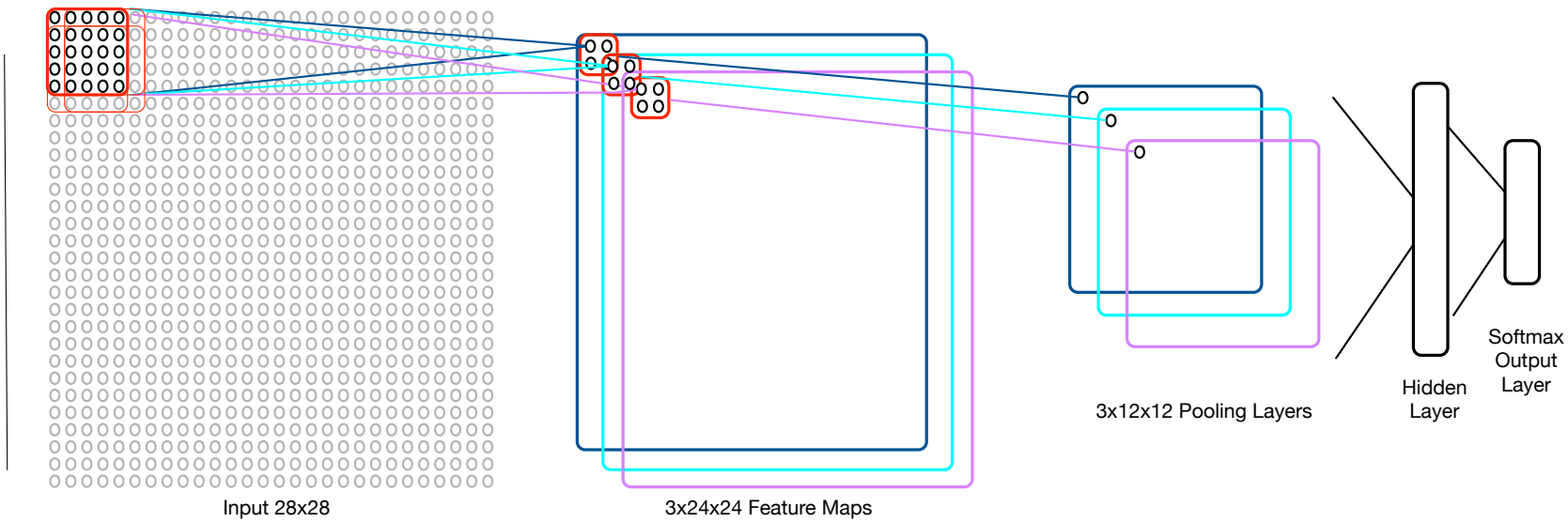
Convolutional Networks (part 2)

Steve Renals

Machine Learning Practical — MLP Lecture 8
11 November 2015

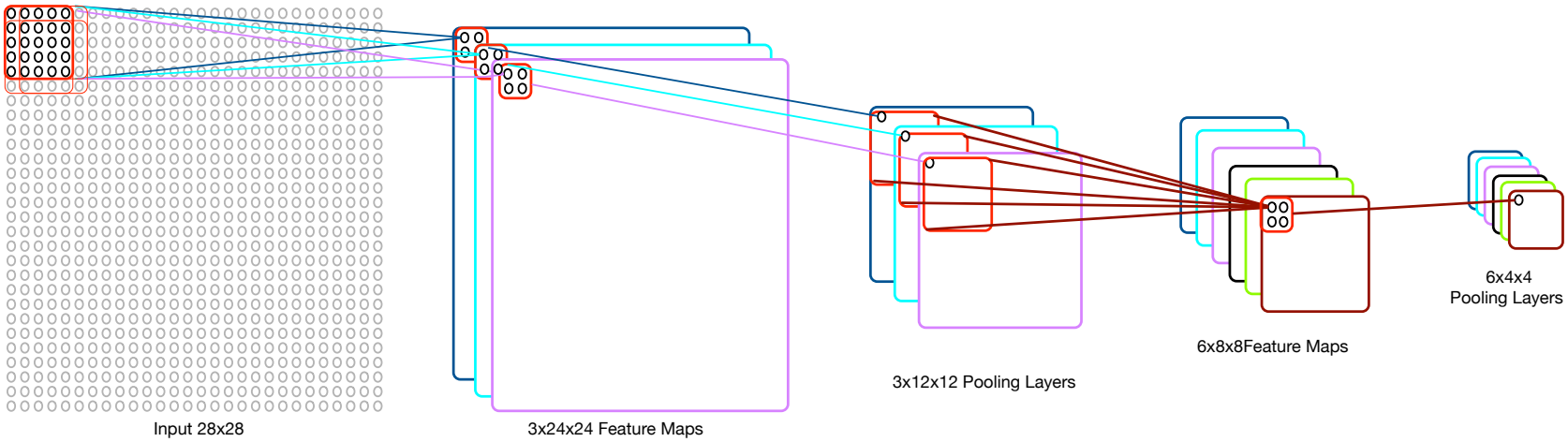
Recap: Convolutional Network



Simple ConvNet:

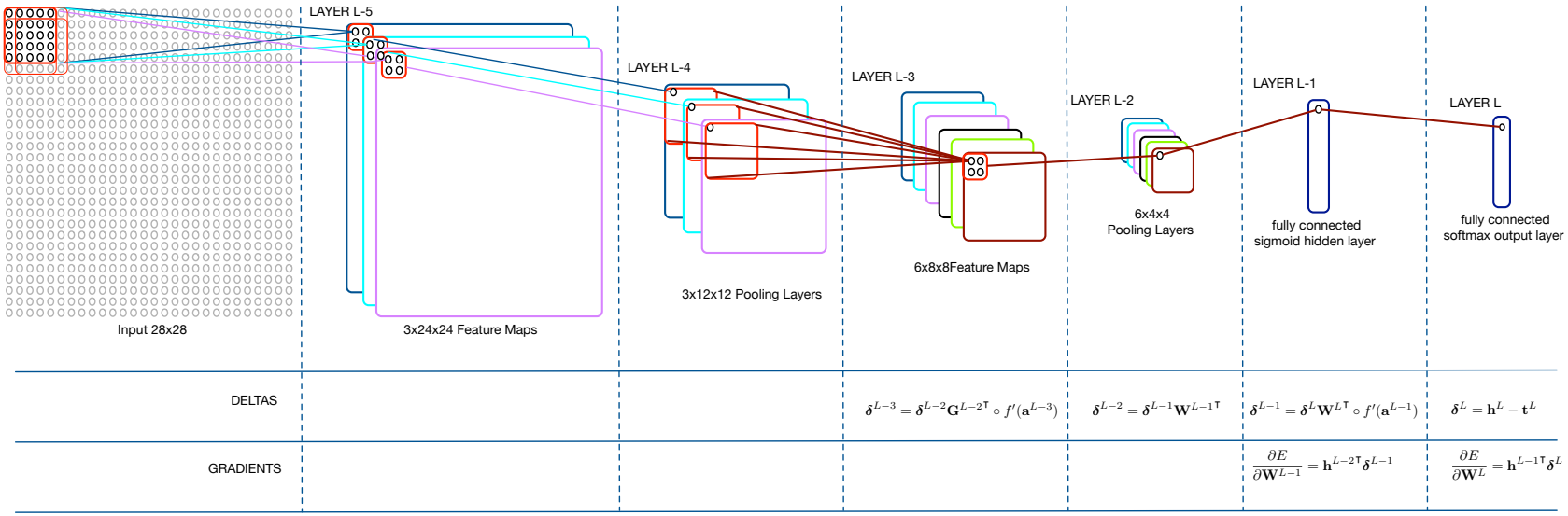
- One convolutional layer with max-pooling
- Final fully connected hidden layer (no sharing weight)
- Softmax output layer

Recap: Stacking convolutional layers



- Local receptive fields
- Weight sharing
- Pooling/subsampling

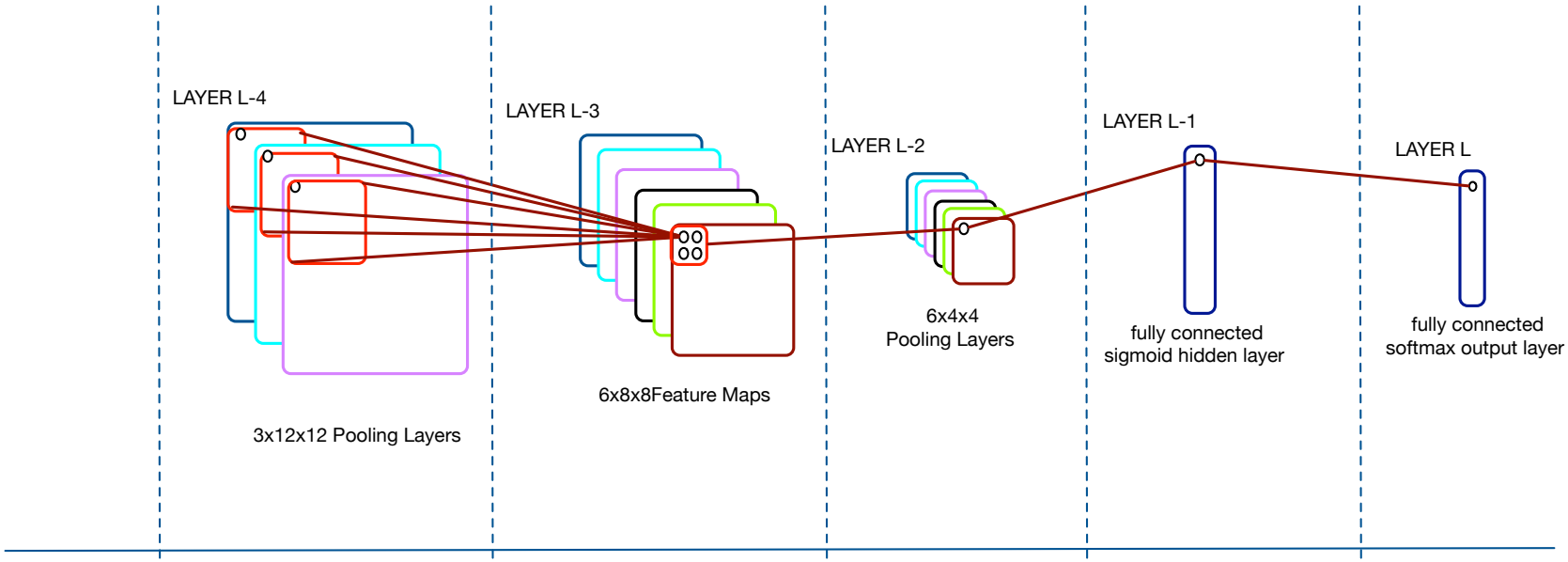
Training Convolutional Networks – Pooling Layer



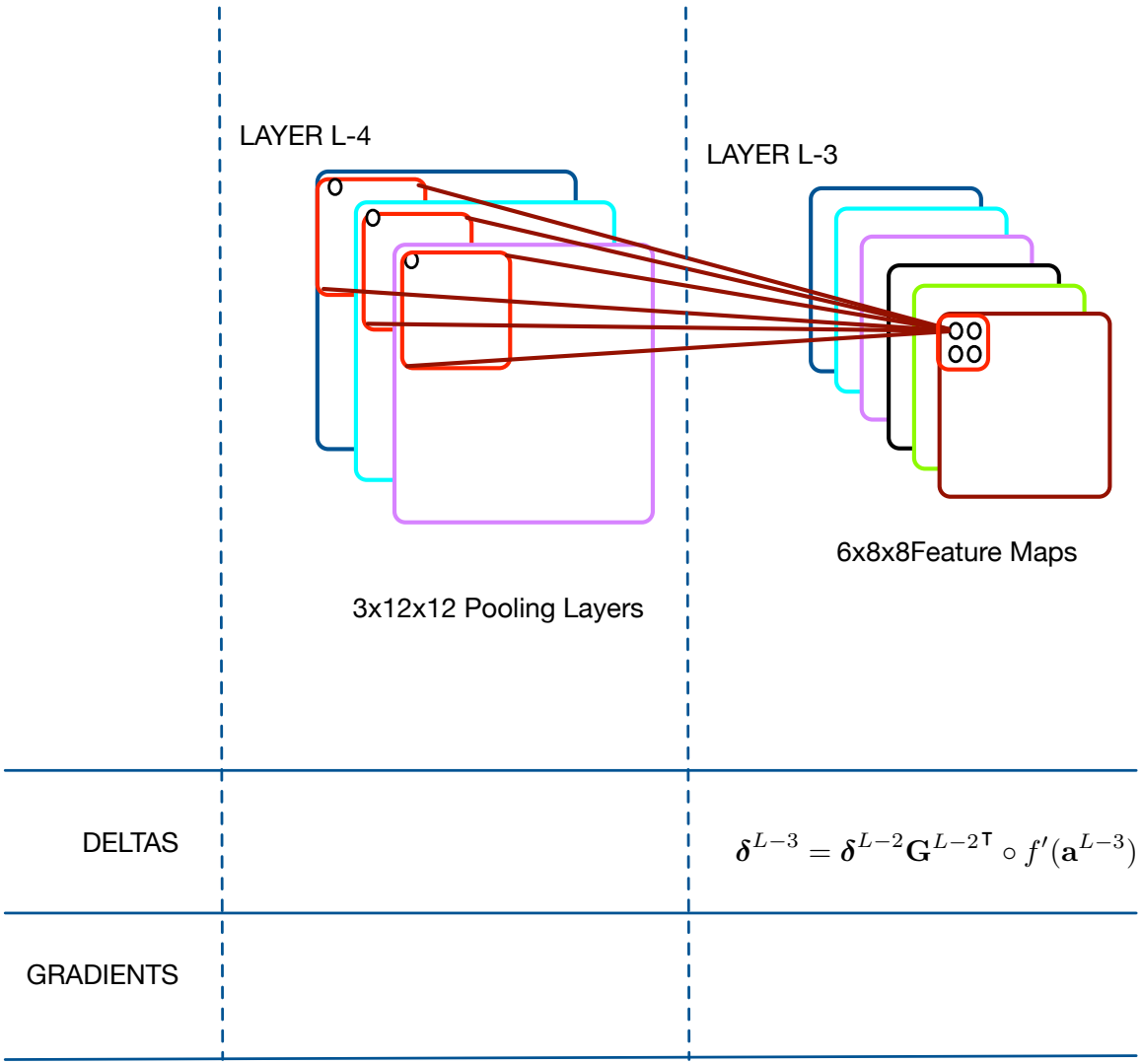
Notes:

- Matrix **G** is a "pseudo-weight matrix" for max-pooling which is set during the forward propagation: element $G_{ba} = 1$ if feature map unit b is contained in max-pool a and is the maximum value for the current input. Unlike the real weight matrices, note that **G** is different for each item in the minibatch

Training Convolutional Networks – Pooling Layer

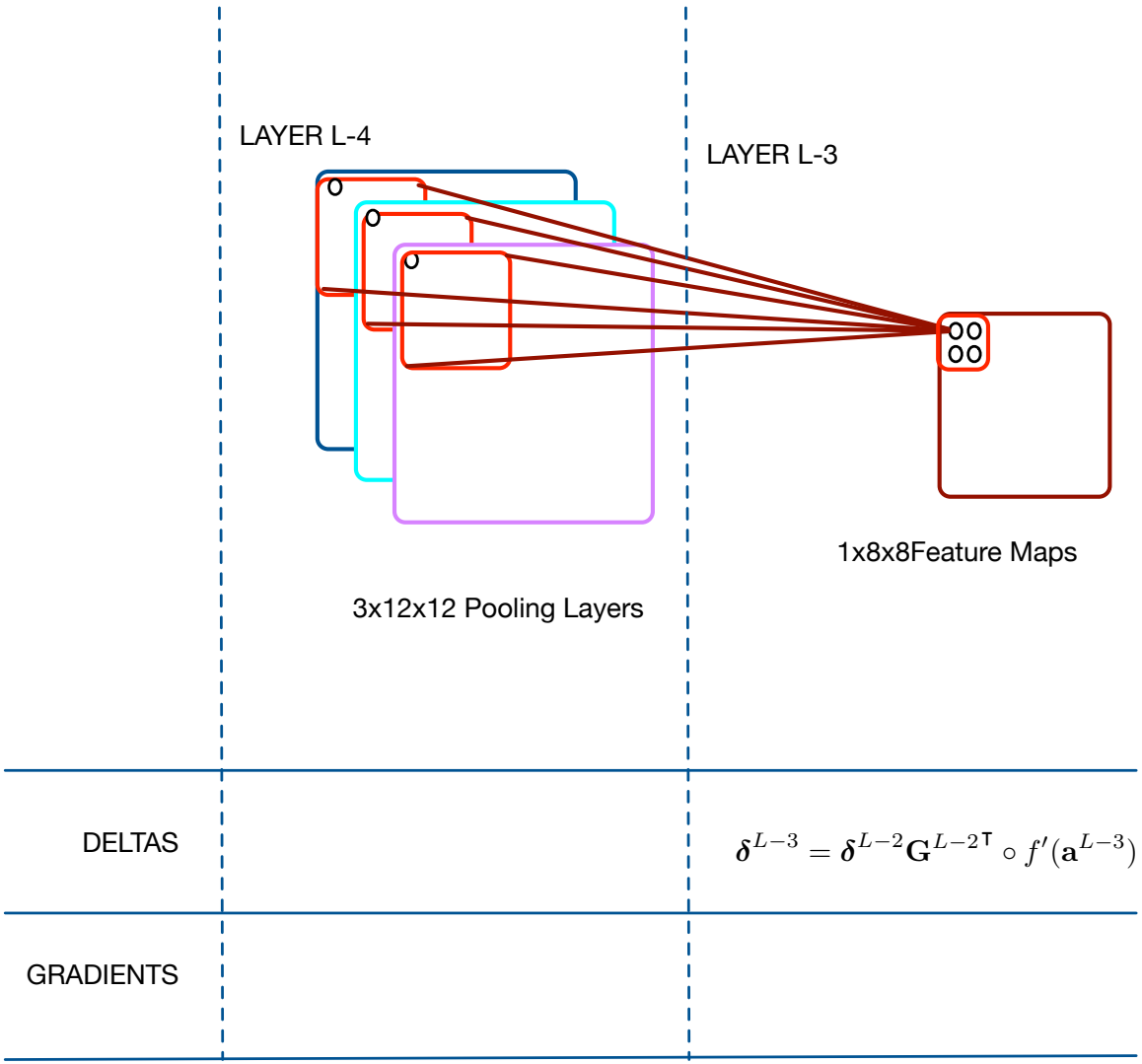


Training Convolutional Networks – Convolutional Layer



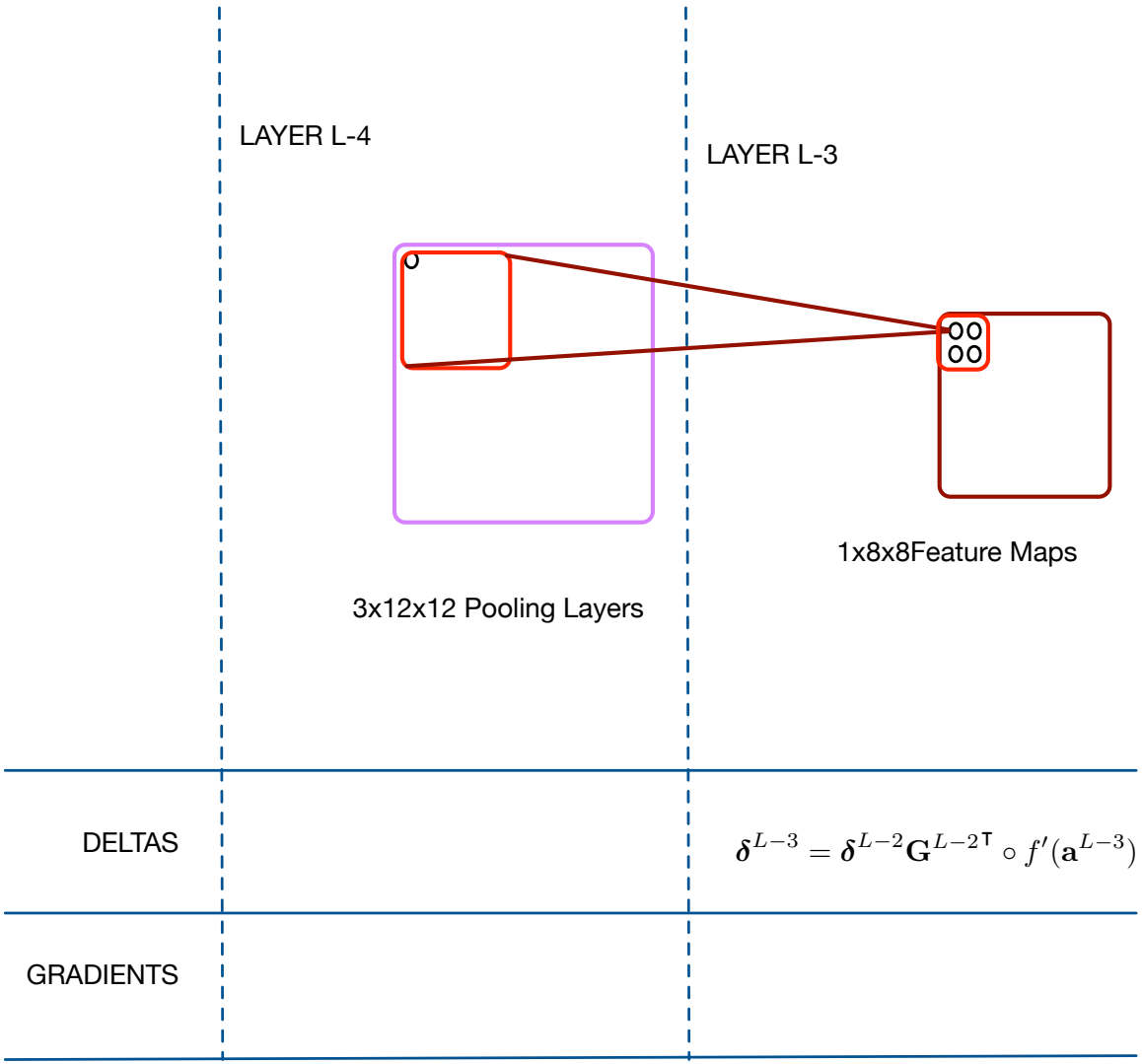
Training the convolutional layer is more complicated

Training Convolutional Networks – Convolutional Layer



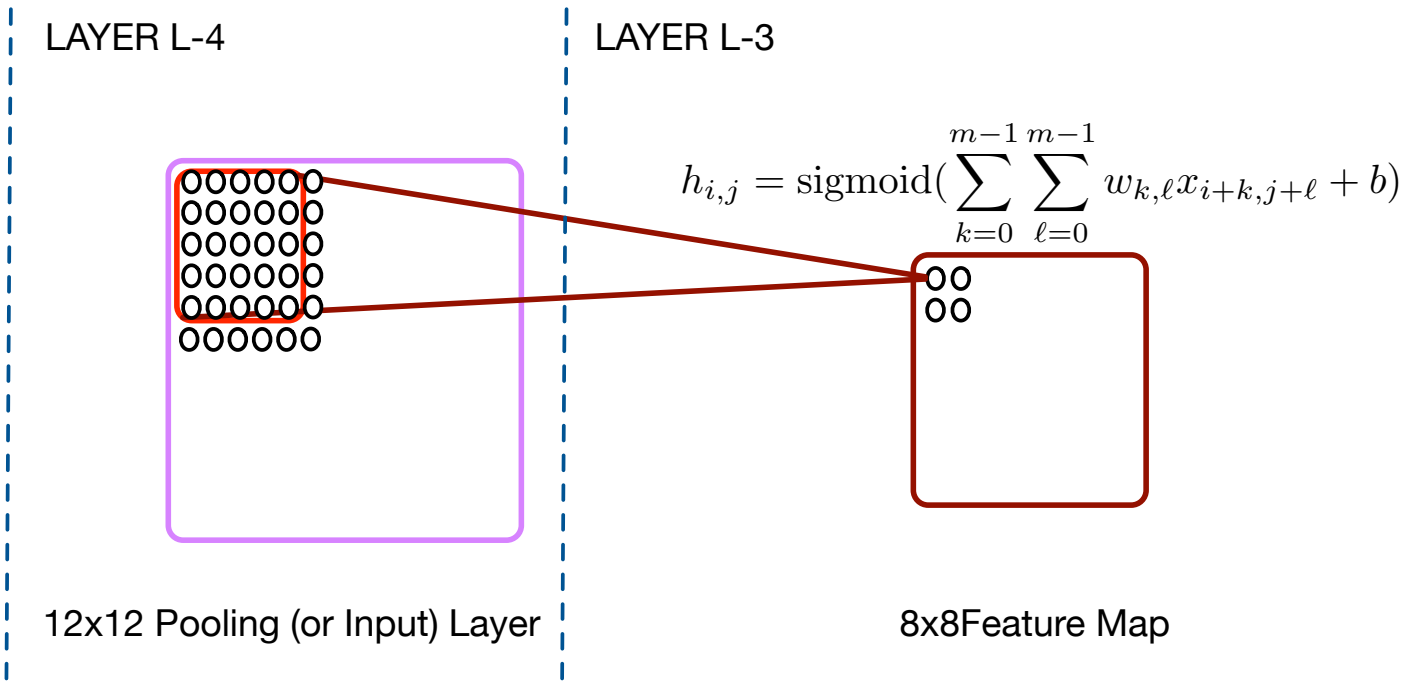
Only need to consider one pooling layer

Training Convolutional Networks – Convolutional Layer



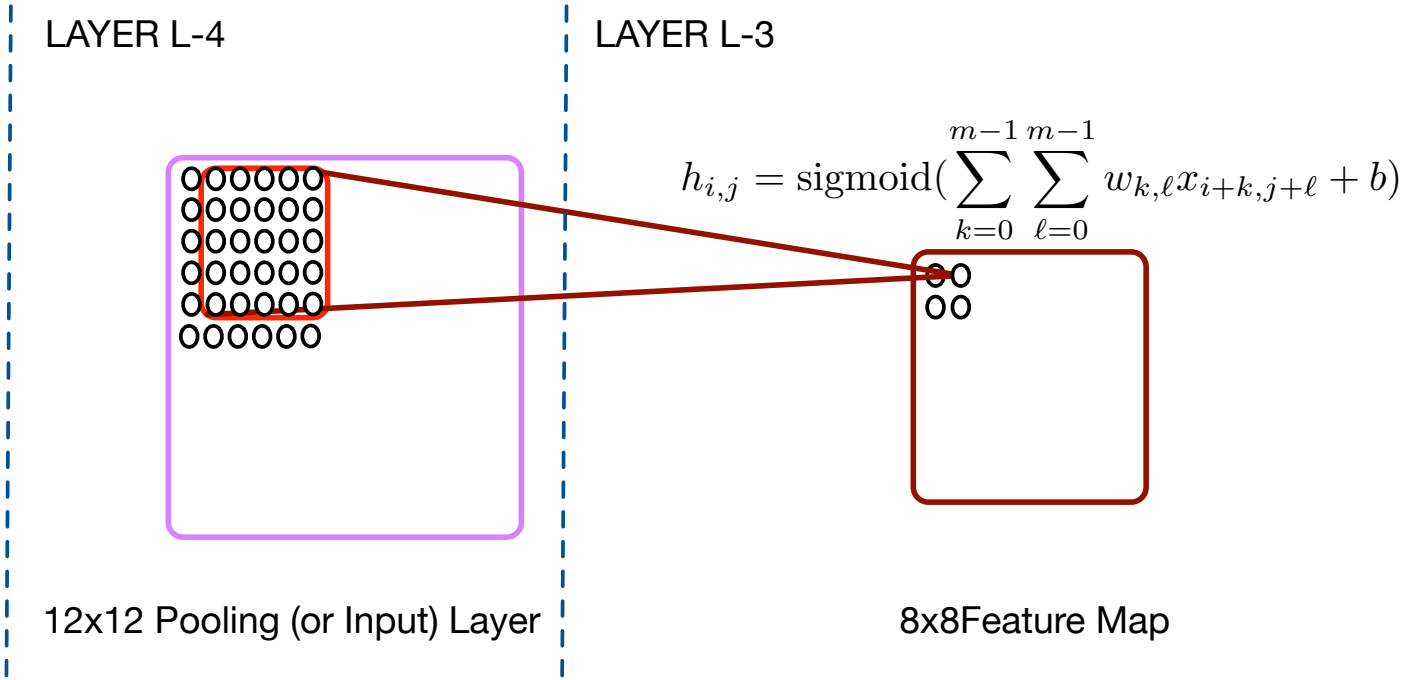
Simplify by only considering one feature map

Convolutional Layer – Forward Prop



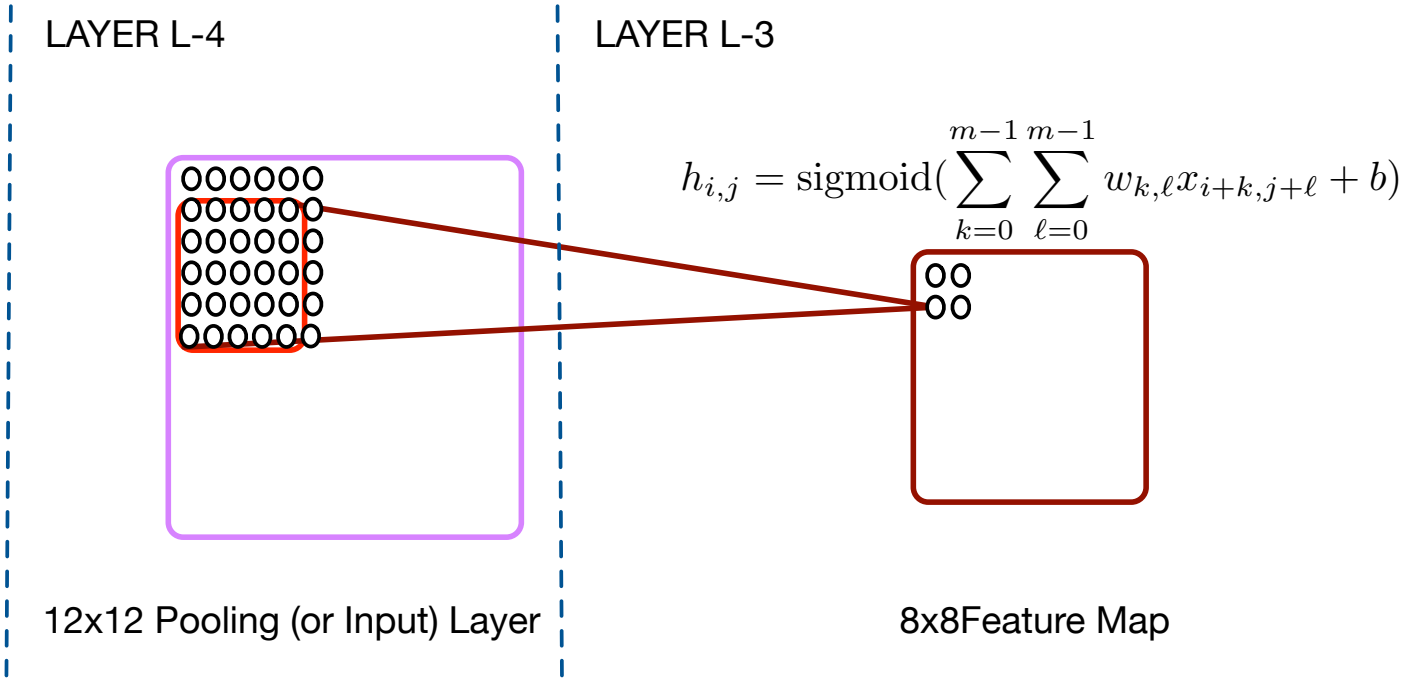
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)

Convolutional Layer – Forward Prop



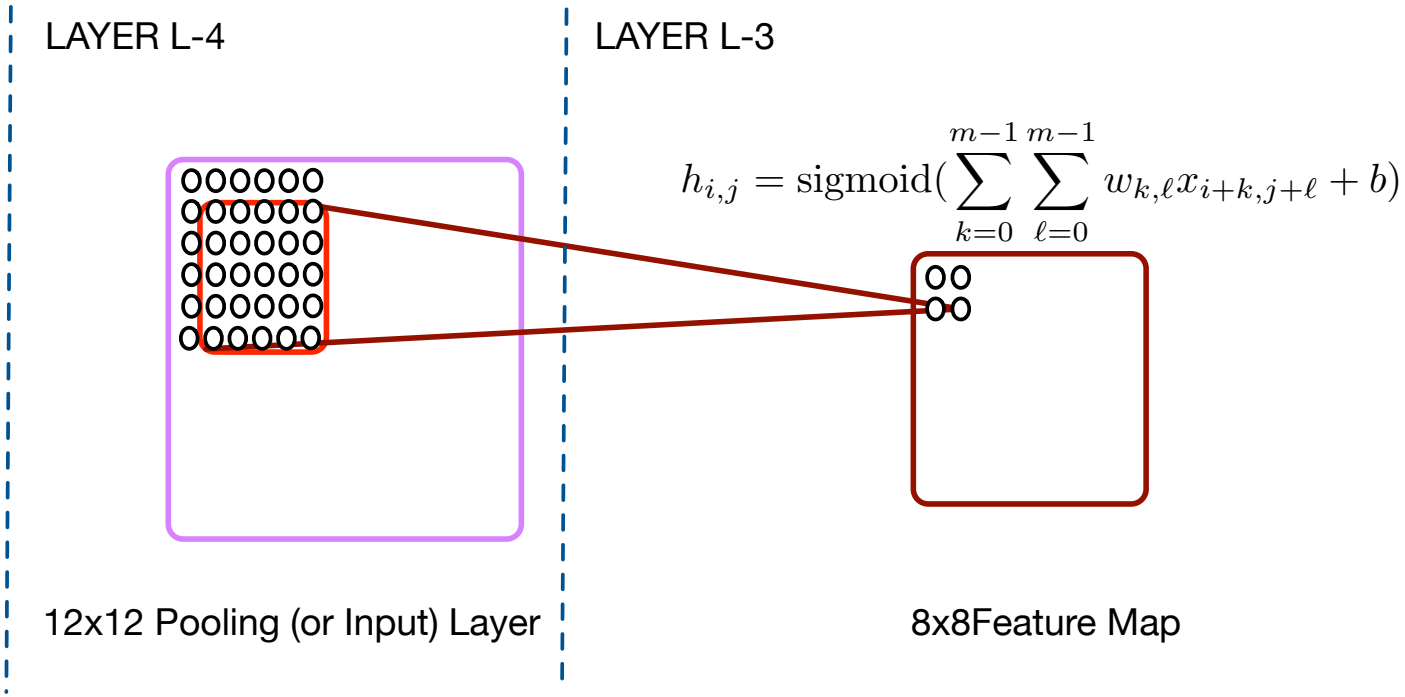
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)

Convolutional Layer – Forward Prop



In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)

Convolutional Layer – Forward Prop



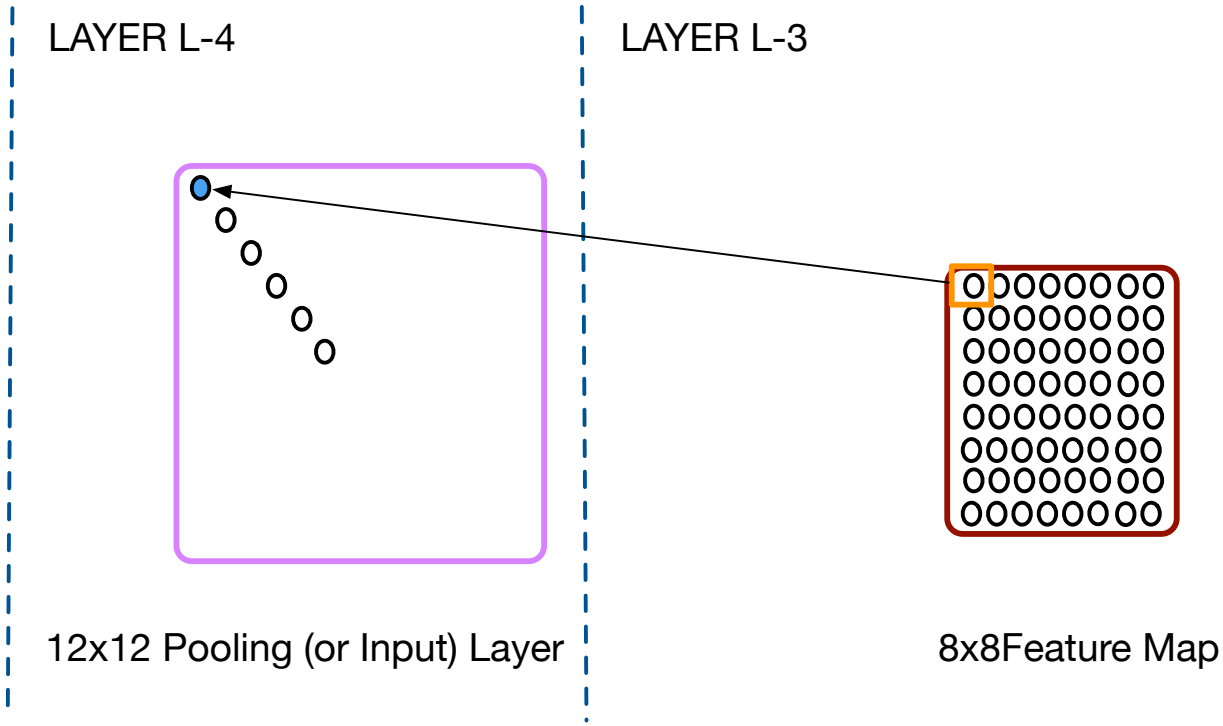
In the forward propagation, each hidden unit is connected to a region of input units (the receptive field)

Convolutional Layer – Back Prop

For backprop we need to consider the region of hidden units connected to each input unit.

Convolutional Layer – Back Prop

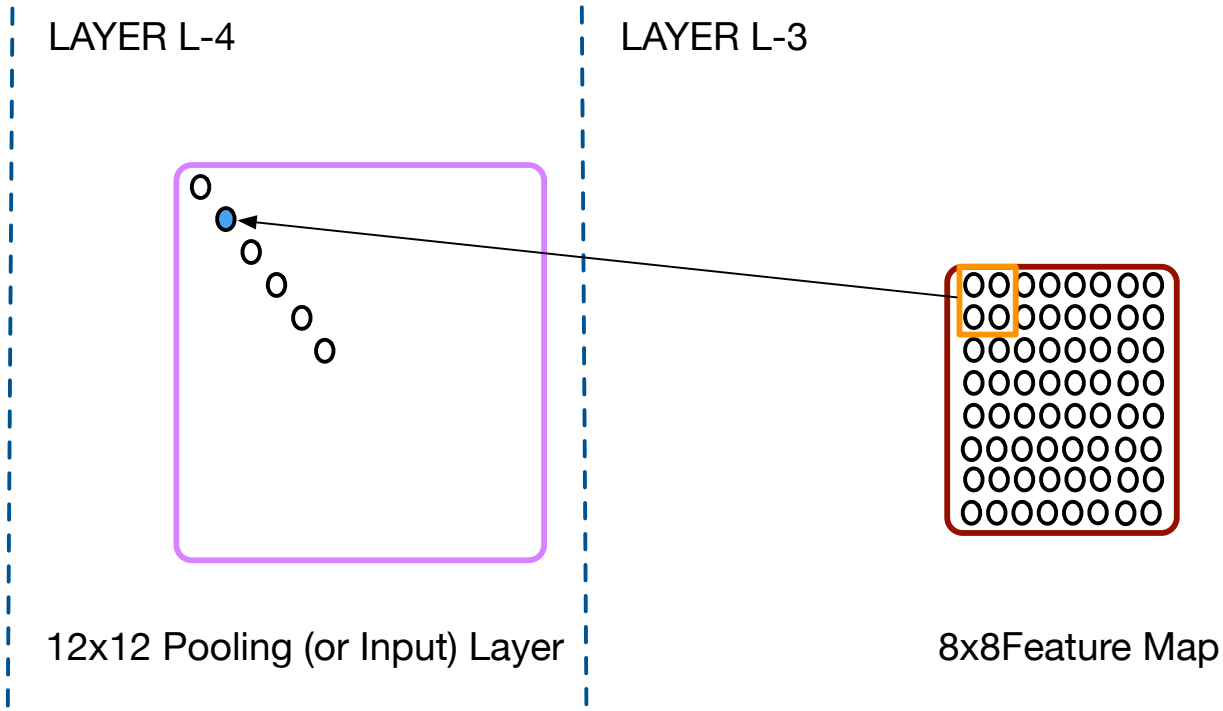
For backprop we need to consider the region of hidden units connected to each input unit.



The top-left input unit (1,1) is connected to just one hidden unit

Convolutional Layer – Back Prop

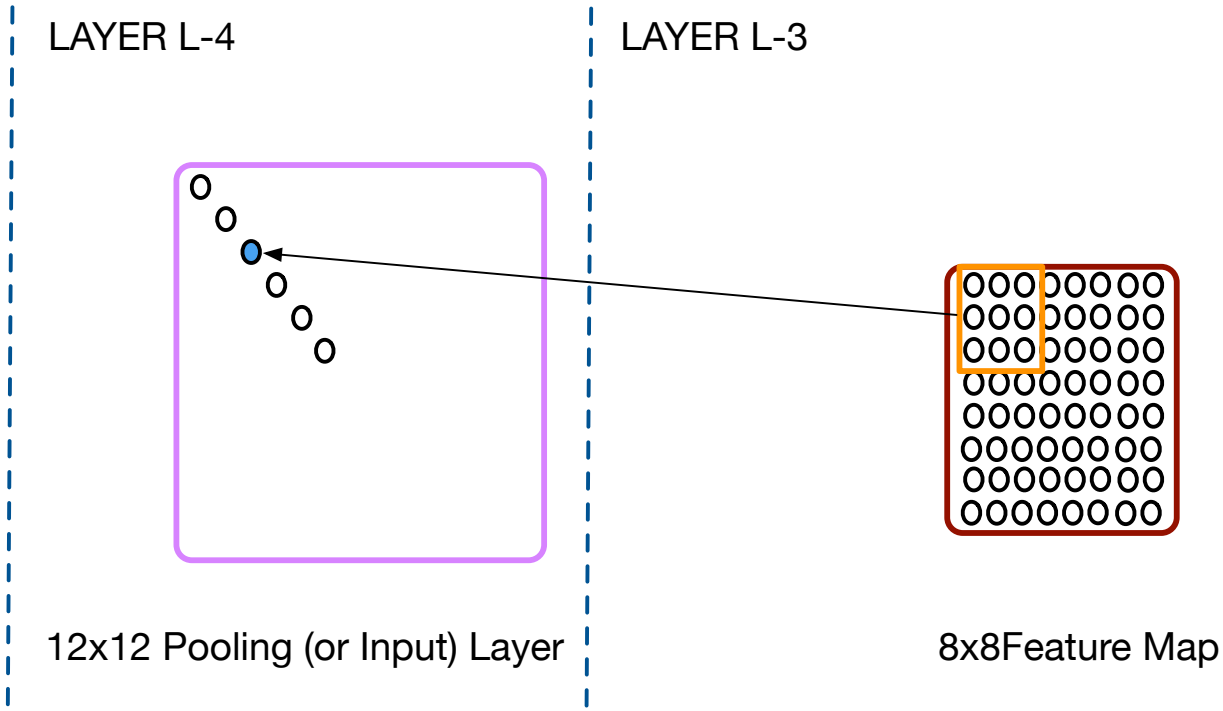
For backprop we need to consider the region of hidden units connected to each input unit.



Input unit (2,2) is in the receptive fields of $2 \times 2 = 4$ hidden units

Convolutional Layer – Back Prop

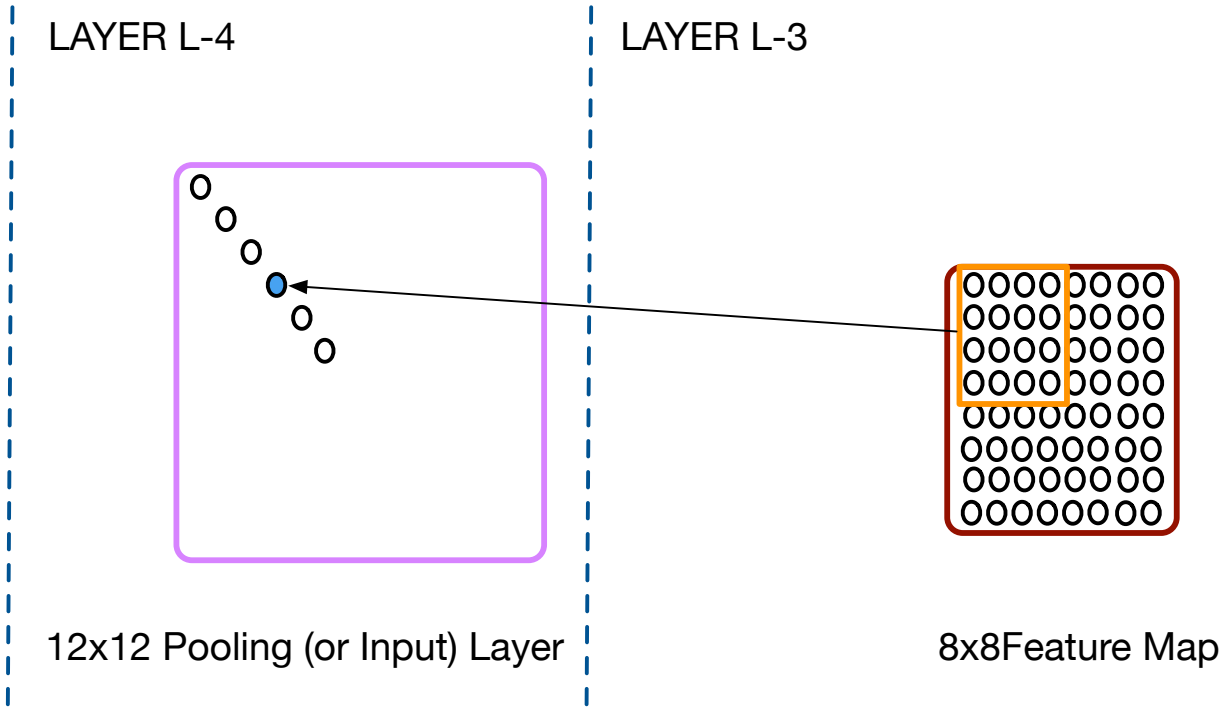
For backprop we need to consider the region of hidden units connected to each input unit.



(3,3) is in the receptive fields of $3 \times 3 = 9$ hidden units

Convolutional Layer – Back Prop

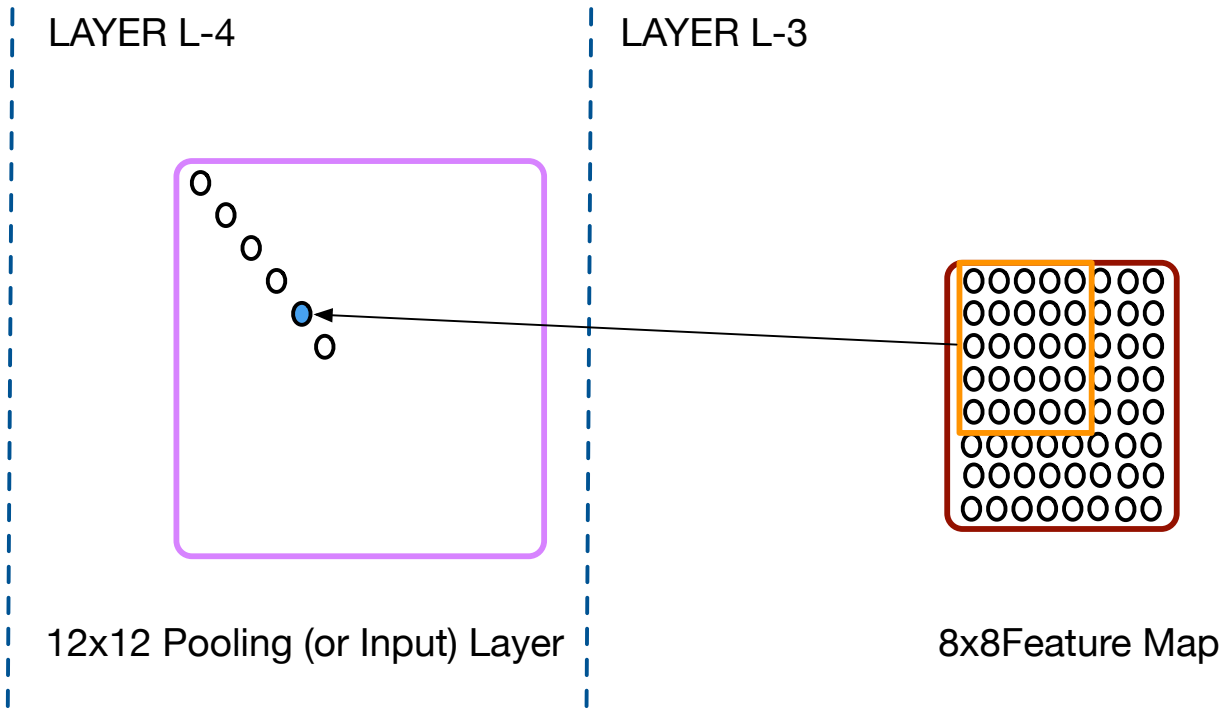
For backprop we need to consider the region of hidden units connected to each input unit.



(4,4) is in the receptive fields of $4 \times 4 = 16$ hidden units

Convolutional Layer – Back Prop

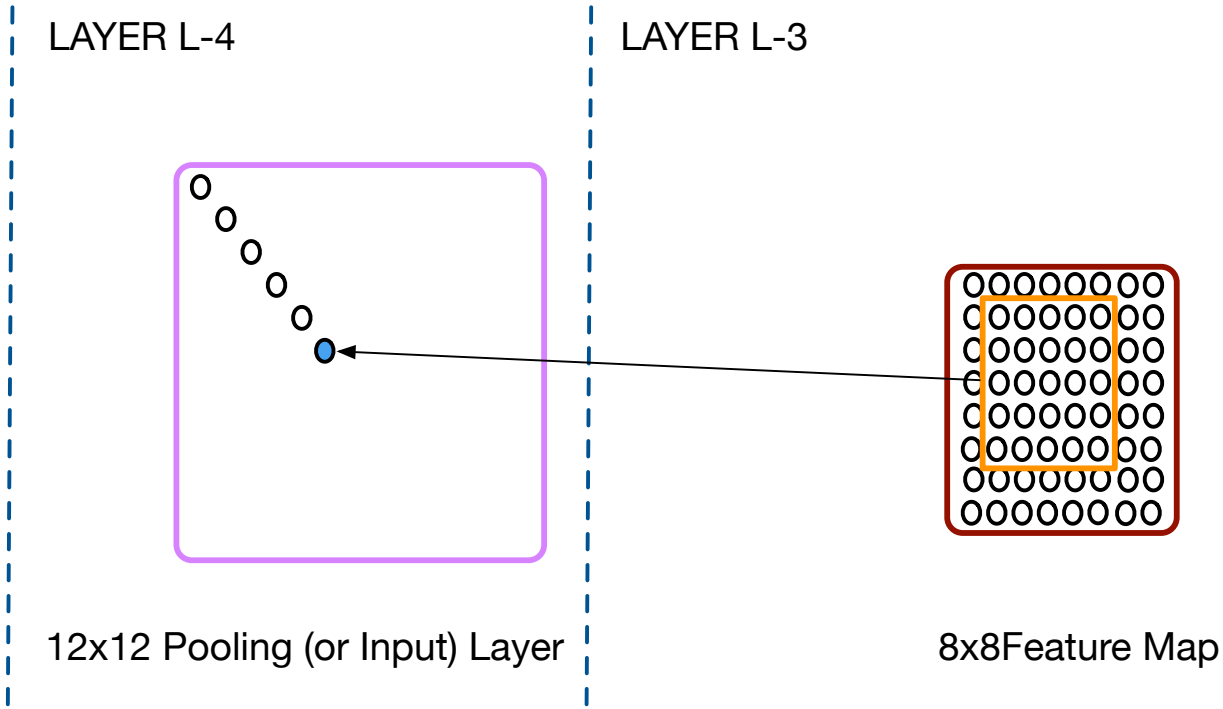
For backprop we need to consider the region of hidden units connected to each input unit.



(5,5) and all units away from the edge are in the receptive fields of $5 \times 5 = 25$ hidden units

Convolutional Layer – Back Prop

For backprop we need to consider the region of hidden units connected to each input unit.



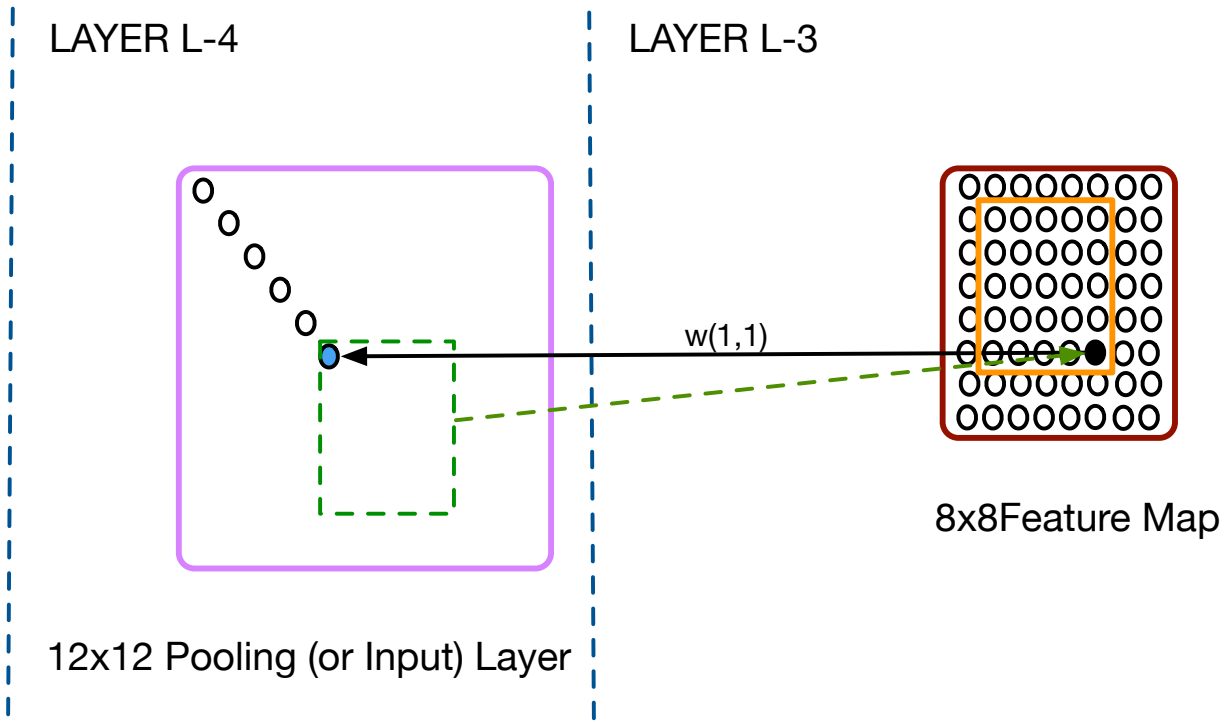
(5,5) and all units away from the edge are in the receptive fields of $5 \times 5 = 25$ hidden units

Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:

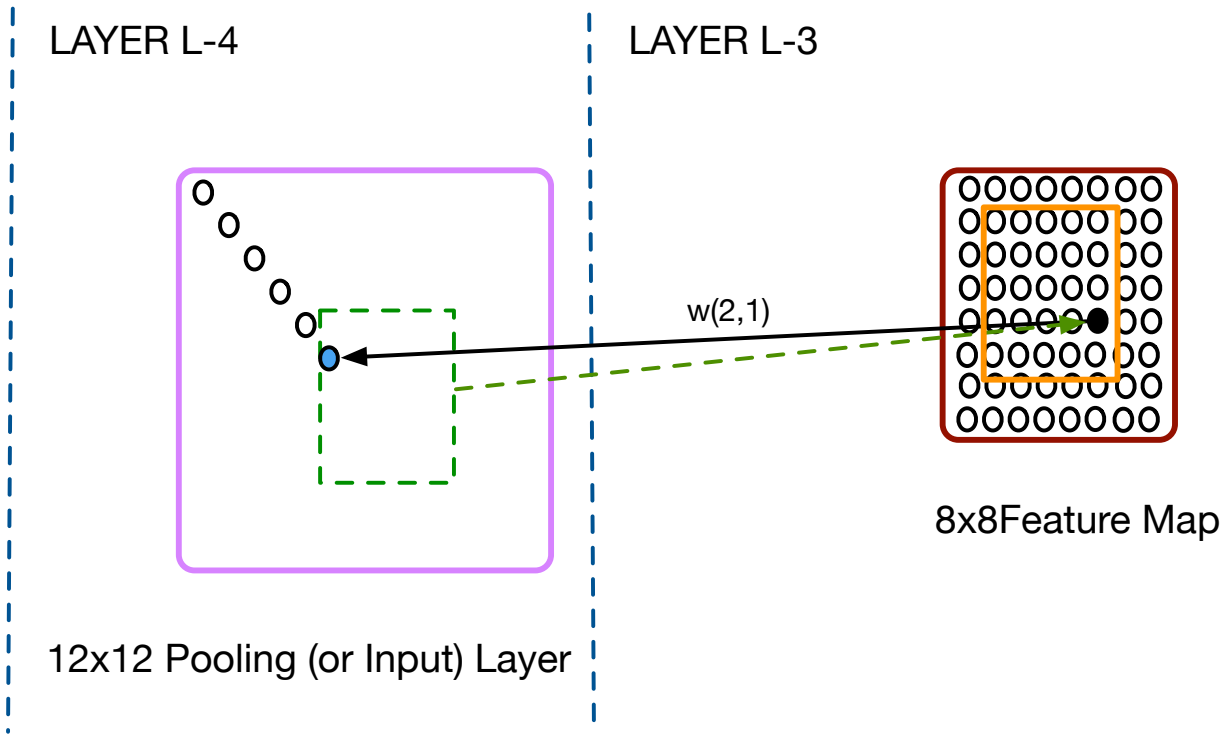


Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:

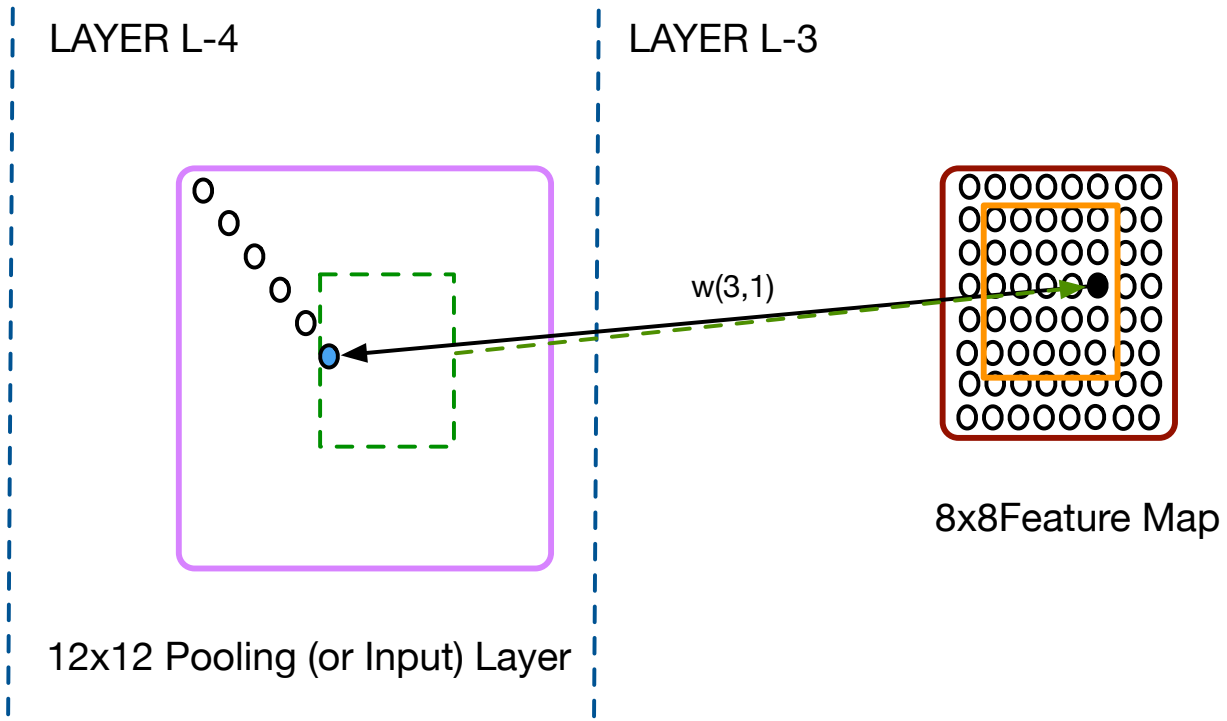


Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:

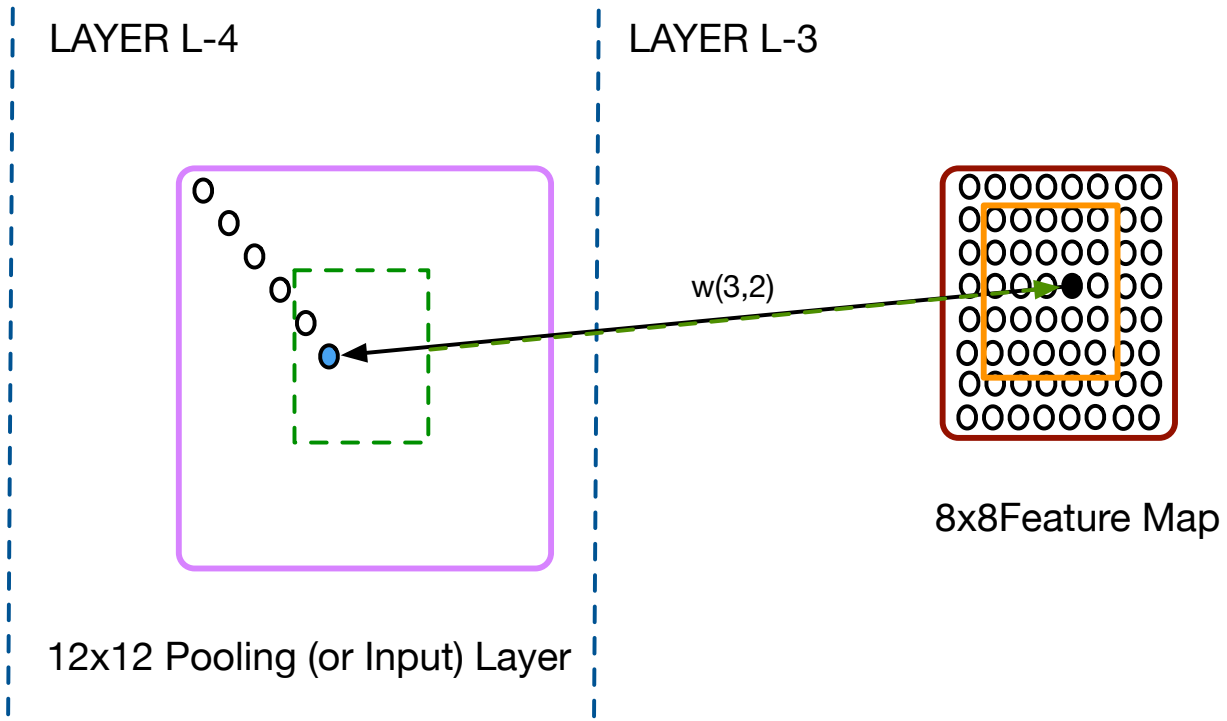


Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:

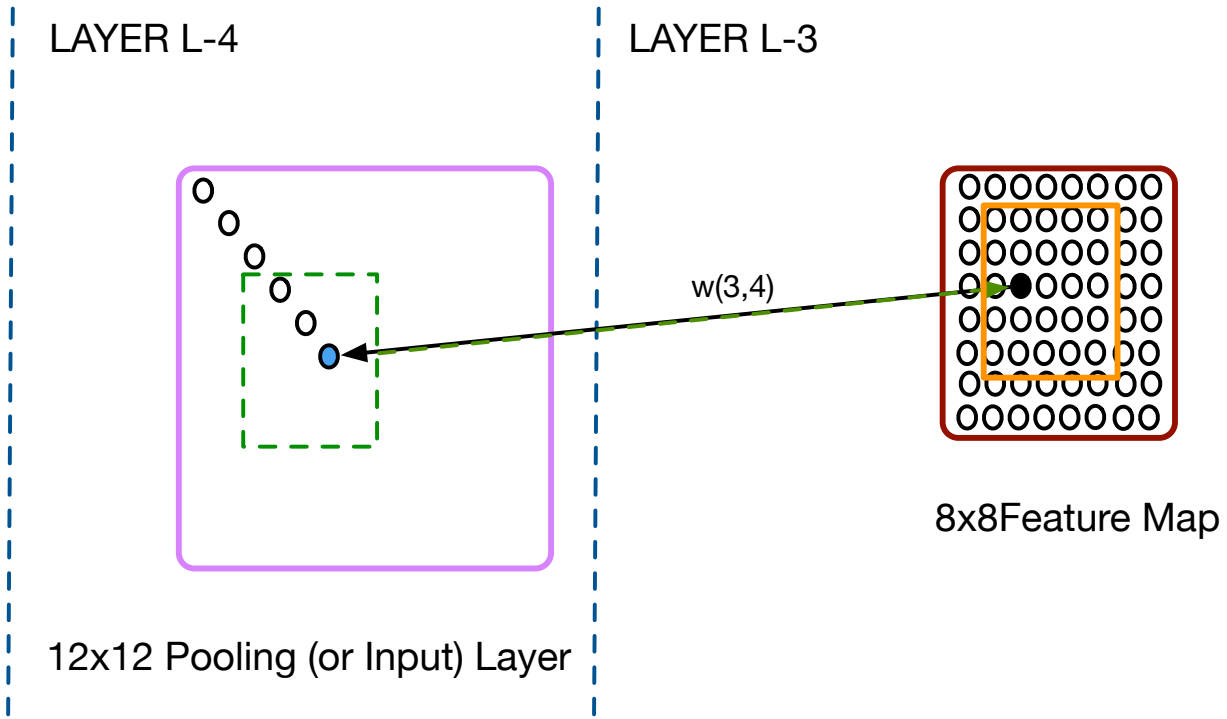


Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:

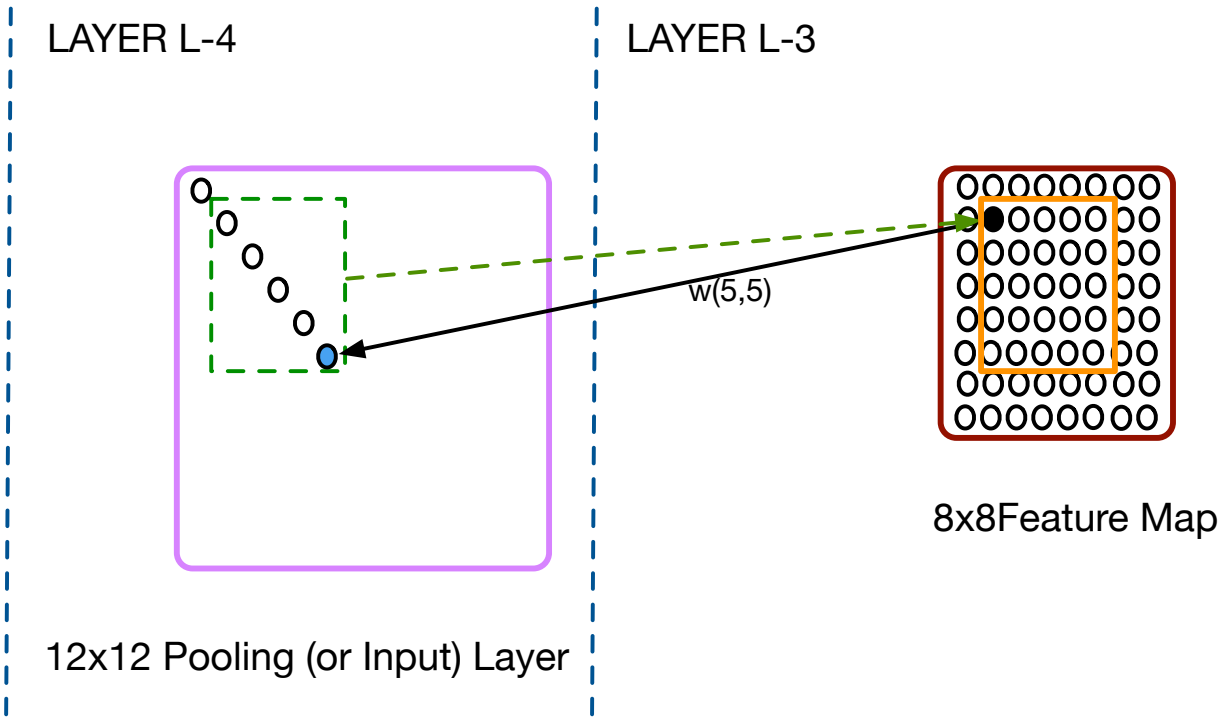


Convolutional Layer – Back Prop

As usual we want to back-propagate the δ values:

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

Look at the shared weights used for back prop:



Backprop as convolution

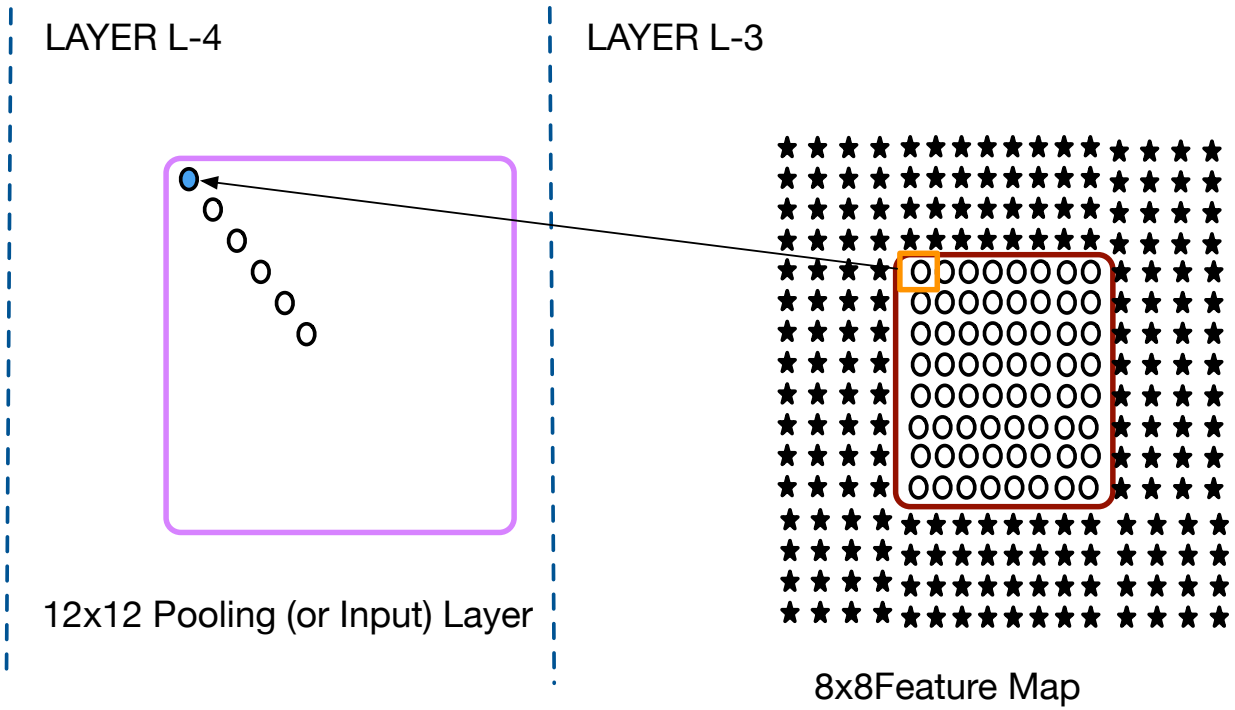
If we have an $m \times m$ kernel size, we can pad the feature map with $(m-1)$ rows and columns of 0s top and bottom, left and right.

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°* as shown before

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.

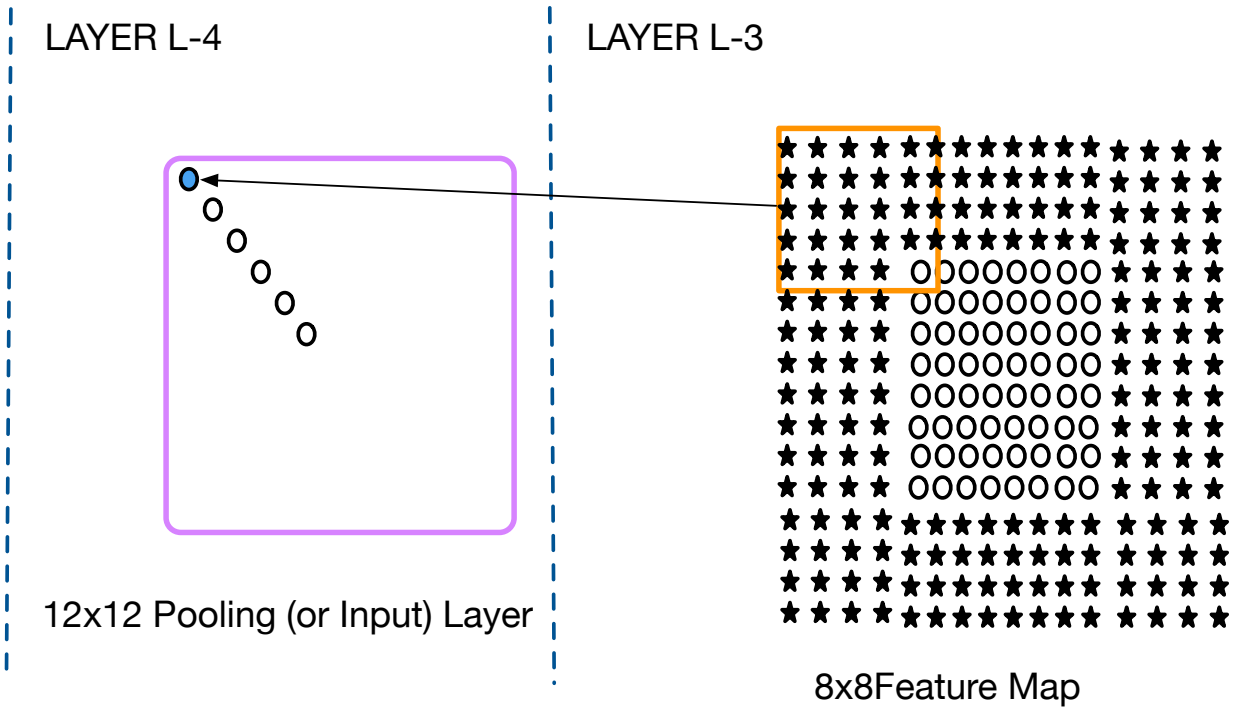
Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* as shown before



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.

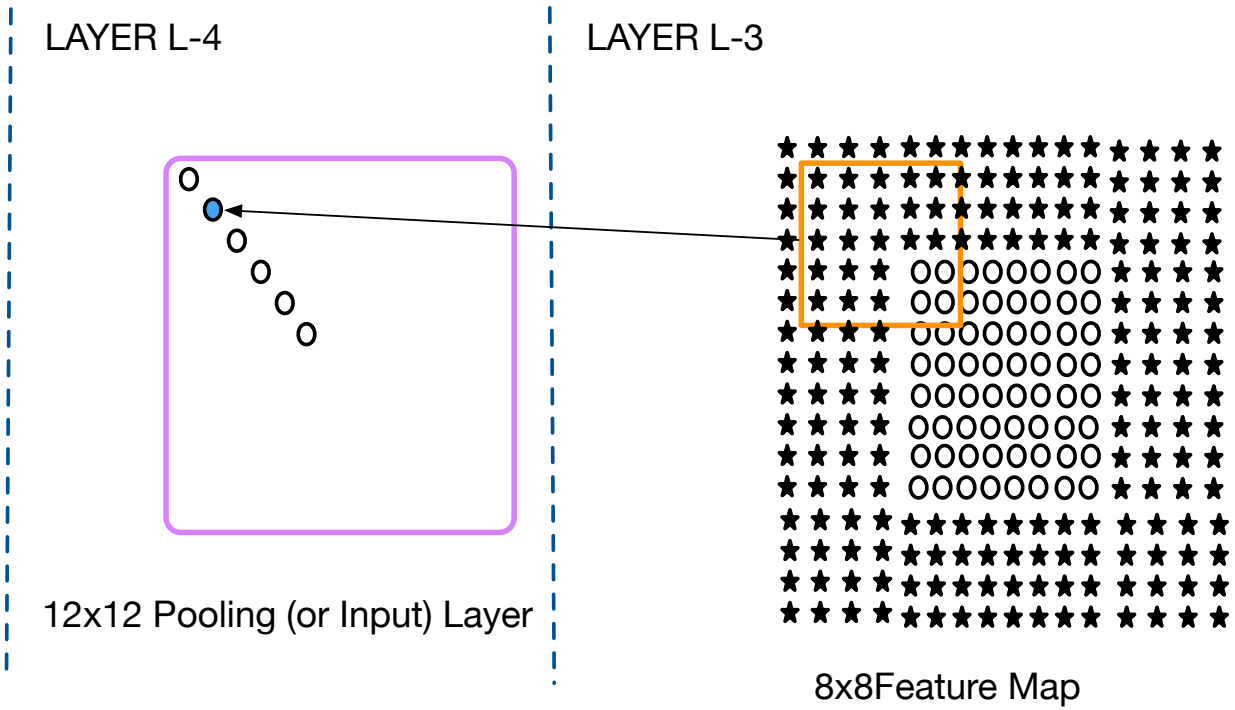
Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* as shown before



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.

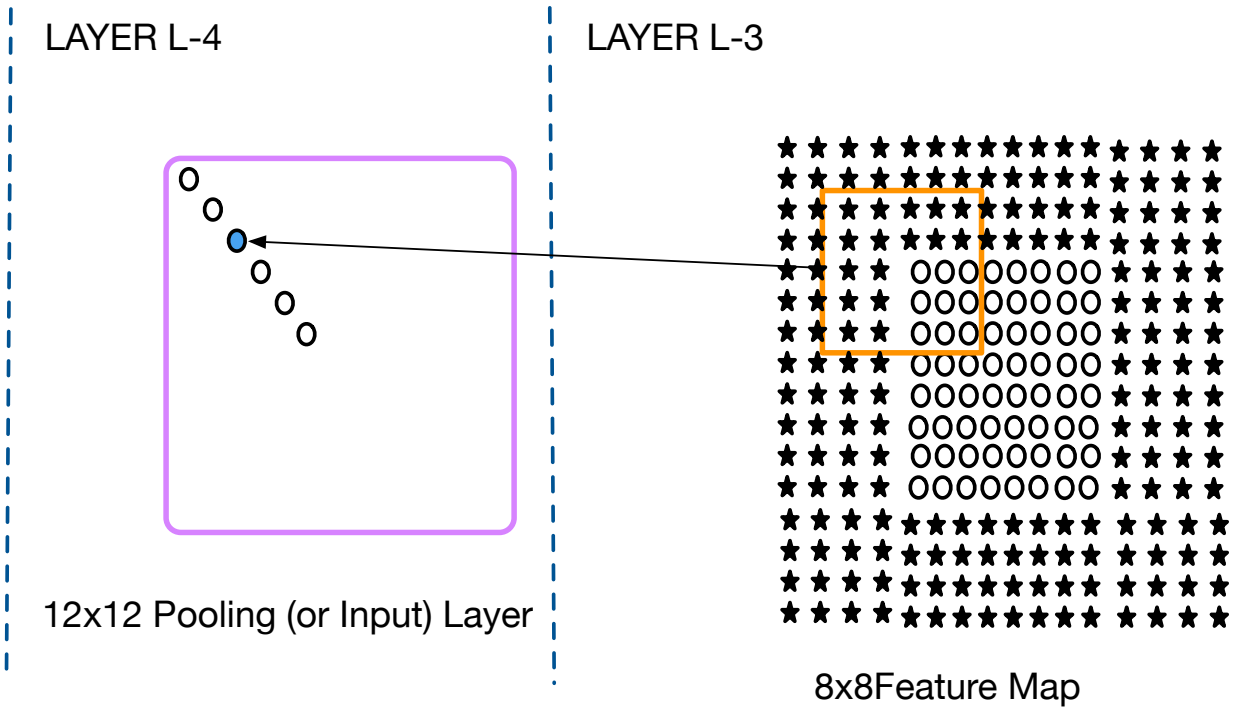
Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* as shown before



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.

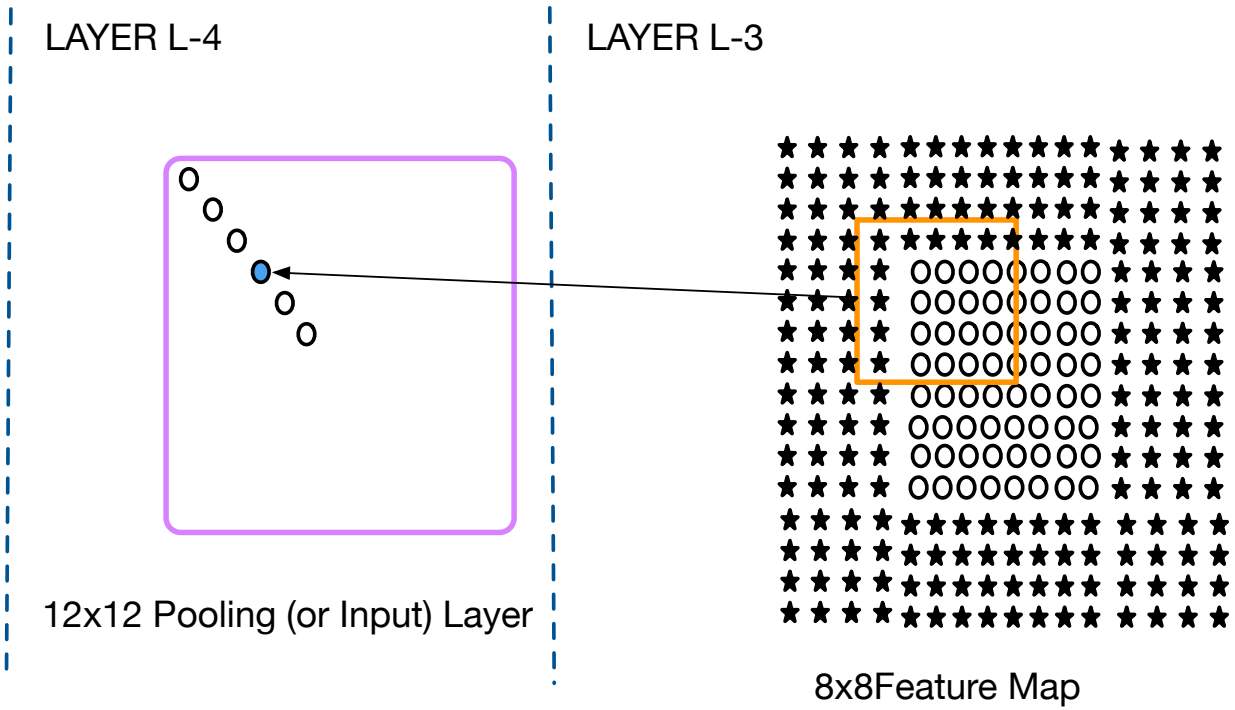
Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* as shown before



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.

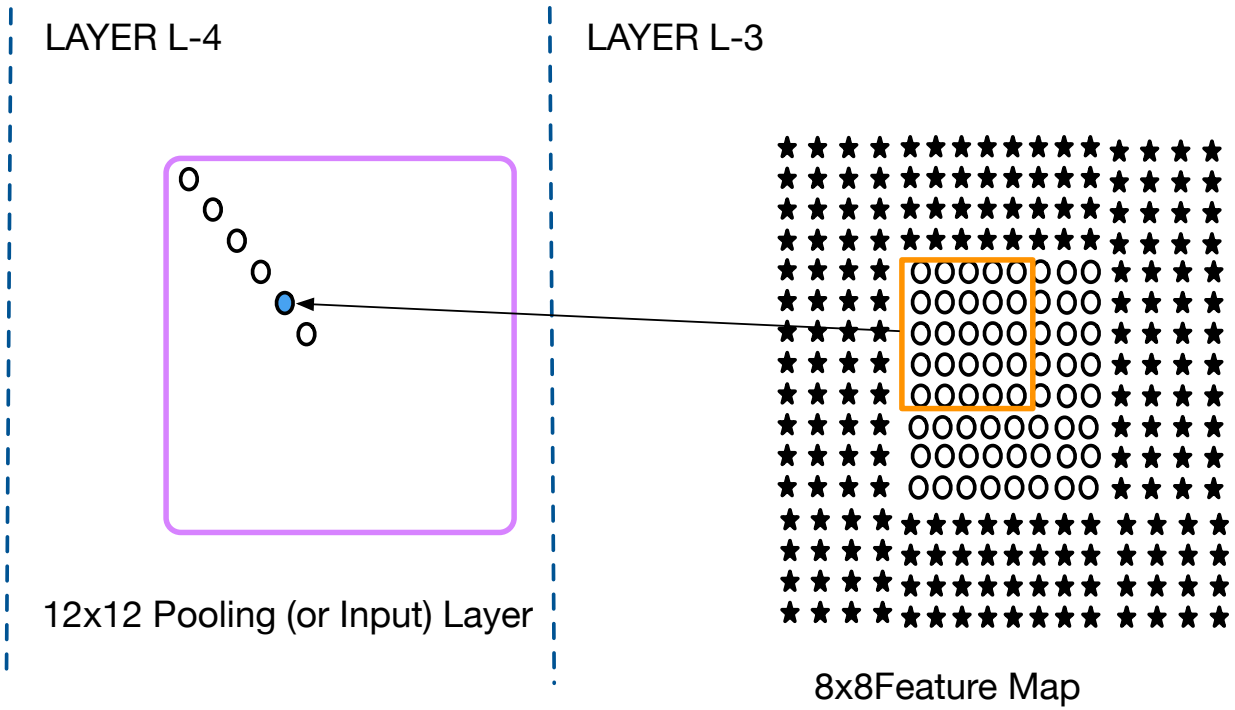
Back prop can then be carried out as a convolution using the weight matrix to scan the padded feature map... BUT the *weight matrix is rotated by 180°* as shown before



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°* as shown before



Convolutional Layer – Back Prop

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

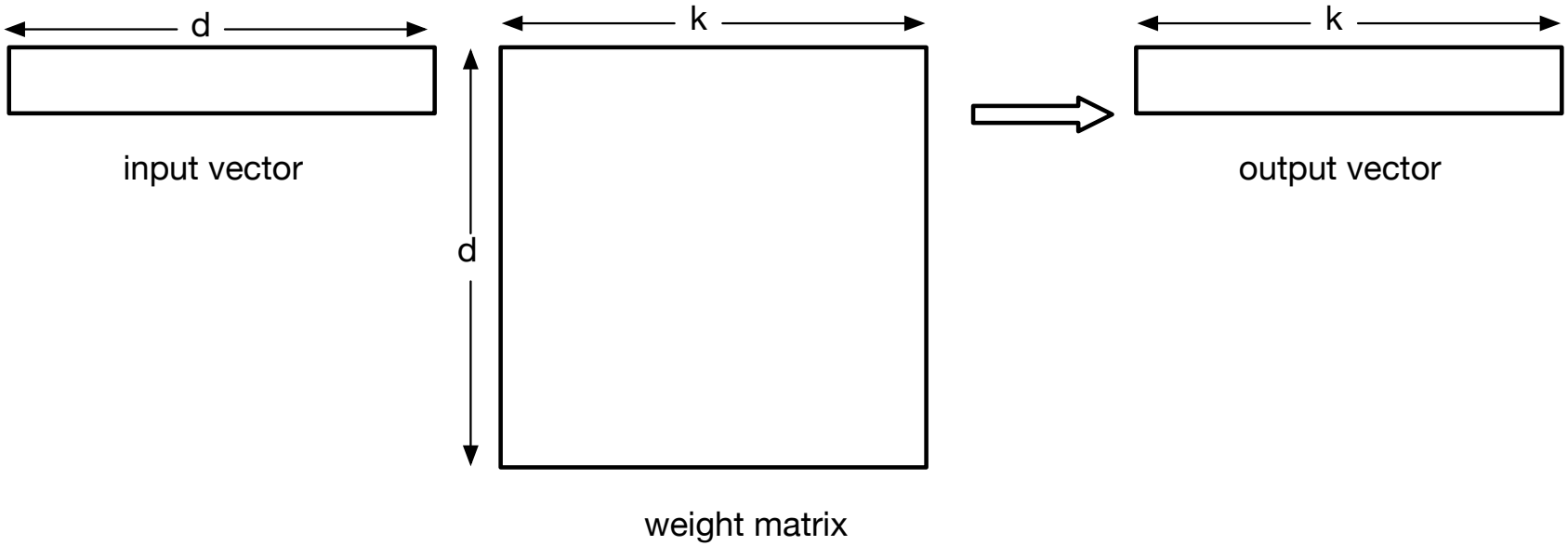
$$\mathbf{h}^{L-3} = \text{sigmoid}(\mathbf{W}^{L-3} * \mathbf{h}^{L-4} + \mathbf{b}^{L-3})$$

And we can write the back-prop as:

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

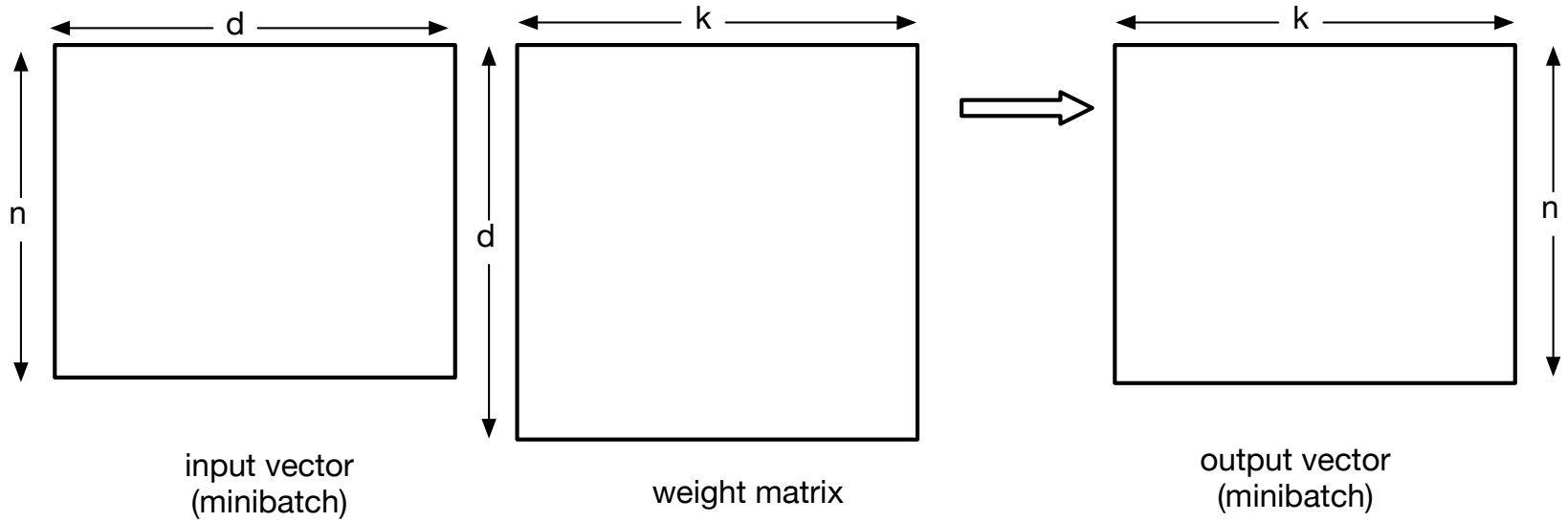
Implementing multilayer networks

Example at a time:



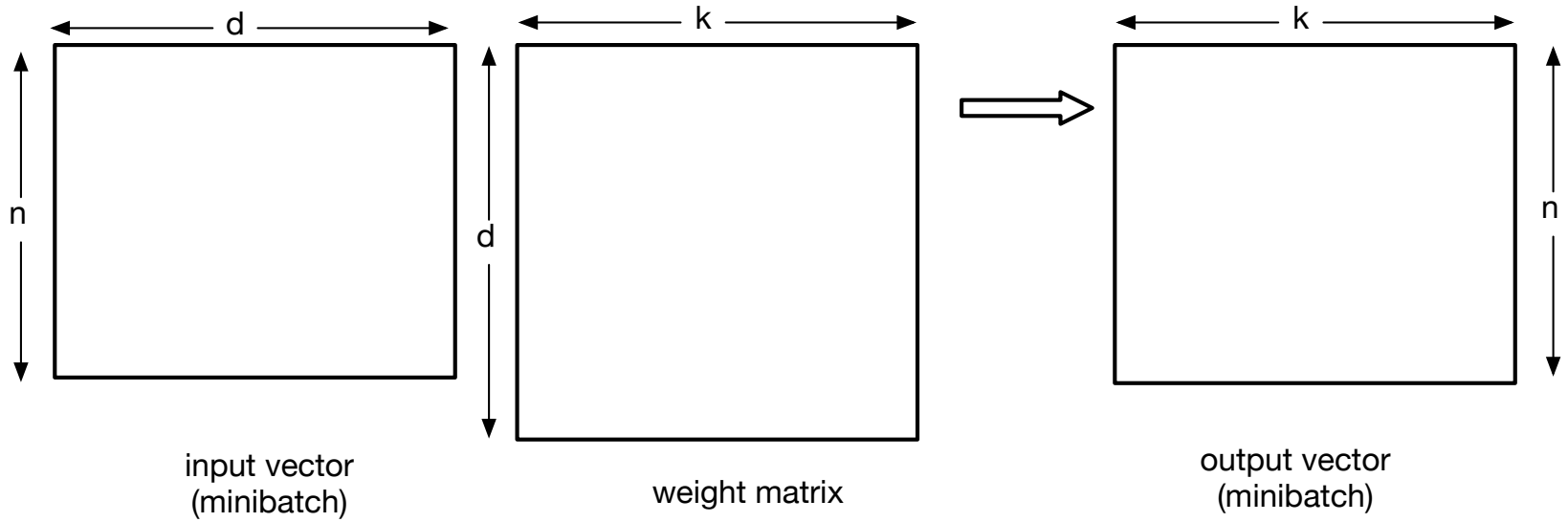
Implementing multilayer networks

Minibatch:



Implementing multilayer networks

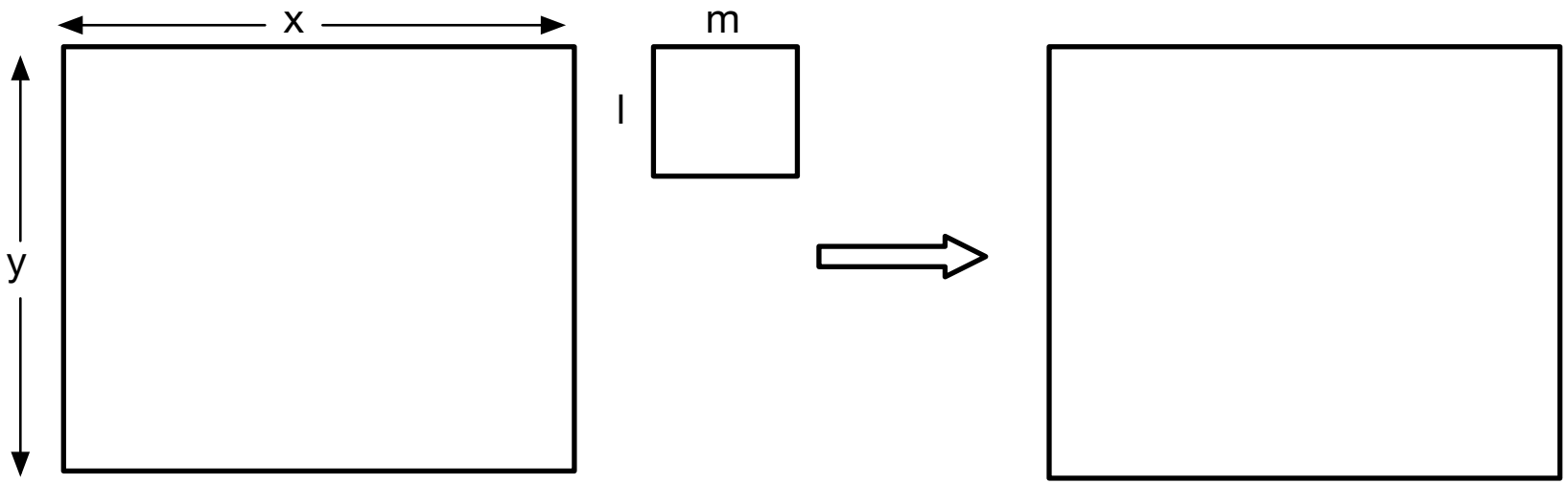
Minibatch:



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

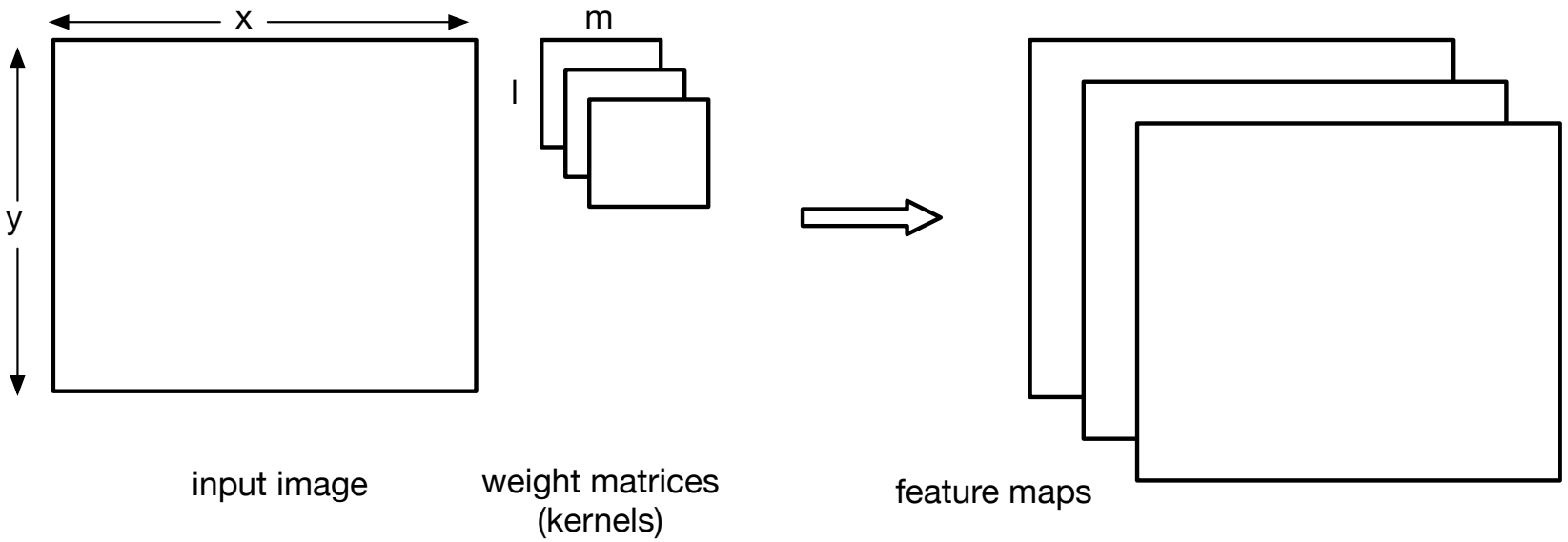
Implementing Convolutional Networks

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



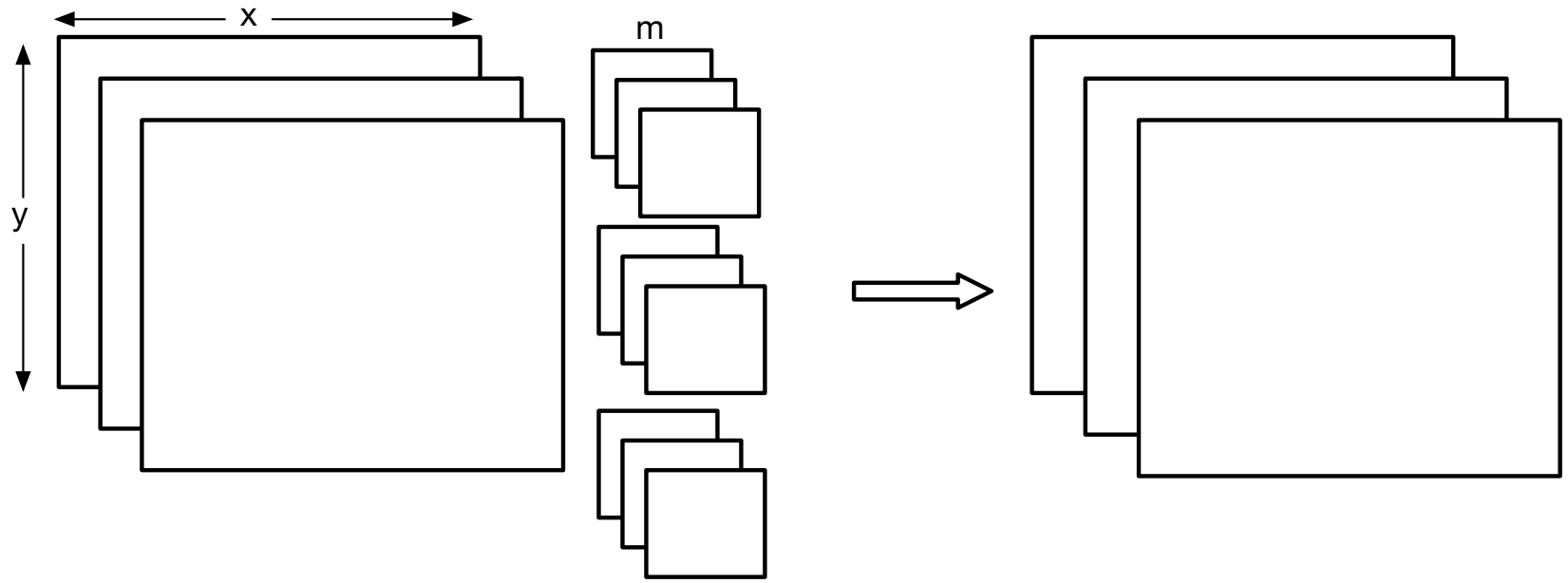
Implementing Convolutional Networks

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



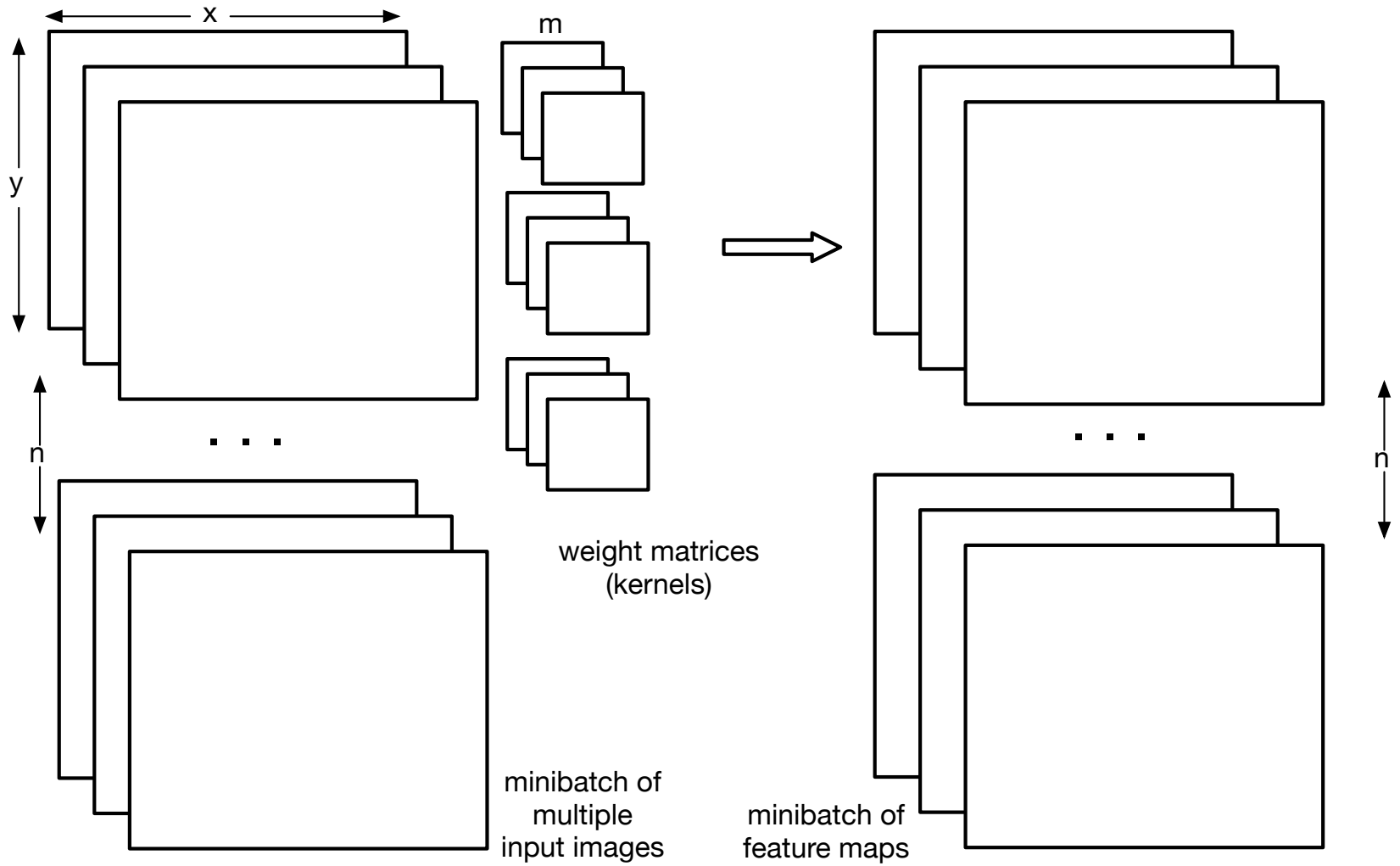
Implementing Convolutional Networks

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

Recent advances using convolutional networks

ImageNet Classification

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

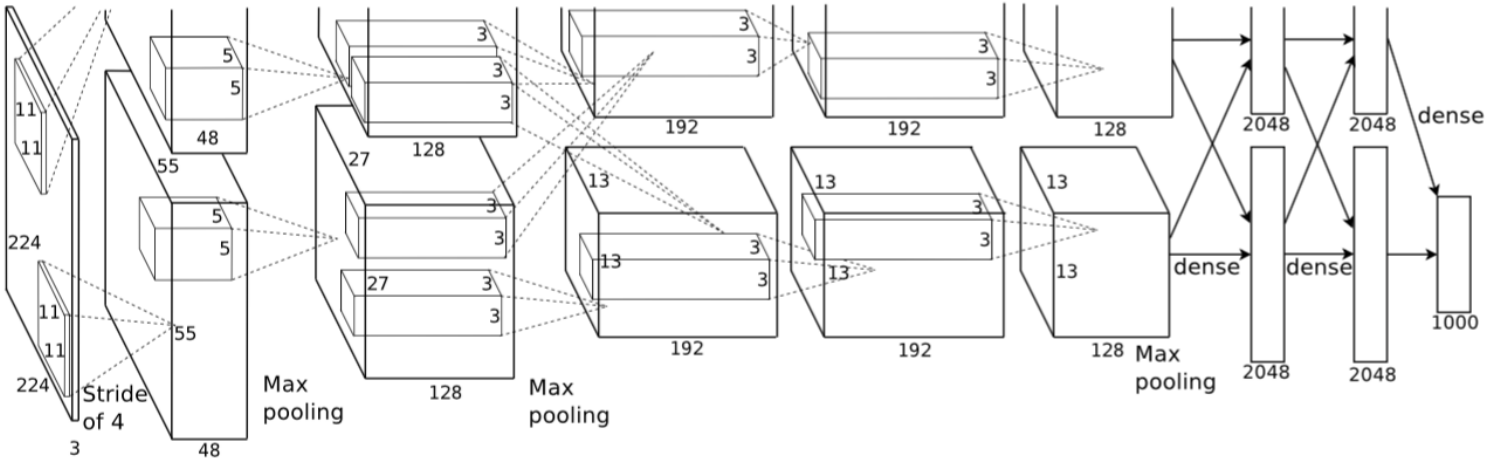


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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%**

Face recognition

Taigman et al, "DeepFace: Closing the Gap to Human-Level Performance in Face Verification", CVPR-2014. http://www.mihantarjomeh.com/wp-content/uploads/2015/01/DeepFace-Closing-the-Gap-to-Human-Level.pdf

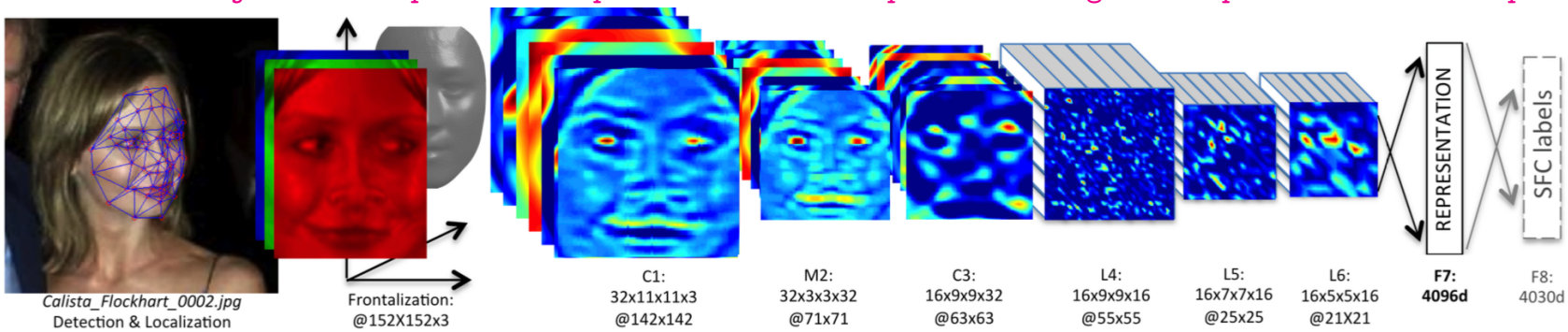


Figure 2. **Outline of the *DeepFace* architecture**. A front-end of a single convolution-pooling-convolution filtering on the rectified input, followed by three locally-connected layers and two fully-connected layers. Colors illustrate feature maps produced at each layer. The net includes more than 120 million parameters, where more than 95% come from the local and fully connected layers.

Method	Accuracy ± SE	Protocol
Joint Bayesian [6]	0.9242 ±0.0108	restricted
Tom-vs-Pete [4]	0.9330 ±0.0128	restricted
High-dim LBP [7]	0.9517 ±0.0113	restricted
TL Joint Bayesian [5]	0.9633 ±0.0108	restricted
DeepFace-single	**0.9592** ±0.0029	unsupervised
DeepFace-single	**0.9700** ±0.0028	restricted
DeepFace-ensemble	**0.9715** ±0.0027	restricted
DeepFace-ensemble	**0.9735** ±0.0025	unrestricted
Human, cropped	0.9753	

Table 3. Comparison with the state-of-the-art on the *LFW* dataset.

Method	Accuracy (%)	AUC	EER
MBGS+SVM- [31]	78.9 ±1.9	86.9	21.2
APEM+FUSION [22]	79.1 ±1.5	86.6	21.4
STFRD+PMML [9]	79.5 ±2.5	88.6	19.9
VSOF+OSS [23]	79.7 ±1.8	89.4	20.0
DeepFace-single	**91.4** ±1.1	96.3	8.6

Table 4. Comparison with the state-of-the-art on the *YTF* dataset.

Action recognition in video

Simonyan and Zisserman, "Two-Stream Convolutional Networks for Action Recognition in Videos", NIPS-2014.

http://papers.nips.cc/paper/

5353-two-stream-convolutional-networks-for-action-recognition-in-videos.pdf

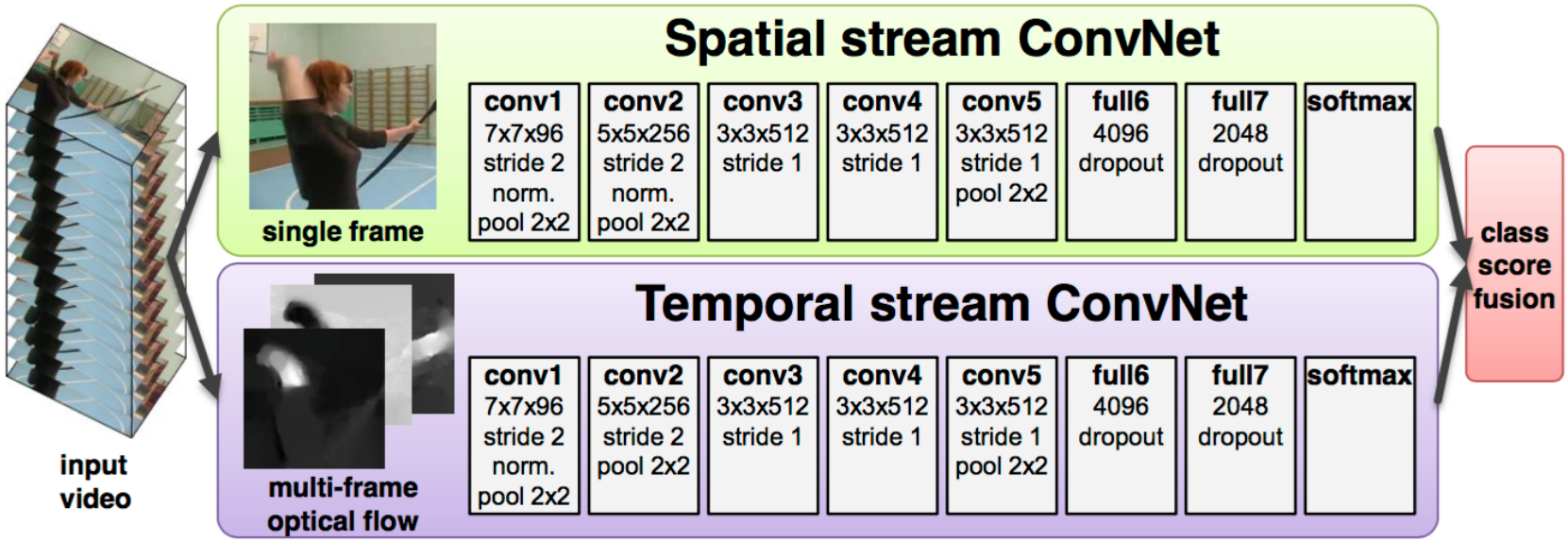


Figure 1: **Two-stream architecture for video classification.**

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:
  Yoshua Bengio et al, *Deep Learning* (ch 9)
  http://goodfeli.github.io/dlbook/contents/convnets.html