

Convolution and Pooling

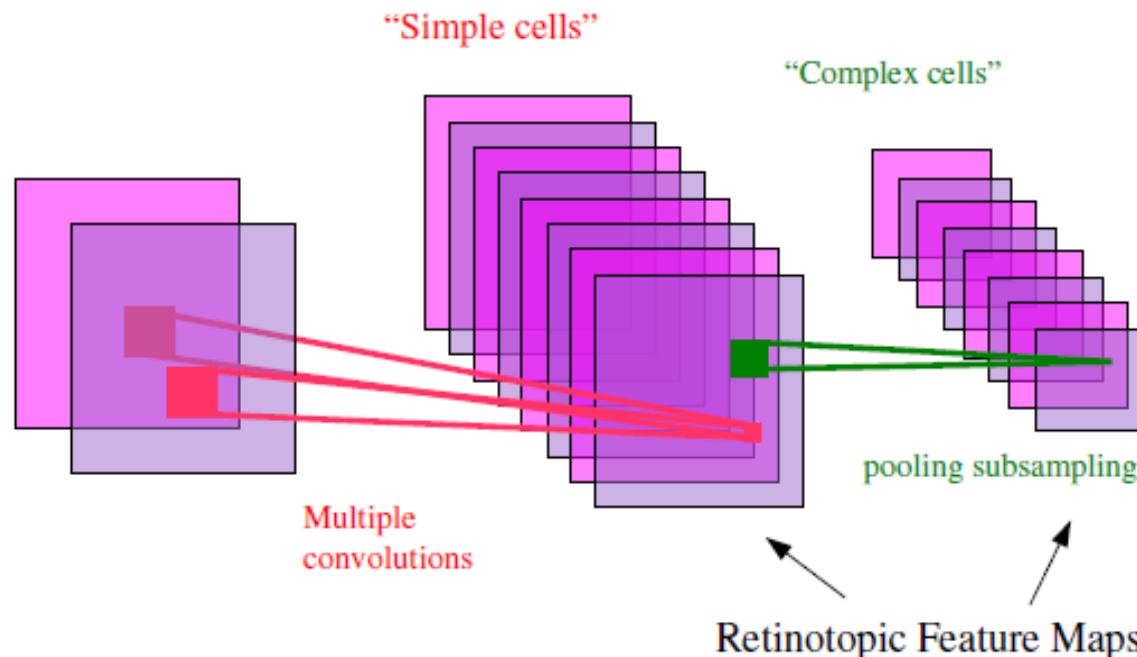
Minlie Huang

aihuang@tsinghua.edu.cn

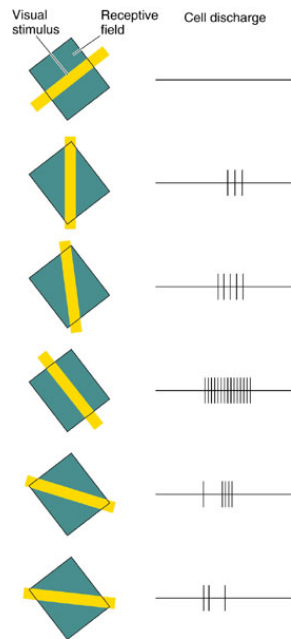
Dept. of Computer Science and Technology
Tsinghua University

Local detectors and shift invariance in the cortex

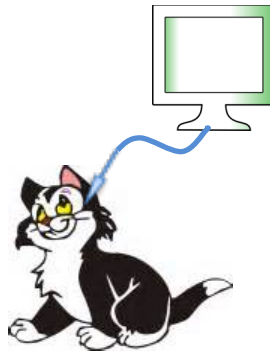
- (Hubel & Wiesel 1962) won 1981 Physiology or Medicine Nobel Prize
 - Simple cells detect local features
 - complex cells “pool” the outputs of simple cells within a retinotopic neighborhood



Simple Cell and Complex Cell



Simple cell
(location and orientation)



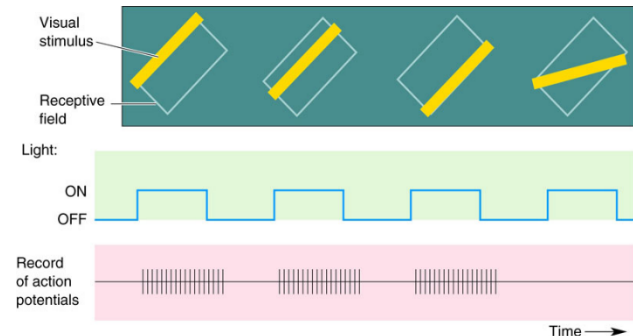
Nobel Prize 1981



D. Hubel



T. Wiesel



Complex cell
(only orientation)

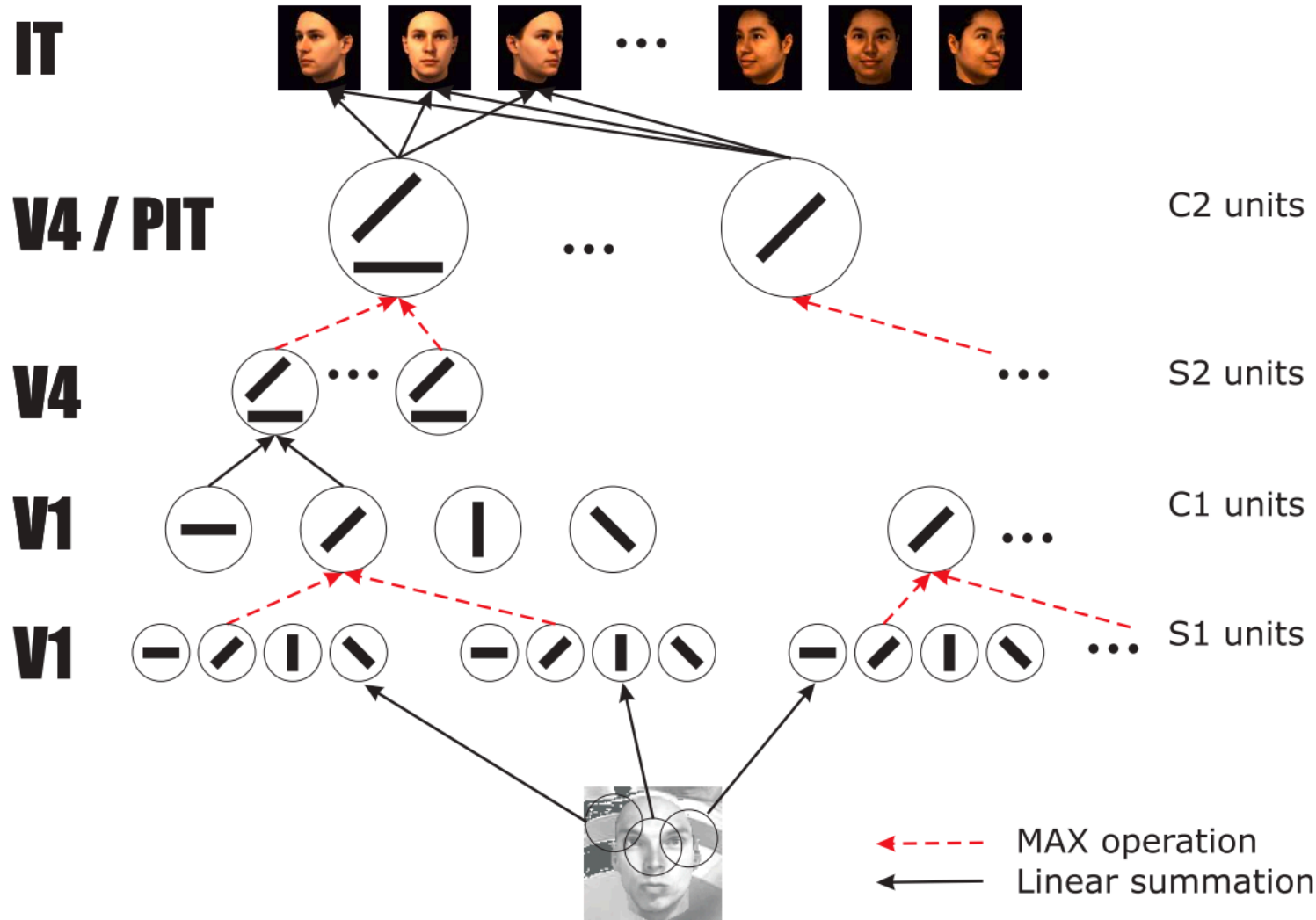
Bear, Connors, Paradiso, 2006

The multistage Hubel-Wiesel architecture

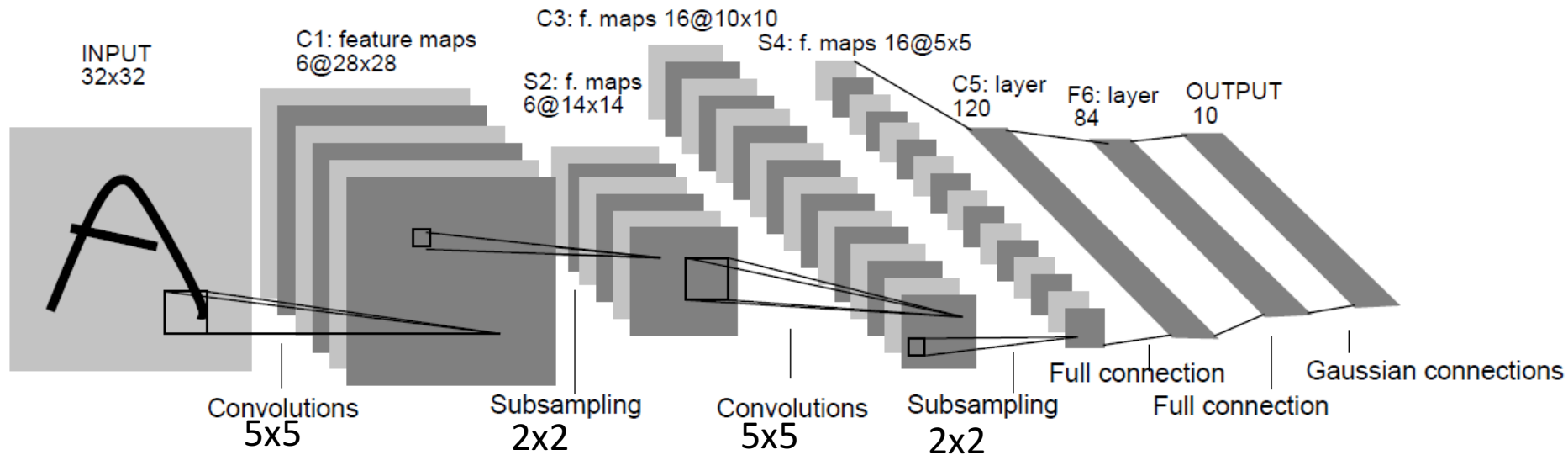
- Building a complete artificial vision system
 - Stack multiple stages of simple cells / complex cells layers
 - Higher stages compute more global, more invariant features
 - Stack a classification layer on top
- Models
 - Neocognitron [Fukushima 1971-1982]
 - **convolutional net [LeCun 1988-2007]**
 - HMAX [Poggio 2002-2006]
 - fragment hierarchy [Ullman 2002-2006]
 - HMAX [Lowe 2006]

HMAX: Hierarchical models of object recognition in cortex

View-tuned units (VTUs). Ex: face units



Convolutional network

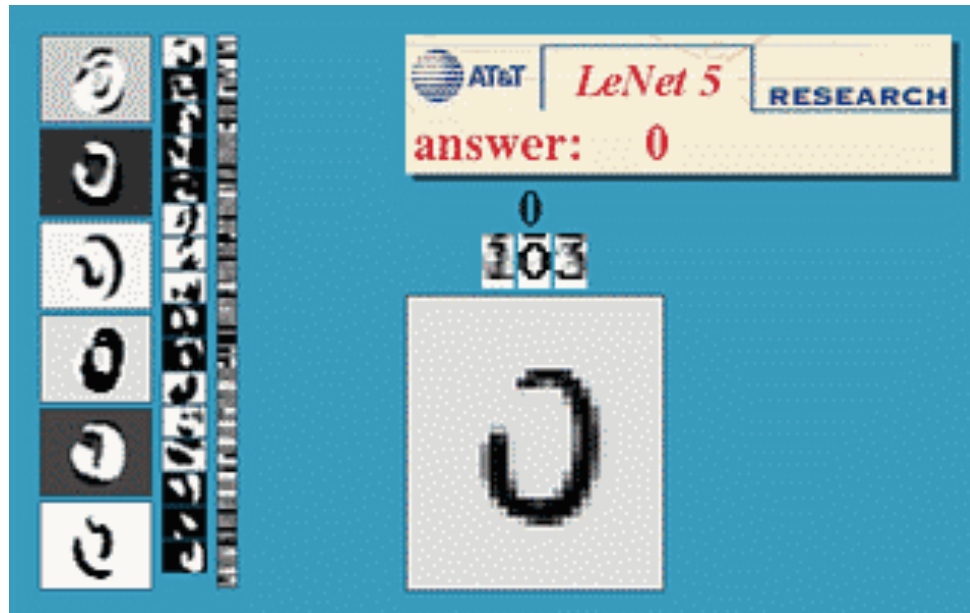


- Local connections and weight sharing
- C layers: convolution
 - Output $y_i = f(\sum_{\Omega} w_j x_j + b)$ where Ω is the patch size, $f(\cdot)$ is the sigmoid function, w and b are parameters
- S layers: subsampling (avg pooling)
 - Output $y_i = f\left(\frac{1}{|\Omega|} \sum_{\Omega} x_j\right)$ where Ω is the pooling size

LeNet5 Demo

LeCun's homepage

<http://yann.lecun.com/exdb/lenet/index.html>

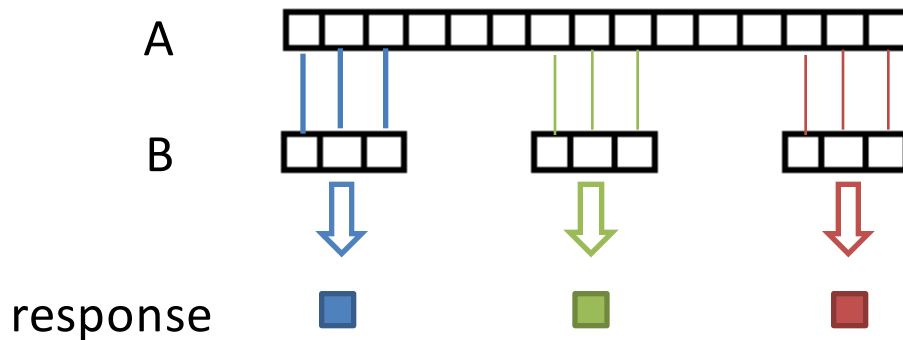


Outline

- Convolution
- Pooling
- Review of multi-layer perceptron
- Stochastic gradient descent

Motivation

- Suppose there are two 1D sequences A and B where the length of B is smaller than that of A
- Compute the similarity between B and each part of A
- Naively, we could slide B on A and calculate the similarity one by one



But this process is very slow!

Cosine similarity between two vectors x and y :

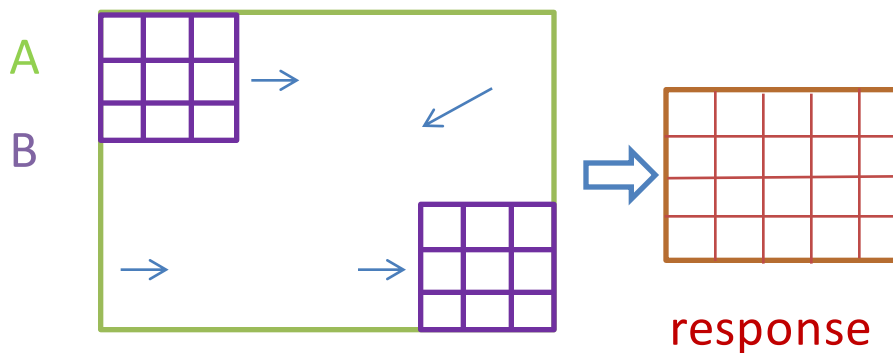
$$s \equiv \cos \theta = \frac{x^T y}{||x|| ||y||}$$

$$= \sum_i x_i y_i$$

if the two vectors have unit length

Motivation

- Suppose there are two 2D images A and B where the size of B is smaller than that of A
- Compute the similarity between B and each part of A
- Naively, we could slide B on A and calculate the similarity one by one



Cosine similarity between two matrices x and y :

$$s = \sum_{i,j} x_{ij} y_{ij}$$

if the two matrices have unit Frobenius norm

But this process is very slow! We have other choices...

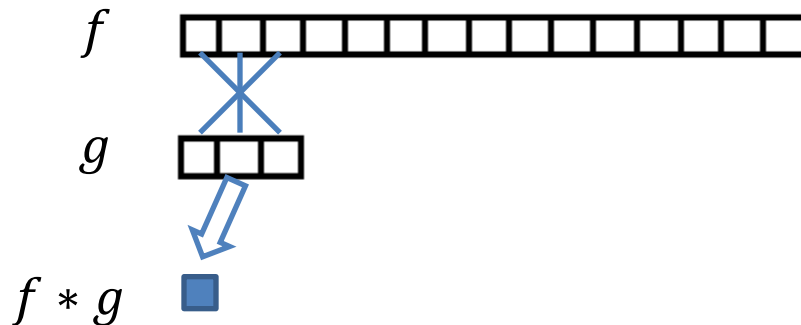
1D convolution

- Continuous convolution

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau$$

- Discrete convolution (for finite length sequences)

$$(f * g)[n] \triangleq \sum_{m=1}^M f[n - m]g[m]$$



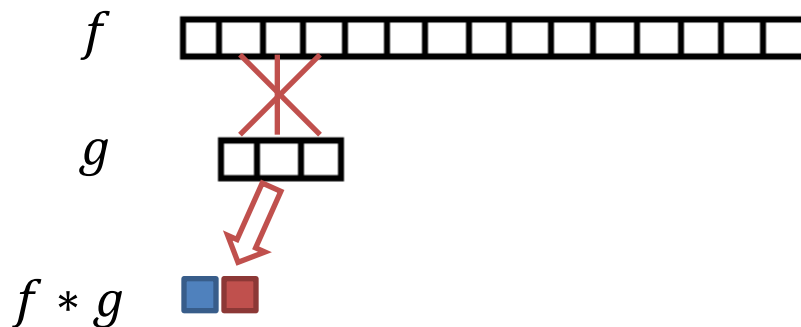
1D convolution

- Continuous convolution

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau$$

- Discrete convolution (for finite length sequences)

$$(f * g)[n] \triangleq \sum_{m=1}^M f[n - m]g[m]$$



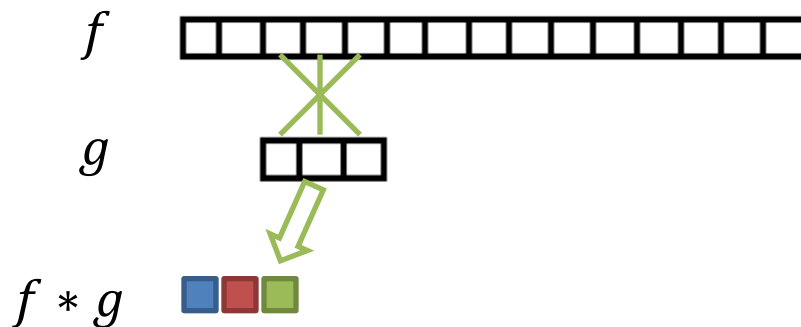
1D convolution

- Continuous convolution

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau$$

- Discrete convolution (for finite length sequences)

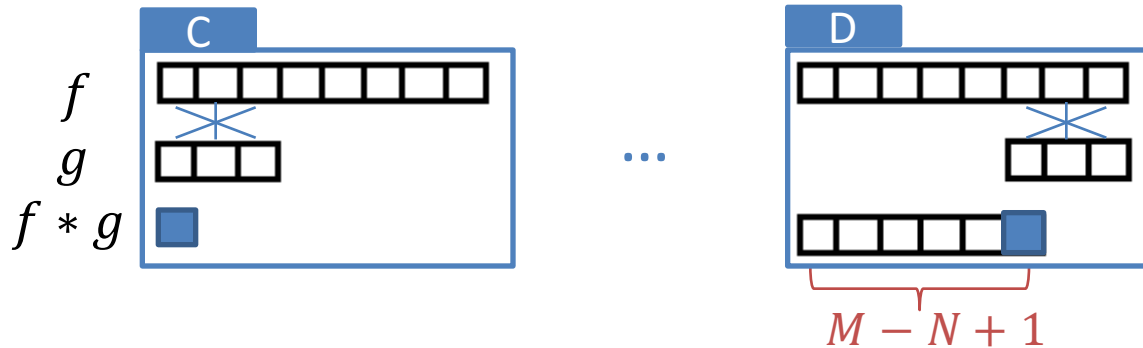
$$(f * g)[n] \triangleq \sum_{m=1}^M f[n - m]g[m]$$



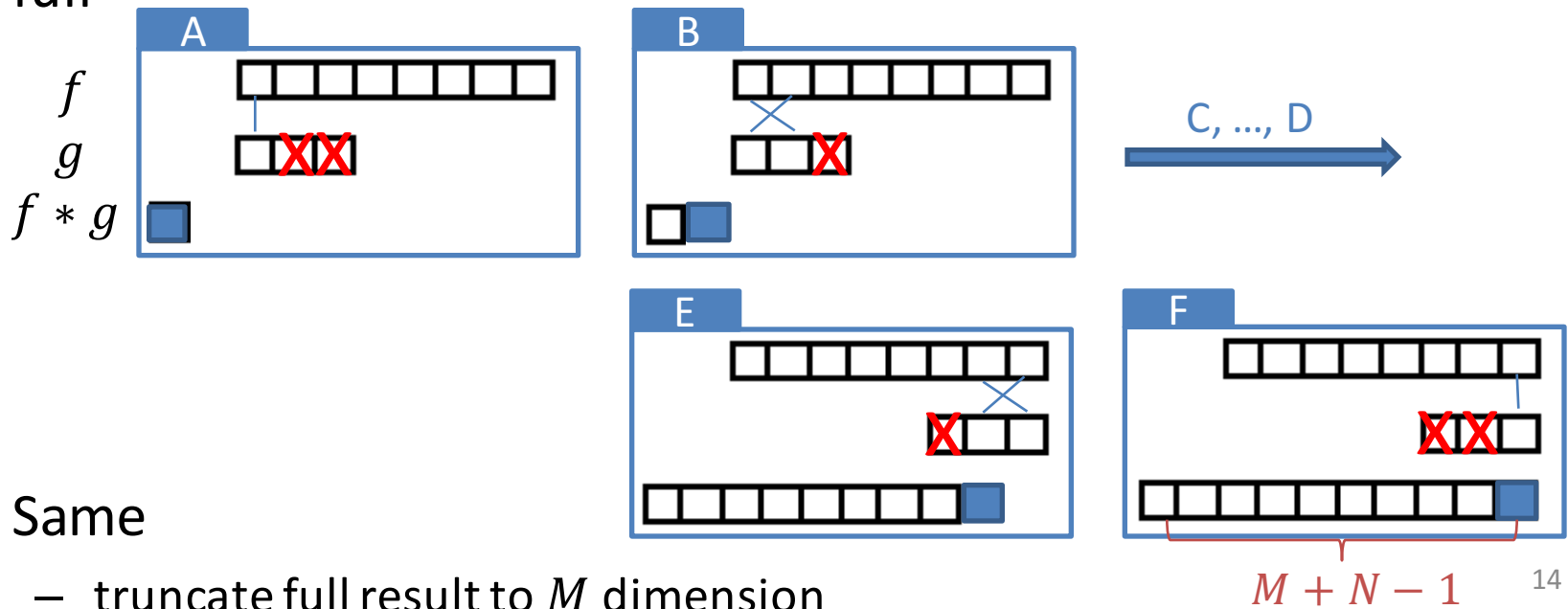
Three shapes of convolution

Length of $f: M$, length of $g: N$, where $M \geq N$

- valid



- full



- Same

– truncate full result to M dimension

Example

- Suppose there are two sequences

$$f = [0, 1, 2, -1, 3]$$

$$g = [1, 1, 0]$$

- Then

$$(f * g)_{valid} = [3, 1, 2]$$

$$(f * g)_{full} = [0, 1, 3, 1, 2, 3, 0]$$

- Python commands

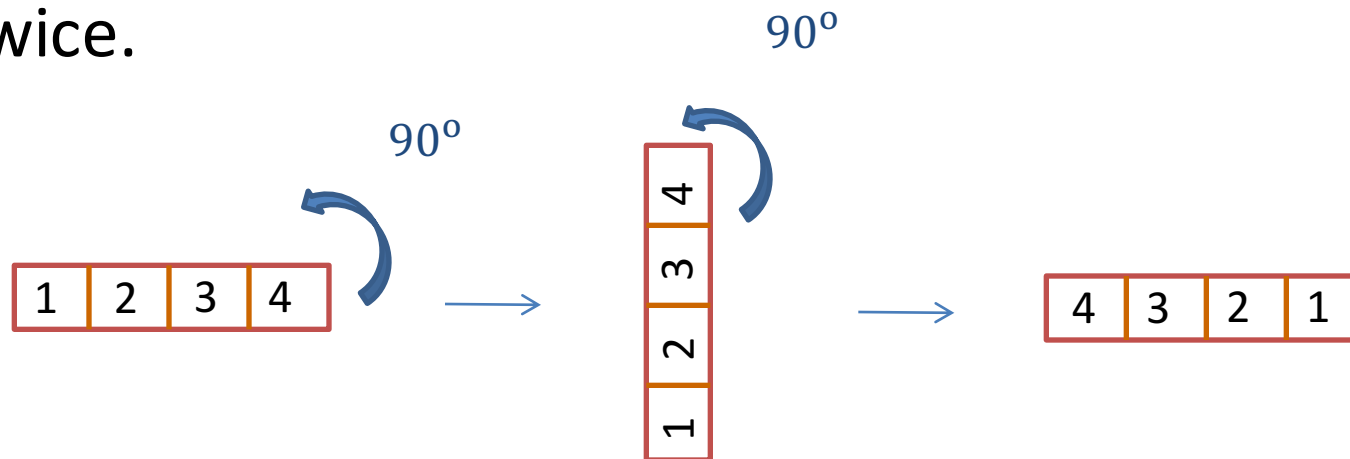
```
>> f=[0, 1, 2, -1, 3]
>> g=[1, 1, 0]
>> convolve(f,g, 'valid')
>> convolve(f,g, 'full')
```

Relationship between similarity and convolution

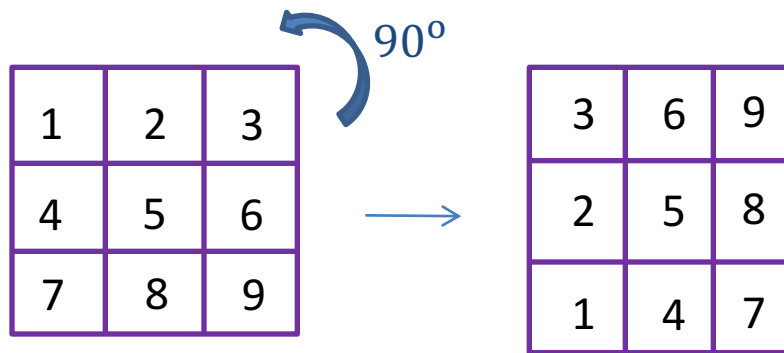
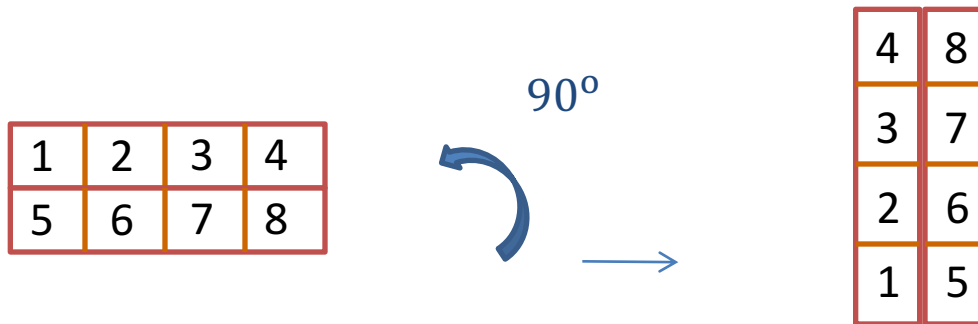
- Calculating the similarity between sequence g and each part of sequence f is equivalent to calculating $f * \tilde{g}$ where

$$\tilde{g}_1 = g_N, \tilde{g}_2 = g_{N-1}, \dots, \tilde{g}_N = g_1$$

- In Python, the above flip operation can be realized by the command `flip()` or applying the command `rot90()` twice.



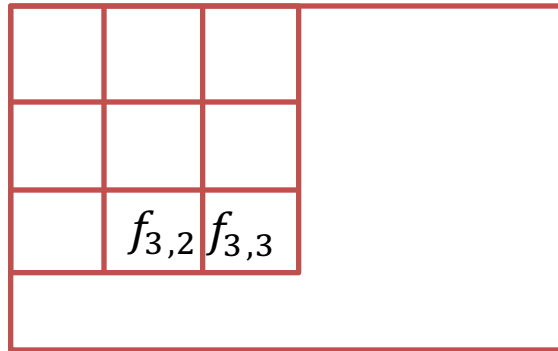
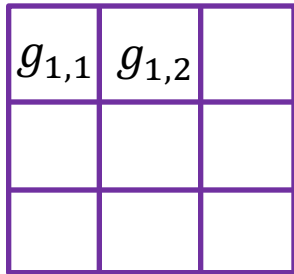
Matrix Rotation



2D convolution

- Suppose there are two matrices f and g with sizes $M \times N$ and $K_1 \times K_2$, respectively, where $M \geq K_1, N \geq K_2$
- Discrete convolution of the two matrices

$$h[m, n] = (f * g)[m, n] \triangleq \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} f[m - k_1, n - k_2] g[k_1, k_2]$$



When $m = 4, n = 4$

$$\begin{aligned} (f * g)_{m,n} &= f_{3,3}g_{1,1} + f_{3,2}g_{1,2} + f_{3,1}g_{1,3} \\ &+ f_{2,3}g_{2,1} + \dots \end{aligned}$$

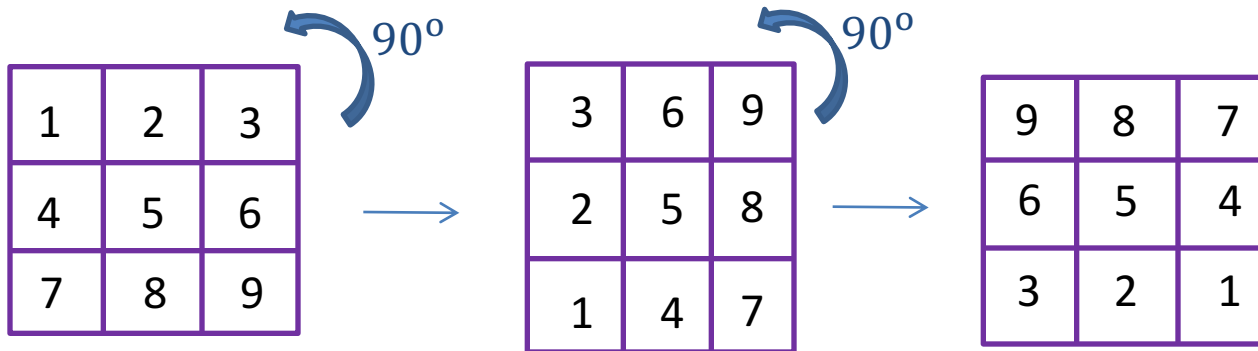
- valid shape: the size of h is $(M - K_1 + 1) \times (N - K_2 + 1)$
- **full shape**: the size of h is $(M + K_1 - 1) \times (N + K_2 - 1)$
- same shape: the size of h is $M \times N$

Relationship between similarity and convolution

- Calculating the similarity between matrix g and each part of matrix f is equivalent to calculating $f * \tilde{g}$ where

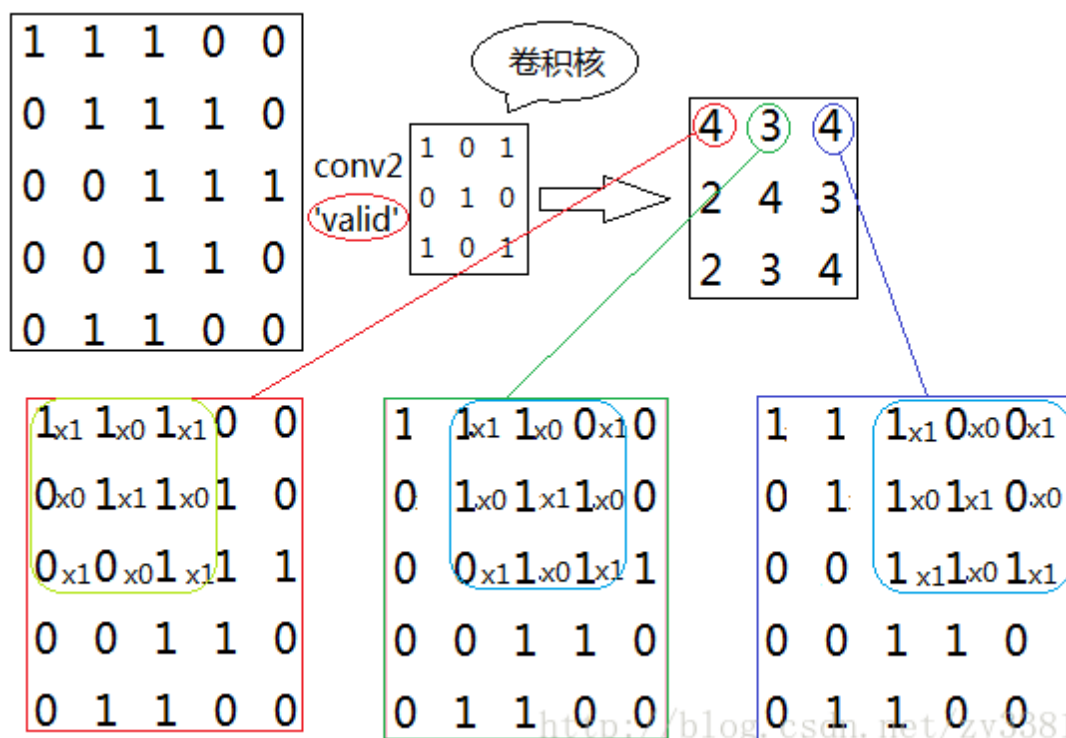
$$\begin{aligned}\tilde{g}_{1,1} &= g_{M,N}, \tilde{g}_{1,2} = g_{M,N-1}, \dots, \tilde{g}_{1,N} = g_{M,1} \\ \tilde{g}_{2,1} &= g_{M-1,N}, \tilde{g}_{2,2} = g_{M-1,N-1}, \dots, \tilde{g}_{2,N} = g_{M-1,1} \\ &\vdots \\ \tilde{g}_{M,1} &= g_{1,N}, \tilde{g}_{M,2} = g_{1,N-1}, \dots, \tilde{g}_{M,N} = g_{1,1}\end{aligned}$$

- In Python, the above flip operation can be realized by applying the command `rot90()` twice



Valid Convolution

- Rotate filter g twice
- Slide from left to right, up to bottom



Full Convolution

- First, pad 0 (K_1-1 rows on the top/bottom, K_2-1 cols on the right/left)
- Second, do valid conv. on the padded f matrix



Python example

```
>> A = randint(0, 4, size=[4, 4])
```

A =

0	0	1	2
2	2	0	0
2	1	2	2
3	0	1	1

```
>> B = randint(0, 3, size=[3, 3]) - 1
```

B =

0	0	-1
1	-1	1
-1	1	1

```
>> C = convolve2d(A, B, 'full')
```

C =

0	0	0	0	-1	-2
0	0	-1	-1	-1	2
2	0	-3	0	1	0
0	-1	4	3	-1	1
1	-2	5	1	4	3
-3	3	2	0	2	1

```
>> D = convolve2d(A, B, 'valid')
```

D =

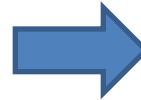
-3	0
4	3

Example

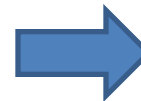
figure



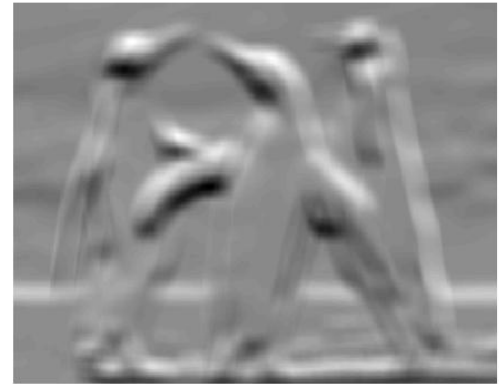
filter



*



feature map



The higher a pixel value (brighter) in the feature map, the more similar between the filter and the corresponding patch in the figure

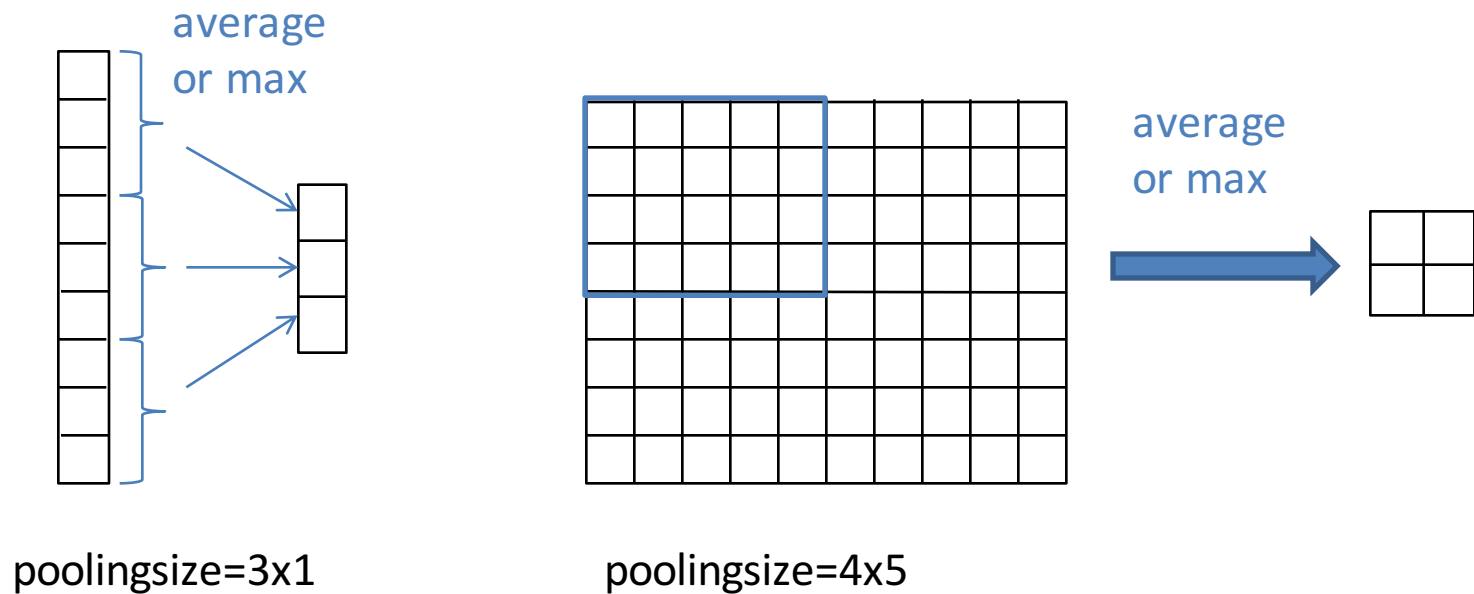
Why do we use convolution?

- Convolution has fast algorithms, e.g., Fast Fourier Transform (FFT)
- It does not slide one signal on the other signal!
- However, when GPU is used, FFT may not be needed as GPU can compute matrix multiplication in parallel
 - Can we transform the similarity calculation to matrix multiplication form?

Outline

- Convolution
- Pooling
- Review of multi-layer perceptron
- Stochastic gradient descent

Pooling in local regions



- Divide the convolved features into *disjoint* regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled features
- Another often used pooling method is L2 pooling (not discussed in this course)

$$p = \sqrt{\sum_{y_{ij} \in \Omega} y_{ij}^2}$$

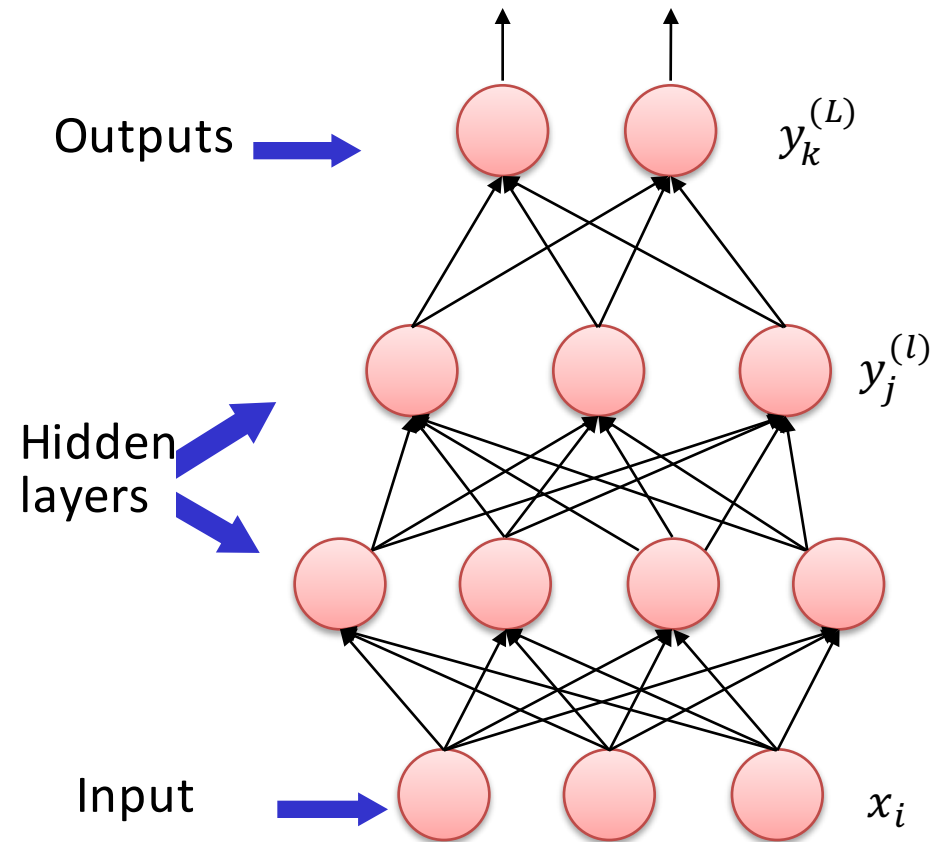
Why do we need pooling

- Reduce the number of features for final classification
 - Consider images of 96×96 pixels. Suppose we have learned 400 features over 8×8 inputs. This results in an output of size $(96 - 8 + 1)^2 \times 400 = 3,168,400$ features per example
- Enlarge the effective region of features in the next layer
 - A feature learned in the pooled maps will have larger effective regions in the pixel space
- Realize invariance
 - After pooling, features tend to be translation invariant in local regions
- This is similar to the receptive fields of visual neurons, whose sizes increase along the visual hierarchy

Outline

- Convolution
- Pooling
- Review of multi-layer perceptron
- Stochastic gradient descent

Multi-layer Perceptron (MLP)



- There are a total of L layers except the input
- Connections:
 - Fully connections between layers
 - No feedback connections between layers
 - No lateral connections in the same layer
- Every neuron receives input from previous layer and fire according to an activation function

Activation functions

- Logistic function

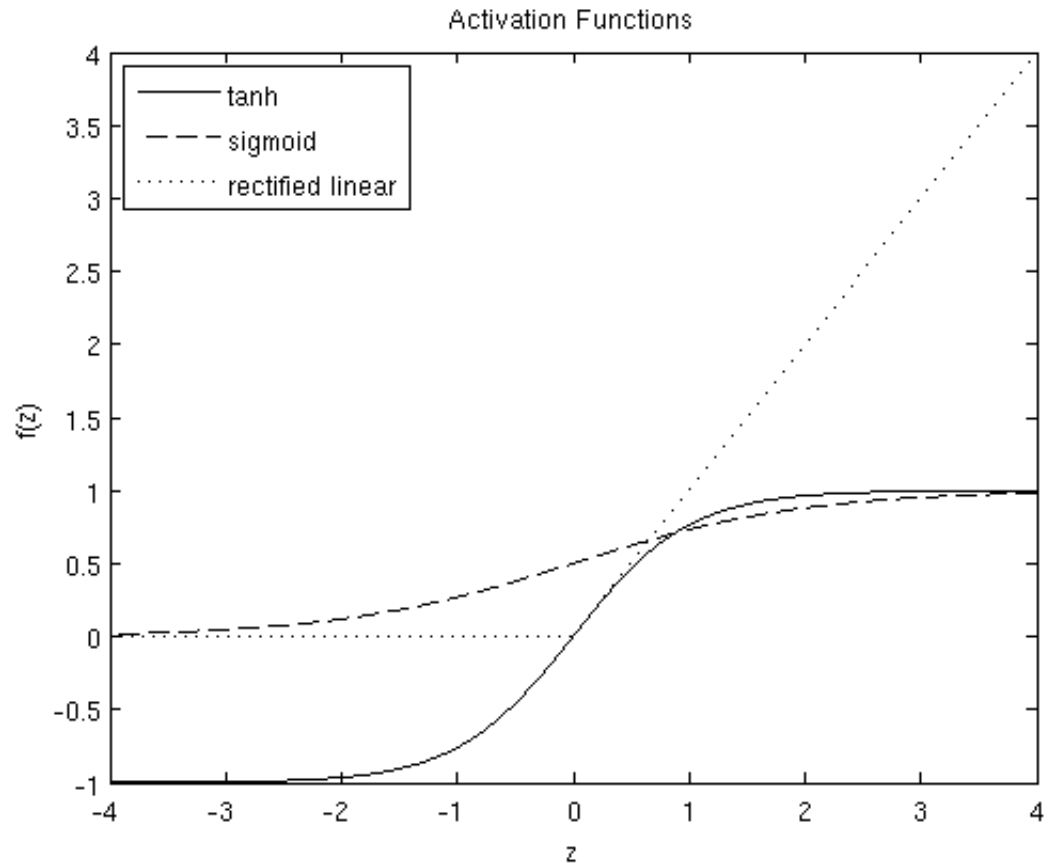
$$f(z) = \frac{1}{1 + \exp(-z)}$$

- Hyperbolic tangent, or tanh, function

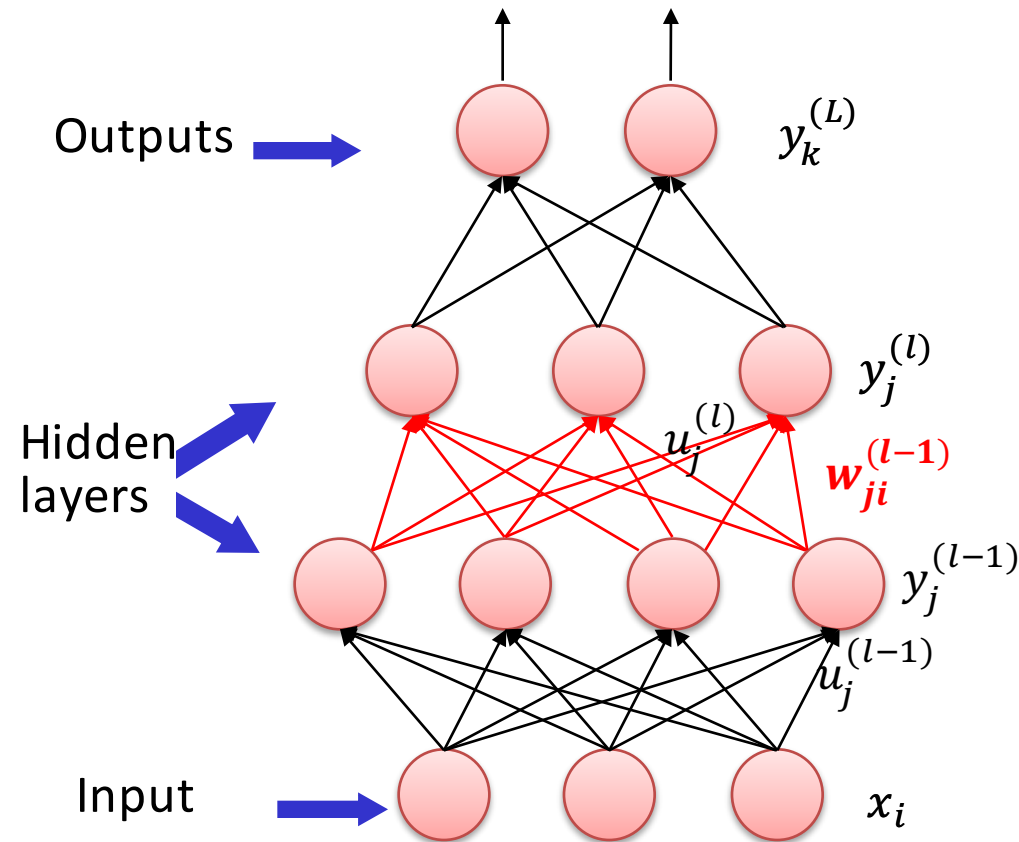
$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- Rectified linear activation function (ReLU)

$$f(z) = \max(0, x)$$



Forward pass



For $l = 1, \dots, L - 1$ calculate the input to neuron j in the l -th layer

$$u_j^{(l)} = \sum_i w_{ji}^{(l-1)} y_i^{(l-1)} + b_j^{(l-1)}$$

and its output

$$y_j^{(l)} = f(u_j^{(l)})$$

where $f(\cdot)$ is activation function

- Note $y^{(0)} = x$
- There are desired outputs t_k for each input sample
- For $l = L$, $f(\cdot)$ depends on the error function

Note the superscript of w (different from Prof. Zhu's slides)

Error functions for BP

- Error function $E = \sum_{n=1}^N E^{(n)}$

where $E^{(n)}$ is the error function for each input sample

- Least square error

$$E^{(n)} = \frac{1}{2} \sum_{k=1}^K (t_k - y_k^{(L)})^2, \quad y_k^{(L)} = \frac{1}{1 + \exp(-w_k^{(L-1)\top} y^{(L-1)} - b_k^{(L-1)})}$$

- Cross-entropy error

$$E^{(n)} = - \sum_{k=1}^K t_k \ln y_k^{(L)}, \quad y_k^{(L)} = \frac{\exp(w_k^{(L-1)\top} y^{(L-1)} + b_k^{(L-1)})}{\sum_{j=1}^K \exp(w_j^{(L-1)\top} y^{(L-1)} + b_j^{(L-1)})}$$

where t_k is target of the form


In what follows, except $E^{(n)}$, for clarity, we will omit the superscript (n) on x, t, u, y etc. for each input sample.

Weight adjustment

- Weight adjustment

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \alpha \frac{\partial E}{\partial w_{ji}^{(l)}} \quad b_j^{(l)} = b_j^{(l)} - \alpha \frac{\partial E}{\partial b_j^{(l)}}$$

Learning rate



- Weight decay is often used on $w_{ji}^{(l)}$ (not necessary on $b_j^{(l)}$) which amounts to adding an additional term on the cost function

$$J = E + \frac{\lambda}{2} \sum_{i,j,l} (w_{ji}^{(l)})^2$$

- Weight adjustment on w is changed to

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \alpha \frac{\partial J}{\partial w_{ji}^{(l)}} = w_{ji}^{(l)} - \alpha \frac{\partial E}{\partial w_{ji}^{(l)}} - \alpha \lambda w_{ji}^{(l)}$$

Local sensitivity for each sample

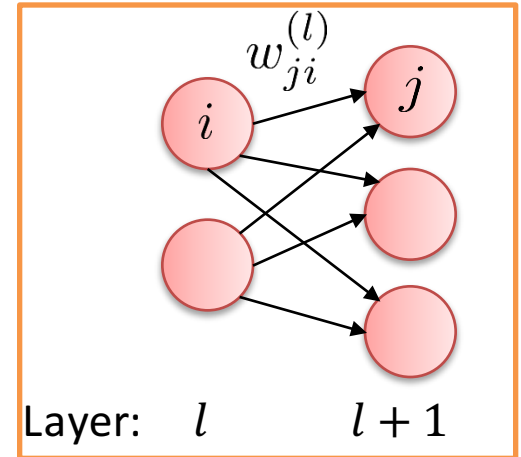
- Define local sensitivity $\delta_i^{(l)} = \frac{\partial E^{(n)}}{\partial u_i^{(l)}}$
- Then for $0 \leq l < L$

$$\frac{\partial E^{(n)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l+1)} \frac{\partial u_j^{(l+1)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l+1)} f(u_i^{(l)})$$

$$\frac{\partial E^{(n)}}{\partial b_j^{(l)}} = \delta_j^{(l+1)}, \text{ since } u_j^{(l+1)} = \sum_i w_{ji}^{(l)} f(u_i^{(l)}) + b_j^{(l)}, \text{ where } f \text{ can be}$$

sigmoid or linear rectifier function and $f(u_i^{(0)}) = x$.

- If $l = L$, i.e., neuron j is an output neuron



Sigmoid function output

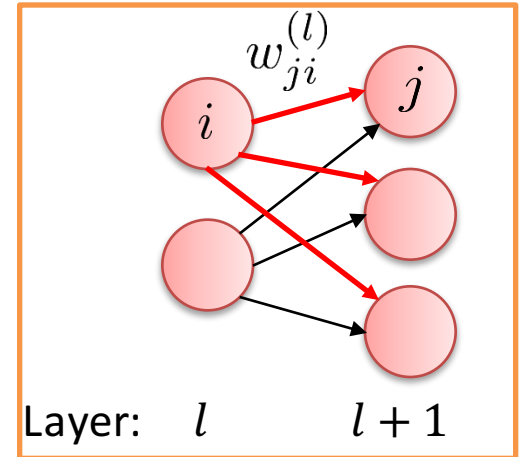
f is sigmoid function

$$\delta_j^{(L)} = \frac{\partial E^{(n)}}{\partial u_j^{(L)}} = \begin{cases} (y_j^{(L)} - t_j) f'(u_j^{(L)}) & \text{least square error} \\ y_j^{(L)} - t_j & \text{cross-entropy error} \end{cases}$$

Softmax function output

Local sensitivity for each sample

- Define local sensitivity $\delta_i^{(l)} = \frac{\partial E^{(n)}}{\partial u_i^{(l)}}$
- If $1 \leq l < L$, i.e., neuron i is a hidden neuron, it has an effect on all neurons in the next layer, therefore its local sensitivity is



$$\delta_i^{(l)} = \frac{\partial E^{(n)}}{\partial u_i^{(l)}} = \sum_j \frac{\partial E^{(n)}}{\partial u_j^{(l+1)}} \frac{\partial u_j^{(l+1)}}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial u_i^{(l)}} = \sum_j \delta_j^{(l+1)} w_{ji}^{(l)} f'(u_i^{(l)})$$

$$u_j^{(l+1)} = \sum_i w_{ji}^{(l)} y_i^{(l)} + b_j^{(l)} \quad y_i^{(l)} = f(u_i^{(l)})$$

where f can be sigmoid or linear rectifier

Therefore we compute $\delta_i^{(l)}$ **backward**, from $l = L, L - 1, \dots, 1$, and in the sequel $\partial E / \partial W^{(l-1)}$ and $\partial E / \partial b^{(l-1)}$ backward.

BP in vector-matrix form

- Local sensitivity $\delta^{(l)} = \left(\frac{\partial E^{(n)}}{\partial u_1^{(l)}}, \frac{\partial E^{(n)}}{\partial u_2^{(l)}}, \dots \right)^T$
- For the output layer L

$$\delta^{(L)} = (y - t) \bullet f'(u^{(L)}) \quad \text{or} \quad \delta^{(L)} = (y - t)$$

Where \bullet denotes element-wise multiplication

- For the hidden layer $1 \leq l < L$

$$\delta^{(l)} = (W^{(l)})^\top \delta^{(l+1)} \bullet f'(u^{(l)})$$

- Calculate the gradients $0 \leq l < L$

$$\frac{\partial E^{(n)}}{\partial w^{(l)}} = \delta^{(l+1)} (f(u^{(l)}))^\top, \quad \frac{\partial E^{(n)}}{\partial b^{(l)}} = \delta^{(l+1)}$$

- Update weights

$$W^{(l)} = W^{(l)} - \alpha \sum_n \frac{\partial E^{(n)}}{\partial W^{(l)}} - \alpha \lambda W^{(l)}, \quad b^{(l)} = b^{(l)} - \alpha \sum_n \frac{\partial E^{(n)}}{\partial b^{(l)}}$$

for each sample n

→ sum over n

Implementation

- Run forward process
 - Calculate $f(u^l)$ and $f'(u^l)$ for $l = 1, 2, \dots, L$
- Run backward process
 - Calculate $\delta^{(l)}$ and $\partial E / \partial W^{(l-1)}, \partial E / \partial b^{(l-1)}$ for $l = L, L - 1, \dots, 1$
- Update $W^{(l)}$ and $b^{(l)}$ for $l = 0, 1, \dots, L - 1$
- Modular programming ← Basic idea of Caffe
 - Implement the layer as a class and provide functions for forward calculation and backward calculation, respectively
 - The forward functions and backward functions differ according to the type of the layer, e.g., input layer, hidden layer, softmax output layer, sigmoid output layer, etc.
 - Then you can design different structures of MLP by specifying the layer modules in a main file

Outline

- Convolution
- Pooling
- Review of multi-layer perceptron
- Stochastic gradient descent

Batch gradient descent vs SGD

- It's straightforward to compute the cost and gradient for the entire training set
- Very slow and sometimes intractable on a single machine if the dataset is too big to fit in main memory.
- Batch modes don't give an easy way to incorporate new data in an **online** setting.
- Stochastic gradient descent (SGD) addresses both issues

Formulation

- The standard gradient descent algorithm updates the parameters θ of the objective $J(\theta)$ as,

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta)$$

where $J(\theta)$ denotes the cost over the full training set

- SGD updates and computes the gradient of the parameters using only a single or a few training examples. The new update is given by,

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta; x^{(i)}, t^{(i)})$$

with a pair $(x^{(i)}, t^{(i)})$ from the training set.

- Often a minibatch is used (e.g., size 256) instead of a single example.
 - This reduces the **variance** in the parameter update and can lead to more stable convergence
 - This allows the computation to take advantage of highly **optimized matrix operations** that should be used in a well vectorized computation of the cost and gradient

Learning rate

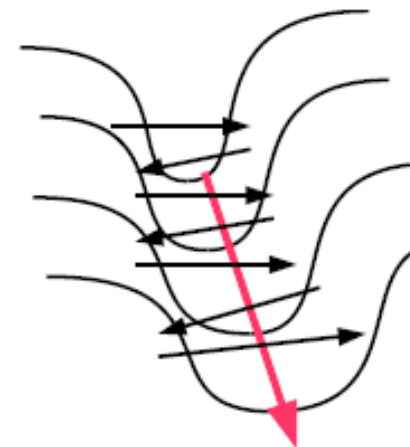
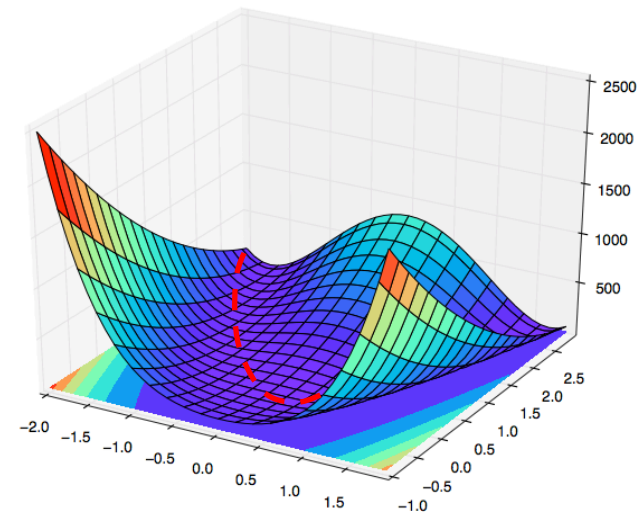
- In SGD the learning rate α is typically much smaller than that in batch gradient descent
 - There is much more variance in the update.
- Choosing the proper learning rate and schedule
 - A small enough constant learning rate that gives stable convergence in the initial epoch or halve the value of the learning rate as convergence slows down.
 - Evaluate a held-out set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold.
 - **Anneal the learning rate at each iteration t as $\frac{a}{b+t}$ where a and b dictate the initial learning rate.**

Order of training samples

- If the data is given in **some meaningful order**, this can bias the gradient and lead to poor convergence
- Generally a good method to avoid this is to **randomly shuffle the data** prior to each epoch of training.

Pathological curvature

- The objective has the form of a long shallow ravine leading to the optimum and steep walls on the sides
 - as seen in the well-known Rosenbrock function
- The objectives of deep architectures have this form near local optima and thus standard SGD tends to oscillate across the narrow ravine



Black arrows: gradient descent paths

https://en.wikipedia.org/wiki/Rosenbrock_function

Momentum

- Momentum is one method for pushing the objective more quickly along the shallow ravine
- The momentum update is given by,

$$\begin{aligned}v_{t+1} &= \gamma v_t + \alpha \nabla_{\theta} J(\theta; x^{(i)}, t^{(i)}) \\ \theta_{t+1} &= \theta_t - v_t\end{aligned}$$

- v_t is the current velocity vector
- The learning rate α may need to be smaller when using momentum since the magnitude of the gradient will be larger
- $\gamma \in (0,1]$ determines for how many iterations the previous gradients are incorporated into the current update. Generally γ is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher

Summary

- Convolution
 - A fast method for computing similarity
 - Akin to “simple cell”
- Pooling
 - Translation invariance
 - Akin to “complex cell”
- Review of multi-layer perceptron
- Stochastic gradient descent
 - Batch mode vs minibatch mode
 - momentum