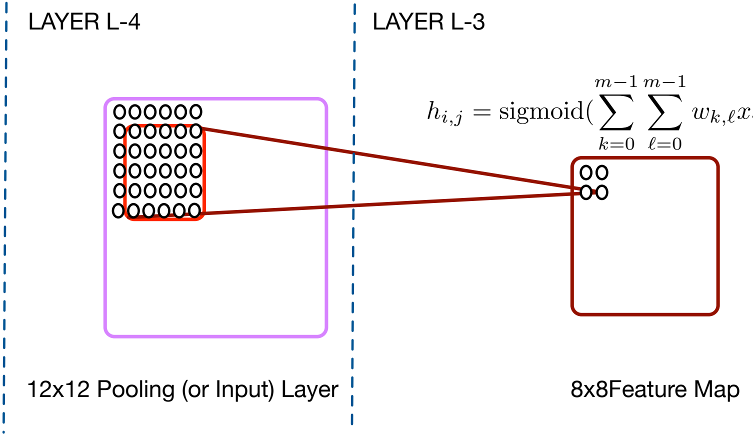
Convolutional Layer – Forward Pass



$$h_{i,j} = \text{sigmoid}\left(\sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} w_{k,\ell} x_{i+k,j+\ell} + b\right)$$

Forward pass: each hidden unit connected to a region of input units (receptive field)

Convolutional Layer – Forward Pass



$$h_{i,j} = \text{sigmoid}\left(\sum_{k=0}^{m-1}\sum_{\ell=0}^{m-1} w_{k,\ell}\, x_{i+k,j+\ell} + b\right)$$

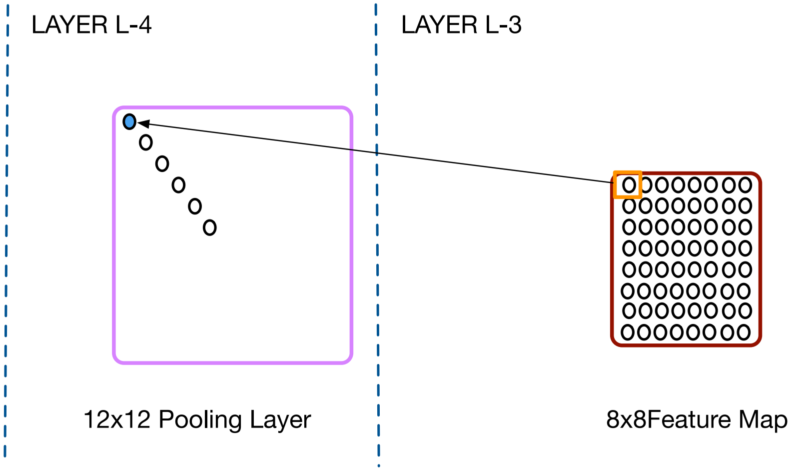
Forward pass: each hidden unit connected to a region of input units (receptive field)

Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.

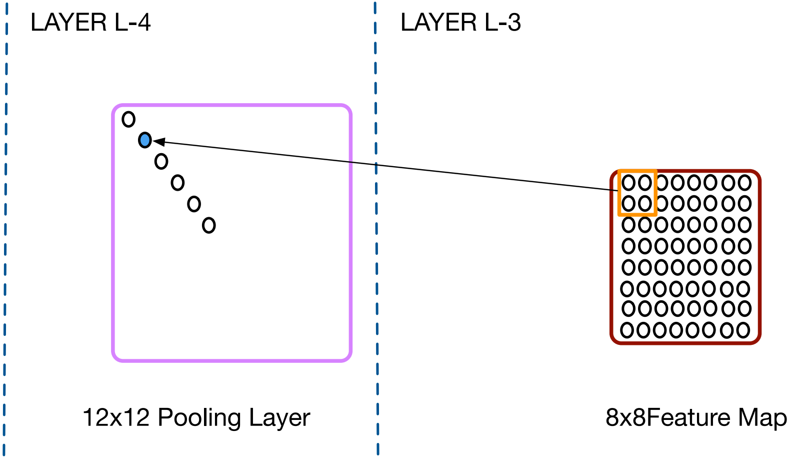
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.



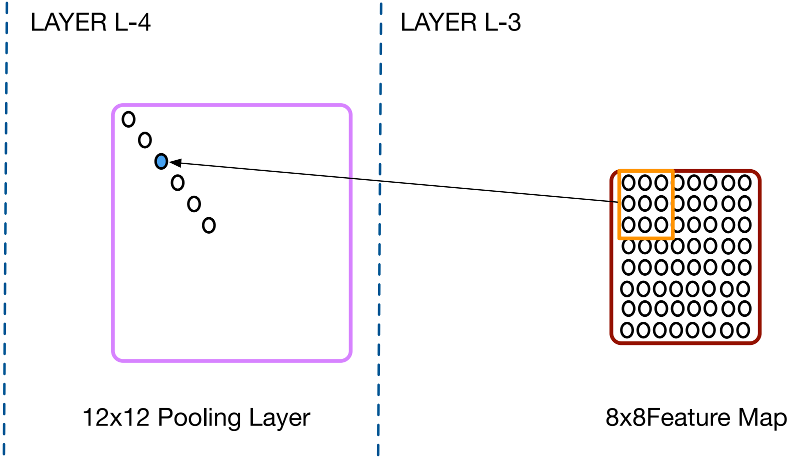
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.



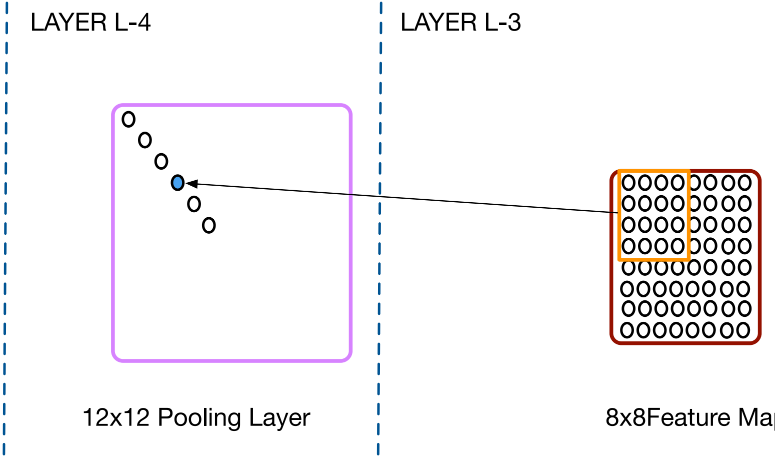
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.



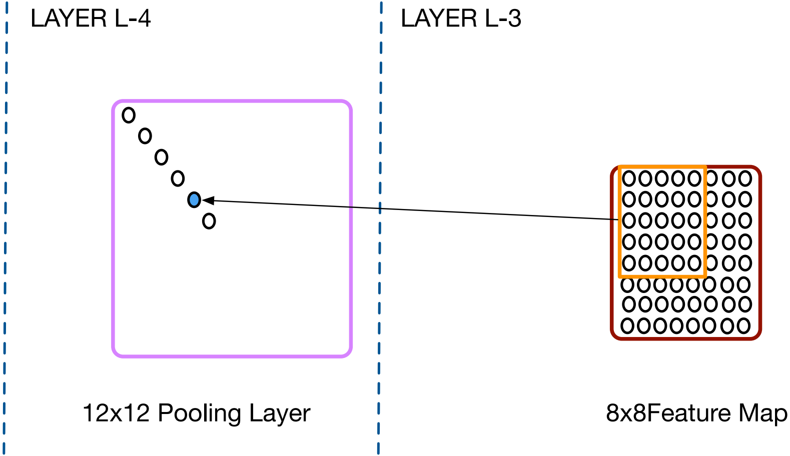
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.



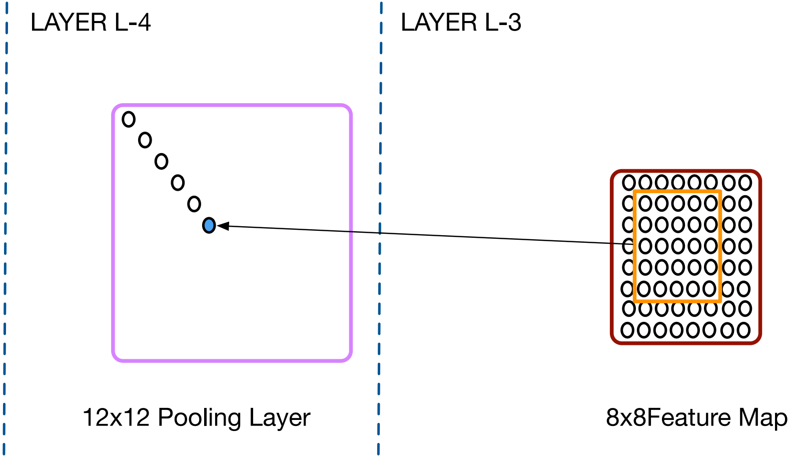
Convolutional Layer – Backward Pass

Backward pass: consider the region of hidden units connected to each input unit.



Convolutional Layer – Backward Pass

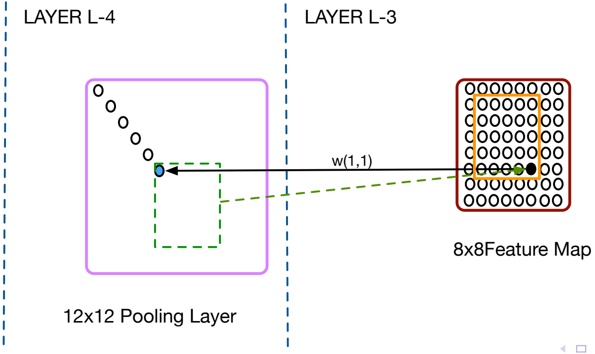
Backward pass: consider the region of hidden units connected to each input unit.



As usual we want to back-propagate the gradients:

$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

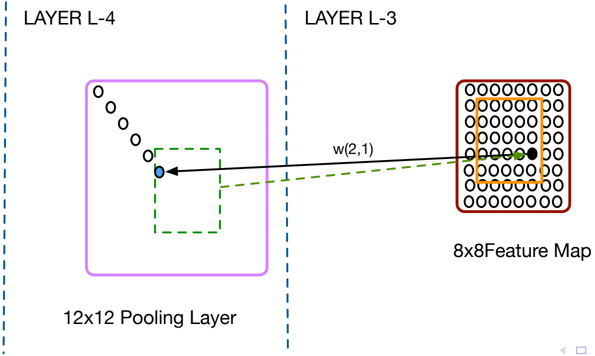


Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

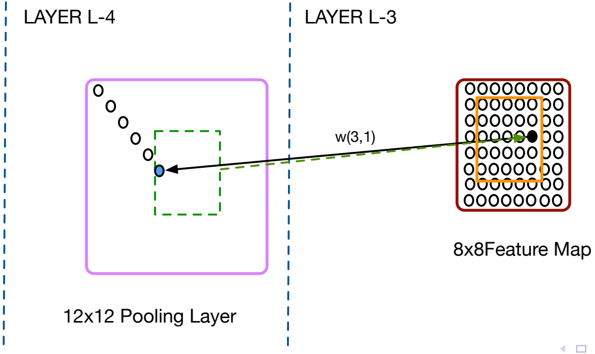


Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

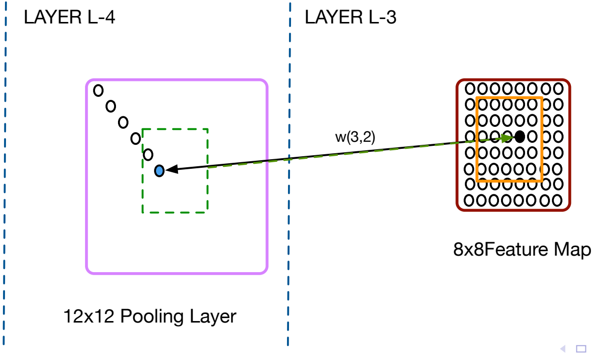


Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

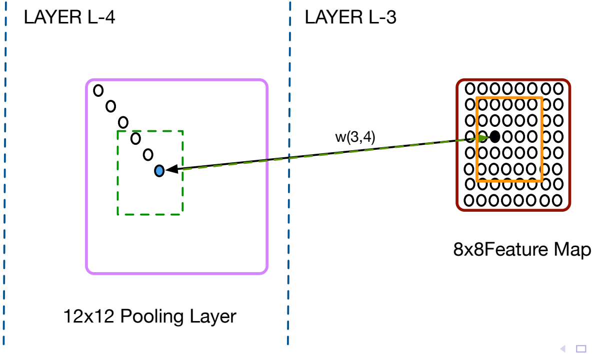


Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

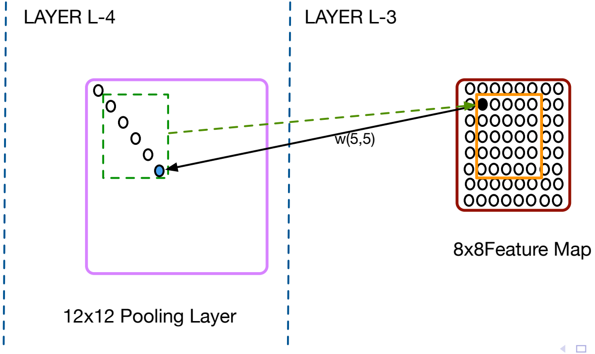


Convolutional Layer – Back Propagation

As usual we want to back-propagate the gradients:

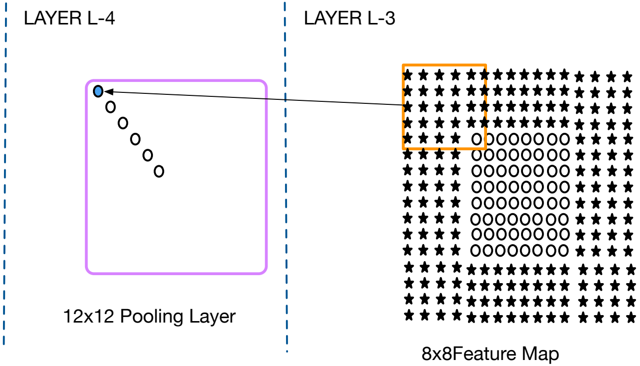
$$g_s^{L-4} = \sum_{j \in \text{connected to } s} w_{js} g_j^{L-3} f'(a_s)$$

Look at the shared weights used for back prop:

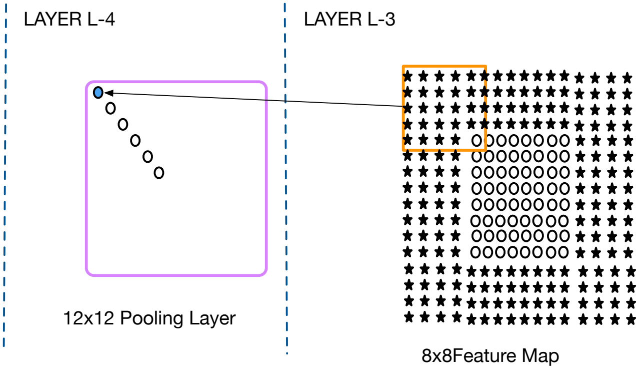


Backprop as convolution

If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



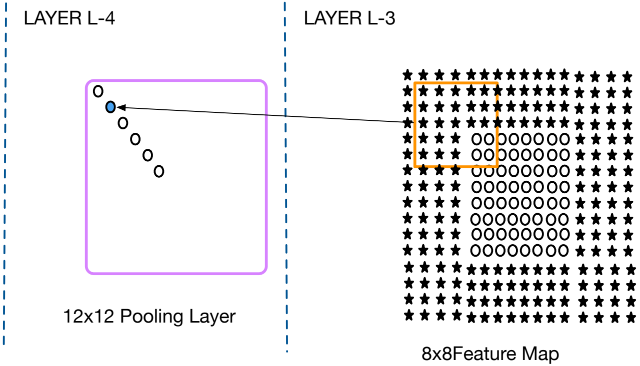
If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* (flipped)

Backprop as convolution

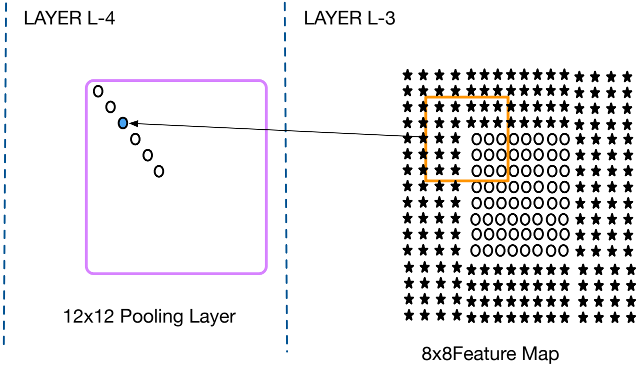
If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* (flipped)

Backprop as convolution

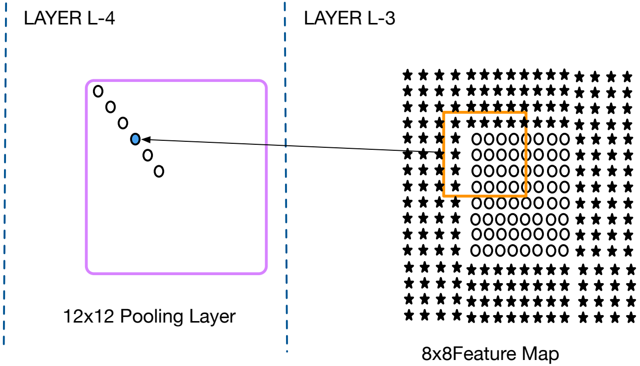
If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* (flipped)

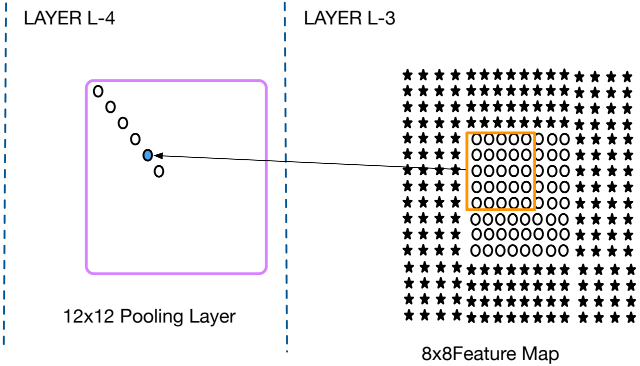
Backprop as convolution

If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* (flipped)

If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.



Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* (flipped)

Convolutional Layer – Back Prop

Back-propagation in a convolutional layer, is also a convolution.
But we have to *rotate* the weight matrix $\mathbf{W}$ by 180° (flip the weight matrix), $\mathbf{W}^R$
Using the convolution operator we saw we can write the forward pass as:

$$\mathbf{a}^{L-3} = \mathbf{W}^{L-3} * \mathbf{h}^{L-4} + \mathbf{b}^{L-3} \quad ; \quad \boldsymbol{h}^{L-3} = f(\boldsymbol{a}^{L-3})$$

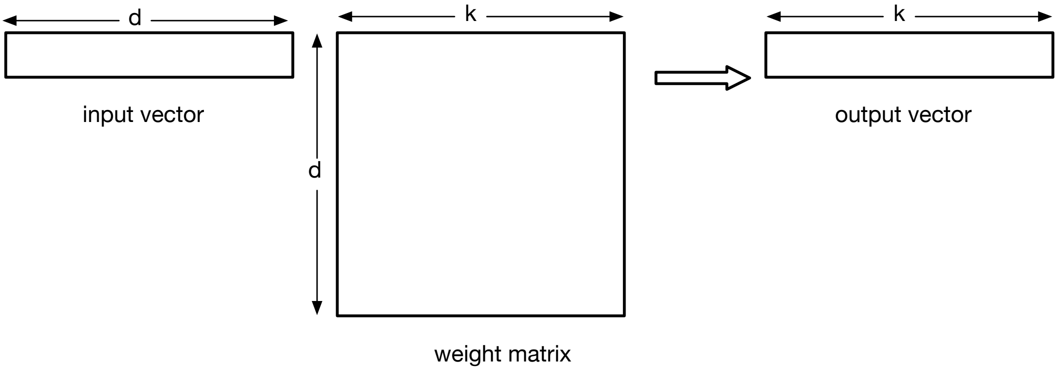
And we can write the back-propagation as:

$$\boldsymbol{g}^{L-4} = \mathbf{W}^{L-3^R} * \boldsymbol{g}^{L-3} \circ f'(\mathbf{a}^{\mathbf{L-4}})$$

- The backward pass flips the weight matrix compared with the forward pass
- If the forward pass is a correlation, the backward pass is a convolution
- If the forward pass is a convolution, the backward pass is a correlation
- (Either is OK)

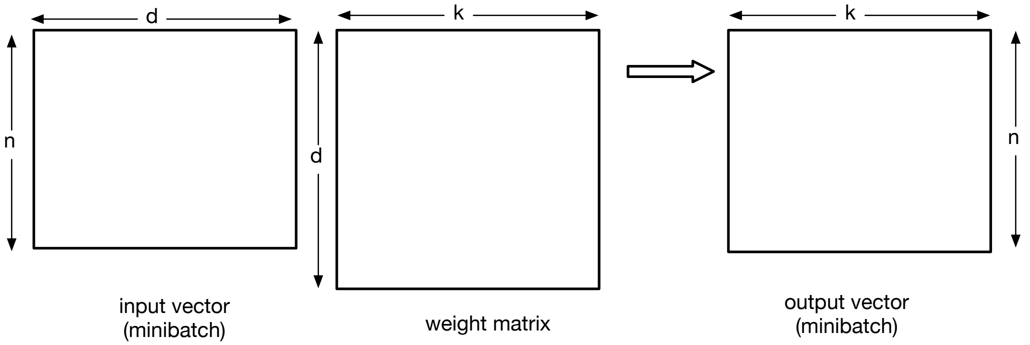
Implementing multilayer networks

Example at a time:



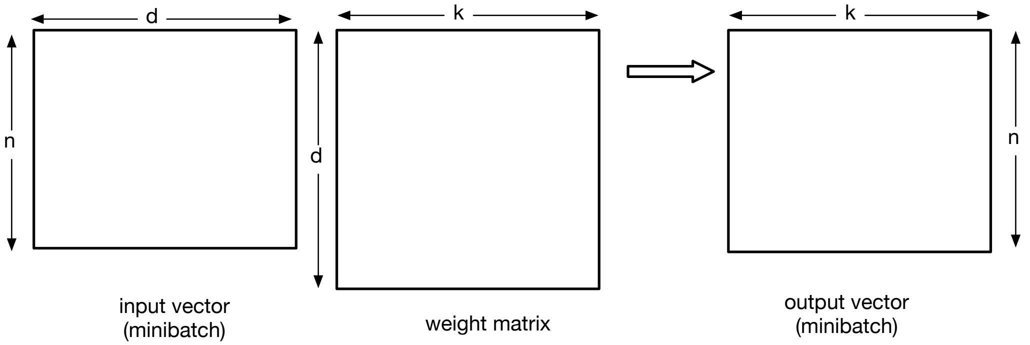
Implementing multilayer networks

Minibatch:



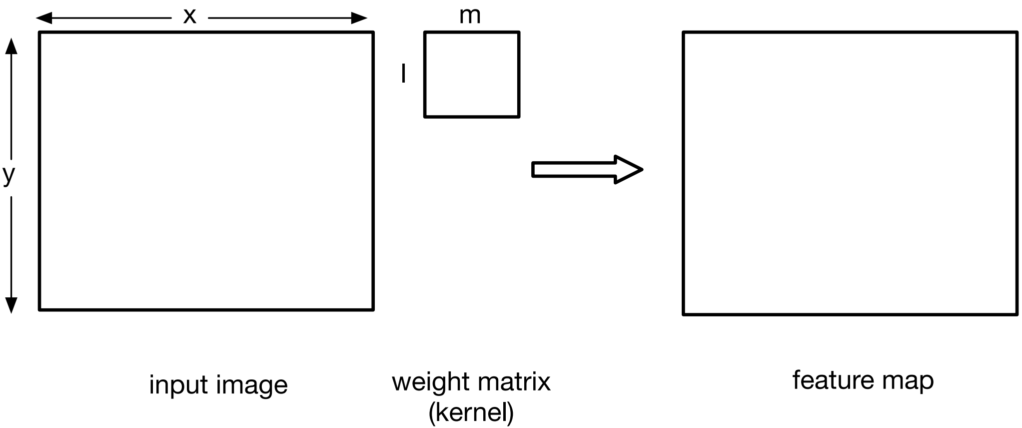
Implementing multilayer networks

Minibatch:



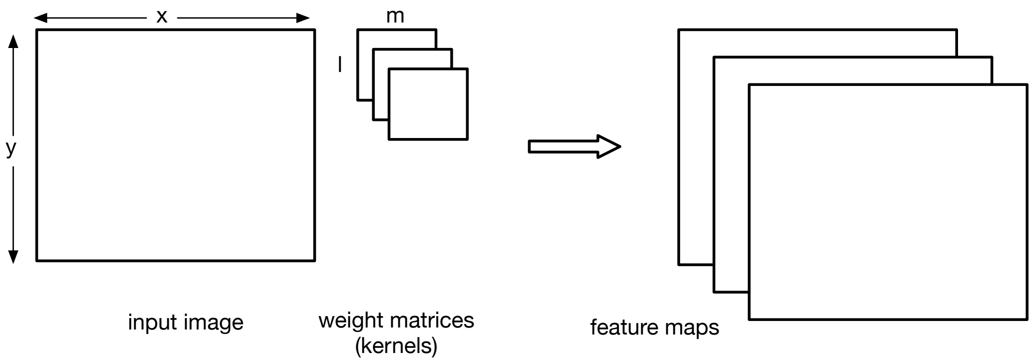
input dimension x minibatch: Represent each layer as a 2-dimension matrix, where each row corresponds to a training example, and the number of minibatch examples is the number of rows

Example at a time, single input image, single feature map:

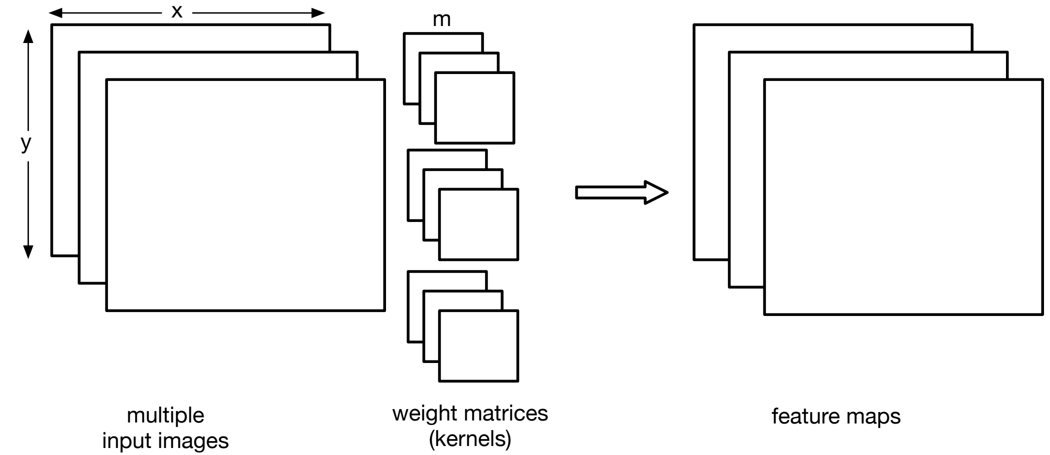


Implementing Convolutional Networks

Example at a time, single input image, multiple feature map:

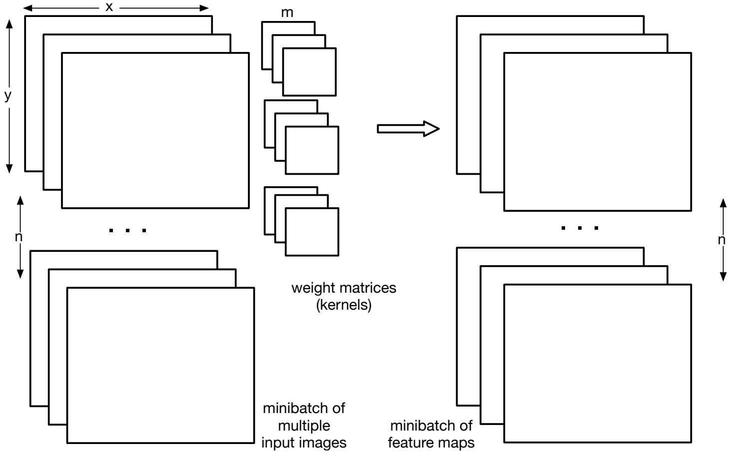


Example at a time, multiple input images, multiple feature map:



Implementing Convolutional Networks

Minibatch, multiple input images, multiple feature map:



Implementing Convolutional Networks

- Inputs / layer values:
  - Each input image (and convlutional and pooling layer) is 2-dimensions (x,y)
  - If we have multiple feature maps, then that is a third dimension
  - And the minibatch adds a fourth dimension
  - Thus we represent each input (layer values) using a 4-dimension *tensor* (array): (minibatch-size, num-fmaps, x, y)
- Weight matrices (kernels)
  - Each weight matrix used to scan across an image has 2 spatial dimensions (x,y)
  - If there are multiple feature maps to be computed, then that is a third dimension
  - Multiple input feature maps adds a fourth dimension
  - Thus the weight matrices are also represented using a 4-dimension tensor: (num-fmaps-in, num-fmaps-out, x, y)

4D tensors in numpy

Both forward and back prop thus involves multiplying 4D tensors. There are various ways to do this:

- Explicitly loop over the dimensions: this results in simpler code, but can be inefficient. Although using cython to compile the loops as C can speed things up
- Serialisation: By replicating input patches and weight matrices, it is possible to convert the required 4D tensor multiplications into a large dot product. Requires careful manipulation of indices!
- Convolutions: use explicit convolution functions for forward and back prop, rotating for the backprop

Deep convolutional networks

Krizhevsky, Sutskever and Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS-2012.

http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

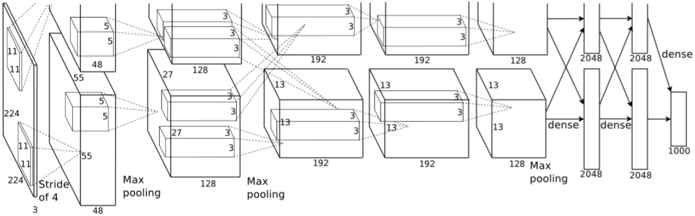


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

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

ImageNet Classification ("VGGNet")

Simonyan and Zisserman, "Very Deep Convolutional Networks for Large-Scale Visual Recognition", ILSVRC-2014. http://www.robots.ox.ac.uk/~vgg/research/very_deep/

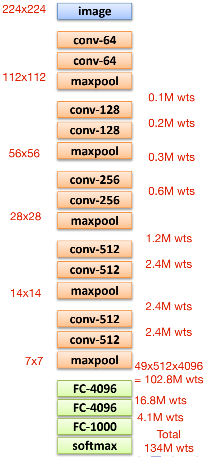
Network Design

Key design choices:

- 3x3 conv. kernels – very small
- conv. stride 1 – no loss of information

Other details:

- Rectification (ReLU) non-linearity
- 5 max-pool layers (x2 reduction)
- no normalisation
- 3 fully-connected (FC) layers



Deep Residual Learning ("ResNets")

He et al, "Deep Residual Learning for Image Recognition", CVPR-2016.

http://arxiv.org/abs/1512.03385

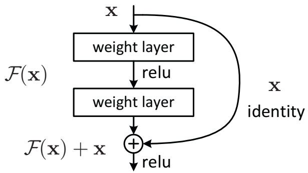
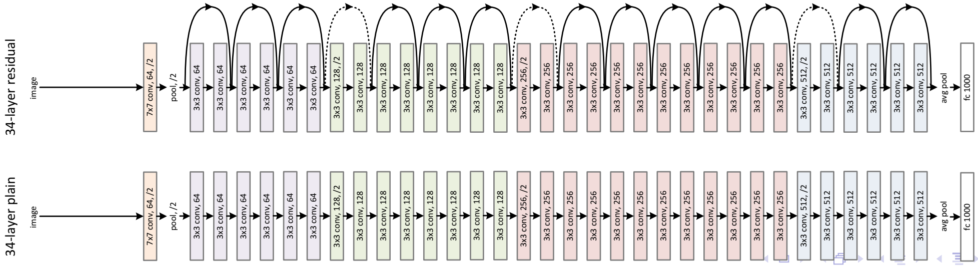


Figure 2. Residual learning: a building block.

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**



Summary

- Convolutional networks include local receptive fields, weight sharing, and pooling leading
- Backprop training can also be implemented as a "reverse" convolutional layer (with the weight matrix rotated)
- Implement using 4D tensors:
  - Inputs / Layer values: minibatch-size, number-fmaps, x, y
  - Weights: number-fmaps-in, number-fmaps-out, x, y
- Reading:
  Goodfellow et al, *Deep Learning* (ch 9)
  http://www.deeplearningbook.org/contents/convnets.html