CSC 311: Introduction to Machine Learning

Lecture 6 - Neural Networks II

Michael Zhang   Chandra Gummaluru

University of Toronto, Winter 2023

Outline

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 $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{w}$, the average loss over all the training examples.
- Let's look at how we can calculate $\mathrm{d}\mathcal{L}/\mathrm{d}\mathbf{w}$.

Example: Two-Layer Neural Network

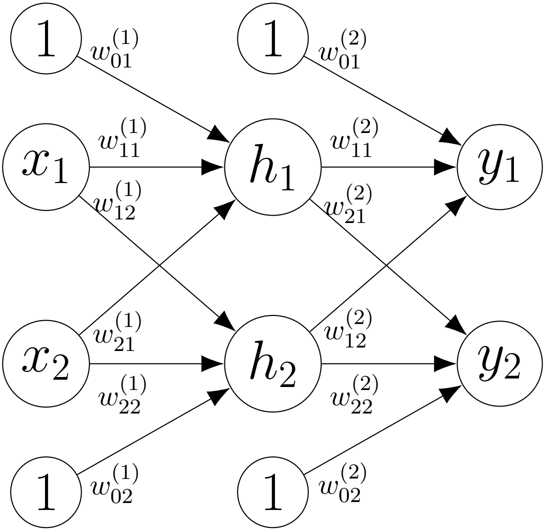


Figure: Two-Layer Neural Network

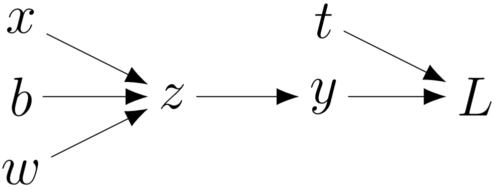
Computations for Two-Layer Neural Network

A neural network computes a composition of functions.

$$
\begin{aligned}
z_1^{(1)} &= w_{01}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 \\
h_1 &= \sigma(z_1) \\
z_1^{(2)} &= w_{01}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{21}^{(2)} \cdot h_2 \\
y_1 &= z_1 \\
z_2^{(1)} &= \\
h_2 &= \\
z_2^{(2)} &= \\
y_2 &= \\
L &= \frac{1}{2}\left((y_1 - t_1)^2 + (y_2 - t_2)^2\right)
\end{aligned}
$$

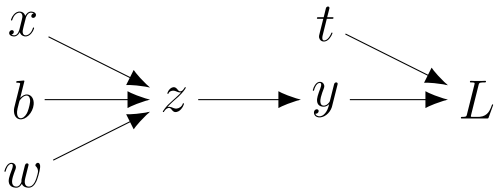
Simplified Example: Logistic Least Squares

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



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 $z = f(g(x))$.

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \frac{\mathrm{d}y}{\mathrm{d}x}$$

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:

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

Logistic Least Squares: Gradient for b

Computing the gradient for b:

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

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:

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

Computing the gradient for b:

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

Computing the loss:

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

Structured Way of Computing Gradients

Computing the gradients:

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

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,1$$

Computing the loss:

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

Error Signal Notation

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

Computing the loss:

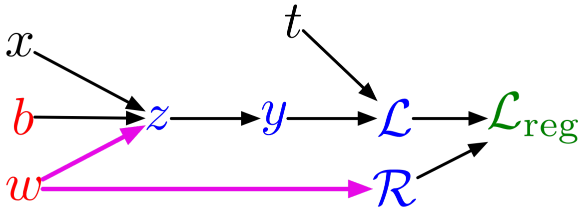
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

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

Computation Graph has a Fan-Out > 1

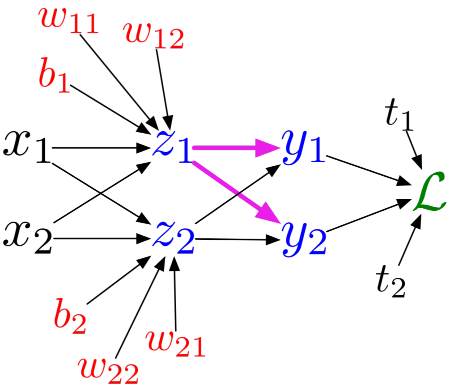
L_2-Regularized Regression



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

Computation Graph has a Fan-Out > 1

Softmax Regression



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

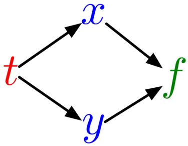
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

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

Multi-variate Chain Rule

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

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



Example:

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

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

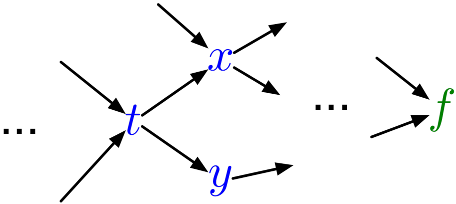
Multi-variate Chain Rule

In the context of back-propagation:

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:

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

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 for which we're trying to compute gradients.

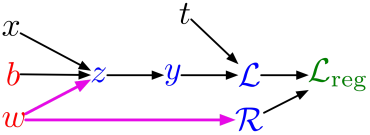
- forward pass:

$$\text{For } i = 1, \ldots, N, \\ \text{Compute } v_i \text{ as a function of Parents}(v_i).$$

- backward pass:

$$\text{For } i = N-1, \ldots, 1, \\ \bar{v_i} = \sum_{j \in \text{Children}(v_i)} \bar{v_j} \frac{\partial v_j}{\partial v_i}$$

Backpropagation for Regularized Logistic Least Squares



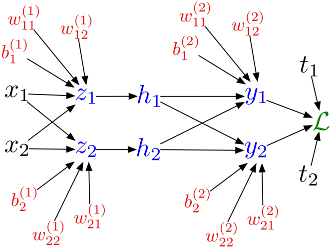
Forward pass:

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

Backward pass:

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

Backpropagation for Two-Layer Neural Network



Forward pass:

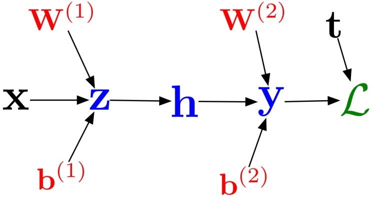
$$
\begin{aligned}
z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
h_i &= \sigma(z_i) \\
y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
\mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\
\overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\
\overline{b_k^{(2)}} &= \overline{y_k} \\
\overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\
\overline{z_i} &= \overline{h_i}\, \sigma'(z_i) \\
\overline{w_{ij}^{(1)}} &= \overline{z_i}\, x_j \\
\overline{b_i^{(1)}} &= \overline{z_i}
\end{aligned}
$$

Backpropagation for Two-Layer Neural Network

In vectorized form:



Forward pass:

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

Backward pass:

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

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

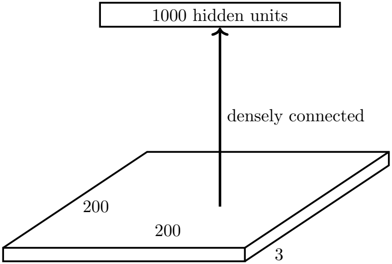
- Suppose we construct our networks out of a series of “primitive” operations (e.g., add, multiply) with specified routines for computing derivatives.
- Autodifferentiation performs backprop in a completely mechanical and automatic way.
- Many autodiff libraries: PyTorch, Tensorflow, Jax, etc.
- Although 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.

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 200×200 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.

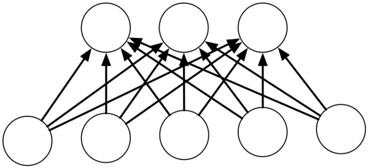
Shared Structures in the Network

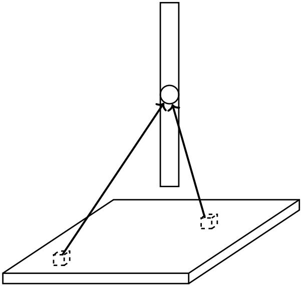
- Some features, e.g. edges, corners, contours, object parts, may be useful in multiple locations in the image.

- We want feature detectors that are applicable in multiple locations in the image.

Convolution Layers

Fully connected layers:

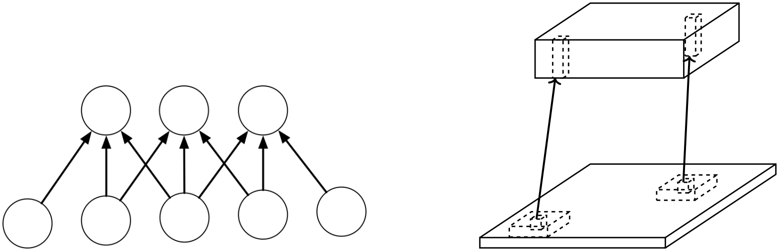




Each hidden unit looks at the entire image.

Convolution Layers

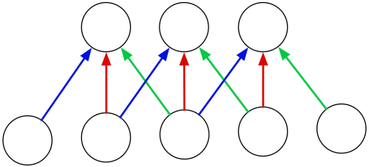
Locally connected layers:

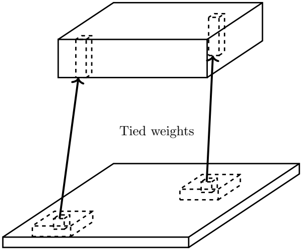


Each set of hidden units looks at a small region of the image.

Convolution Layers

Convolution layers:

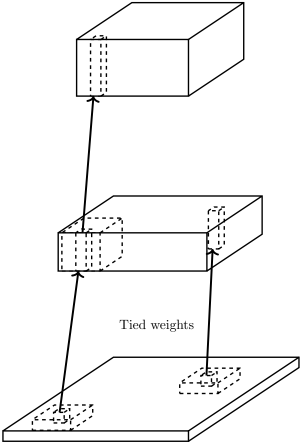




Each set of hidden units looks at a small region of the image, and the weights are shared between all image locations.

Going Deeply Convolutional

Convolution layers can be stacked:



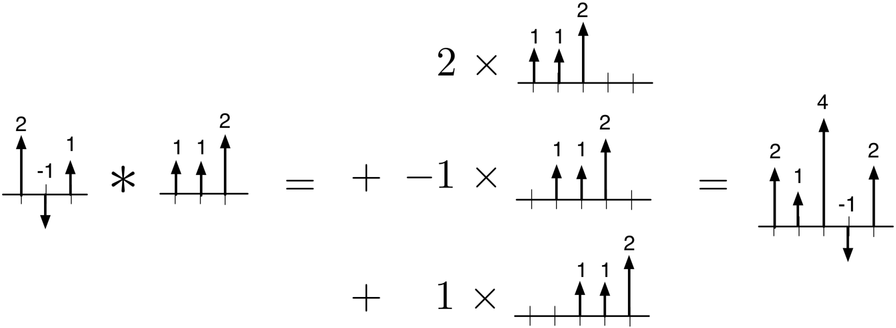
1-D Convolution

We have two signals/arrays x and w.

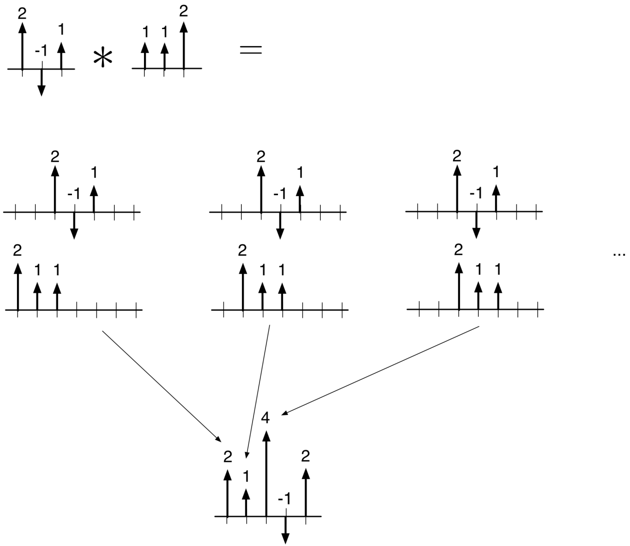
- x is an input signal (e.g. a waveform or an image).
- w is a set of k weights (also referred to as a kernel or filter).
- Often zero pad x to an infinite array

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

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


2
-1
1
*
1 1 2
=
2 × 1 1 2
+ −1 × 1 1 2
+ 1 × 1 1 2
=
2 1 4 -1 2

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:

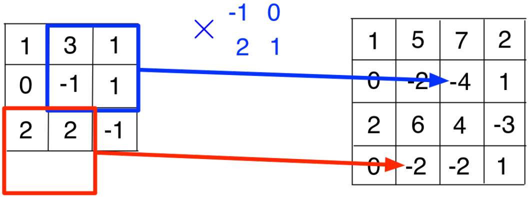
$$(x * w)[i, j] = \sum_s \sum_t x[i - s, j - t] * w[s, t].$$

2-D Convolution: Translate-and-Scale

$$
\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array}
\ast
\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array}
=
\begin{array}{l}
1 \times \begin{array}{|c|c|c|c|}\hline 1 & 3 & 1 & \\ \hline 0 & -1 & 1 & \\ \hline 2 & 2 & -1 & \\ \hline & & & \\ \hline \end{array} \\[1em]
+\,2 \times \begin{array}{|c|c|c|c|}\hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline & & & \\ \hline \end{array} \\[1em]
+\,-1 \times \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline \end{array}
\end{array}
=
\begin{array}{|c|c|c|c|}\hline 1 & 5 & 7 & 2 \\ \hline 0 & -2 & -4 & 1 \\ \hline 2 & 6 & 4 & -3 \\ \hline 0 & -2 & -2 & 1 \\ \hline \end{array}
$$

2-D Convolution: Flip-and-Filter

$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} \ast \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array}$$



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?

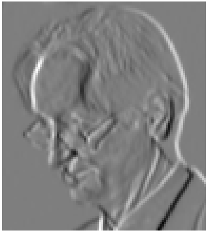
$*$

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

Example 3: What does this convolution kernel do?

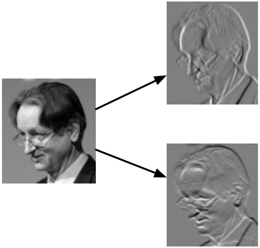
$*$

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



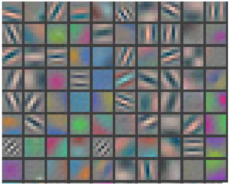
Convolution Layer in Convolutional Networks

- 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



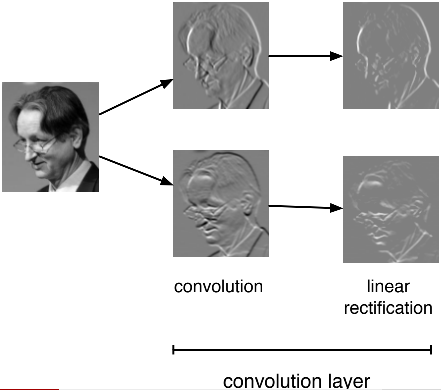
(Zeiler and Fergus, 2013, Visualizing and understanding convolutional networks)

Non-linearity in Convolutional Networks

Common to apply a linear rectification nonlinearity:

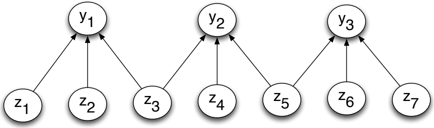
$$y_i = \max(z_i, 0).$$

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

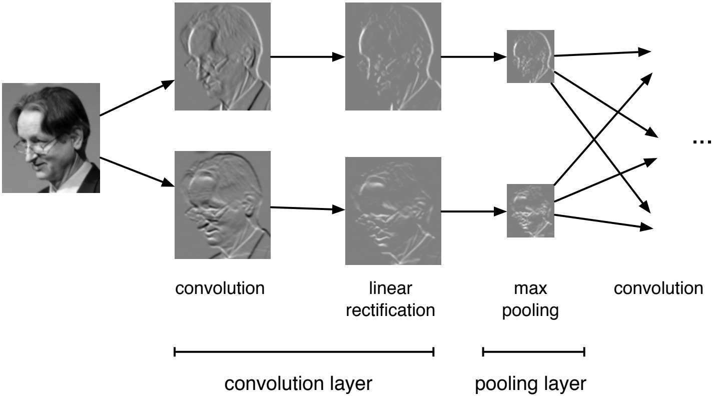
These layers reduce the size of the representation and build in in-variance to small transformations.



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

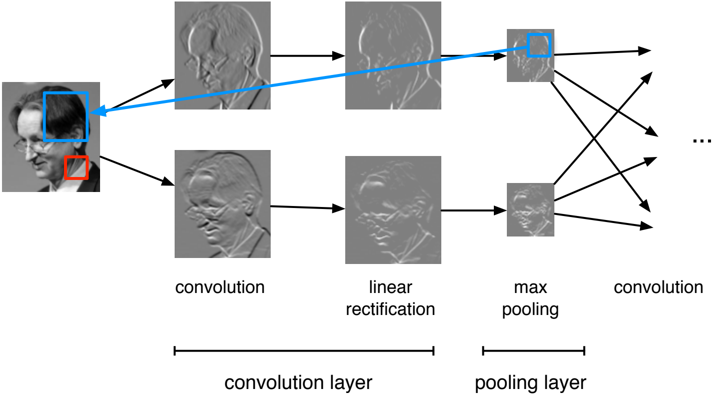
$$y_i = \max_{j \text{ in pooling group}} z_j$$

Convolutional networks



Convolutional Network Structure

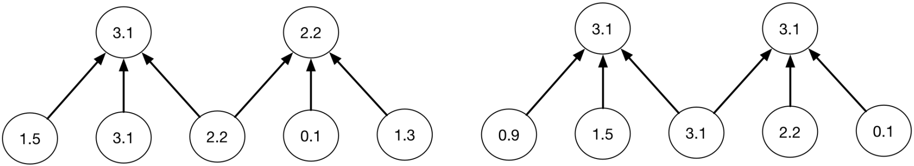
Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in the lower layers.



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.



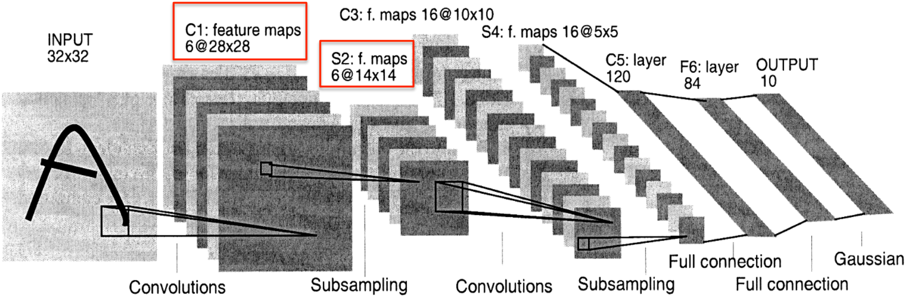
Convolution Layers

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.

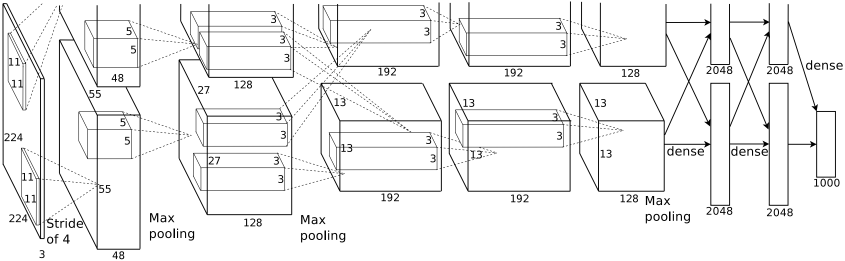
Each unit is connected to each unit within its receptive field in the previous layer. This includes *all* of the previous layer's feature maps.

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%.

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%.

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