CSC 311: Introduction to Machine Learning

Lecture 6 - Neural Nets II

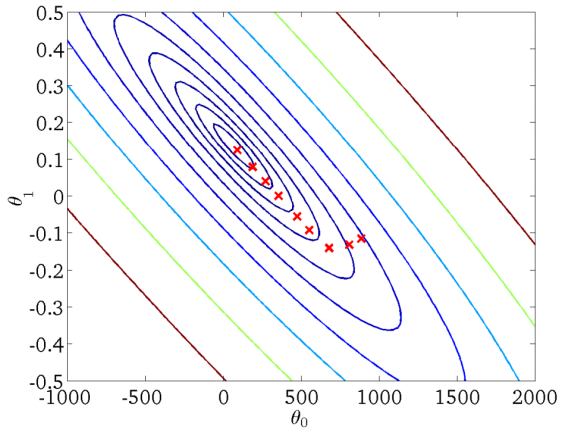
Roger Grosse Rahul G. Krishnan Guodong Zhang

University of Toronto, Fall 2021

Training neural networks with backpropagation

Recap: Gradient Descent

- **Recall:** gradient descent moves opposite the gradient (the direction of steepest descent)



- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers
- Conceptually, not any different from what we've seen so far — just higher dimensional and harder to visualize!
- We want to define a loss $\mathcal{L}$ and compute the gradient of the cost $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{w}$, which is the vector of partial derivatives.
  - This is the average of $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$ over all the training examples, so in this lecture we focus on computing $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$.

Univariate Chain Rule

- Let's now look at how we compute gradients in neural networks.

- We've already been using the univariate Chain Rule.
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t)) = \frac{\mathrm{d}f}{\mathrm{d}x} \frac{\mathrm{d}x}{\mathrm{d}t}.$$

Univariate Chain Rule

Recall: Univariate logistic least squares model

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Let's compute the loss derivatives $\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b}$

Univariate Chain Rule

How you would have done it in calculus class

$$\mathcal{L} = \frac{1}{2}(\sigma(wx+b)-t)^2$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\ &= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 \\ &= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)x\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\ &= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 \\ &= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\ &= (\sigma(wx+b)-t)\sigma'(wx+b)\end{aligned}$$

What are the disadvantages of this approach?

Univariate Chain Rule

A more structured way to do it

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = y - t$$
$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\sigma'(z)$$
$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}x$$
$$\frac{\partial\mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}$$

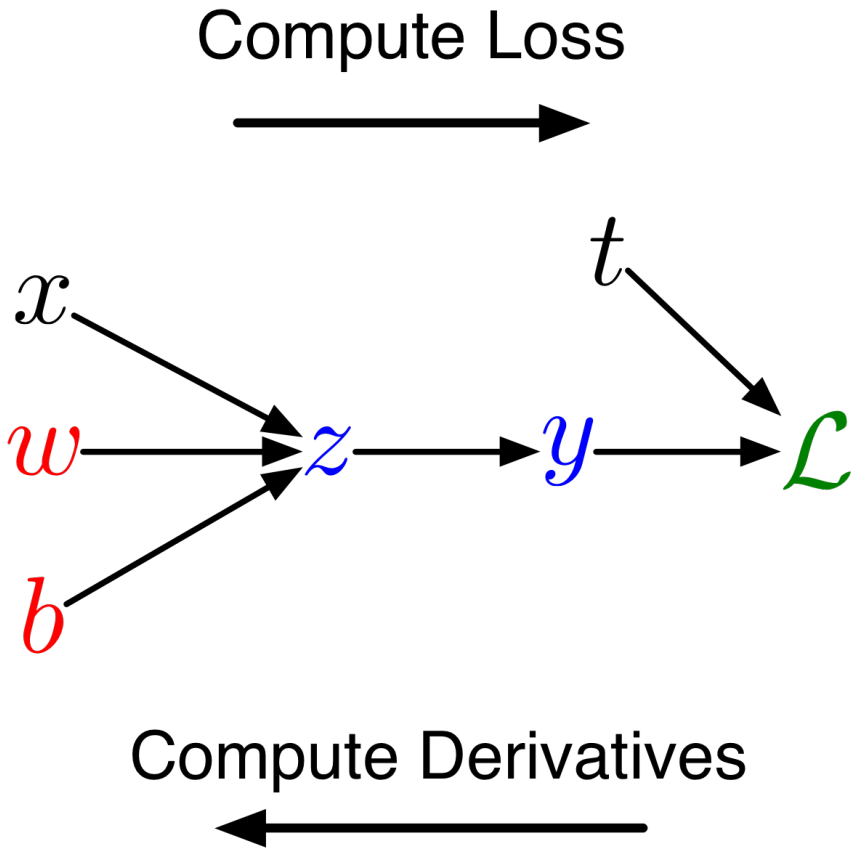
Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Univariate Chain Rule

- We can diagram out the computations using a **computation graph**.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Univariate Chain Rule

A slightly more convenient notation:

- Use $\overline{y}$ to denote the derivative $\mathrm{d}\mathcal{L}/\mathrm{d}y$, sometimes called the **error signal**.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

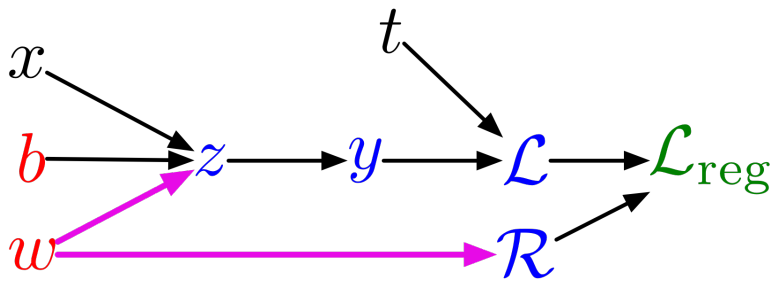
Computing the derivatives:

$$\overline{y} = y - t$$
$$\overline{z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,x$$
$$\overline{b} = \overline{z}$$

Multivariate Chain Rule

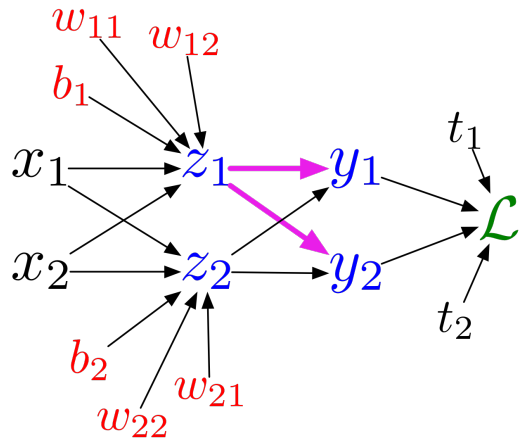
Problem: what if the computation graph has **fan-out** > 1?
This requires the **Multivariate Chain Rule**!

L_2-Regularized regression



$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Softmax regression

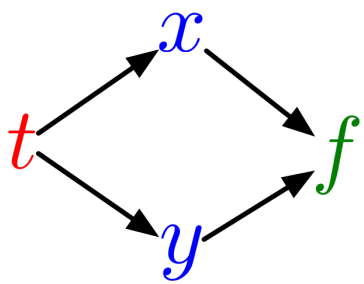


$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$
$$\mathcal{L} = -\sum_k t_k \log y_k$$

Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$



- Example:

$$\begin{aligned} f(x, y) &= y + e^{xy} \\ x(t) &= \cos t \\ y(t) &= t^2 \end{aligned}$$

- Plug in to Chain Rule:

$$\begin{aligned} \frac{\mathrm{d}f}{\mathrm{d}t} &= \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t \end{aligned}$$

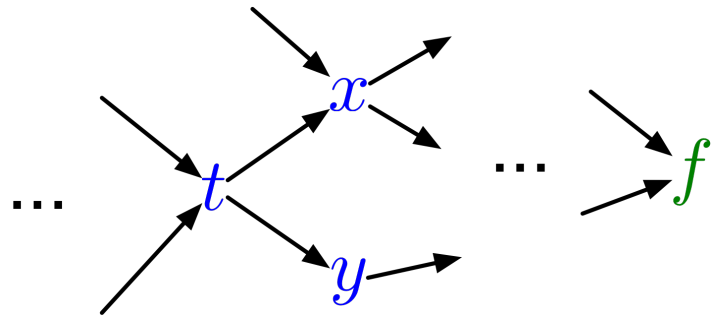
Multivariable Chain Rule

- In the context of backpropagation:

Mathematical expressions to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program



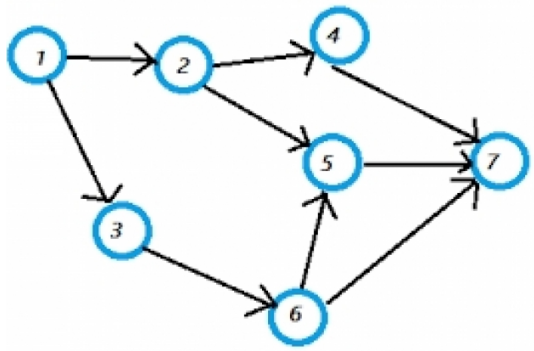
- In our notation:

$$\bar{t} = \bar{x}\frac{\mathrm{d}x}{\mathrm{d}t} + \bar{y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Backpropagation

Full backpropagation algorithm:

Let $v_1, \ldots, v_N$ be a **topological ordering** of the computation graph (i.e. parents come before children.)



v_N denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass

For $i = 1, \ldots, N$

Compute v_i as a function of $\text{Pa}(v_i)$

backward pass

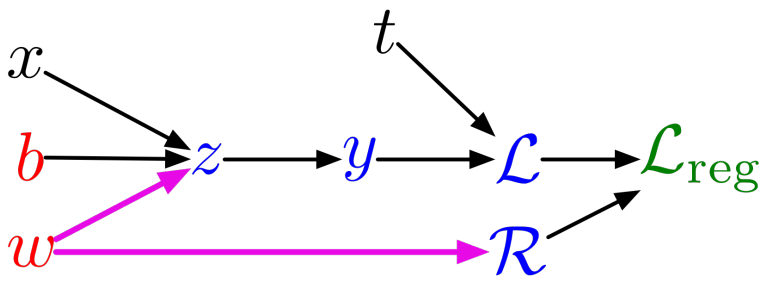
$\overline{v_N} = 1$

For $i = N-1, \ldots, 1$

$\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

Backpropagation

Example: univariate logistic least squares regression



Forward pass:

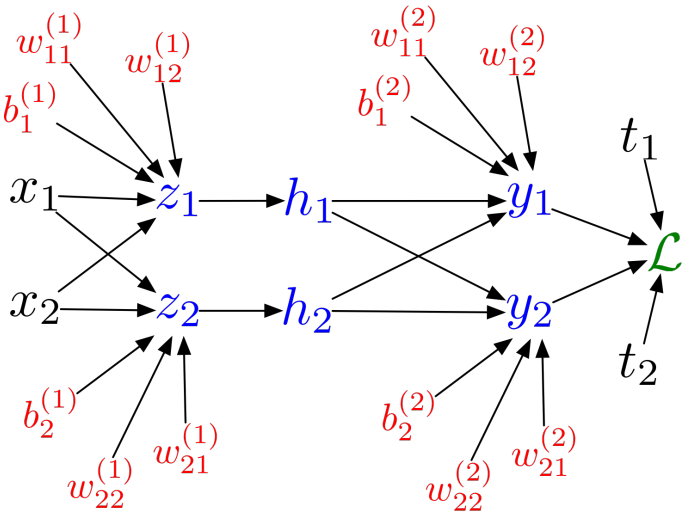
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Backward pass:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$
$$\overline{\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} = \overline{\mathcal{L}_{\text{reg}}}\,\lambda$$
$$\overline{\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} = \overline{\mathcal{L}_{\text{reg}}}$$
$$\overline{y} = \overline{\mathcal{L}}\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = \overline{\mathcal{L}}\,(y - t)$$
$$\overline{z} = \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} = \overline{z}\,x + \overline{\mathcal{R}}\,w$$
$$\overline{b} = \overline{z}\,\frac{\partial z}{\partial b} = \overline{z}$$

Backpropagation

Multilayer Perceptron (multiple outputs):



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}}\,(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

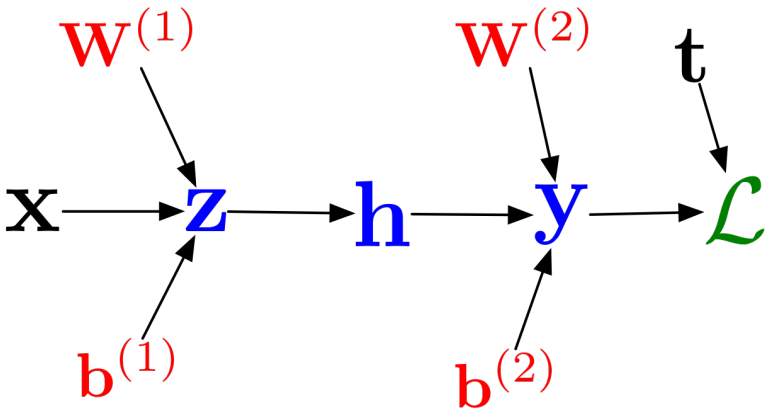
$$\overline{z_i} = \overline{h_i}\, \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i}\, x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

Backpropagation

In vectorized form:



Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$
$$\overline{\mathbf{y}} = \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t})$$
$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^\top$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = \mathbf{W}^{(2)\top}\overline{\mathbf{y}}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$
$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^\top$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Computational Cost

- Computational cost of forward pass: one add-multiply operation per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Computational cost of backward pass: two add-multiply operations per weight

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$
$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

- Rule of thumb: the backward pass is about as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.

Backpropagation

- Backprop is the algorithm for efficiently computing gradients in neural nets.
- Gradient descent with gradients computed via backprop is used to train the overwhelming majority of neural nets today.
  - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.
- Despite its practical success, backprop is believed to be neurally implausible.

Pytorch, Tensorflow, et al. (Optional)

- If we construct our networks out of a series of "primitive" operations (e.g., add, multiply) with specified routines for computing derivatives, backprop can be done in a completely mechanical, and automatic, way.
- This is called autodifferentiation or just autodiff.
- There are many autodiff libraries (e.g., PyTorch, Tensorflow, Jax, etc.)
- Practically speaking, autodiff automates the backward pass for you — but it's still important to know how things work under the hood.
- In CSC413, you'll learn more about how autodiff works and use an autodiff framework to build complex neural networks.

Convolutional Networks

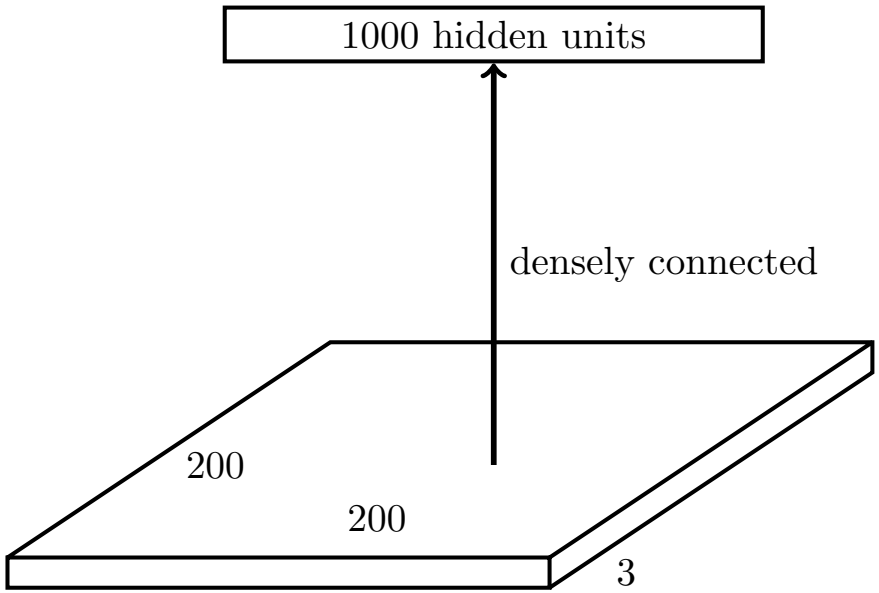
Overview

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

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

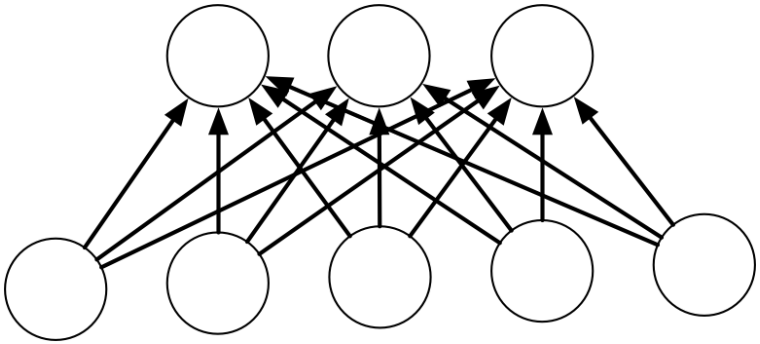
The same sorts of features that are useful in analyzing one part of the image will probably be useful for analyzing other parts as well.

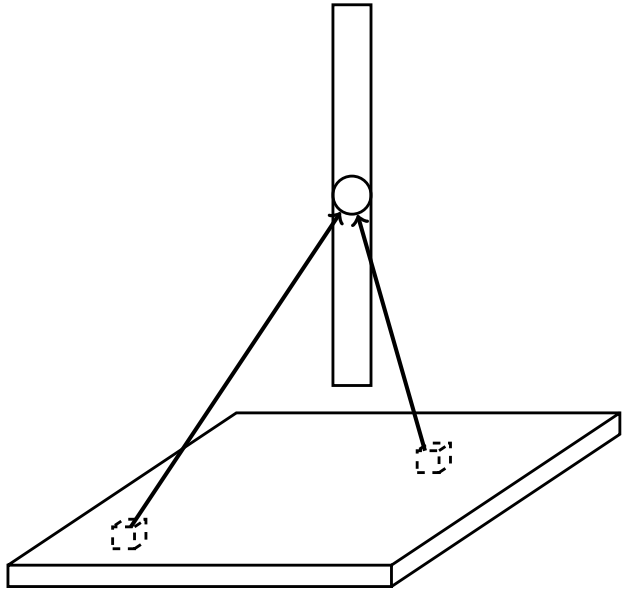
E.g., edges, corners, contours, object parts

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

Convolution Layers

Fully connected layers:

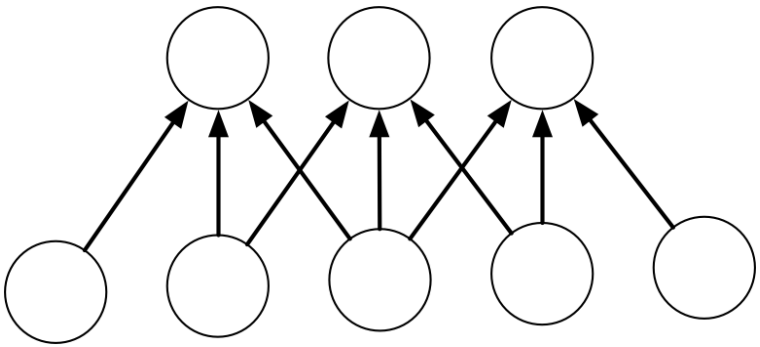


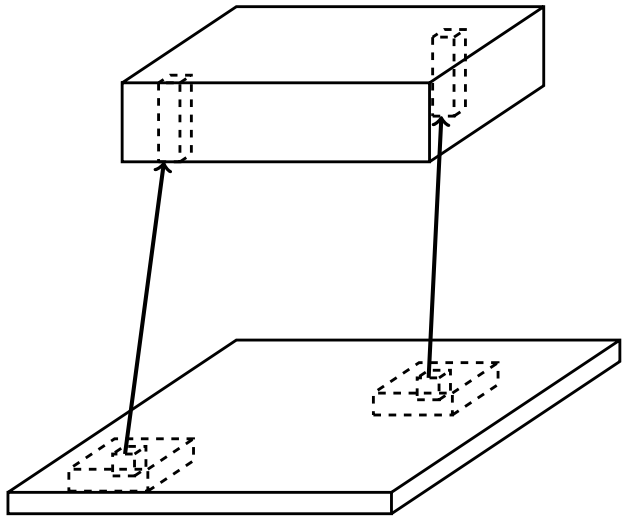


Each hidden unit looks at the entire image.

Convolution Layers

Locally connected layers:

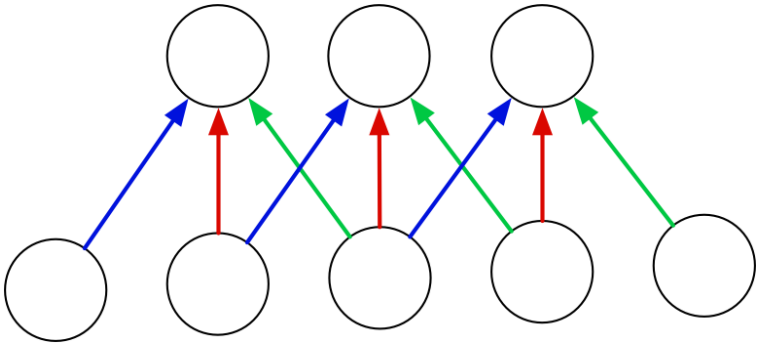


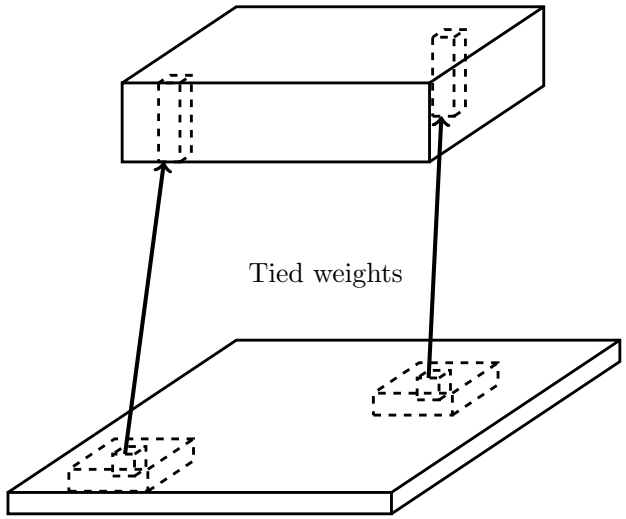


Each column of hidden units looks at a small region of the image.

Convolution Layers

Convolution layers:

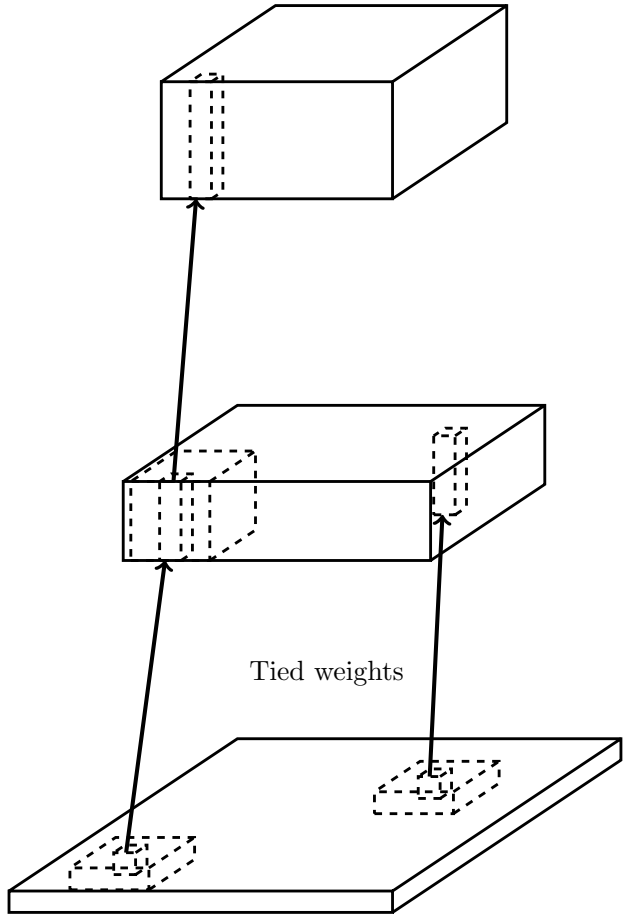




Each column of hidden units looks at a small region of the image, and the weights are shared between all image locations.

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. Convolution is another useful high-level operation.

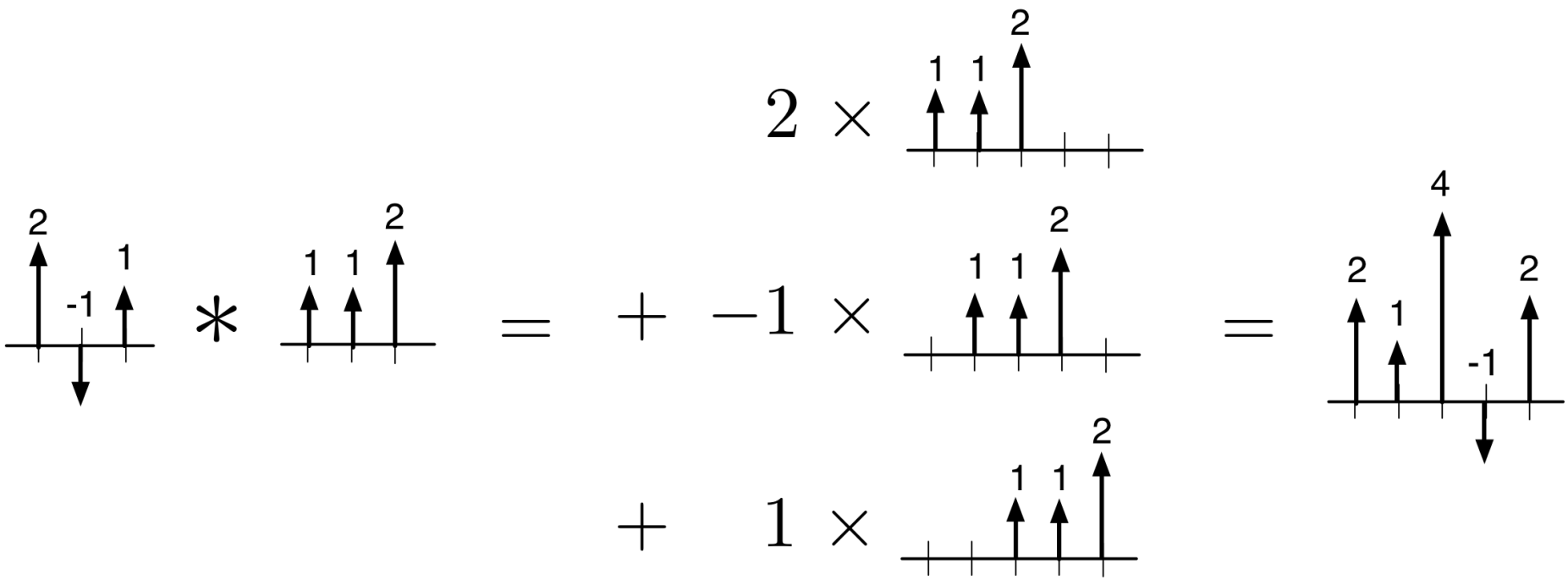
Let's look at the 1-D case first. If a and b are two arrays, the convolution is defined as:

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

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

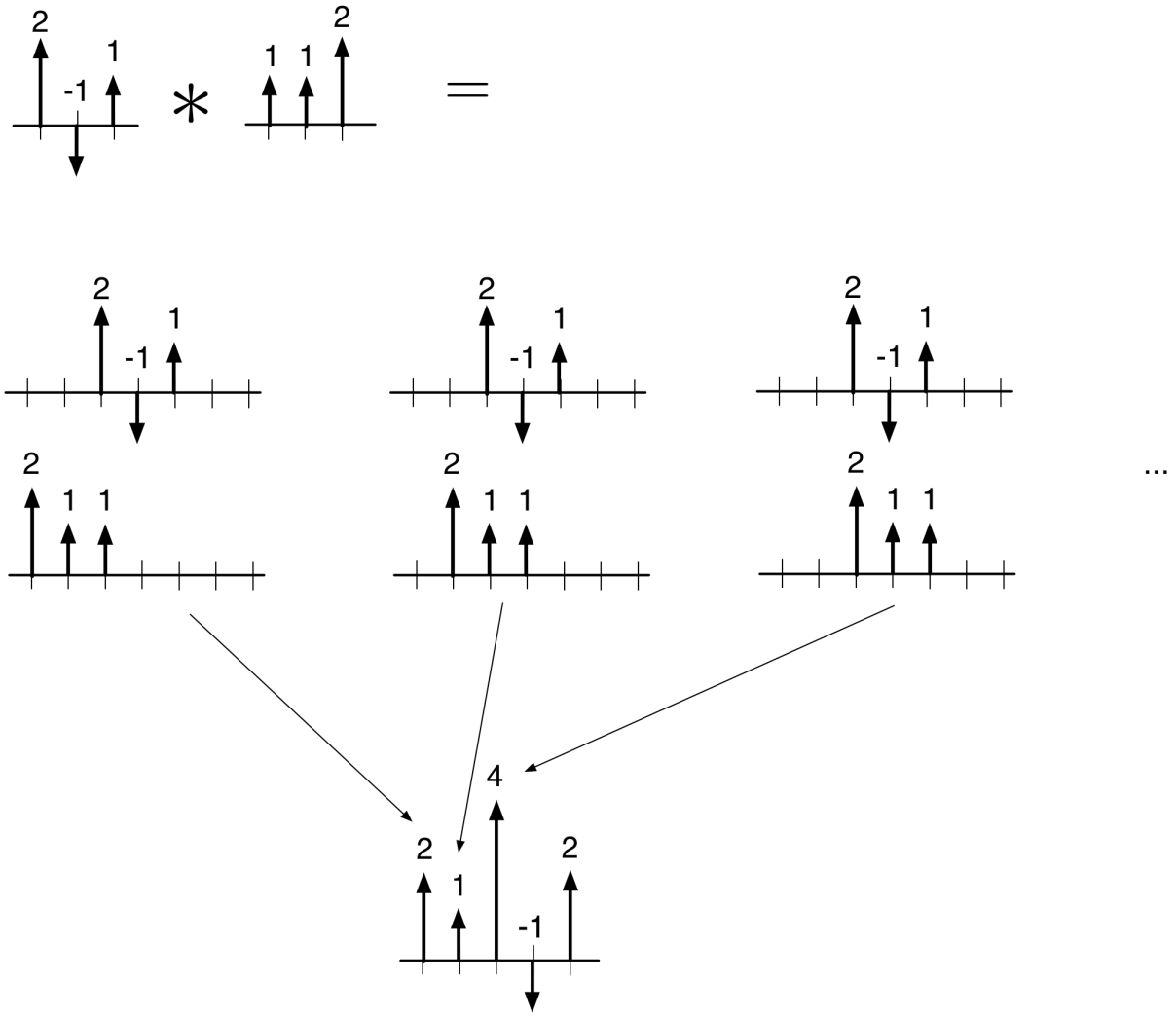
Convolution

Method 1: translate-and-scale



Convolution

Method 2: flip-and-filter



Convolution

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:

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

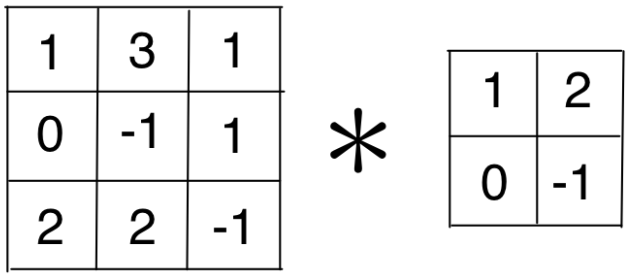
2-D Convolution

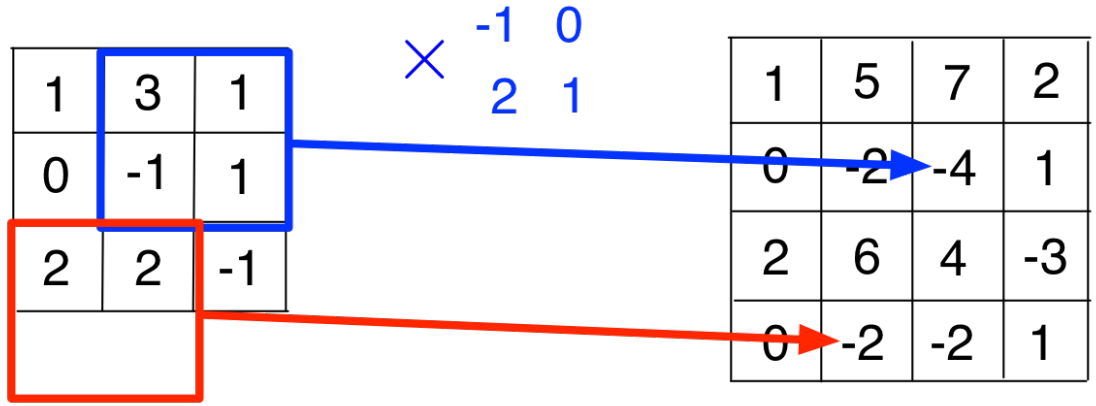
Method 1: Translate-and-Scale

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

2-D Convolution

Method 2: Flip-and-Filter (note that when used as a neural net layer, the flipping step is often omitted)





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

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

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

2-D Convolution

What does this convolution kernel do?

$*$

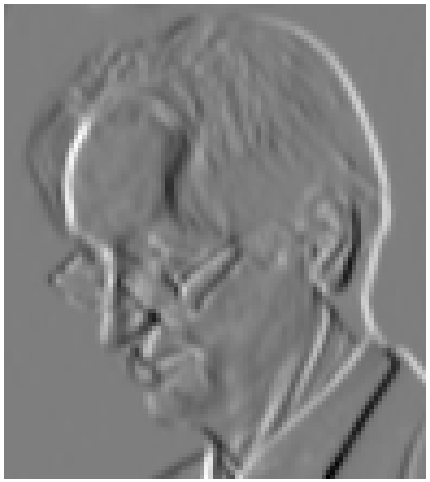
1	0	-1
2	0	-2
1	0	-1

2-D Convolution

What does this convolution kernel do?

$*$

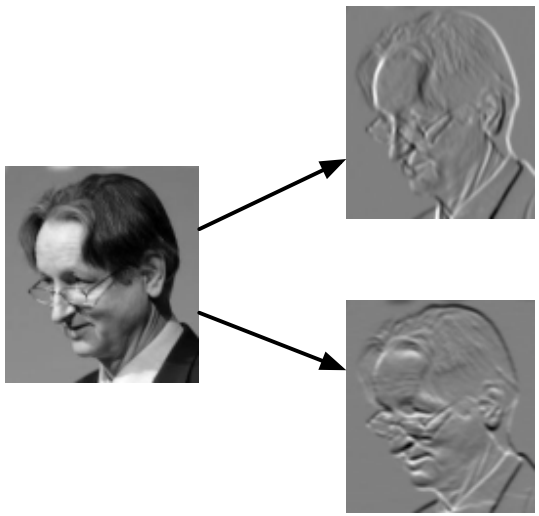
1	0	-1
2	0	-2
1	0	-1



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.

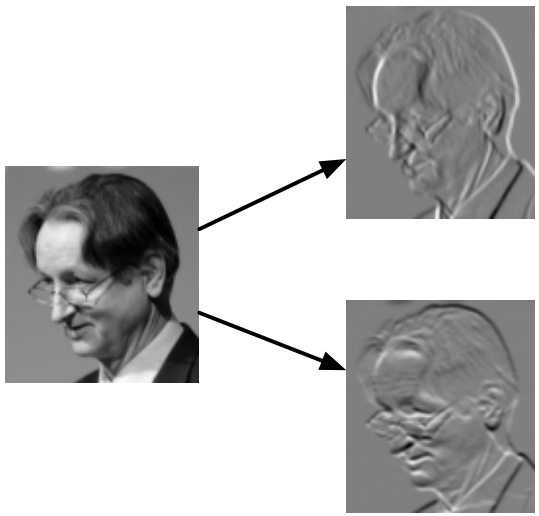


convolution

Convolutional networks

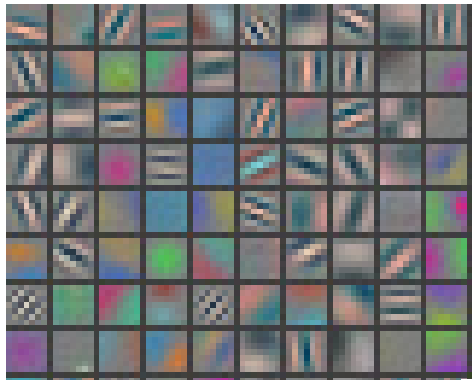
Let's finally turn to convolutional networks. These have two kinds of layers: detection layers (or convolution layers), and pooling layers.

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

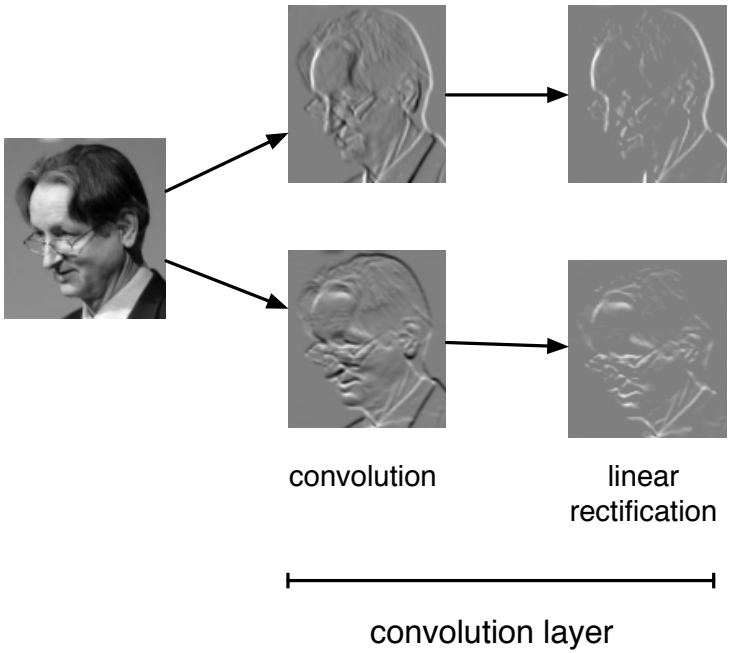
Example first-layer filters



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

Convolutional networks

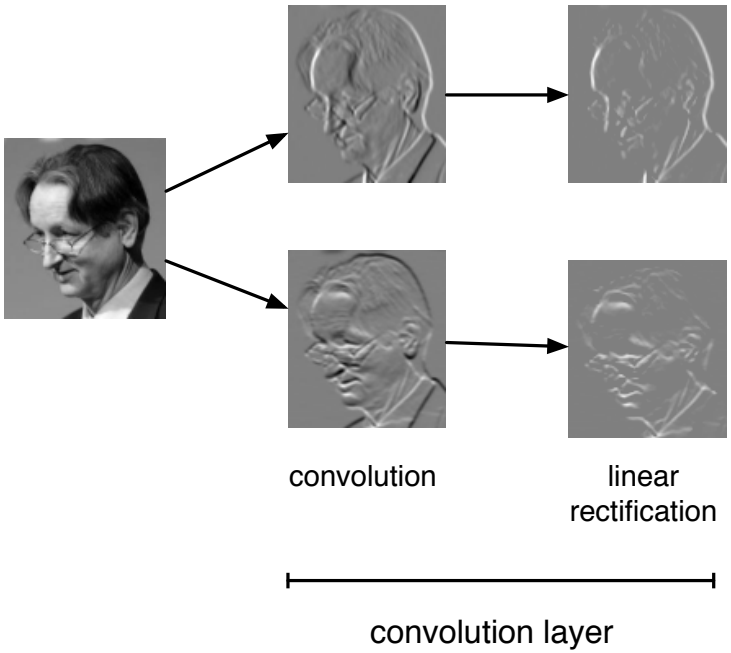
It's common to apply a linear rectification nonlinearity: $y_i = \max(z_i, 0)$



Why might we do this?

Convolutional networks

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

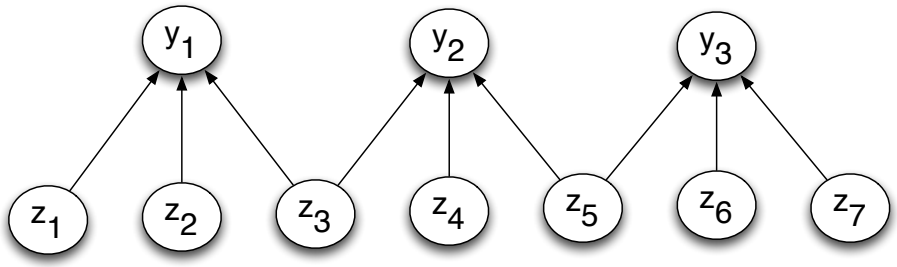


Why might we do this?

- 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

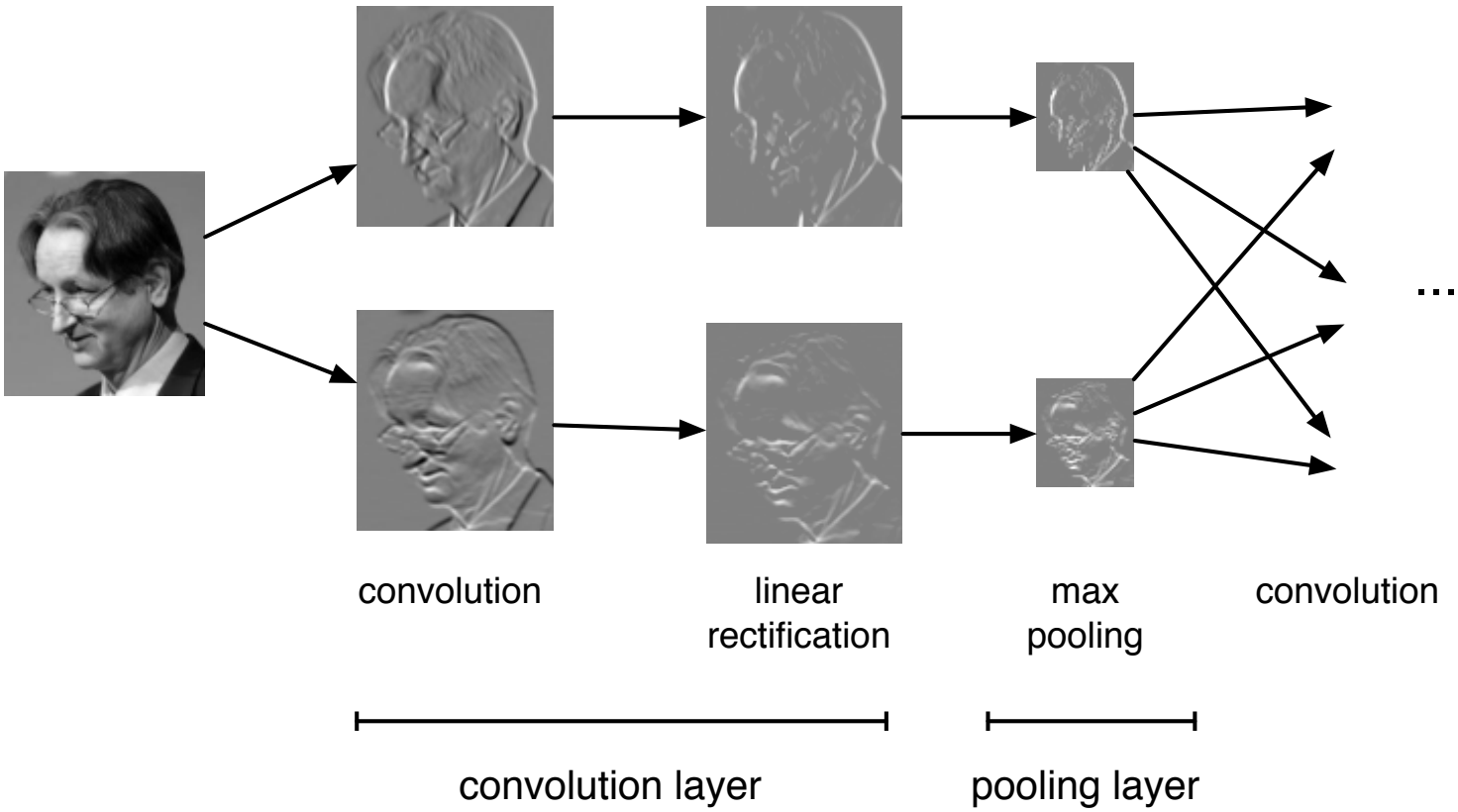
The other type of layer in a pooling layer. These layers reduce the size of the representation and build in invariance to small transformations.



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

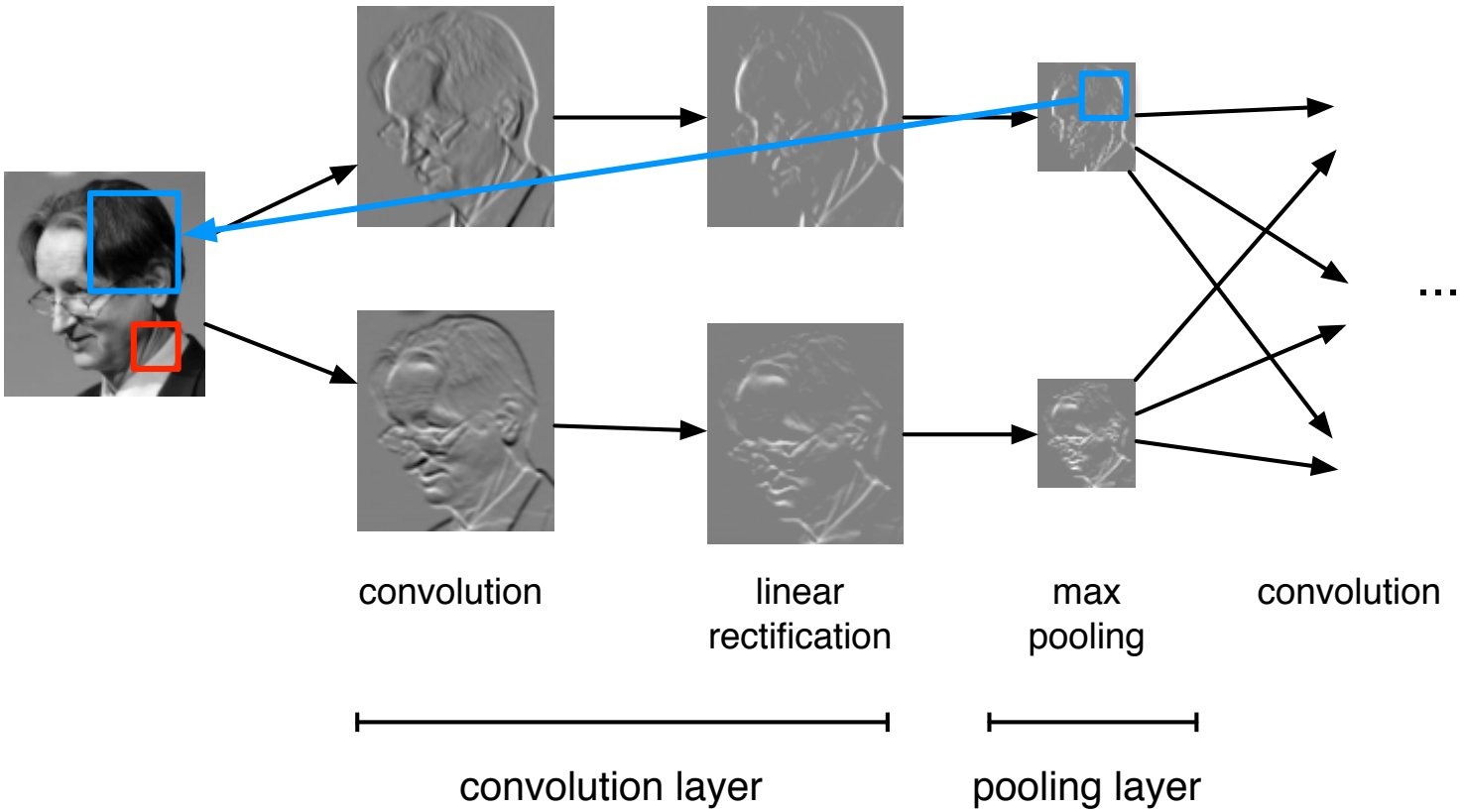
$$y_i = \max_{j \text{ in pooling group}} z_j$$

Convolutional networks



Convolutional networks

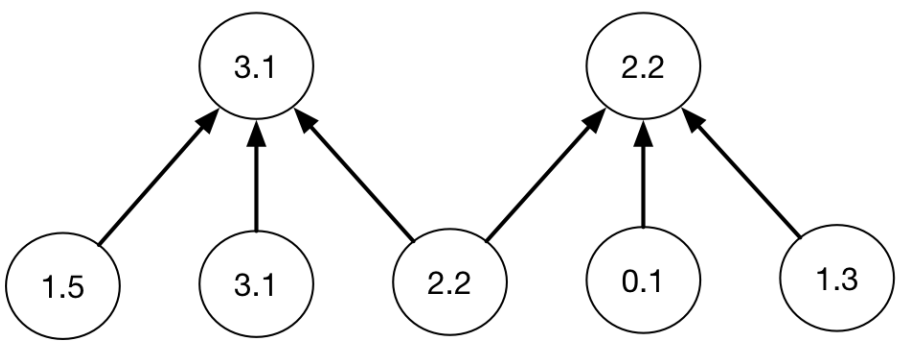
Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in the lower layers.

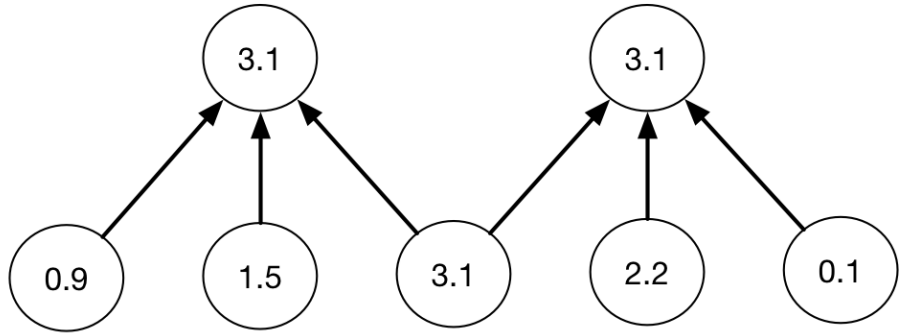


Equivariance and Invariance

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

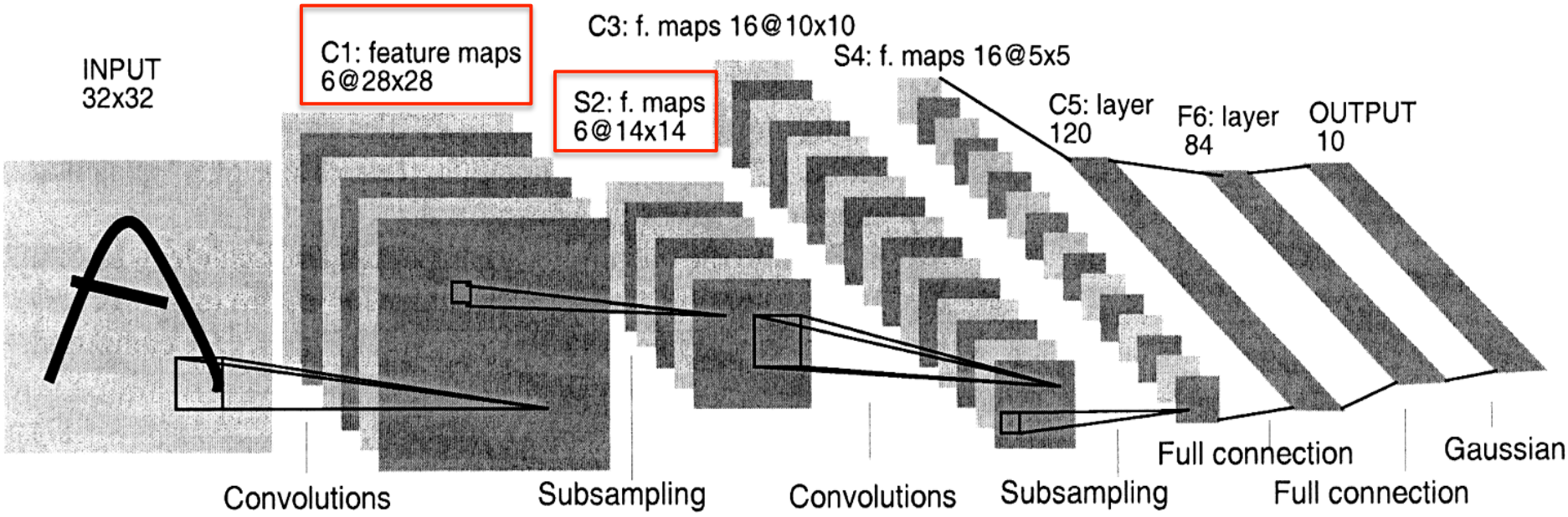
Each layer consists of several feature maps, or channels each of which is an array.

- 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.

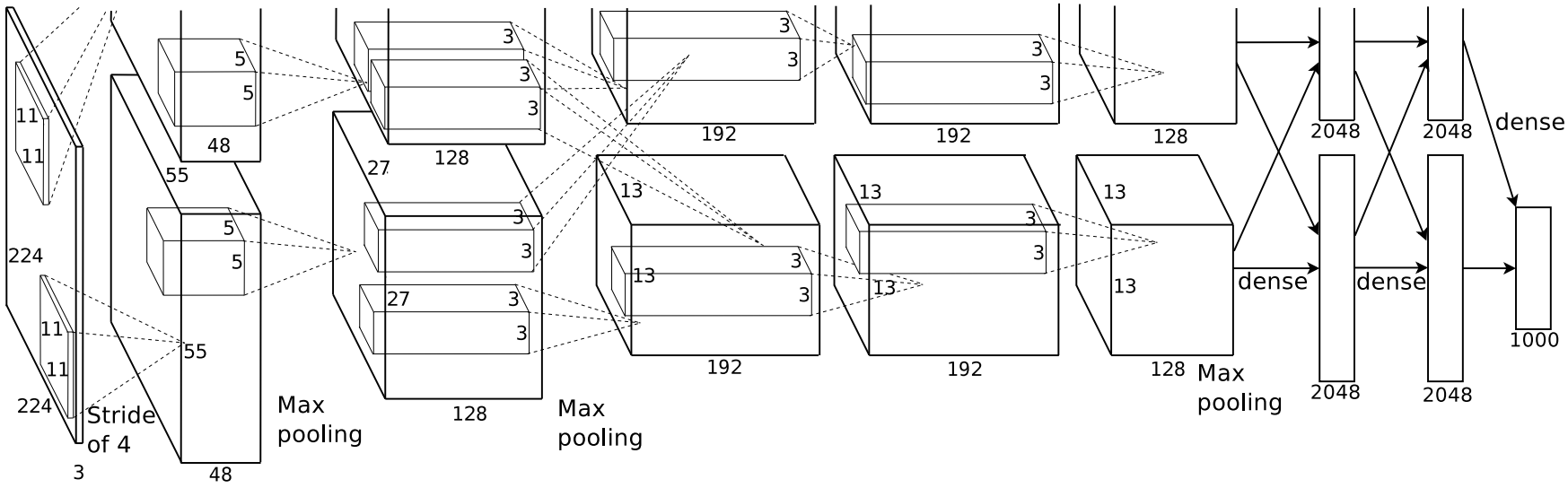
LeNet

Here's the LeNet architecture, which was applied to handwritten digit recognition on MNIST in 1998:



AlexNet

- AlexNet, essentially like LeNet but scaled up in every way (more layers, more units, more connections, etc.):



(Krizhevsky et al., 2012)

- AlexNet's stunning performance on the ImageNet competition is what got everyone excited about deep learning in 2012.

Classification

ImageNet results over the years. There are 1000 classes. Note that errors are top-5 errors (the network gets to make 5 guesses), so chance = 0.5%.

Year	**Model**	**Top-5 error**
2010	Hand-designed descriptors + SVM	28.2%
2011	Compressed Fisher Vectors + SVM	25.8%
2012	AlexNet	16.4%
2013	a variant of AlexNet	11.7%
2014	GoogLeNet	6.6%
2015	deep residual nets	4.5%

Human-level performance is around 5.1%.

They stopped running the object recognition competition because the performance is already so good.