

# CSC 311: Introduction to Machine Learning

## Lecture 6 - Neural Networks II

Rahul G. Krishnan

University of Toronto, Fall 2023

# Outline

- 1 Back-Propagation
- 2 Convolutional Networks

1 Back-Propagation

2 Convolutional Networks

# Learning Weights in a Neural Network

- Goal is to learn weights in a multi-layer neural network using gradient descent.
- Weight space for a multi-layer neural net: one set of weights for each unit in every layer of the network
- Define a loss  $\mathcal{L}$  and compute the gradient of the cost  $d\mathcal{J}/d\mathbf{w}$ , the average loss over all the training examples.
- Let's look at how we can calculate  $d\mathcal{L}/d\mathbf{w}$ .

## Example: Two-Layer Neural Network

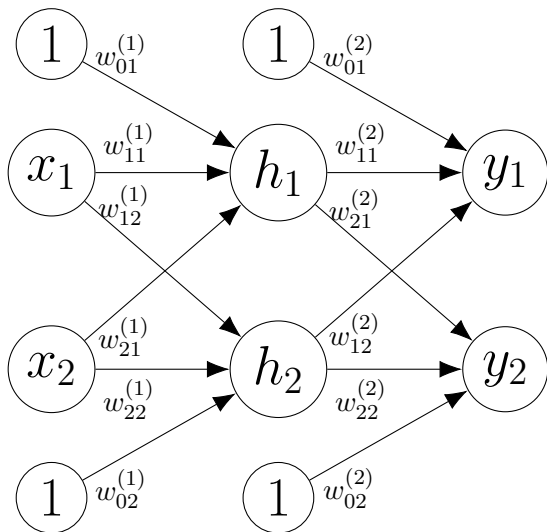


Figure: Two-Layer Neural Network

# Computations for Two-Layer Neural Network

A neural network computes a composition of functions.

$$z_1^{(1)} = w_{01}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2$$

$$h_1 = \sigma(z_1^{(1)})$$

$$z_1^{(2)} = w_{01}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{12}^{(2)} \cdot h_2$$

$$y_1 = z_1^{(2)}$$

$$z_2^{(1)} =$$

$$h_2 =$$

$$z_2^{(2)} =$$

$$y_2 =$$

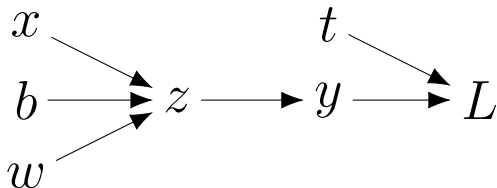
$$L = \frac{1}{2} \left( (y_1 - t_1)^2 + (y_2 - t_2)^2 \right)$$

# Simplified Example: Logistic Least Squares

$$z = wx + b$$

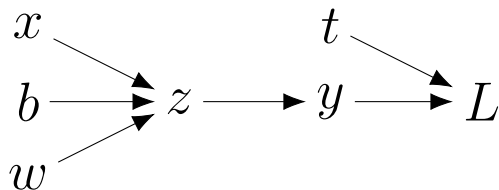
$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



# Computation Graph

- The nodes represent the inputs and computed quantities.
- The edges represent which nodes are computed directly as a function of which other nodes.





# Uni-variate Chain Rule

Let  $z = f(y)$  and  $y = g(x)$  be uni-variate functions.  
Then  $z = f(g(x))$ .

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

# Univariate Chain Rule

How you would have done it in calculus class

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

What are the disadvantages of this approach?

# Logistic Least Squares: Gradient for $w$

Computing the gradient for  $w$ :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial w} \\ &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w} \\ &= (y - t) \sigma'(z) x \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) x\end{aligned}$$

Computing the loss:

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

# Logistic Least Squares: Gradient for $b$

Computing the gradient for  $b$ :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \\ &= \\ &= \\ &= \end{aligned}$$

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

# Logistic Least Squares: Gradient for $b$

Computing the gradient for  $b$ :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial b} \\ &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b} \\ &= (y - t) \sigma'(z) 1 \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) 1\end{aligned}$$

Computing the loss:

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

## Comparing Gradient Computations for $w$ and $b$

Computing the gradient for  $w$ :    Computing the gradient for  $b$ :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w} \\ &= (y - t) \sigma'(z) x\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b} \\ &= (y - t) \sigma'(z) 1\end{aligned}$$

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

# Structured Way of Computing Gradients

Computing the gradients:

$$\frac{\partial \mathcal{L}}{\partial y} = (y - t)$$

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y} \sigma'(z)$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} \frac{dz}{dw} = \frac{d\mathcal{L}}{dz} x$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz} \frac{dz}{db} = \frac{d\mathcal{L}}{dz} 1$$

Computing the loss:

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

# Error Signal Notation

- Let  $\bar{y}$  denote the derivative  $d\mathcal{L}/dy$ , called the **error signal**.
- Error signals are just values our program is computing (rather than a mathematical operation).

**Computing the loss:**

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

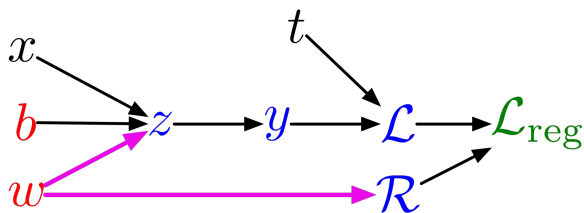
**Computing the derivatives:**

$$\begin{aligned}\bar{y} &= (y - t) \\ \bar{z} &= \bar{y} \sigma'(z) \\ \bar{w} &= \bar{z} x \quad \bar{b} = \bar{z}\end{aligned}$$



# Computation Graph has a Fan-Out $> 1$

## $L_2$ -Regularized Regression



$$z = wx + b$$

$$y = \sigma(z)$$

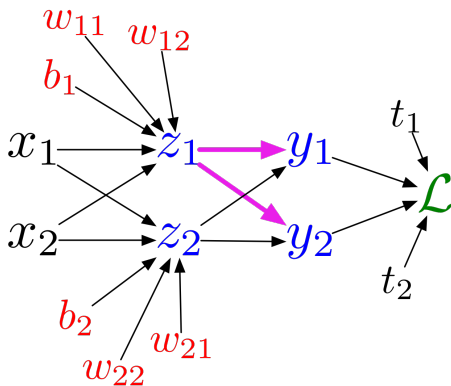
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

# Computation Graph has a Fan-Out > 1

## Softmax Regression



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$

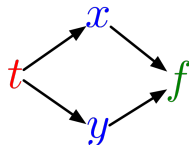
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

$$\mathcal{L} = - \sum_k t_k \log y_k$$

# Multi-variate Chain Rule

Suppose we have functions  $f(x, y)$ ,  $x(t)$ , and  $y(t)$ .

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



Example:

$$f(x, y) = y + e^{xy}$$

$$x(t) = \cos t$$

$$y(t) = t^2$$

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

$$= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t$$

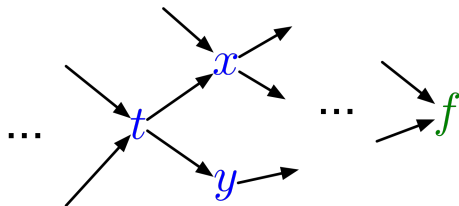
# Multi-variate Chain Rule

In the context of back-propagation:

Mathematical expressions  
to be evaluated

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Values already computed  
by our program



In our notation:

$$\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}$$

## Full Backpropagation Algorithm:

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

$v_N$  denotes the variable for which we're trying to compute gradients.

- forward pass:

For  $i = 1, \dots, N$ ,

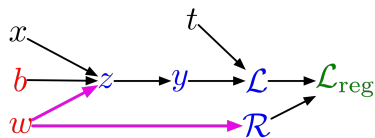
Compute  $v_i$  as a function of  $\text{Parents}(v_i)$ .

- backward pass:

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

$$\bar{v}_i = \sum_{j \in \text{Children}(v_i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

# Backpropagation for Regularized Logistic Least Squares



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:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$

$$\begin{aligned}\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} \\ &= \overline{\mathcal{L}_{\text{reg}}} \lambda\end{aligned}$$

$$\begin{aligned}\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} \\ &= \overline{\mathcal{L}_{\text{reg}}}\end{aligned}$$

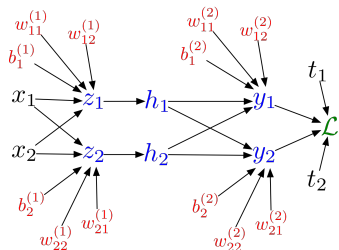
$$\begin{aligned}\overline{y} &= \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \\ &= \overline{\mathcal{L}}(y - t)\end{aligned}$$

$$\begin{aligned}\overline{z} &= \overline{y} \frac{dy}{dz} \\ &= \overline{y} \sigma'(z)\end{aligned}$$

$$\begin{aligned}\overline{w} &= \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw} \\ &= \overline{z}x + \overline{\mathcal{R}}w\end{aligned}$$

$$\begin{aligned}\overline{b} &= \overline{z} \frac{\partial z}{\partial b} \\ &= \overline{z}\end{aligned}$$

# Backpropagation for Two-Layer Neural Network



**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:**

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

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

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

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

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

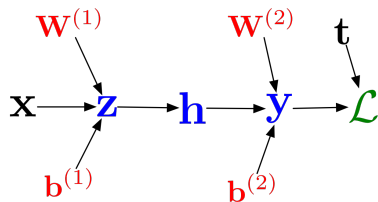
$$\overline{z_i} = \overline{h_i} \sigma'(z_i)$$

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

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

# Backpropagation for Two-Layer Neural Network

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:

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

$$\bar{\mathbf{y}} = \bar{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{\mathbf{W}^{(2)}} = \bar{\mathbf{y}}\mathbf{h}^\top$$

$$\bar{\mathbf{b}}^{(2)} = \bar{\mathbf{y}}$$

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)\top}\bar{\mathbf{y}}$$

$$\bar{\mathbf{z}} = \bar{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{\mathbf{W}^{(1)}} = \bar{\mathbf{z}}\mathbf{x}^\top$$

$$\bar{\mathbf{b}}^{(1)} = \bar{\mathbf{z}}$$



# 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

$$\begin{aligned}\overline{w_{ki}^{(2)}} &= \overline{y_k} h_i \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)}\end{aligned}$$

- One backward pass is 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

- 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.
- We need to be careful with network initialization (should not set all weights = 0)
- Even optimization algorithms 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.

# Auto-Differentiation

- Suppose we construct our networks out of a series of “primitive” operations (e.g., add, multiply) with specified routines for computing derivatives.
- [Autodifferentiation](#) performs backprop in a completely mechanical and automatic way.
- Many autodiff libraries: PyTorch, Tensorflow, Jax, etc.
- Although autodiff automates the backward pass for you, it’s still important to know how things work under the hood.
- In CSC413, learn more about how autodiff works and use an autodiff framework to build complex neural networks.

1 Back-Propagation

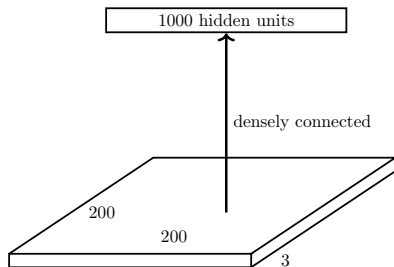
2 Convolutional Networks

# Robust to Transformations

- Must be robust to transformations or distortions:
  - ▶ change in pose/viewpoint
  - ▶ change in illumination
  - ▶ deformation
  - ▶ occlusion (some objects are hidden behind others)
- We would like the network to be **invariant**:  
if the image is transformed slightly,  
the classification shouldn't change.

## Too Many Parameters

Want to train a network that takes a  $200 \times 200$  RGB image as input.



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

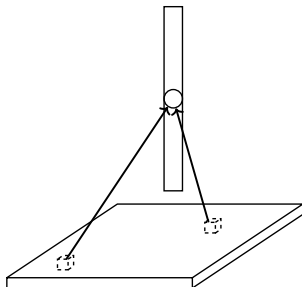
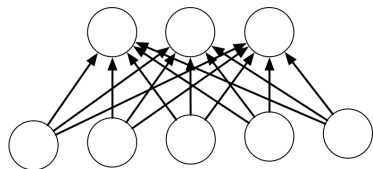
Too many parameters! Input size =  $200 \times 200 \times 3 = 120\text{K}$ .  
Parameters =  $120\text{K} \times 1000 = 120$  million.

# Shared Structures in the Network

- Some features, e.g. edges, corners, contours, object parts, may be useful in multiple locations in the image.
  
- We want feature detectors that are applicable in multiple locations in the image.

# Convolution Layers

Fully connected layers:

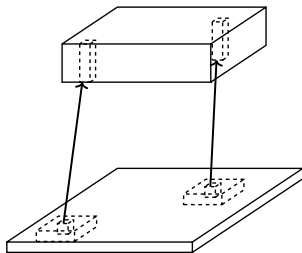
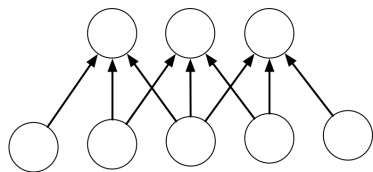


Each hidden unit looks at the entire image.



# Convolution Layers

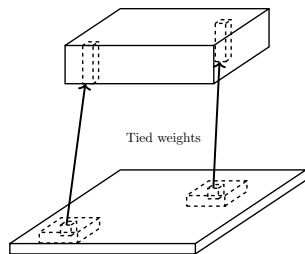
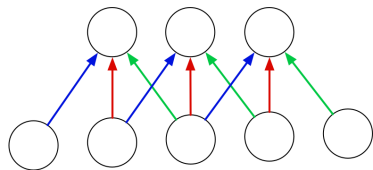
Locally connected layers:



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

# Convolution Layers

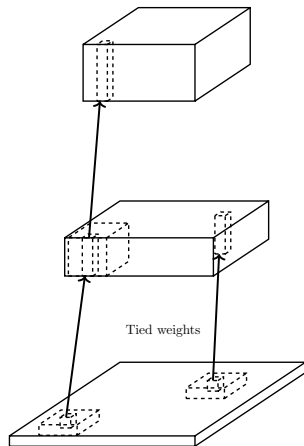
Convolution layers:



Each set 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:



# 1-D Convolution

We have two signals/arrays  $x$  and  $w$ .

- $x$  is an input signal (e.g. a waveform or an image).
- $w$  is a set of  $k$  weights (also referred to as a kernel or filter).
- Often zero pad  $x$  to an infinite array

The  $t$ -th value in the convolution is defined below.

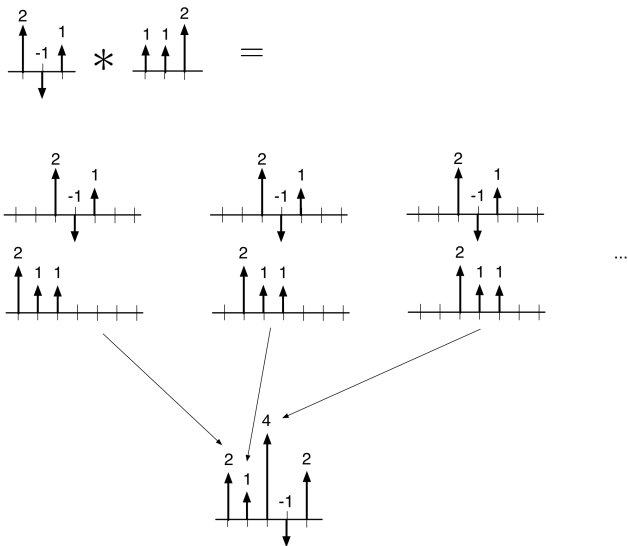
$$(x * w)[t] = \sum_{\tau=0}^{k-1} x[t - \tau]w[\tau].$$

# Convolution Method 1: Translate-And-Scale

The diagram illustrates the convolution of two discrete signals using the Translate-And-Scale method. It shows the following steps:

- Input 1:** A discrete signal with values 2, -1, and 1 at three discrete time steps.
- Input 2:** A discrete signal with values 1, 1, and 2 at three discrete time steps.
- Step 1:** The convolution is expressed as the sum of three scaled and translated versions of the second signal:
  - $+ 2 \times$  (Signal 2 shifted left by 1 step)
  - $- 1 \times$  (Signal 2)
  - $+ 1 \times$  (Signal 2 shifted right by 1 step)
- Step 2:** The resulting signal is the sum of these three components, yielding values 2, 1, 4, -1, and 2 at five discrete time steps.

# Convolution Method 2: Flip-And-Filter



# 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  $x$  and  $w$  are two 2-D arrays, then:

$$(x * w)[i, j] = \sum_s \sum_t x[i - s, j - t] * w[s, t].$$



## 2-D Convolution: Translate-and-Scale

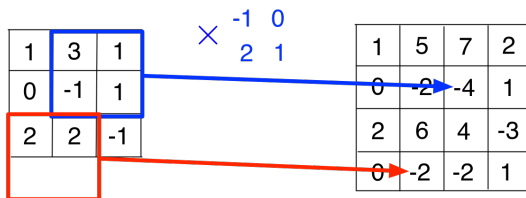
$$\begin{array}{|c|c|c|} \hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array} = 1 \times \begin{array}{|c|c|c|c|} \hline 1 & 3 & 1 & \\ \hline 0 & -1 & 1 & \\ \hline 2 & 2 & -1 & \\ \hline & & & \\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

## 2-D Convolution: Flip-and-Filter

1	3	1
0	-1	1
2	2	-1

 \* 

1	2
0	-1



## Example 1: What does this convolution kernel do?



\*

0	1	0
1	4	1
0	1	0



## Example 2: What does this convolution kernel do?



\*

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



## Example 3: What does this convolution kernel do?



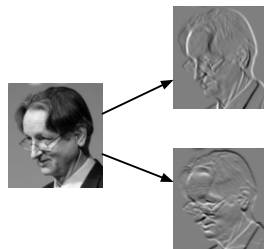
\*

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



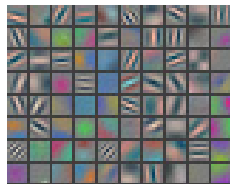
# Convolution Layer in Convolutional Networks

- Two types of layers: **convolution layers** (or detection layer), and **pooling layers**.
- The convolution layer has a set of filters and produces a set of feature maps.
- Each feature map is a result of convolving the image with a filter.



convolution

Example first-layer filters



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

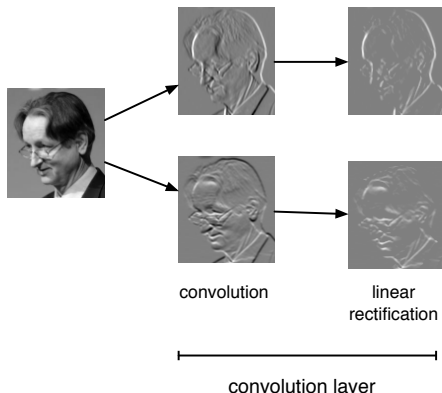
# Non-linearity in Convolutional Networks

Common to apply a linear rectification nonlinearity:

$$y_i = \max(z_i, 0).$$

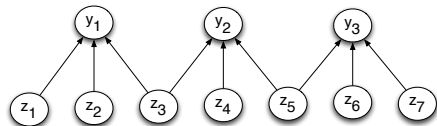
Why might we do this?

Convolution is a linear operation. Therefore, we need a nonlinearity, otherwise 2 convolution layers would be no more powerful than 1.



# Pooling Layers

These layers reduce the size of the representation and build in in-variance to small transformations.

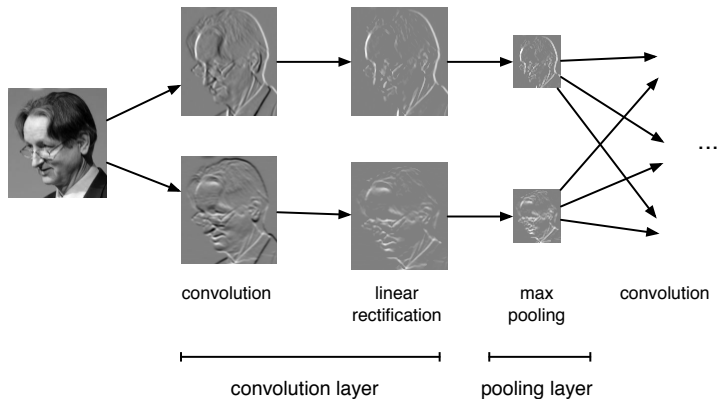


Most commonly, we use [max-pooling](#), which computes the maximum value of the units in a [pooling group](#):

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

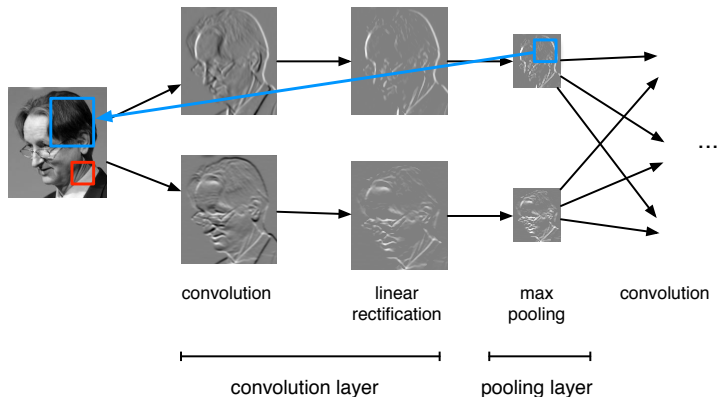


# Convolutional networks



# Convolutional Network Structure

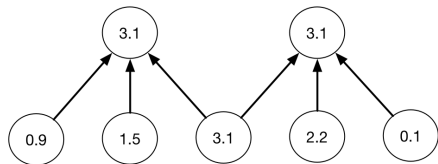
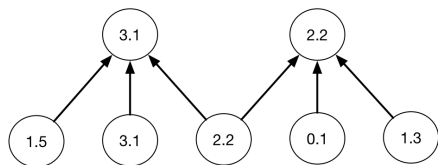
Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in the lower layers.



# Equivariance and Invariance

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.
- Want 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

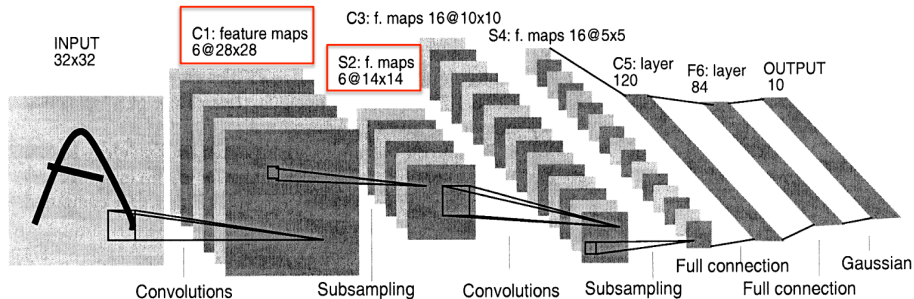
Each layer consists of several **feature maps**, or **channels** each of which is an array.

- 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.

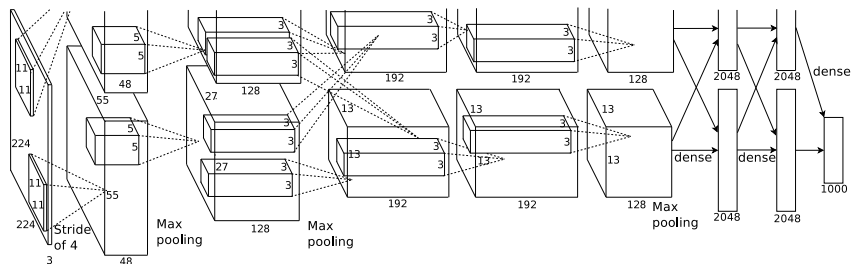
# LeNet

The LeNet architecture applied to handwritten digit recognition on MNIST in 1998:



# AlexNet

AlexNet, 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.

# ImageNet Results Over the Years

There are 1000 classes. Top-5 errors mean that the network can make 5 guesses for each image. So chance is 0.5%.

<b>Year</b>	<b>Model</b>	<b>Top-5 error</b>
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%.

No longer running the object recognition competition because the performance is already so good.