COMS 4776 Lecture 7: Convolutional Neural Networks

Richard Zemel

Overview

So far in the course, we've seen two types of layers:

- fully connected layers
- embedding layers (i.e. lookup tables)

Different layers could be stacked together to build powerful models.
Let's add another layer type: convolution layers
Conv layers are very useful building blocks for computer vision applications.

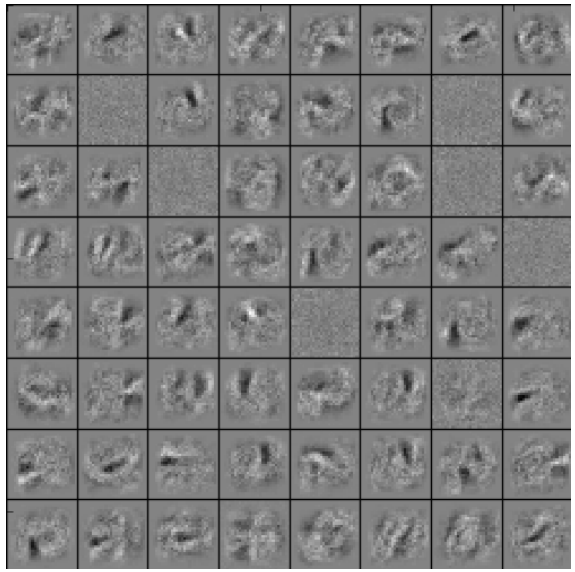
Overview

What makes vision hard?

- Vison needs to be robust to a lot of transformations or distortions:
  - change in pose/viewpoint
  - change in illumination
  - deformation
  - occlusion (some objects are hidden behind others)
- Many object categories can vary wildly in appearance (e.g. chairs)
- Geoff Hinton: "Imaging a medical database in which the age of the patient sometimes hops to the input dimension which normally codes for weight!"

Overview

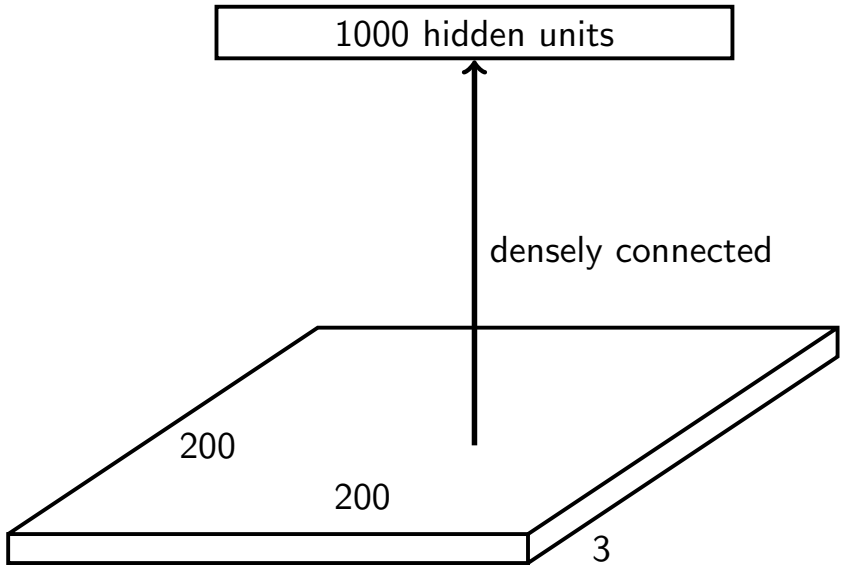
Recall we looked at some hidden layer features for classifying handwritten digits:



This isn't going to scale to full-sized images.

Overview

Suppose we want to train a network that takes a 200×200 RGB image as input.

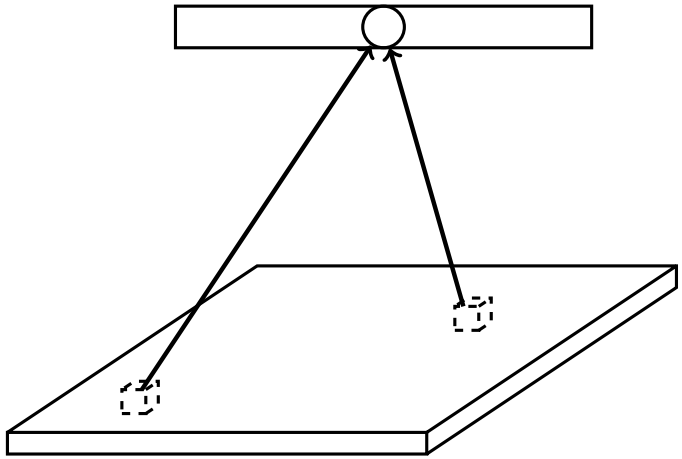


What is the problem with having this as the first layer?

- Too many parameters! Input size $= 200 \times 200 \times 3 = 120$K. Parameters $= 120\text{K} \times 1000 = 120$ million.
- What happens if the object in the image shifts a little?

Overview

In the fully connected layer, each feature (hidden unit) looks at the entire image.
Since the image is a BIG thing, we end up with lots of parameters.



But, do we really expect to learn a useful feature at the first layer which depends on pixels that are spatially far away ?
The far away pixels will probably belong to completely different objects (or object sub-parts). Very little correlation.
We want the incoming weights to focus on local patterns of the input image.

Overview

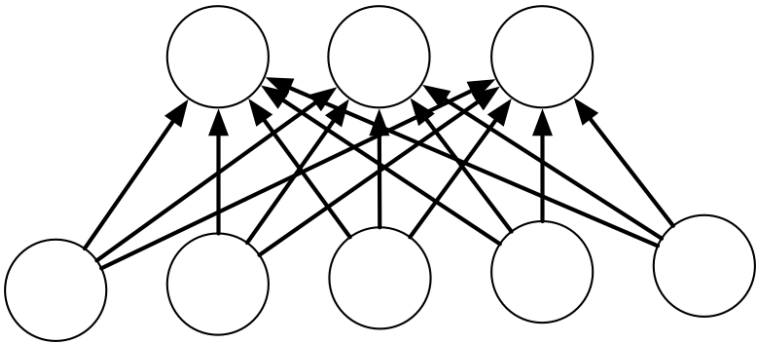
The same sorts of features that are useful in analyzing one part of the image will probably be useful for analyzing other parts as well.

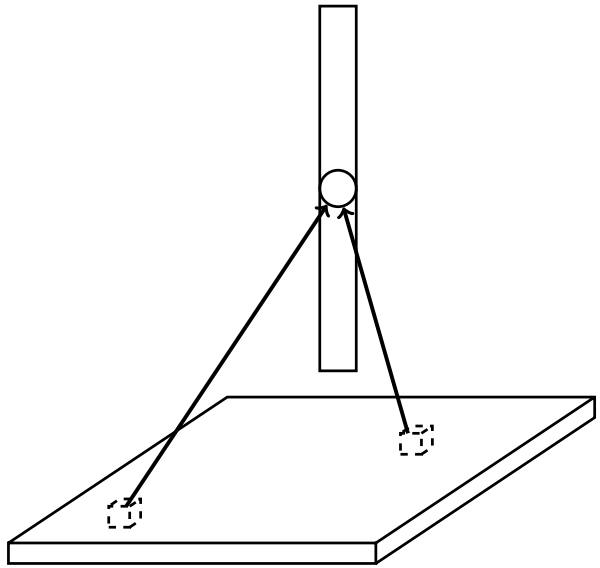
E.g., edges, corners, contours, object parts

We want a neural net architecture that lets us learn a set of feature detectors shared at all image locations.

Convolution Layers

Fully connected layers:

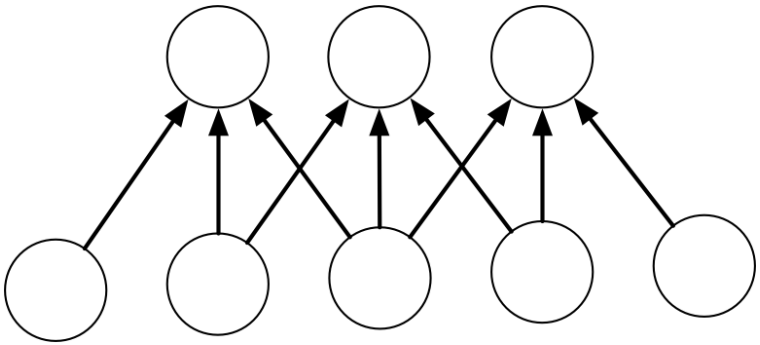


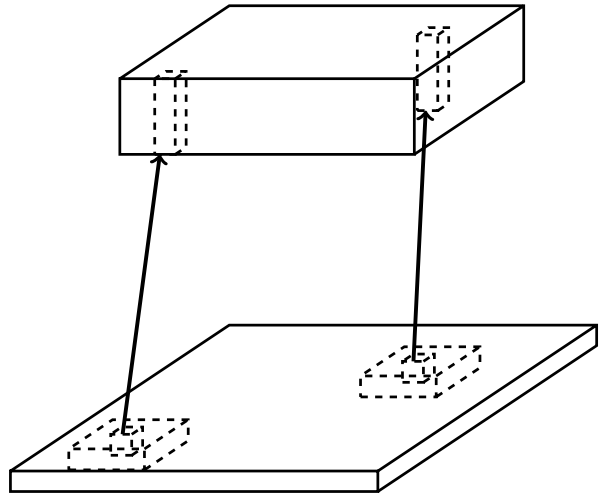


Each hidden unit looks at the entire image.

Convolution Layers

Locally connected layers:

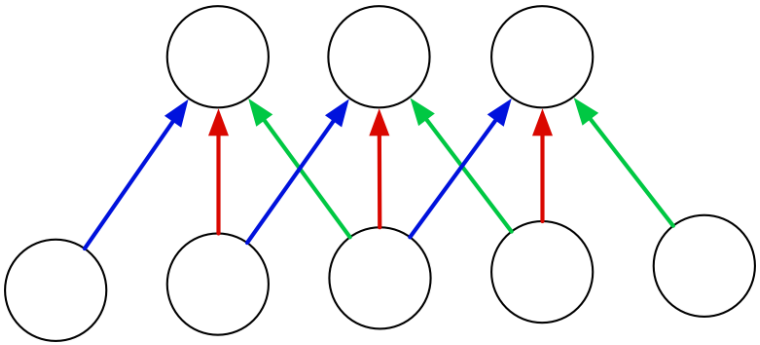


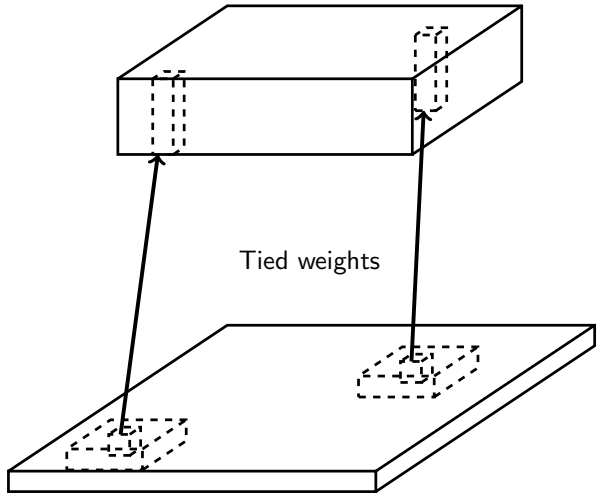


Each column of hidden units looks at a small region of the image.

Convolution Layers

Convolution layers:

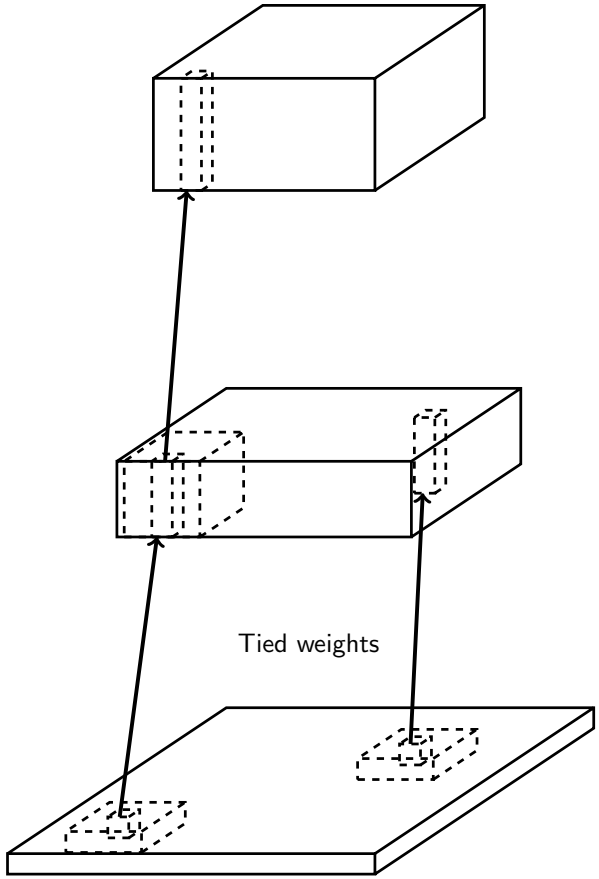




Each column of hidden units looks at a small region of the image, and the weights are shared between all image locations.

Going Deeply Convolutional

Convolution layers can be stacked:



Convolution

We've already been vectorizing our computations by expressing them in terms of matrix and vector operations.

Now we'll introduce a new high-level operation, convolution. Here the motivation isn't computational efficiency — we'll see more efficient ways to do the computations later. Rather, the motivation is to get some understanding of what convolution layers can do.

Convolution

We've already been vectorizing our computations by expressing them in terms of matrix and vector operations.

Now we'll introduce a new high-level operation, convolution. Here the motivation isn't computational efficiency — we'll see more efficient ways to do the computations later. Rather, the motivation is to get some understanding of what convolution layers can do.

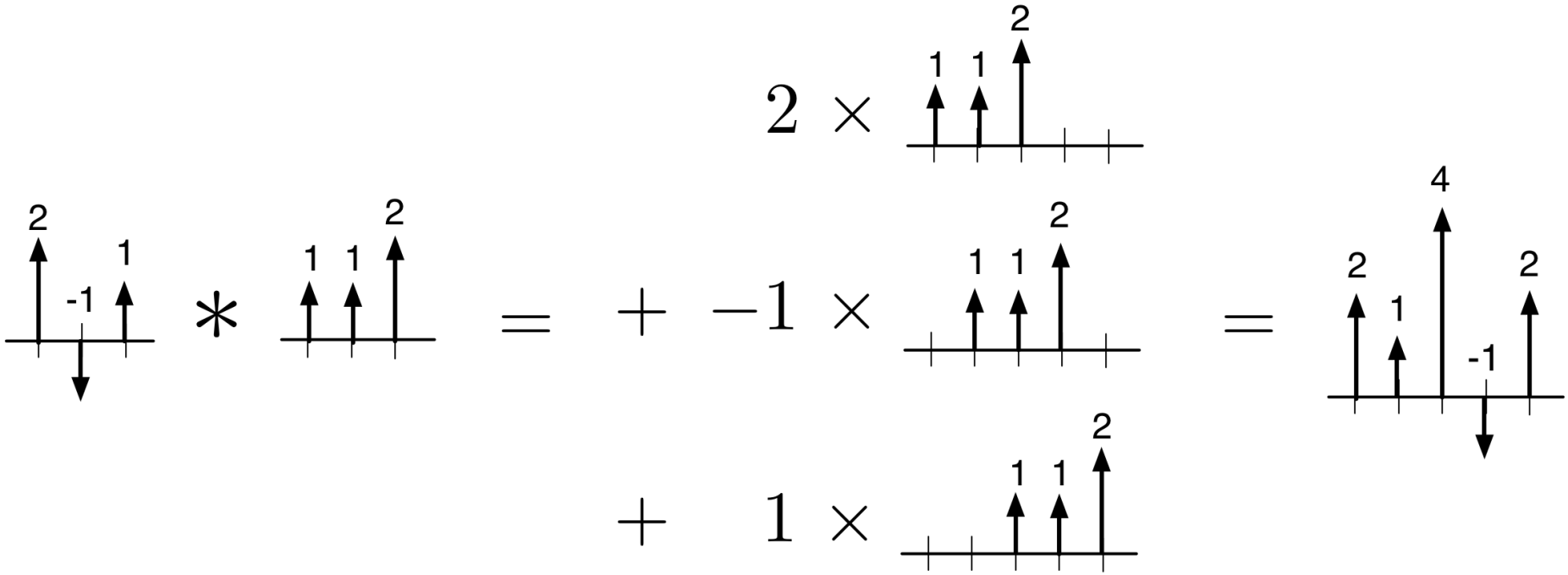
Let's look at the 1-D case first. If a and b are two arrays,

$$(a * b)_t = \sum_\tau a_\tau b_{t-\tau}.$$

Note: indexing conventions are inconsistent. We'll explain them in each case.

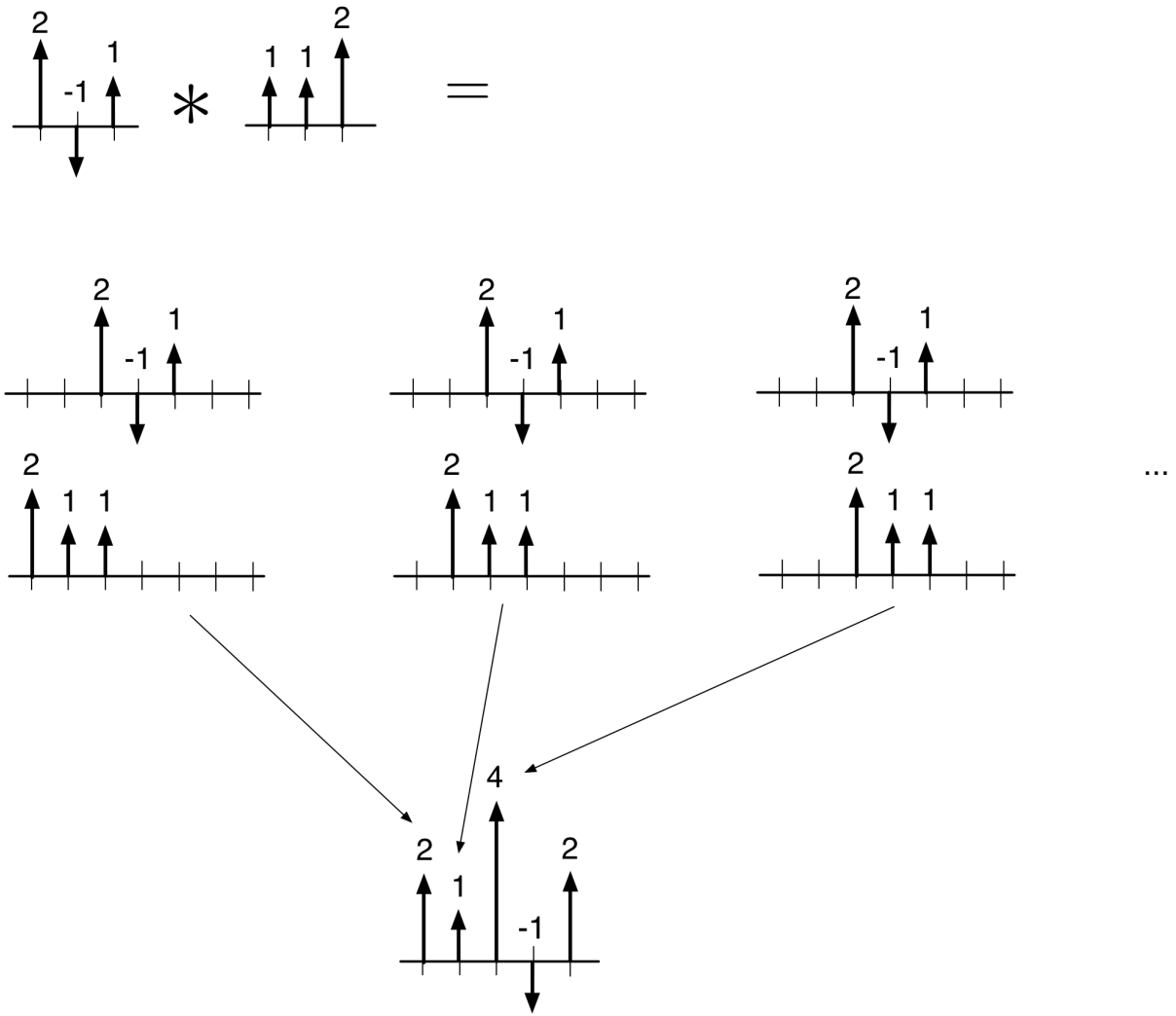
Convolution

Method 1: translate-and-scale



Convolution

Method 2: flip-and-filter



Convolution

Convolution can also be viewed as matrix multiplication:

$$(2, -1, 1) * (1, 1, 2) = \begin{pmatrix} 1 & & \\ 1 & 1 & \\ 2 & 1 & 1 \\ & 2 & 1 \\ & & 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix}$$

Aside: This is how convolution is typically implemented. (More efficient than the fast Fourier transform (FFT) for modern conv nets on GPUs!)

Convolution

Some properties of convolution:

- Commutativity

$$a * b = b * a$$

- Linearity

$$a * (\lambda_1 b + \lambda_2 c) = \lambda_1 a * b + \lambda_2 a * c$$

2-D Convolution

2-D convolution is defined analogously to 1-D convolution.

If A and B are two 2-D arrays, then:

$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}.$$

2-D Convolution

Method 1: Translate-and-Scale

$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array} = 1 \times \begin{array}{|c|c|c|c|}\hline 1 & 3 & 1 & \\ \hline 0 & -1 & 1 & \\ \hline 2 & 2 & -1 & \\ \hline & & & \\ \hline \end{array} + 2 \times \begin{array}{|c|c|c|c|}\hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline & & & \\ \hline \end{array} + -1 \times \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 1 & 5 & 7 & 2 \\ \hline 0 & -2 & -4 & 1 \\ \hline 2 & 6 & 4 & -3 \\ \hline 0 & -2 & -2 & 1 \\ \hline \end{array}$$

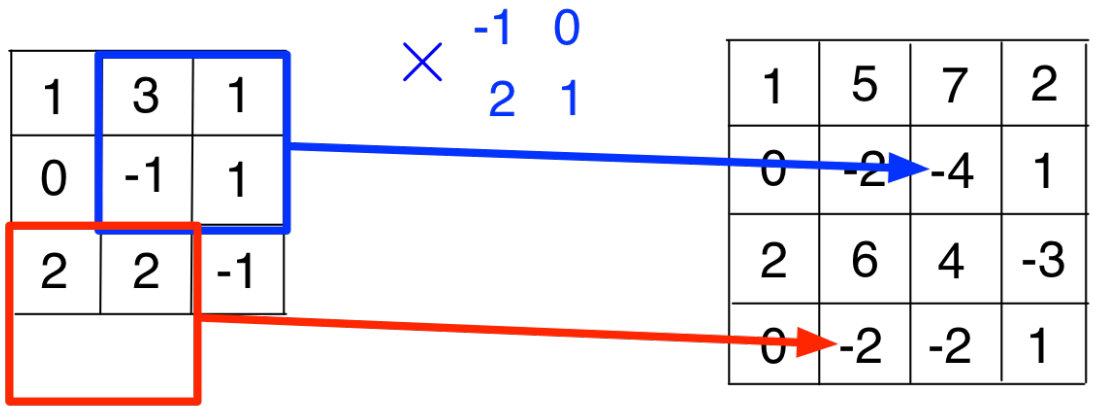
2-D Convolution

Method 2: Flip-and-Filter

1	3	1
0	-1	1
2	2	-1

$*$

1	2
0	-1



2-D Convolution

The thing we convolve by is called a kernel, or filter.

What does this convolution kernel do?

$*$

0	1	0
1	4	1
0	1	0

2-D Convolution

The thing we convolve by is called a kernel, or filter.

What does this convolution kernel do?

$*$

0	1	0
1	4	1
0	1	0

Answer: Blur

Note: We call the resulting image an "activation map" by the kernel

2-D Convolution

What does this convolution kernel do?

$*$

0	-1	0
-1	8	-1
0	-1	0

2-D Convolution

What does this convolution kernel do?

$*$

0	-1	0
-1	8	-1
0	-1	0

Answer: Sharpen

2-D Convolution

What does this convolution kernel do?

$*$

0	-1	0
-1	4	-1
0	-1	0

2-D Convolution

What does this convolution kernel do?

$*$

0	-1	0
-1	4	-1
0	-1	0

Answer: Edge Detection

2-D Convolution

What does this convolution kernel do?

$*$

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

2-D Convolution

What does this convolution kernel do?

$*$

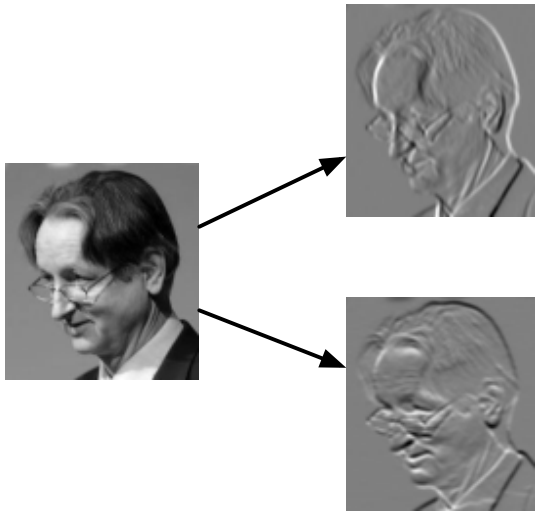
1	0	-1
2	0	-2
1	0	-1

Answer: Stronger Edge Detection

Convolutional networks

Let's finally turn to convolutional networks. These have two kinds of layers: detection layers (or convolution layers), and pooling layers.

The convolution layer has a set of filters. Its output is a set of feature maps, each one obtained by convolving the image with a filter.

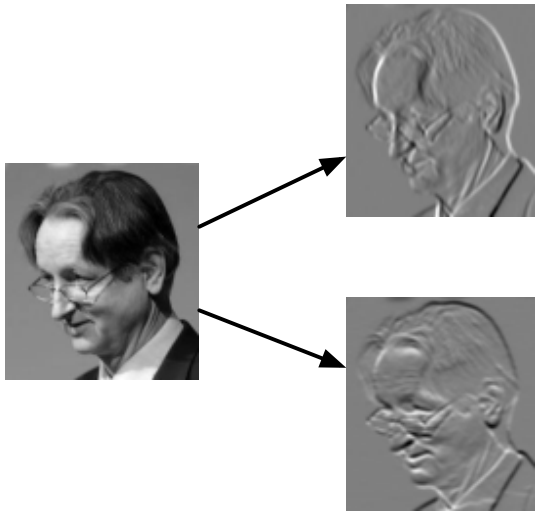


convolution

Convolutional networks

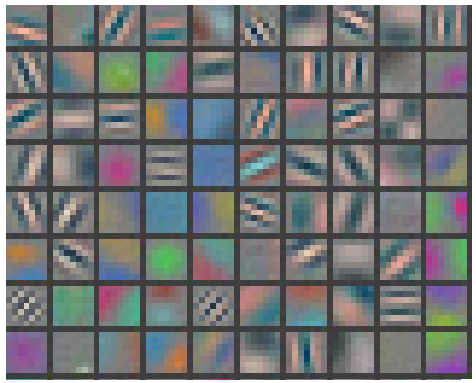
Let's finally turn to convolutional networks. These have two kinds of layers: detection layers (or convolution layers), and pooling layers.

The convolution layer has a set of filters. Its output is a set of feature maps, each one obtained by convolving the image with a filter.



convolution

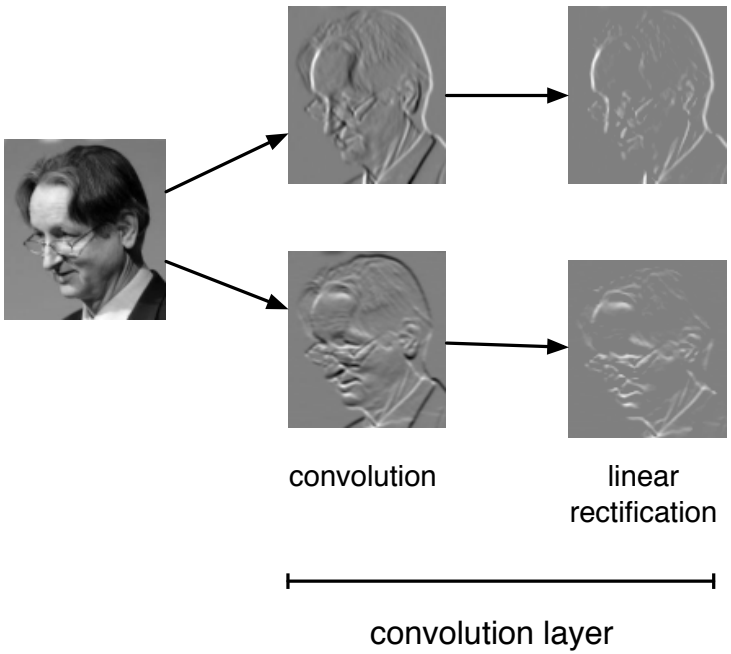
Example first-layer filters



(Zeiler and Fergus, 2013, Visualizing and understanding convolutional networks)

Convolutional networks

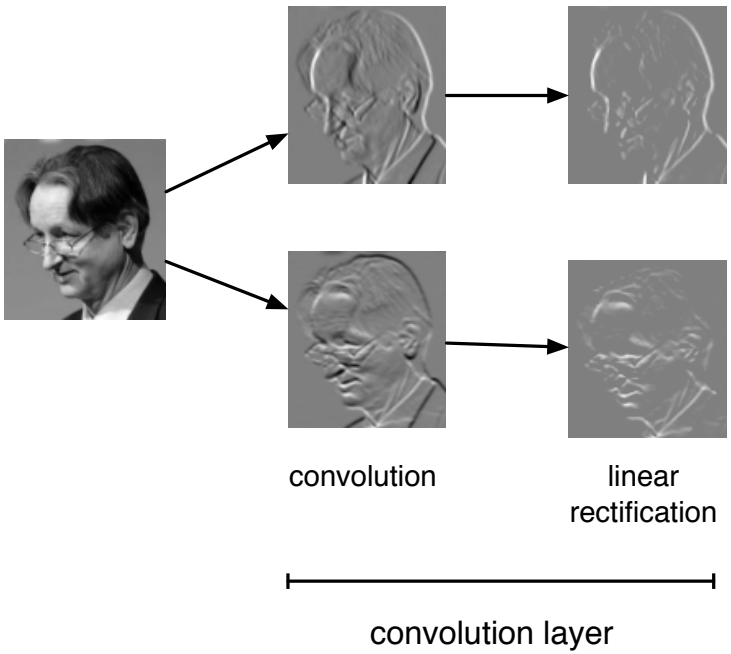
It's common to apply a linear rectification nonlinearity: $y_i = \max(z_i, 0)$



Why might we do this?

Convolutional networks

It's common to apply a linear rectification nonlinearity: $y_i = \max(z_i, 0)$

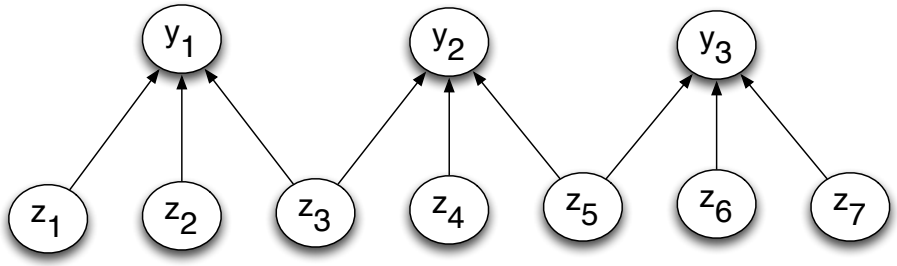


Why might we do this?

- Convolution is a linear operation. Therefore, we need a nonlinearity, otherwise 2 convolution layers would be no more powerful than 1.
- Two edges in opposite directions shouldn't cancel

Pooling layers

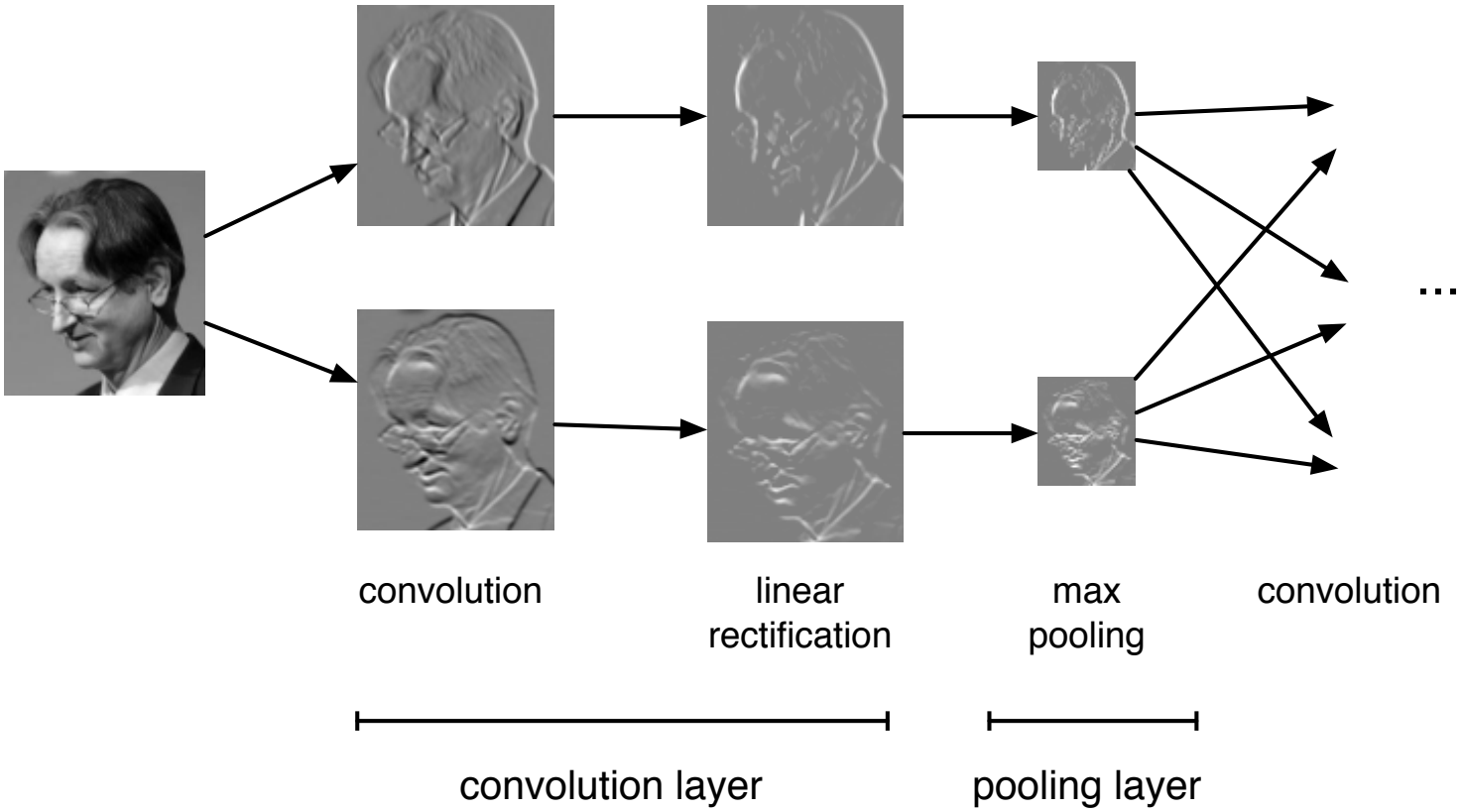
The other type of layer in a pooling layer. These layers reduce the size of the representation and build in invariance to small transformations.



Most commonly, we use max-pooling, which computes the maximum value of the units in a pooling group:

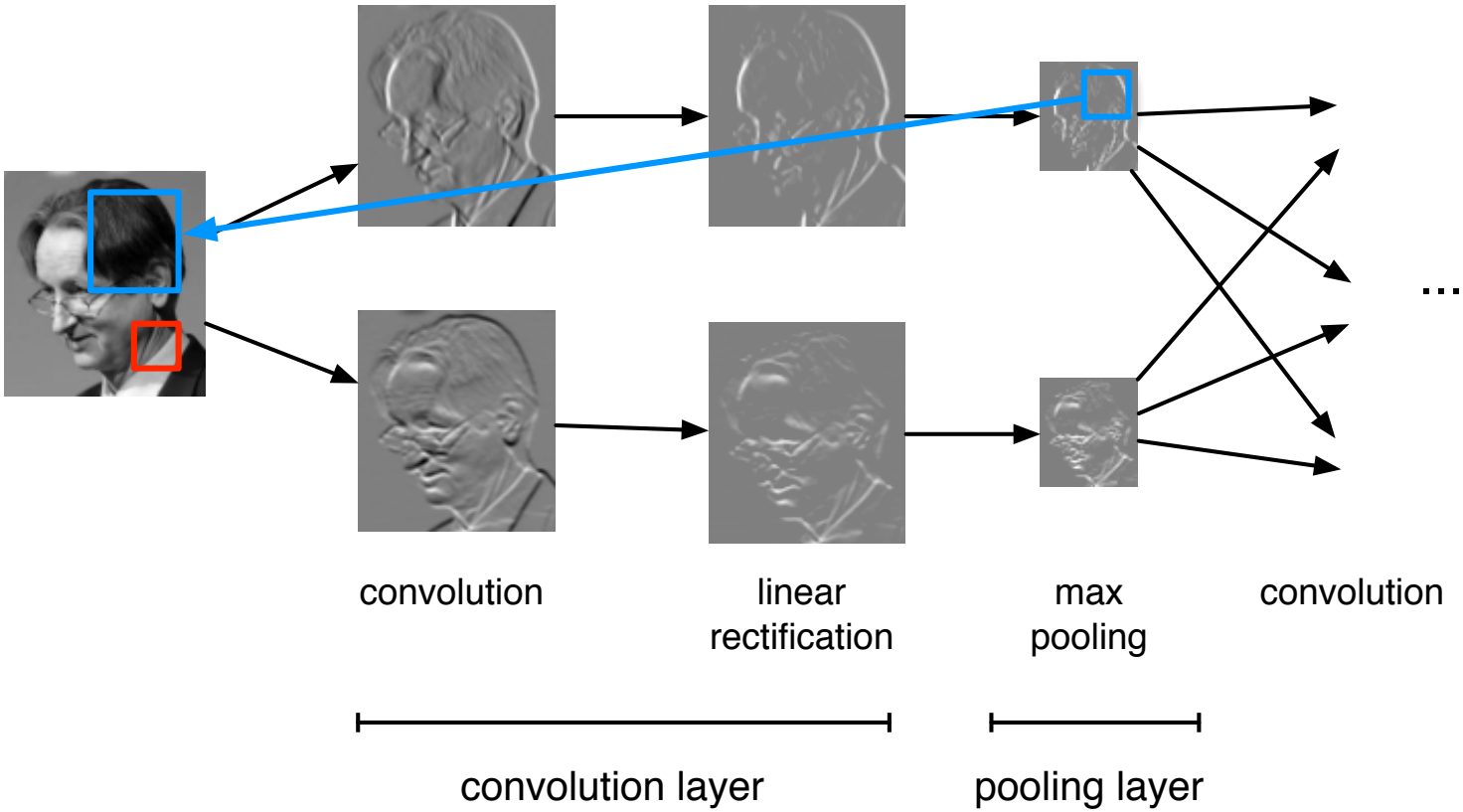
$$y_i = \max_{j \text{ in pooling group}} z_j$$

Convolutional networks



Convolutional networks

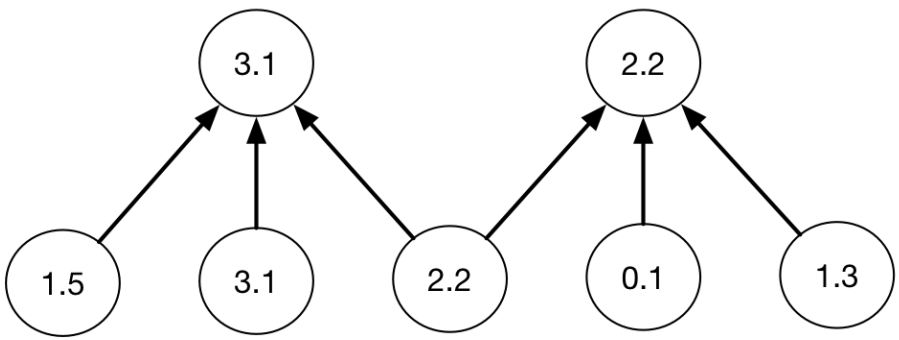
Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in the lower layers.

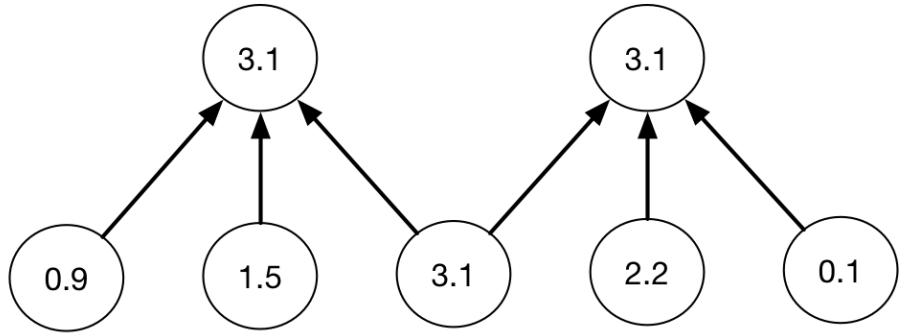


Equivariance and Invariance

We said the network's responses should be robust to translations of the input. But this can mean two different things.

- Convolution layers are equivariant: if you translate the inputs, the outputs are translated by the same amount.
- We'd like the network's predictions to be invariant: if you translate the inputs, the prediction should not change.
- Pooling layers provide invariance to small translations.





Convolution Layers

Each layer consists of several feature maps, or channels each of which is an array.

- If the input layer represents a grayscale image, it consists of one channel. If it represents a color image, it consists of three channels.

Each unit is connected to each unit within its receptive field in the previous layer. This includes *all* of the previous layer's feature maps.

Convolution Layers

For simplicity, focus on 1-D signals (e.g. audio waveforms). Suppose the convolution layer's input has J feature maps and its output has I feature maps. Let t index the locations. Suppose the convolution kernels have radius R, i.e. dimension $K = 2R + 1$.

Each unit in a convolution layer receives inputs from all the units in its receptive field in the previous layer:

$$y_{i,t} = \sum_{j=1}^{J} \sum_{\tau=-R}^{R} w_{i,j,\tau} x_{j,t+\tau}.$$

In terms of convolution,

$$\mathbf{y}_i = \sum_j \mathbf{x}_j * \text{flip}(\mathbf{w}_{i,j}).$$

Backprop Updates (Optional)

How do we train a conv net? With backprop, of course!

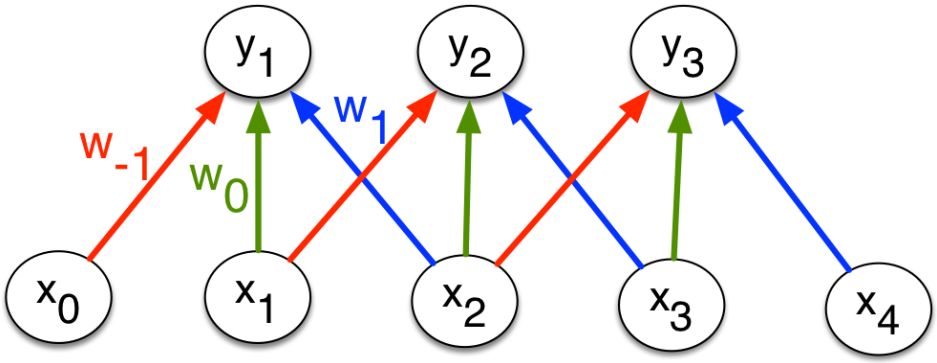
Recall what we need to do. Backprop is a message passing procedure, where each layer knows how to pass messages backwards through the computation graph. Let's determine the updates for convolution layers.

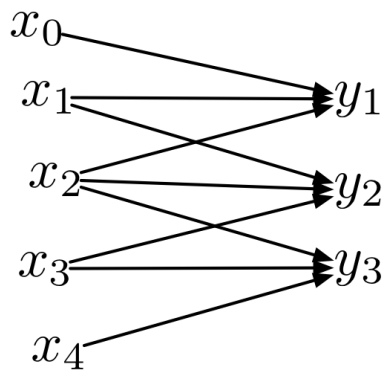
- We assume we are given the loss derivatives $\overline{y_{i,t}}$ with respect to the output units.
- We need to compute the cost derivatives with respect to the input units and with respect to the weights.

The only new feature is: how do we do backprop with tied weights?

Backprop Updates (Optional)

Consider the computation graph for the inputs:





Each input unit influences all the output units that have it within their receptive fields. Using the multivariate Chain Rule, we need to sum together the derivative terms for all these edges.

Backprop Updates (Optional)

Recall the formula for the convolution layer:

$$y_{i,t} = \sum_{j=1}^{J} \sum_{\tau=-R}^{R} w_{i,j,\tau} x_{j,t+\tau}.$$

We compute the derivatives, which requires summing over all the outputs units which have the input unit in their receptive field:

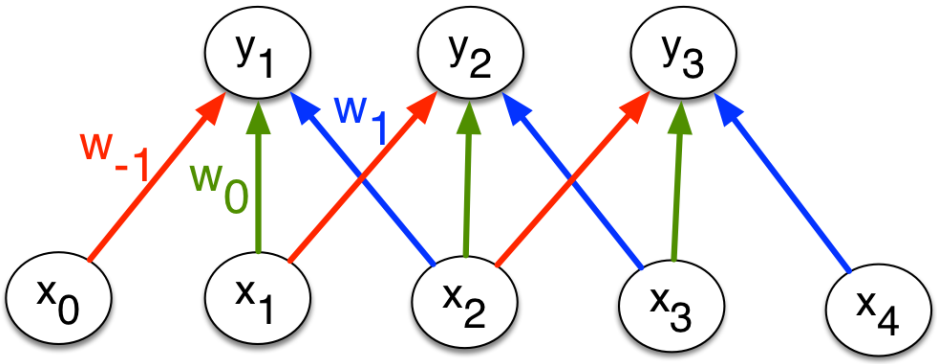
$$\begin{aligned}\overline{x_{j,t}} &= \sum_{\tau} \overline{y_{i,t-\tau}} \frac{\partial y_{i,t-\tau}}{\partial x_{j,t}} \\ &= \sum_{\tau} \overline{y_{i,t-\tau}} \, w_{i,j,\tau}\end{aligned}$$

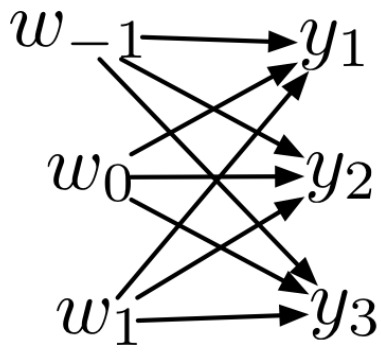
Written in terms of convolution,

$$\overline{\mathbf{x}_j} = \overline{\mathbf{y}_i} * \mathbf{w}_{i,j}.$$

Backprop Updates (Optional)

Consider the computation graph for the weights:





Each of the weights affects all the output units for the corresponding input and output feature maps.

Backprop Updates (Optional)

Recall the formula for the convolution layer:

$$y_{i,t} = \sum_{j=1}^{J} \sum_{\tau=-R}^{R} w_{i,j,\tau} x_{j,t+\tau}.$$

We compute the derivatives, which requires summing over all spatial locations:

$$\begin{aligned} \overline{w_{i,j,\tau}} &= \sum_t \overline{y_{i,t}} \frac{\partial y_{i,t}}{\partial w_{i,j,\tau}} \\ &= \sum_t \overline{y_{i,t}} \, x_{j,t+\tau} \end{aligned}$$