



清华大学 计算机科学与技术系  
Department of Computer Science and Technology, Tsinghua University

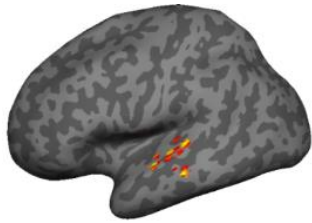
Duke-Tsinghua Machine Learning Summer School  
2017/7/27

# Multi-layer Perceptron and convolutional neural network

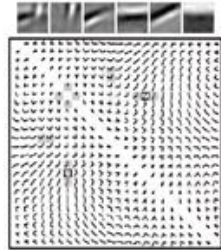
Xiaolin Hu

Department of Computer Science & Technology  
Tsinghua University

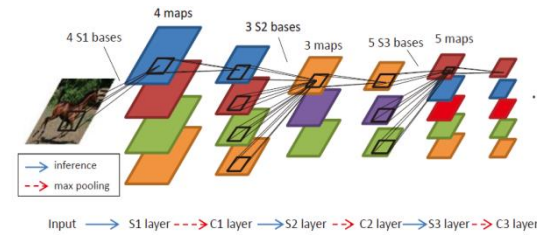
# My Interests



Zhang et al., 2016



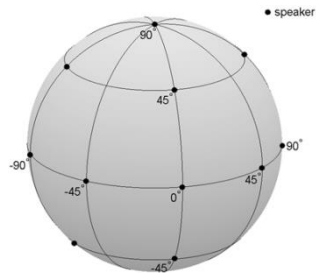
Hu et al., 2012, 2014



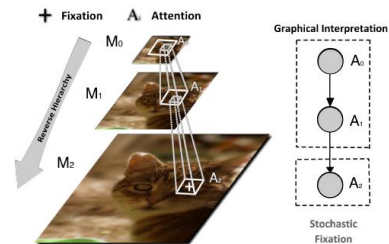
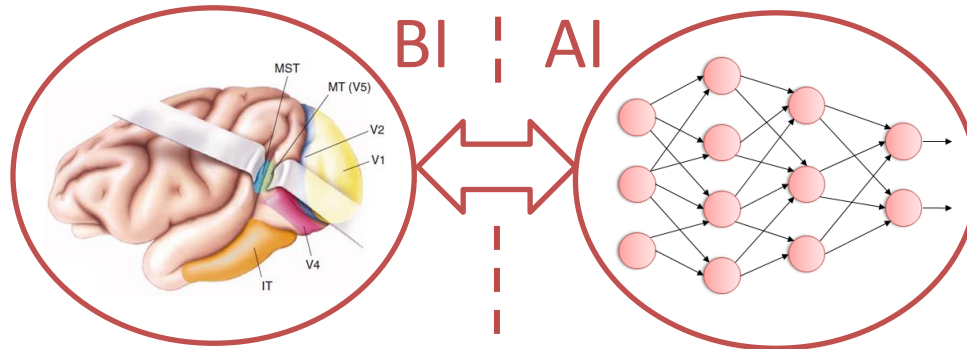
Hu et al., 2014



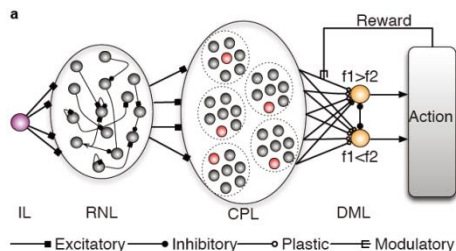
Liang et al., 2013;  
Wu et al., 2013;  
Li et al., 2015



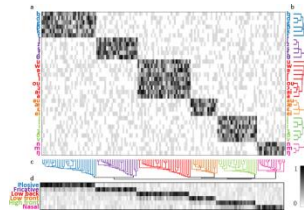
Zhang et al., 2015



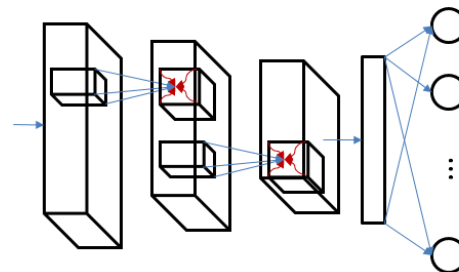
Shi et al., 2014



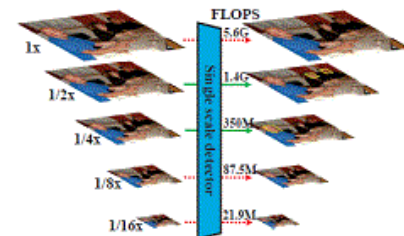
Cheng et al., 2015



Zhang et al.,  
under review



Liang et al., 2015a; 2015b



Hao et al., 2017

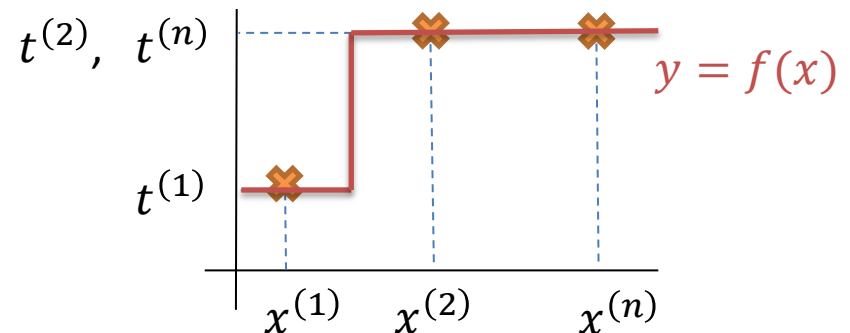
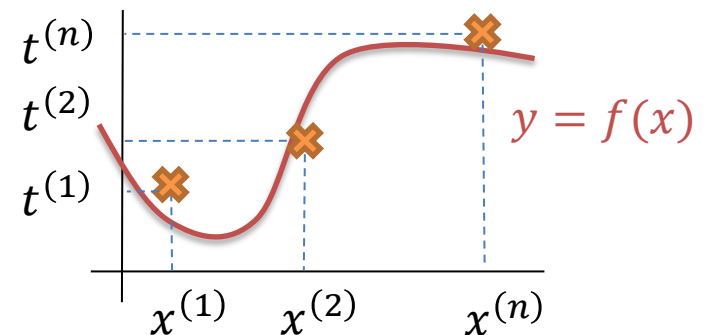
# Outline

- Regression and classification
- Multi-layer perceptron
- Convolutional neural network
- Applications
- Practical tricks

# Regression and classification

Given a set of data points  $x^{(n)} \in R^m$  and the corresponding labels  $t^{(n)} \in \Omega$ :  $\{(x^{(1)}, t^{(1)}), (x^{(2)}, t^{(2)}), \dots, (x^{(N)}, t^{(N)})\}$ , for a new data point  $x$ , predict its label

- The goal is to find a mapping
$$f: R^m \rightarrow \Omega$$
- If  $\Omega$  is a continuous set, this is called regression
- If  $\Omega$  is a discrete set, this is called classification



# Linear regression

- $f(x)$  is linear

$$f(x) = w^T x + b$$

where  $w \in R^n, b \in R$ .

- The cost function can be chosen as the **least square error**

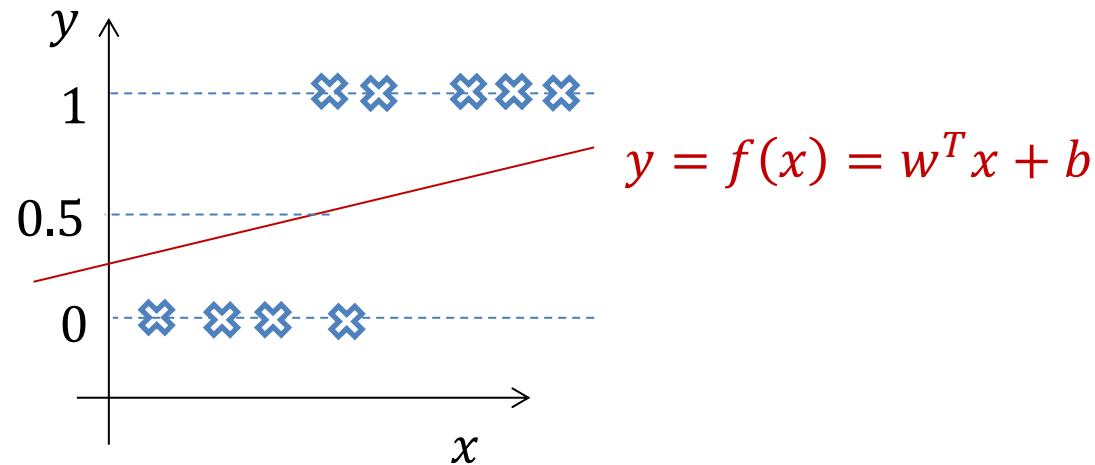
$$E = \sum_{n=1}^N (f(x^{(n)}) - t^{(n)})^2 = \sum_{n=1}^N (w^T x^{(n)} + b - t^{(n)})^2$$

- Find optimal  $w^*$  and  $b^*$  by minimizing the cost function

$$\begin{aligned} \nabla_w E &= \sum_{n=1}^N (w^T x^{(n)} + b - t^{(n)}) x^{(n)} = 0 \\ \nabla_b E &= \sum_{n=1}^N (w^T x^{(n)} + b - t^{(n)}) = 0 \end{aligned} \quad \left. \vphantom{\sum_{n=1}^N} \right\} w^*, b^*$$

# Linear regression as classification

- Suppose  $t \in \{0,1\}$ . Consider the 1D case



- Regression
  - Prediction  $y = f(x)$  which is continuous
- Classification
  - Prediction  $y = \begin{cases} 1, & \text{if } f(x) \geq 0.5 \\ 0, & \text{if } f(x) < 0.5 \end{cases}$

# Representation of class labels

- For classification, given  $\{(x^{(1)}, t^{(1)}), \dots, (x^{(N)}, t^{(N)})\}$ , the goal is to find a mapping from  $x^{(n)}$  to  $t^{(n)}$

$$f: R^m \rightarrow \Omega$$

where  $\Omega$  is a discrete set

- $t^{(n)}$  can be a (discrete) scalar or vector

*Suppose there are 5 classes in total*

Seldom  
used

*Scalar representation*

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

$$t^{(3)} = 3$$

*Vector representation*

$$t^{(1)} = (1, 0, 0, 0, 0)^T$$

$$t^{(3)} = (0, 0, 1, 0, 0)^T$$

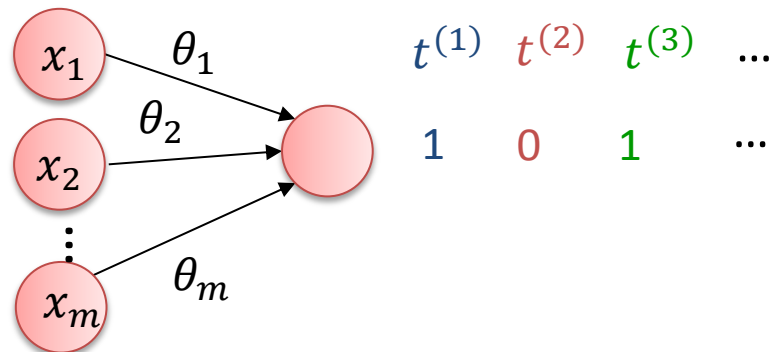
Usually  
used

➤ 1-of-K representation

➤ Property:  $t_k^{(n)} \in \{0, 1\}$ ;  $\sum_k t_k^{(n)} = 1$

# 2-class problems

- For 2-class problems, one 0-1 unit is enough for representing a label



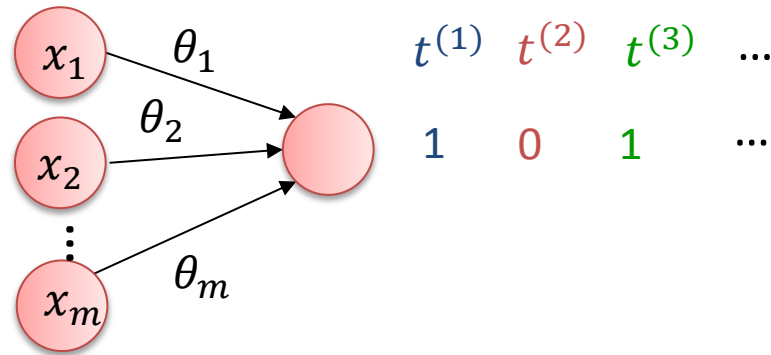
This representation has been used in linear classification by minimizing

$$\sum_{n=1}^N (f(x^{(n)}) - t^{(n)})^2 = \sum_{n=1}^N (w^T x^{(n)} + b - t^{(n)})^2$$



# Logistic regression

- For 2-class problems, one 0-1 unit is enough for representing a label

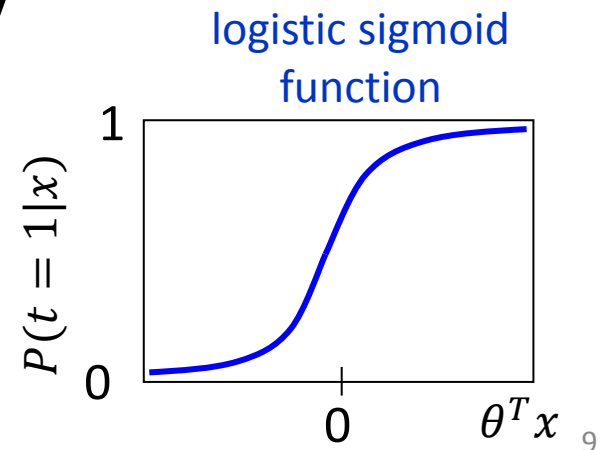


- We try to learn a conditional probability

$$P(t = 1|x) = \frac{1}{1 + \exp(-\theta^\top x)} \triangleq h(x)$$

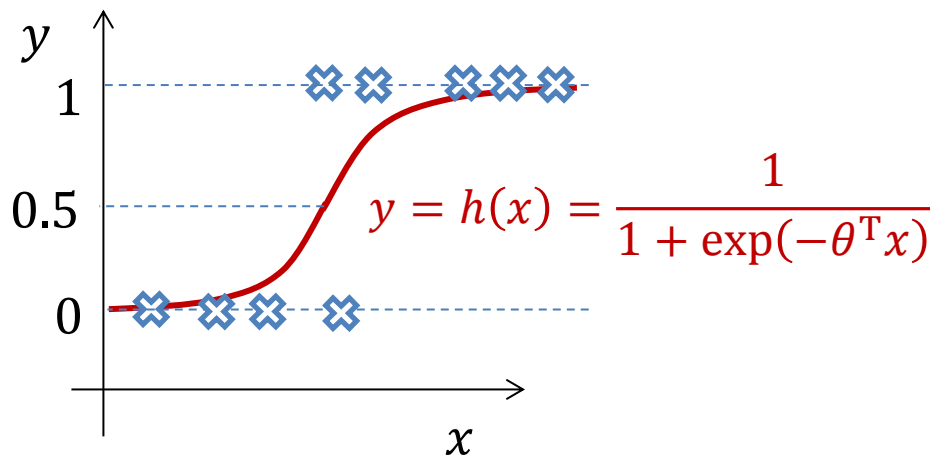
$$P(t = 0|x) = 1 - P(t = 1|x) = 1 - h(x)$$

where  $x$  is input and  $t$  is label



# Logistic regression

- Our goal is to search for a value of  $\theta$  so that the probability  $P(t = 1|x) = h(x)$  is
  - large when  $x$  belongs to class 1 and
  - small when  $x$  belongs to class 0 (so that  $P(t = 0|x)$  is large)
- As we're estimating a conditional probability (continuous), it's "regression"



Regression

Prediction  $y = h(x)$

Classification

Prediction  $y = \begin{cases} 1, & \text{if } h(x) \geq 0.5 \\ 0, & \text{if } h(x) < 0.5 \end{cases}$

Or equivalently

$y = \begin{cases} 1, & \text{if } \theta^T x \geq 0 \\ 0, & \text{if } \theta^T x < 0 \end{cases}$

# Questions

$$y = h(x) = \frac{1}{1 + \exp(-\theta^T x)}$$

- Why not use a linear function for  $h(x)$ 
  - $P(t = 1|x) = h(x)$  should be a probability
- Why should  $h(x)$  be a probability?
  - You'll see the reason soon
- Can we minimize the least square error between  $h(x)$  and  $t$ ?

$$E(\theta) = \frac{1}{N} \sum_{n=1}^N (h(x^{(n)}) - t^{(n)})^2$$

- Yes you can, but here we introduce another error function

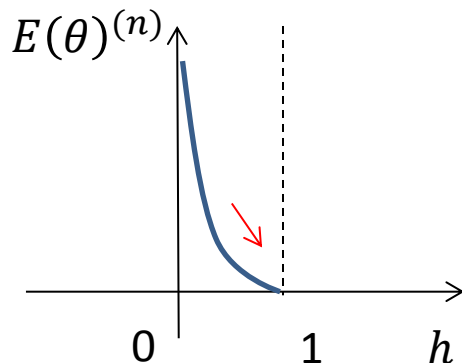
# Cross-entropy error function

- For a set of training examples with binary labels  $\{(x^{(n)}, t^{(n)}) : n = 1, \dots, N\}$  define the *cross-entropy* error function

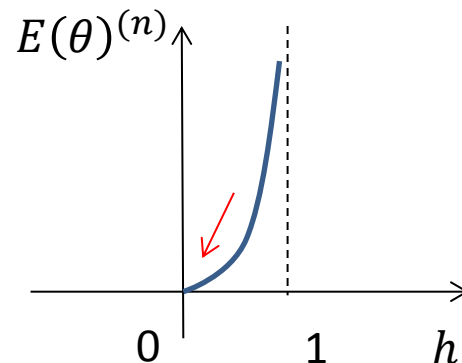
$$E(\theta) = -\frac{1}{N} \sum_{n=1}^N (t^{(n)} \ln(h(x^{(n)})) + (1 - t^{(n)}) \ln(1 - h(x^{(n)})))$$

$$E(\theta)^{(n)} = -t^{(n)} \ln(h(x^{(n)})) - (1 - t^{(n)}) \ln(1 - h(x^{(n)}))$$

➤ If  $t^{(n)} = 1$ , then  
 $E(\theta)^{(n)} = -\ln(h)$



➤ If  $t^{(n)} = 0$ , then  
 $E(\theta)^{(n)} = -\ln(1 - h)$



# Maximum likelihood formulation

- Why do we have this error function?
- For a dataset  $\{(x^{(1)}, t^{(1)}), \dots, (x^{(N)}, t^{(N)})\}$  where  $t^{(n)} \in \{0, 1\}$ , the data likelihood function is

$$p(t^{(1)}, \dots, t^{(N)} | \theta) = \prod_{n=1}^N h(x^{(n)})^{t^{(n)}} (1 - h(x^{(n)}))^{1-t^{(n)}}$$

- Maximizing the likelihood is equivalent to minimizing

$$\begin{aligned} E(\theta) &= -\frac{1}{N} \ln p(t^{(1)}, \dots, t^{(N)}) \\ &= -\frac{1}{N} \sum_{n=1}^N (t^{(n)} \ln h(x^{(n)}) + (1 - t^{(n)}) \ln(1 - h(x^{(n)}))) \end{aligned}$$

# Training and testing

$$E(\theta) = -\frac{1}{N} \sum_{n=1}^N (t^{(n)} \ln h(x^{(n)}) + (1 - t^{(n)}) \ln(1 - h(x^{(n)})))$$

- Calculate the gradient (**exercise**)

$$\nabla E(\theta) = \frac{1}{N} \sum_n x^{(n)} (h(x^{(n)}) - t^{(n)})$$

- Some regularization term can be incorporated into the cost function

$$J(\theta) = E(\theta) + \lambda ||\theta||^2 / 2$$

- **Training:** learn  $\theta$  to minimize the cost function

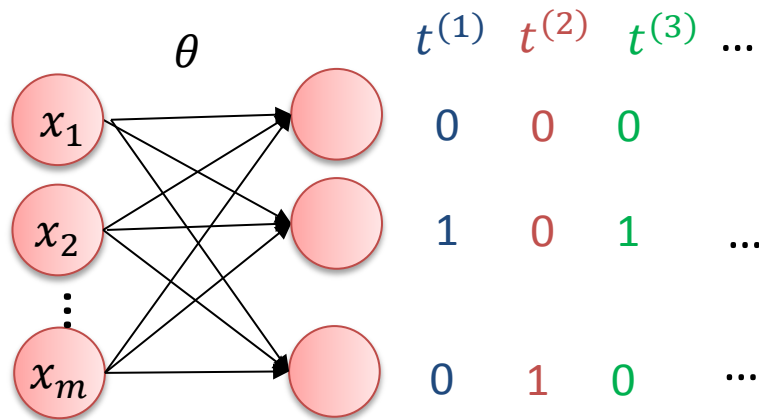
$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

where  $\alpha$  is the learning rate

- **Testing:** for a new input  $x$ , if  $P(t = 1|x) > P(t = 0|x)$  then we predict the input as class 1, and 0 otherwise

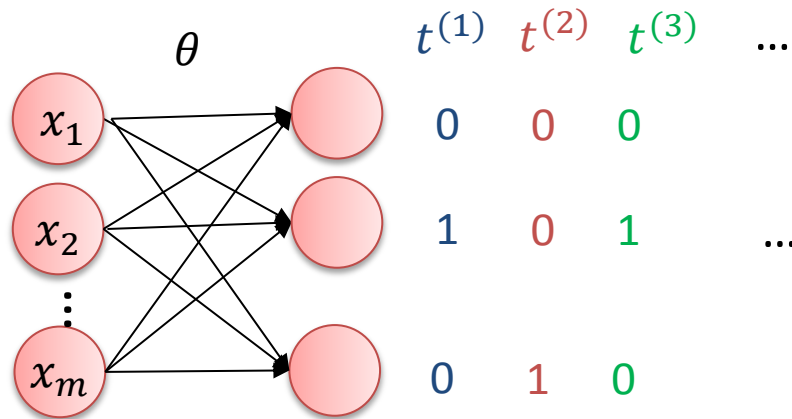
# More than two classes

- For  $K$ -class problems ( $K > 2$ ),
  - One unit is enough for representing a label if it can take discrete values, e.g.,  $0, 1, 2, \dots, K$   $\leftarrow$  scalar representation
  - $K$  0-1 units can be also used to represent a label  $\leftarrow$  vector representation



# More than two classes

- For  $K$ -class problems ( $K > 2$ ),  $K$  0-1 units is used to represent a label



Note that  $\sum_k t_k^{(n)} = 1$

- We try to learn a hypothesis  $h(x)$  of the form

$$h(x) \triangleq \begin{bmatrix} P(t_1 = 1|x; \theta) \\ P(t_2 = 1|x; \theta) \\ \vdots \\ P(t_K = 1|x; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^K \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix}$$




# More than two classes

Then 
$$h_k(x) = P(t_k = 1|x) = \frac{\exp(\theta^{(k)\top} x)}{\sum_{j=1}^K \exp(\theta^{(j)\top} x)}$$

- Given a test input  $x$ , estimate  $P(t_k = 1|x)$  for each value of  $k = 1, \dots, K$ .
- Goal: search for a value of  $\theta$  so that the probability  $P(t_k = 1|x)$  is
  - large when  $x$  belongs to the  $k$ -th class and
  - small when  $x$  belongs to other classes

where 
$$\theta = \begin{bmatrix} \theta^{(1)} & \theta^{(2)} & \dots & \theta^{(K)} \\ | & | & | & | \end{bmatrix}.$$

- Since  $h_k(x)$  is a (continuous) probability, we need to transform it into discrete values for classification ←How? 

# Softmax function

$$h_k(x) = P(t_k = 1|x) = \frac{\exp(\theta^{(k)\top} x)}{\sum_{j=1}^K \exp(\theta^{(j)\top} x)}$$

- The following function is called *softmax* function

$$\psi(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} = \frac{\exp(z_i)}{\exp(z_i) + \sum_{j \neq i} \exp(z_j)} \in (0, 1) \quad \text{💬}$$

- If  $z_i > z_j$  for all  $j \neq i$ 
  - Then  $\psi(z_i) > \psi(z_j)$  for all  $j \neq i$  but it is smaller than 1
- If  $z_i \gg z_j$  for all  $j \neq i$ , 💬
  - then  $\psi(z_i) \rightarrow 1$  and  $\psi(z_j) \rightarrow 0$  for  $j \neq i$ .

# Error function

- Least square error function

$$E = \frac{1}{N} \sum_{n=1}^N E^{(n)} \quad \text{where} \quad E^{(n)} = \frac{1}{2} \| \overset{\text{vector}}{h(x^{(n)})} - t^{(n)} \|^2$$

- Cross-entropy error function

$$E(\theta) = \frac{1}{N} \sum_{n=1}^N E^{(n)}(\theta), \quad \text{where} \quad E^{(n)}(\theta) = - \sum_{i=1}^K t_i^{(n)} \ln h_i^{(n)}$$

- In practice, the cross-entropy error function works better

# Cross-entropy error function



- The data likelihood function is

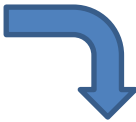


$$p(t^{(1)}, \dots, t^{(N)} | \theta) = \prod_{n=1}^N \prod_{k=1}^K P(t_k^{(n)} = 1 | x^{(n)})^{t_k^{(n)}}$$

- The cross-entropy error function is



$$E(\theta) = -\frac{1}{N} \ln p(t^{(1)}, \dots, t^{(N)})$$

$$= -\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K t_k^{(n)} \ln \frac{\exp(\theta^{(k)\top} x^{(n)})}{\sum_{j=1}^K \exp(\theta^{(j)\top} x^{(n)})}$$


$$E(\theta) = \frac{1}{N} \sum_{n=1}^N E^{(n)}(\theta), \quad E^{(n)}(\theta) = -\sum_{i=1}^K t_i^{(n)} \ln h_i^{(n)}$$

where  $h_i^{(n)} = P(t_i^{(n)} = 1 | x^{(n)}) = \frac{\exp(u_i^{(n)})}{\sum_{j=1}^K \exp(u_j^{(n)})}$ ,  $u_k^{(n)} = \theta^{(k)\top} x^{(n)}$

# Calculate the gradient

- By defining the *local sensitivity*

$$\delta_k^{(n)} \triangleq \frac{\partial E^{(n)}}{\partial u_k^{(n)}} = - \left( t_k^{(n)} - h_k^{(n)} \right) \quad \text{where } u_k^{(n)} = \theta^{(k)\top} x^{(n)}$$



- It can be shown that (the derivation is not covered in this lecture)

$$\frac{\partial E^{(n)}}{\partial \theta^{(k)}} = \delta_k^{(n)} x^{(n)}$$

- The overall gradient

$$\begin{aligned} \nabla_{\theta^{(k)}} E(\theta) &= \frac{1}{N} \sum_{n=1}^N \frac{\partial E^{(n)}}{\partial \theta^{(k)}} = -\frac{1}{N} \sum_{n=1}^N \left( t_k^{(n)} - h_k^{(n)} \right) x^{(n)} \\ &= -\frac{1}{N} \sum_{n=1}^N \left( t_k^{(n)} - P(t_k^{(n)} = 1 | x^{(n)}) \right) x^{(n)} \end{aligned}$$

# Training and testing

- Calculate the gradient of the cross-entropy error function

$$\nabla_{\theta^{(k)}} E(\theta) = -\frac{1}{N} \sum_{n=1}^N \left( t_k^{(n)} - h_k^{(n)} \right) x^{(n)}$$

- As before, some **regularization term** can be incorporated into the cost function

$$J(\theta) = E(\theta) + \lambda ||\theta||^2 / 2$$

- **Training:** minimize the cost function with gradient  $\nabla J(\theta)$

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

where  $\alpha$  is the learning rate

- **Testing:** find the maximum  $P(t_k = 1|x)$  among  $k$  for a new input  $x$

$$P(t_k = 1|x) = \frac{\exp(\theta^{(k)\top} x)}{\sum_{j=1}^K \exp(\theta^{(j)\top} x)}$$

# Stochastic gradient descent

- **Batch gradient descent** algorithm updates the parameters  $\theta$  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 using only a single or a few training examples.

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

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.
  - Reduces the variance in the parameter update
  - Take advantage of highly optimized matrix operations

# Introducing bias

- So far we have assumed

$$h_k(x) = P(t_k = 1|x) = \frac{\exp(u_k^{(n)})}{\sum_{j=1}^K \exp(u_j^{(n)})} \quad u_k^{(n)} = \theta^{(k)\top} x^{(n)}$$

- Sometimes a bias is introduced into  $u_k^{(n)}$  and the parameters become  $\{W, b\}$

$$u_k^{(n)} = W^{(k)\top} x^{(n)} + b^{(k)}$$

- It's easy to show that  $\nabla_{W^{(k)}} E(\theta) = -\frac{1}{N} \sum_{n=1}^N \left( t_k^{(n)} - h_k(x^{(n)}) \right) x^{(n)}$

$$\nabla_{b^{(k)}} E(\theta) = -\frac{1}{N} \sum_{n=1}^N \left( t_k^{(n)} - h_k(x^{(n)}) \right)$$

- Regularization is often applied on  $W$  only

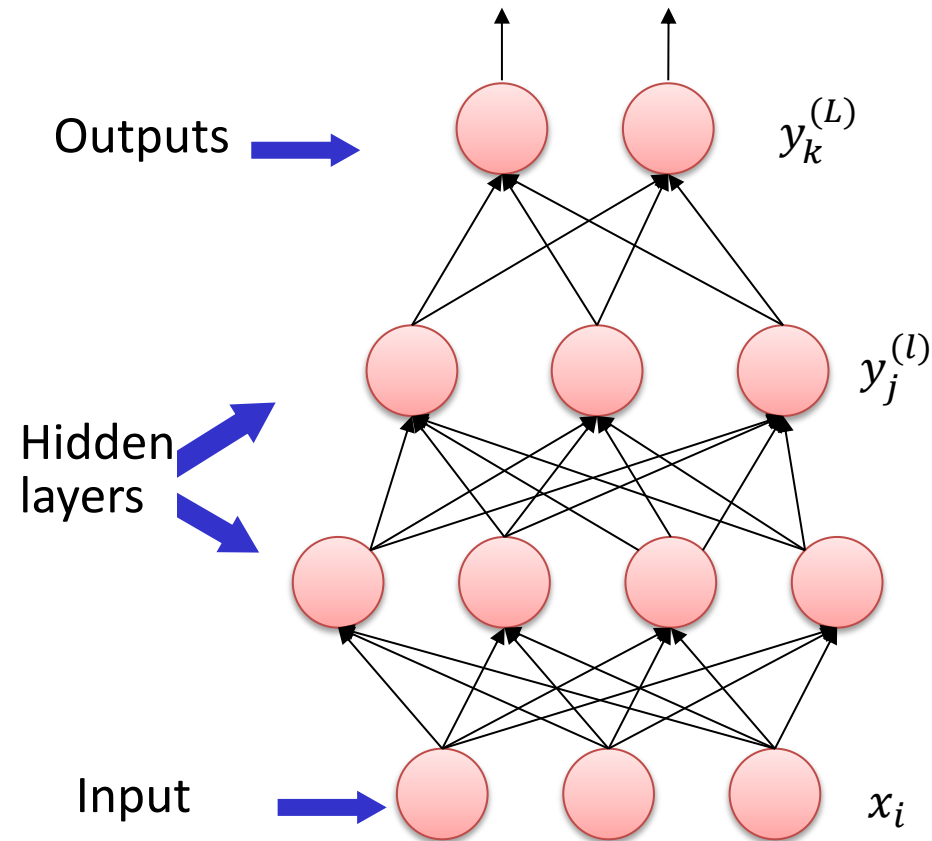
$$J(W, b) = E(W, b) + \lambda \|W\|^2 / 2$$



# Outline

- Regression and classification
- Multi-layer perceptron
- Convolutional neural network
- Applications
- Practical tricks

# Multi-layer Perceptron (MLP)



- There are a total of  $L$  layers except the input
- Connections:
  - Full 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

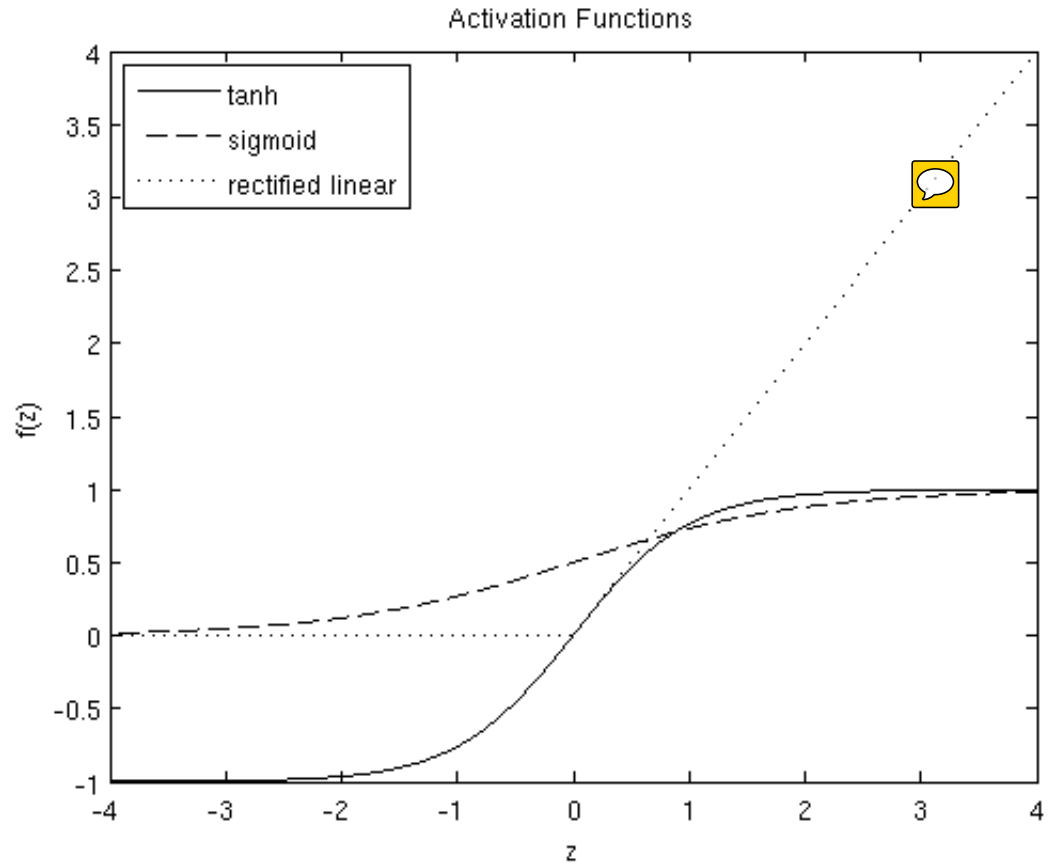
$$f(z) = \frac{1}{1 + \exp(-z)}$$

- Hyperbolic tangent function

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

- Rectified linear activation function (ReLU)

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



# Activation functions

- Logistic function

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

gradient



$$f'(z) = f(z)(1 - f(z))$$

- Hyperbolic tangent function

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

gradient



$$f'(z) = 1 - f(z)^2$$

- Rectified linear activation function (ReLU)

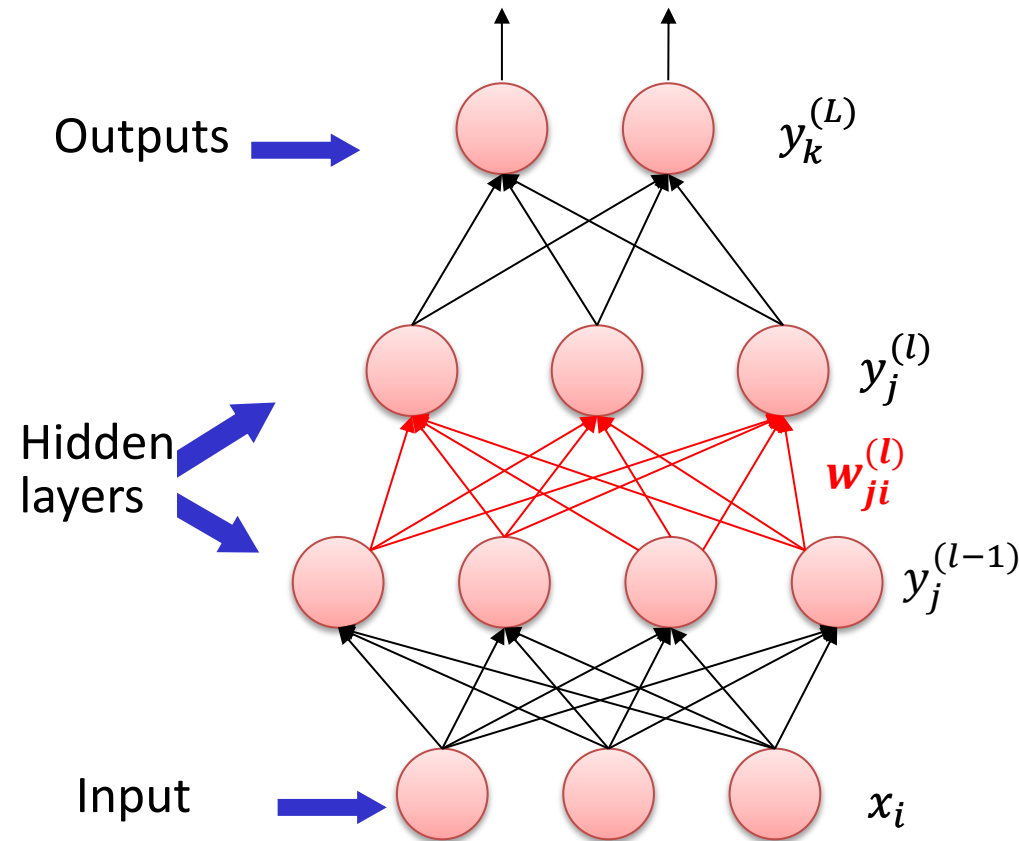
$$f(z) = \max(0, z)$$

gradient



$$f'(z) = \begin{cases} 1, & \text{if } z \geq 0, \\ 0, & \text{else} \end{cases}$$

# Forward pass



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

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

and its output

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

where  $f(\cdot)$  is activation function

- Note  $y^{(0)} = x$
- There are desired outputs  $t$  for each input sample
- For  $l = L$ ,  $f(\cdot)$  can be an activation function or the softmax function

# Error functions for BP

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

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

- Least square error

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



Is ReLU applicable?

- Cross-entropy error

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



where  $t$  is target of the form  $(0, 0, \dots, 1, 0, 0)^T$

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


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

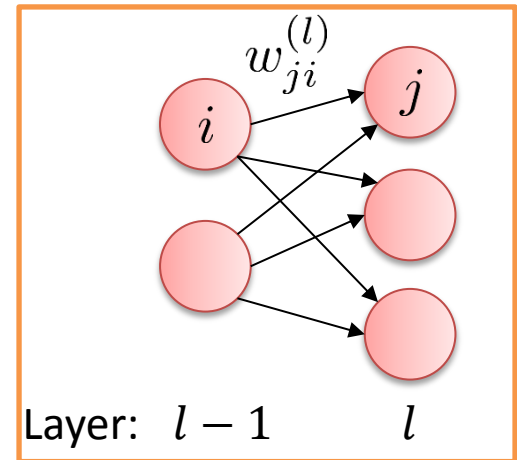
# Gradient and local sensitivity

- Define local sensitivity  $\delta_i^{(l)} = \frac{\partial E^{(n)}}{\partial u_i^{(l)}}$  
- Then for  $1 \leq l \leq L$

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

$$\frac{\partial E^{(n)}}{\partial b_j^{(l)}} = \delta_j^{(l)},$$

since   $u_j^{(l)} = \sum_i w_{ji}^{(l)} f(u_i^{(l-1)}) + b_j^{(l)}$ , where  $f$  is the activation function and  $f(u_i^{(0)}) = x$ .



Computing the gradients amounts to computing the local sensitivity in each layer!

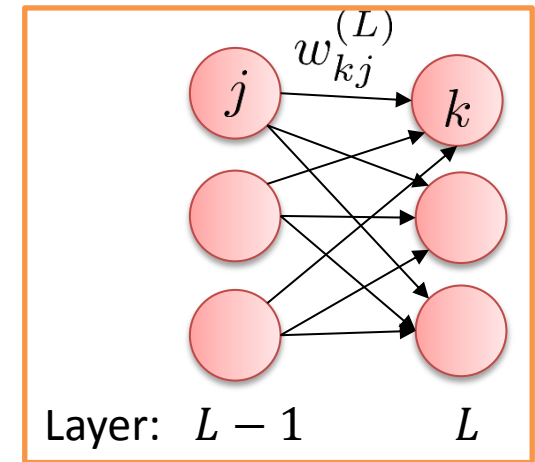


# Local sensitivity for least square error layer

- If least square error is used then the output of the last layer units of MLP are

$$y_k^{(L)} = f(u_k^{(L)}) = f(w_k^{(L)\top} \boxed{y^{(L-1)}} + b_k^{(L)})$$

Output of the units on  
the (L-1)-th layer



where the activation function  $f$  can be

✓ logistic sigmoid

✓ tanh

✓ ReLU

- Recall the error for each sample  $E^{(n)} = \frac{1}{2} \sum_{k=1}^K (t_k - y_k^{(L)})^2$ ,
- Local sensitivity

$$\delta_k^{(L)} \triangleq \frac{\partial E^{(n)}}{\partial u_k^{(L)}} = (y_k^{(L)} - t_k) f'(u_k^{(L)})$$

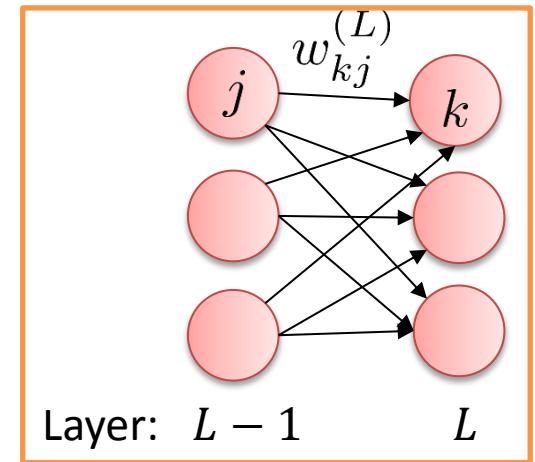
# Recall local sensitivity for softmax layer

- If softmax regression is used in the last layer of an MLP, then  $\theta$  is replaced with  $w^{(L-1)}$  and  $b^{(L-1)}$  and the probabilistic function becomes

Output of the units on  
the (L-1)-th layer



$$y_k^{(L)} \triangleq P(t_k = 1 | \boxed{y^{(L-1)}}) = \frac{\exp(w_k^{(L)\top} y^{(L-1)} + b_k^{(L)})}{\sum_{j=1}^K \exp(w_j^{(L)\top} y^{(L-1)} + b_j^{(L)})}$$



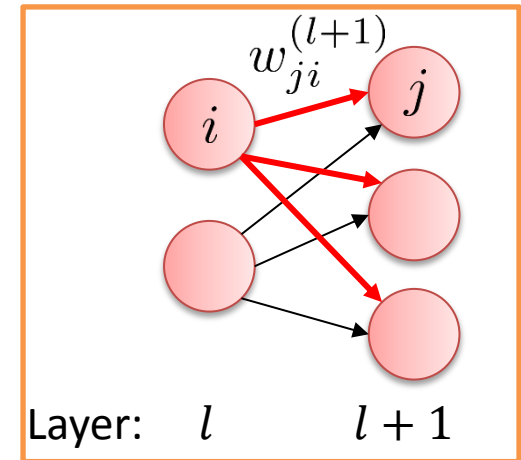
- Local sensitivity

$$\delta_k^{(L)} \triangleq \frac{\partial E^{(n)}}{\partial u_k^{(L)}} = y_k^{(L)} - t_k$$



# Local sensitivity for other layers

- 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



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

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

where  $f$  can be any activation function

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

# Backpropagation 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  $1 \leq l < L$

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

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

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

- Update weights

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

for each sample  $n$

→ sum over  $n$

# Gradient vanishing

- Note that for the hidden layer  $1 \leq l < L$

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

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



we have  $f'(z) = f(z)(1 - f(z)) < 1$

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

we have  $f'(z) = 1 - \tanh^2(z) < 1$

- For these two sigmoid functions,  $\delta^{(l)}$  is smaller and smaller from  $L$  to 1
- The gradient approaches zero in lower layers

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

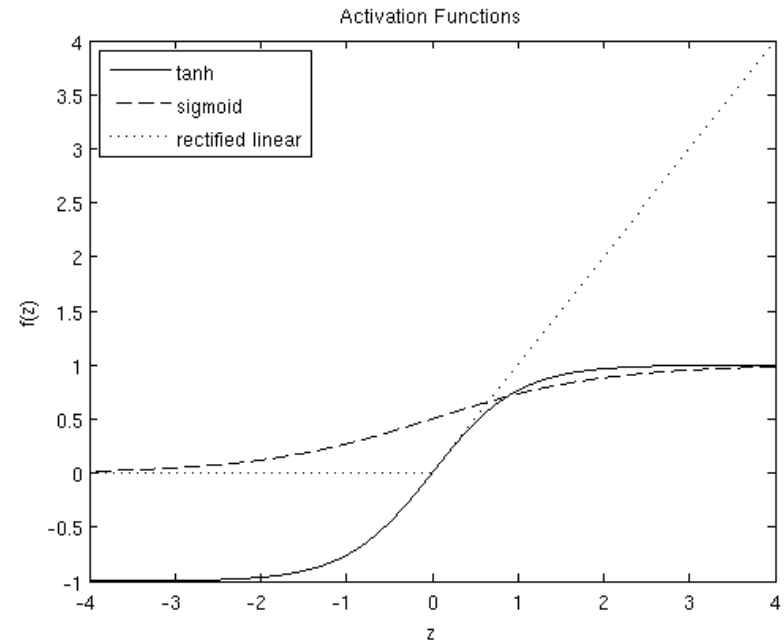
# Rectified linear activation function

- Rectified linear unit (ReLU)

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



- It's derivative is

$$f'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$



- The gradient of  $E$  doesn't have gradient vanishing effect, if only it is nonzero
- The gradient becomes sparse (many components are zero)!

# 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  $\partial E / \partial W^{(l)}, \partial E / \partial b^{(l)}$  for  $l = L, L - 1, \dots, 1$
- Update  $W^{(l)}$  and  $b^{(l)}$  for  $l = 1, 2, \dots, L$
- Modular programming  $\leftarrow$  Basic idea of tensorflow, Caffe, etc.
  - 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

# More flexible setting

- The input layer or hidden layer

$$y_j^{(l)} = f \left( \sum_i w_{ji}^{(l)} y_i^{(l-1)} + b_j^{(l)} \right)$$

can be decomposed into two layers

- Fully connected layer:  $u_j^{(l)} = \sum_i w_{ji}^{(l)} y_i^{(l-1)} + b_j^{(l)}$
- Activation layer:  $y_j^{(l)} = f(u_j^{(l)})$

- The least square error layer  $E^{(n)} = \frac{1}{2} \left\| f(u^{(L)}) - t \right\|^2$

can be decomposed into two layers

- Activation layer:  $y_k^{(L)} = f(u_k^{(L)})$ , where  $f$  can be any function
- Loss layer:  $E^{(n)} = \frac{1}{2} \left\| y^{(L)} - t \right\|^2$



# Question


- Consider the **least square error function** discussed before

$$E^{(n)} = \frac{1}{2} \left\| f(W^{(L)}y^{(L-1)} + b^{(L)}) - t \right\|^2$$

How many layers can be designed? 

Fc layer + activation layer + Euclidean loss layer 

# More flexible setting

- The cross-entropy error layer  $E^{(n)} = -\sum_{k=1}^K t_k \ln f(u_k^{(L)})$  can be decomposed into two layers
  - Softmax layer:  $y_k^{(L)} = f(u_k^{(L)})$ , where  $f$  is the softmax function
  - Loss layer:  $E^{(n)} = -\sum_{k=1}^K t_k \ln y_k^{(L)}$
  - But this is unnecessary! **Why?** 
- Consider this error discussed before

$$E^{(n)} = -\sum_{k=1}^K t_k \ln f\left(\sum_i w_{ki}^{(l)} y_i^{(l-1)} + b_k^{(l)}\right)$$

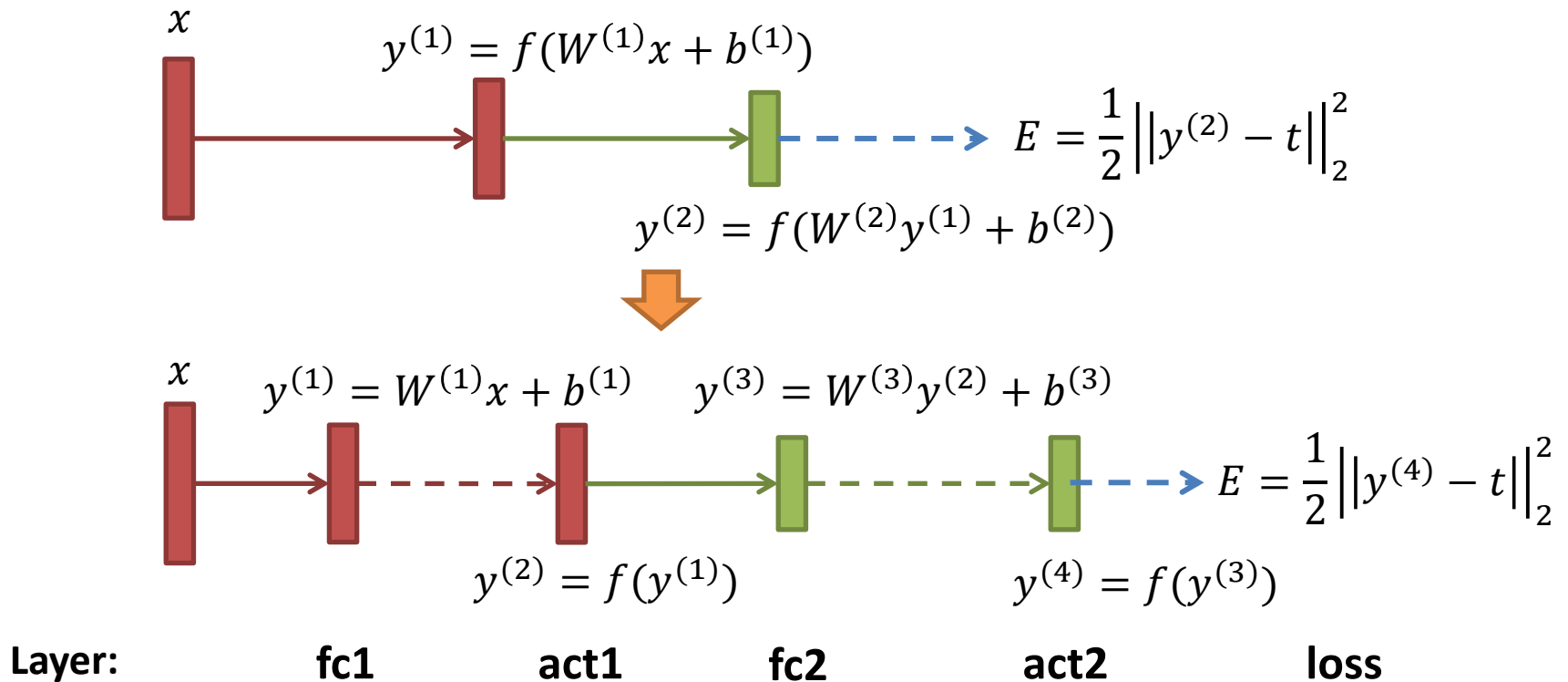
How many layers can be designed?

**Fc layer + softmax crossentropy layer**

# Example

- An MLP with one hidden layer using least square error function

Solid arrow: W/ param.  
Dashed arrow: w/o param.



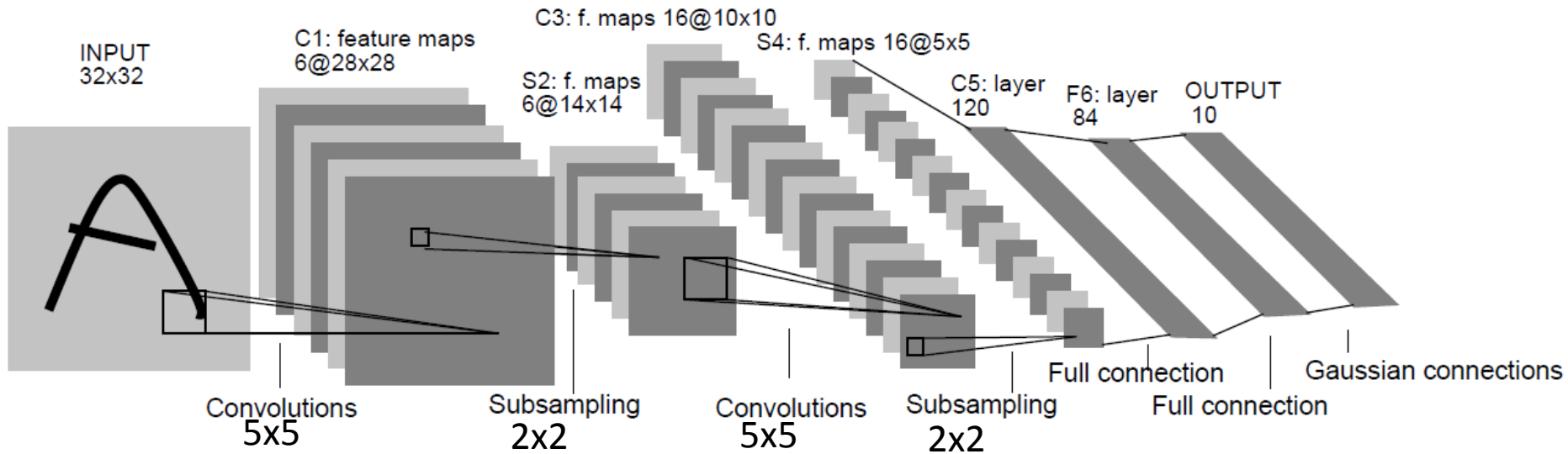
# Problem 1

- Derive the local sensitivity  $\delta$  and gradient  $\partial E^{(n)} / \partial w^{(l)}$  and  $\partial E^{(n)} / \partial b^{(l)}$  where applicable for
  - fully connected layer
  - sigmoid layer
  - ReLU layer
  - Euclidean loss layer
- For convenience, denote the input to the current layer  $l$  by  $u^{(l)}$
- Examples of these layers are shown in the previous slide

# Outline

- Regression and classification
- Multi-layer perceptron
- Convolutional neural network
- Applications
- Practical tricks

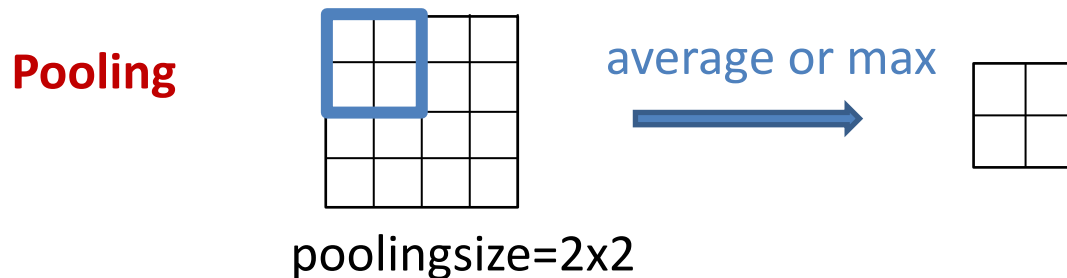
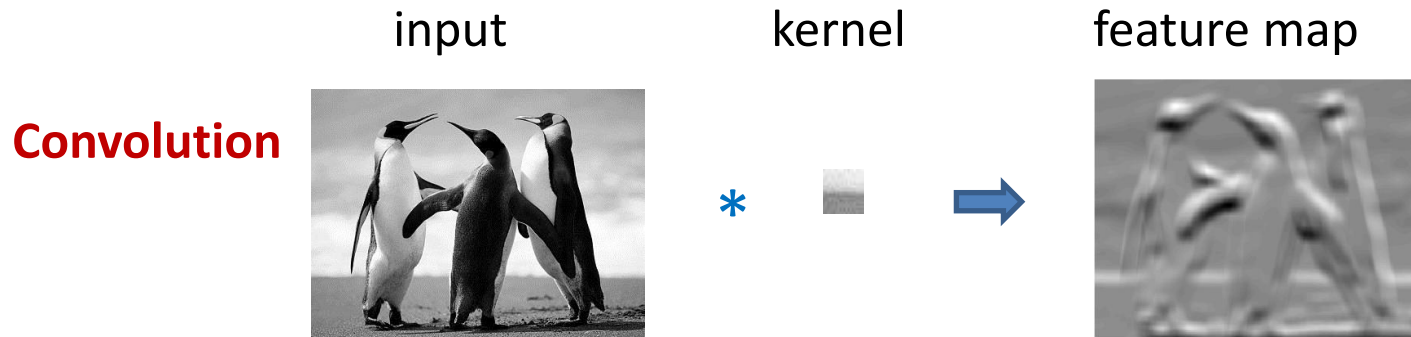
# A typical example



- Local connections and weight sharing
- C layers: convolution
  - Output  $y_i = f(\sum_{\Omega} w_j x_j + b)$  where  $\Omega$  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  $\Omega$  is the pooling size

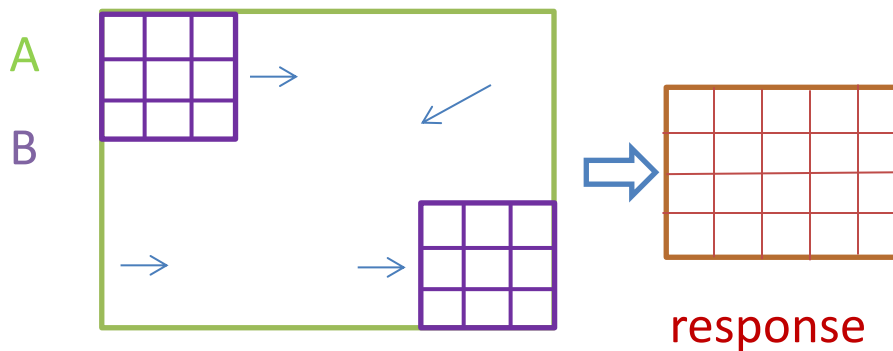
# Two new layers

- Define two additional layers with forward computation and backward computation
  - Convolution layer and pooling layer



# Motivation for convolution

- 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...

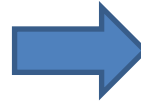


# Example

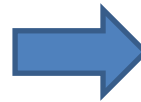
figure



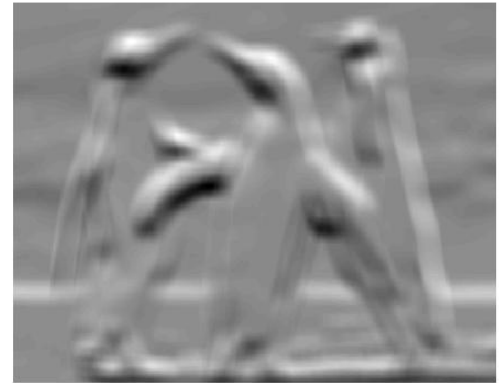
kernel



\*



feature map

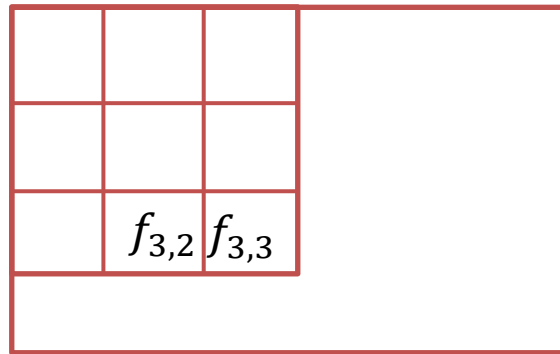
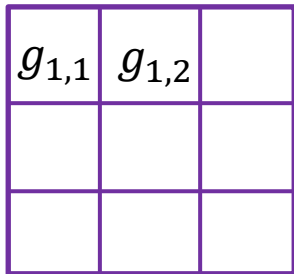


The higher a pixel value (brighter) in the feature map, the more similar between the filter and the corresponding patch in the figure

# 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  $M \times N$

# Matlab example

```
>> A = round(3*rand(4))
```

A =

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

```
>> B = round(2*rand(3))-1
```

B =

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

```
>> C = conv2(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 = conv2(A,B,'valid')
```

D =

-3	0
4	3

# Matlab example

```
>> A = round(3*rand(4))
```

A =

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

```
>> B = round(2*rand(3))-1
```

B =

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

```
>> C = conv2(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 = conv2(A,B,'same')
```

D =

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

# Python example

```
import numpy
from scipy import signal
A = numpy.array([[0,0,1,2],[2,2,0,0],[2,1,2,2],[3,0,1,1]])
B = numpy.array([[0,0,-1],[1,-1,1],[-1,1,1]])
C = signal.convolve2d(A,B,mode='full')
print(C)
C = signal.convolve2d(A,B,mode='valid')
print(C)
C = signal.convolve2d(A,B,mode='same')
print(C)
```

You would obtain the same results as before

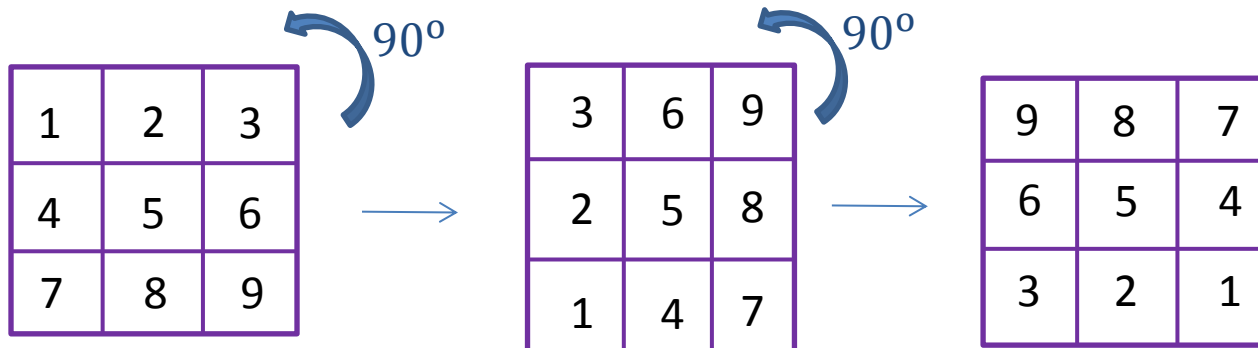
# Relationship between similarity and convolution

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

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

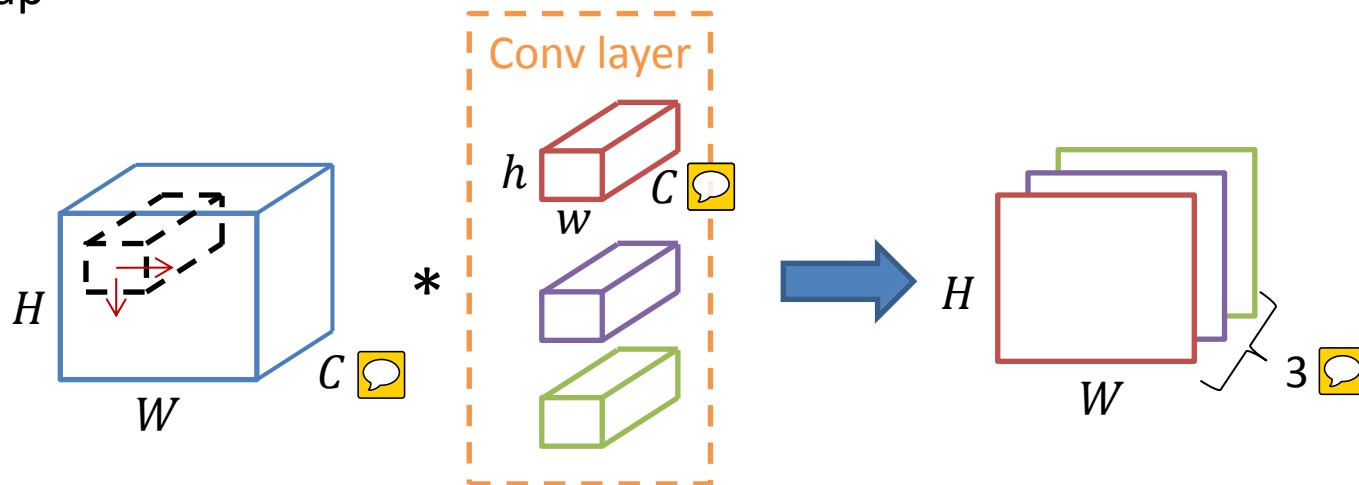
$$\tilde{g}_{M,1} = g_{1,N}, \tilde{g}_{M,2} = g_{1,N-1}, \dots, \tilde{g}_{M,N} = g_{1,1}$$

- In Matlab, the above flip operation can be realized by applying the command `rot90()` twice [or `rot180()`]



# 3D convolution

- Why 3D convolution?
  - The input might be 3D, e.g., RGB channels
- In general we assume the number of channels in the input is the same as that in the kernel (filter)
- Convolve a 2D feature map in the 3D input with the *corresponding* 2D section in the 3D kernel, then sum over all sections to yield one feature map



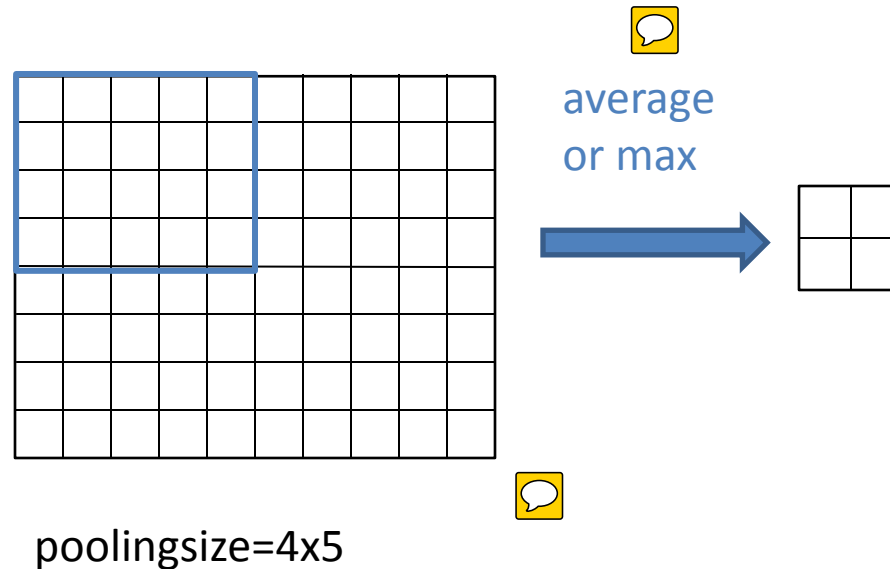
The number of parameters in this layer is  $h \times w \times C \times 3$

# 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
  - We can transform the similarity calculation to matrix multiplication form



# Pooling in local regions




- Divide the convolved features into *disjoint*  $m \times n$  regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled features
- How about 3D input?
  - Channel-wise pooling: if the input has  $C$  channels, then the output also has  $C$  channels

# Why do we need pooling

- Reduce the number of features for final classification
  - Consider images of  $96 \times 96$  pixels. Suppose we have learned 400 features over  $8 \times 8$  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

# Backward computation

- We have discussed the forward computation of the convolutional layer and pooling layer
- For BP 
  - in the convolutional layer, you need to calculate the local sensitivity and parameter gradient
  - in the pooling layer, you need to calculate the local sensitivity
- But this is a little bit complicated
  - Fortunately, we have toolboxes

# Deep learning tools

- Theano @ University of Montreal
- Caffe @ UC Berkley
- TensorFlow @ Google
- Torch @ Facebook

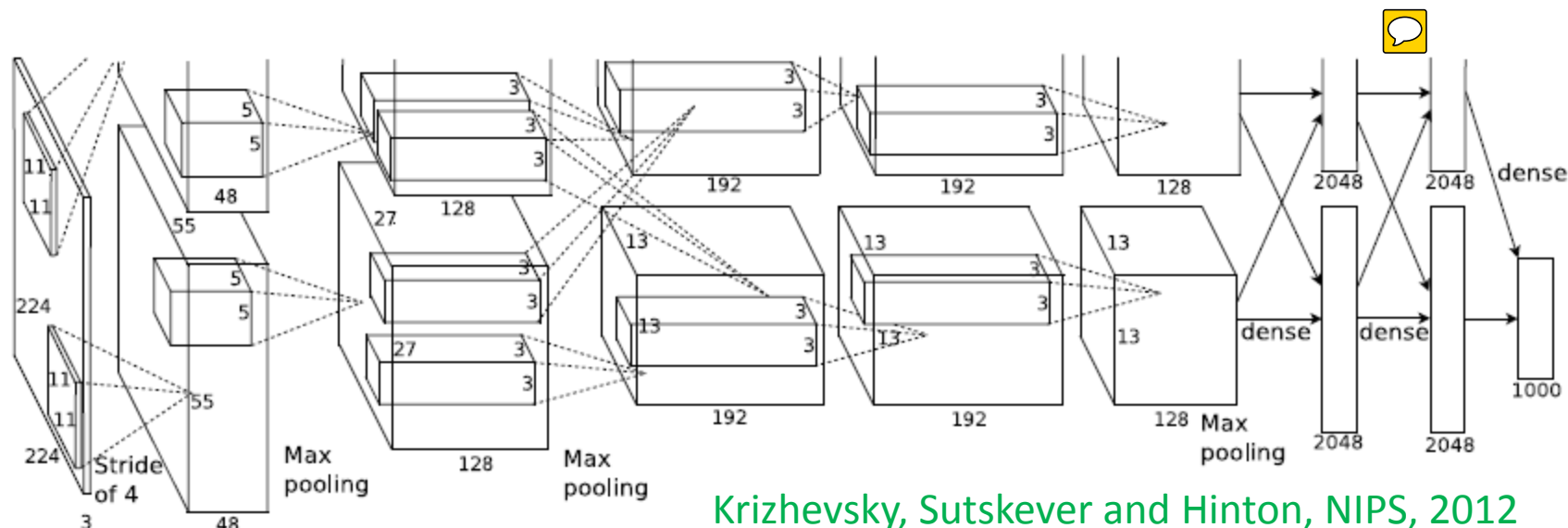
# Construction of CNN

- The convolutional layers and pooling layers can be combined freely with other layers that we have discussed
  - Fully connected layer
  - Sigmoid layer
  - ReLU layer
  - Euclidean loss layer
  - Cross-entropy loss layer
- as well as other layers that we haven't discussed, e.g.,
  - Local response normalization layer (Krizhevsky et al. 2012)
  - Dropout layer (Srivastava et al., 2014)
  - PReLU layer (He et al., 2015)
  - Batch normalization layer (Ioffe and Szegedy, 2015)

# Outline

- Regression and classification
- Multi-layer perceptron
- Convolutional neural network
- Applications
- Practical tricks

# CNN for image classification



- Network dimension: 150,528(input)-253,440-186,624-64,896-64,896-43,264-4096-4096-1000(output)
- In total: 60 million parameters
- Task: classify 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes
- Results: state-of-the-art accuracy on ImageNet

# VGG net

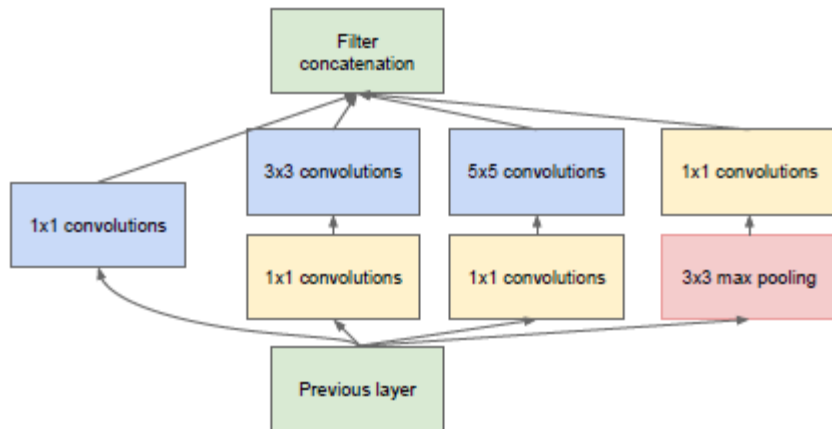
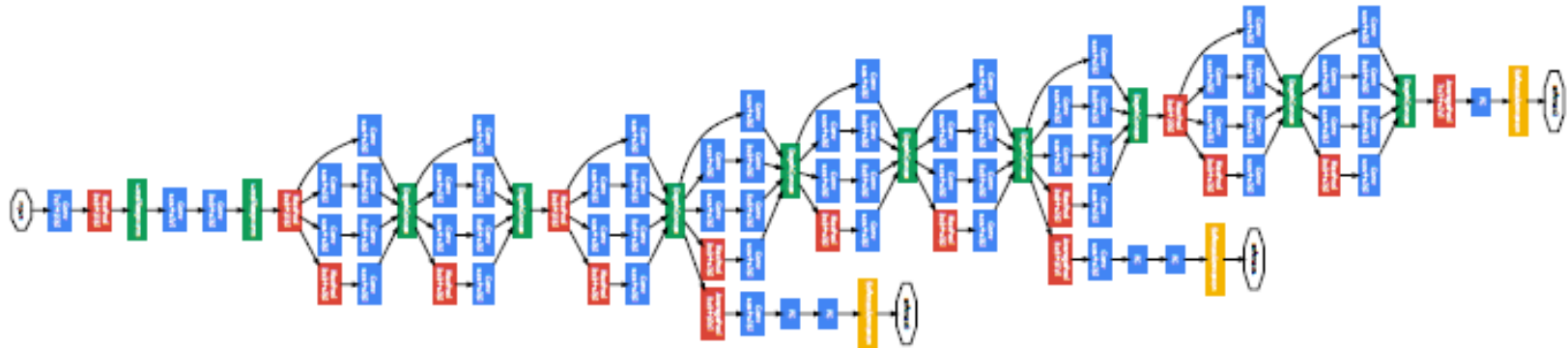
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input ( $224 \times 224$ RGB image)					
conv3-64	conv3-64 <b>LRN</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

- 3\*3 filters are extensively used
- GPU implementation

Simonyan, Zisserman, 2015



# GoogLeNet



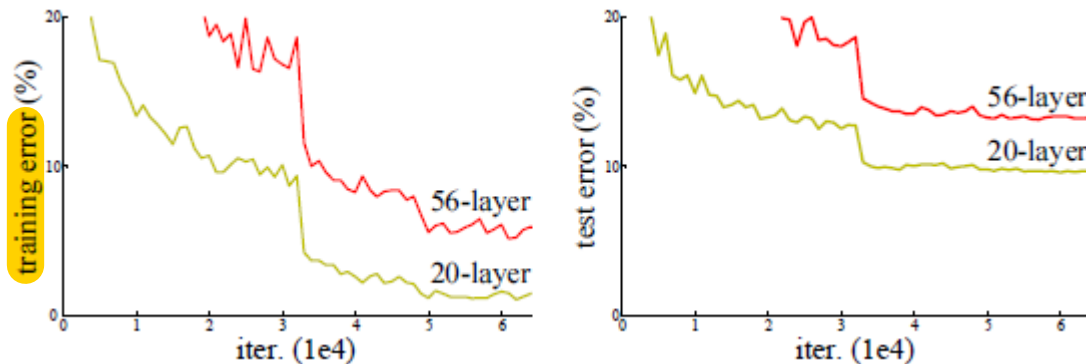
- 22 weight layers
- Small filters (1x1, 3x3, 5x5)
- Two auxiliary classifiers connected to intermediate layers are used to increase the gradient signal for BP algorithm
- A cpu-based implementation on distributed system

Szegedy, et al., 2014

# Deep residual net

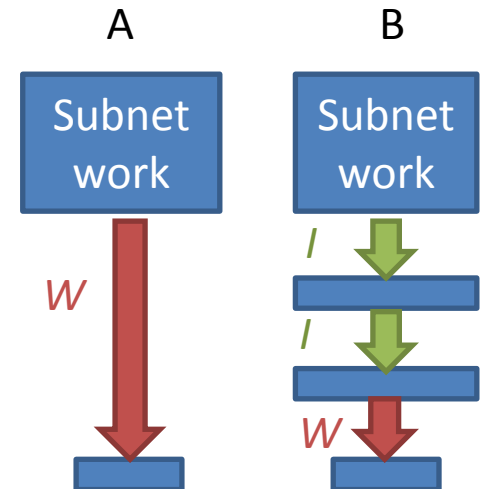
He et al., 2016

- Empirical observations: with the network depth increasing, accuracy gets saturated and then degrades rapidly



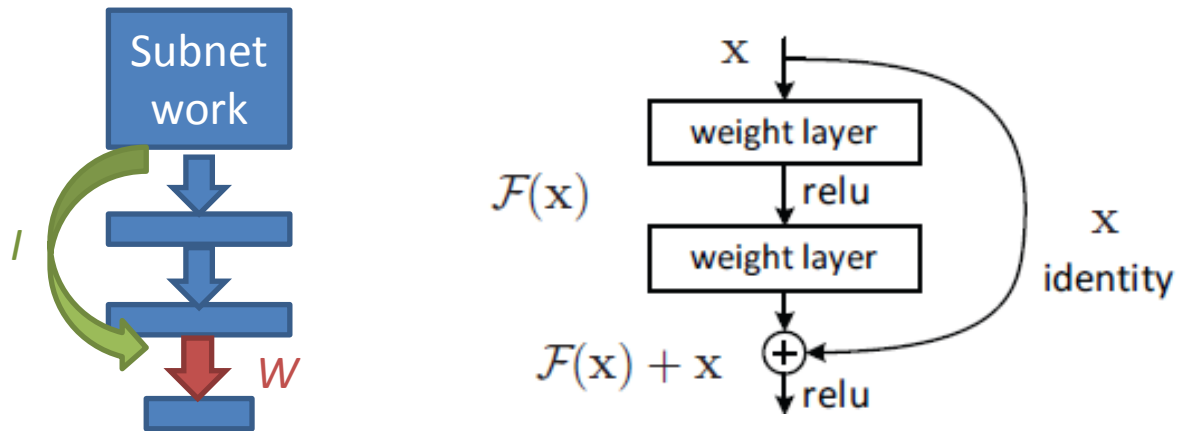
- Is it caused by over-fitting?
- This is counterintuitive!
  - Error of B should not be larger than that of A

Hypo: it's difficult for nonlinear layers to approx. the identity mapping



# Deep residual net

- If this hypothesis is valid, let's explicitly use the identity mapping

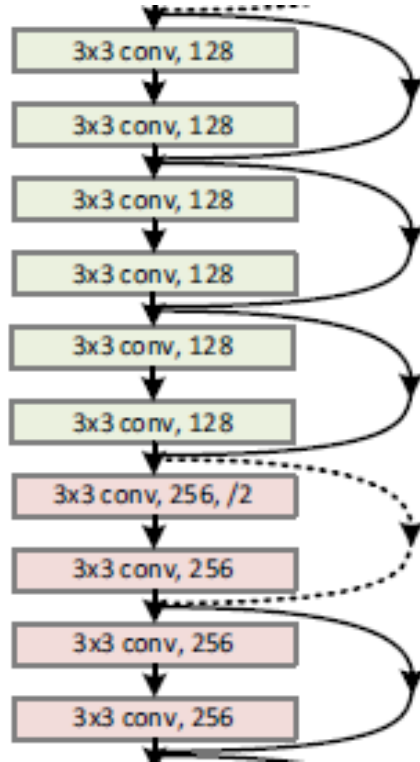


- The nonlinear mapping from input to output  $H(x)$  has two parts

$$H(x) = F(x) + x$$

- Then the two weight layers are learning  $F(x)$ , i.e., the **residual**  $H(x) - x$

# Deep residual net

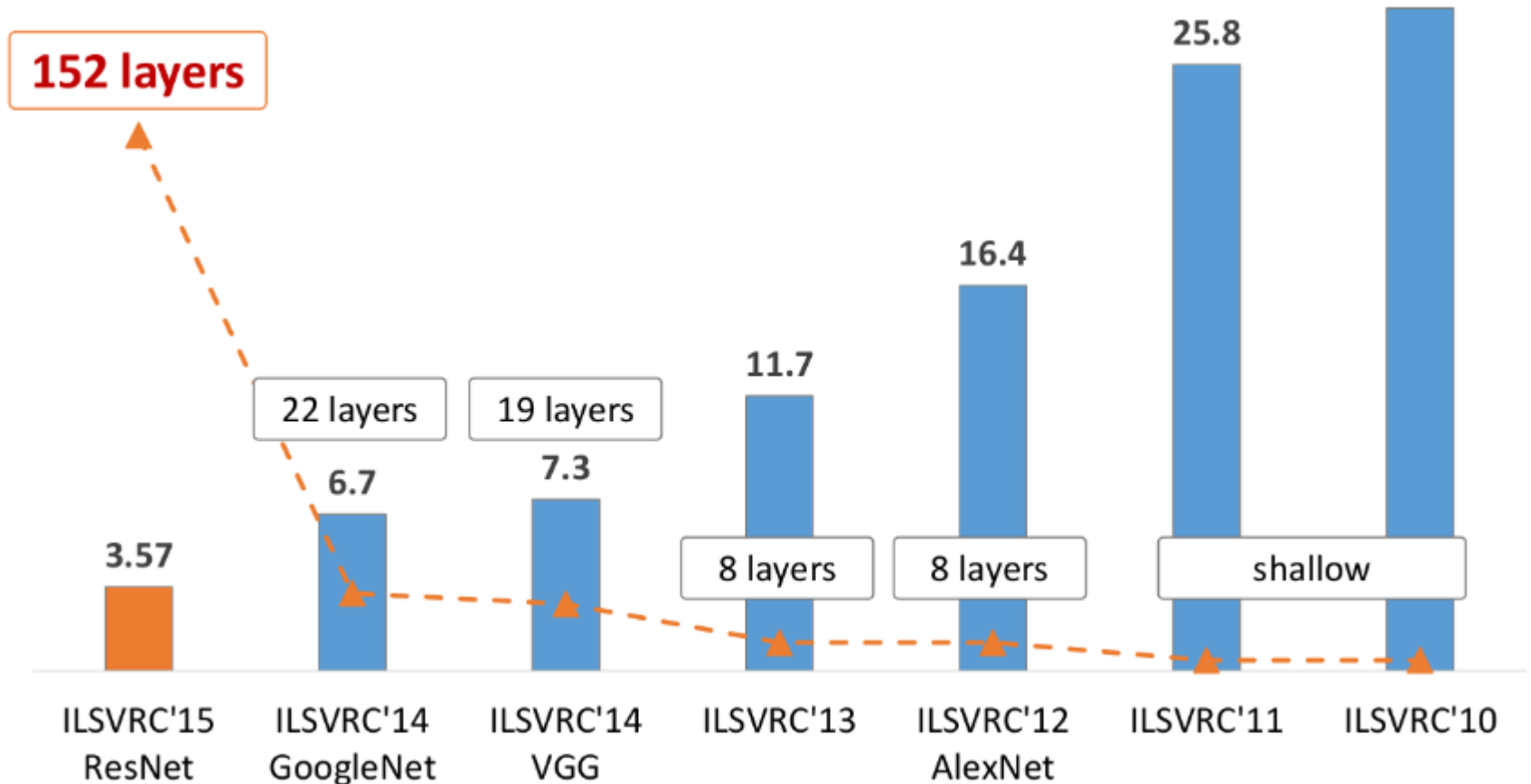


A 152-layer network achieves  
3.57% error rate

method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
<b>ResNet (ILSVRC'15)</b>	<b>3.57</b>

1000-layer model has also been  
tested on CIFAR-10

# The deeper, the better



He, 2015

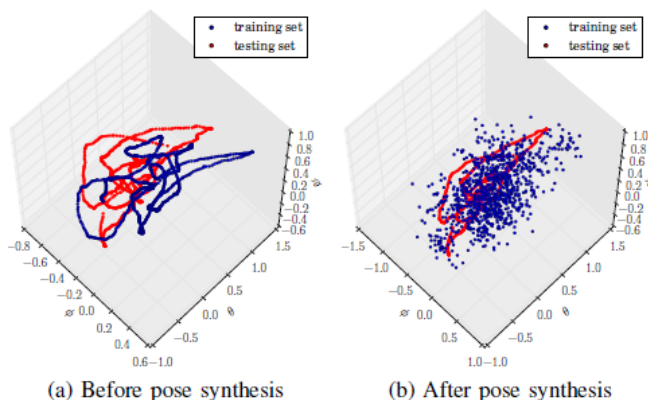
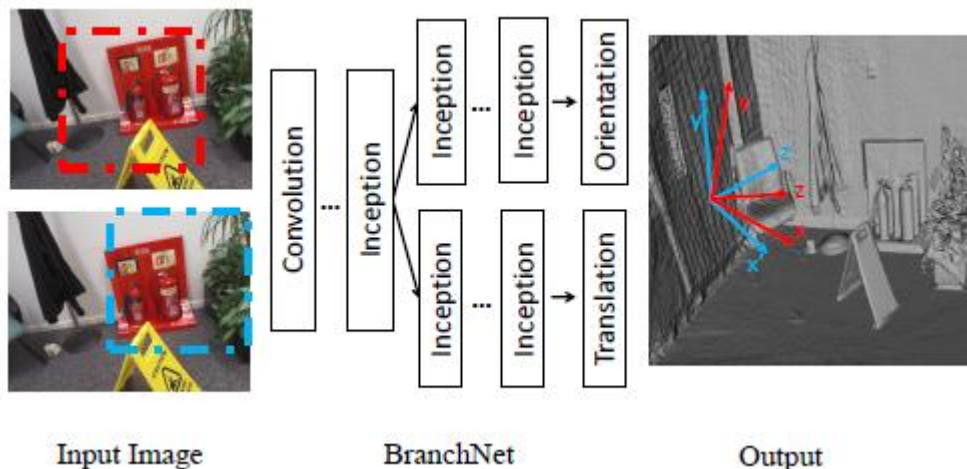
# CNN for regression: camera relocalization

Infer the 6 DOF  
(translation and  
orientation) of the  
camera from an image

$$\text{loss}(I) = \|t - \hat{t}\|^2 + \beta \|e - \hat{e}\|^2$$

where  $t$  is the translation and

$$e = [\sin \phi, \cos \phi, \sin \theta, \cos \theta, \sin \psi, \cos \psi]$$



Scene	BranchNet-Euler6-Aug
Chess	5.17°, 0.18m
Fire	8.99°, 0.34m
Heads	14.15°, 0.20m
Office	7.05°, 0.30m
Pumpkin	5.10°, 0.27m
RedKitchen	7.40°, 0.33m
Stairs	10.26°, 0.38m
Average	8.30°, 0.29m

Wu, Ma, Hu, ICRA'17

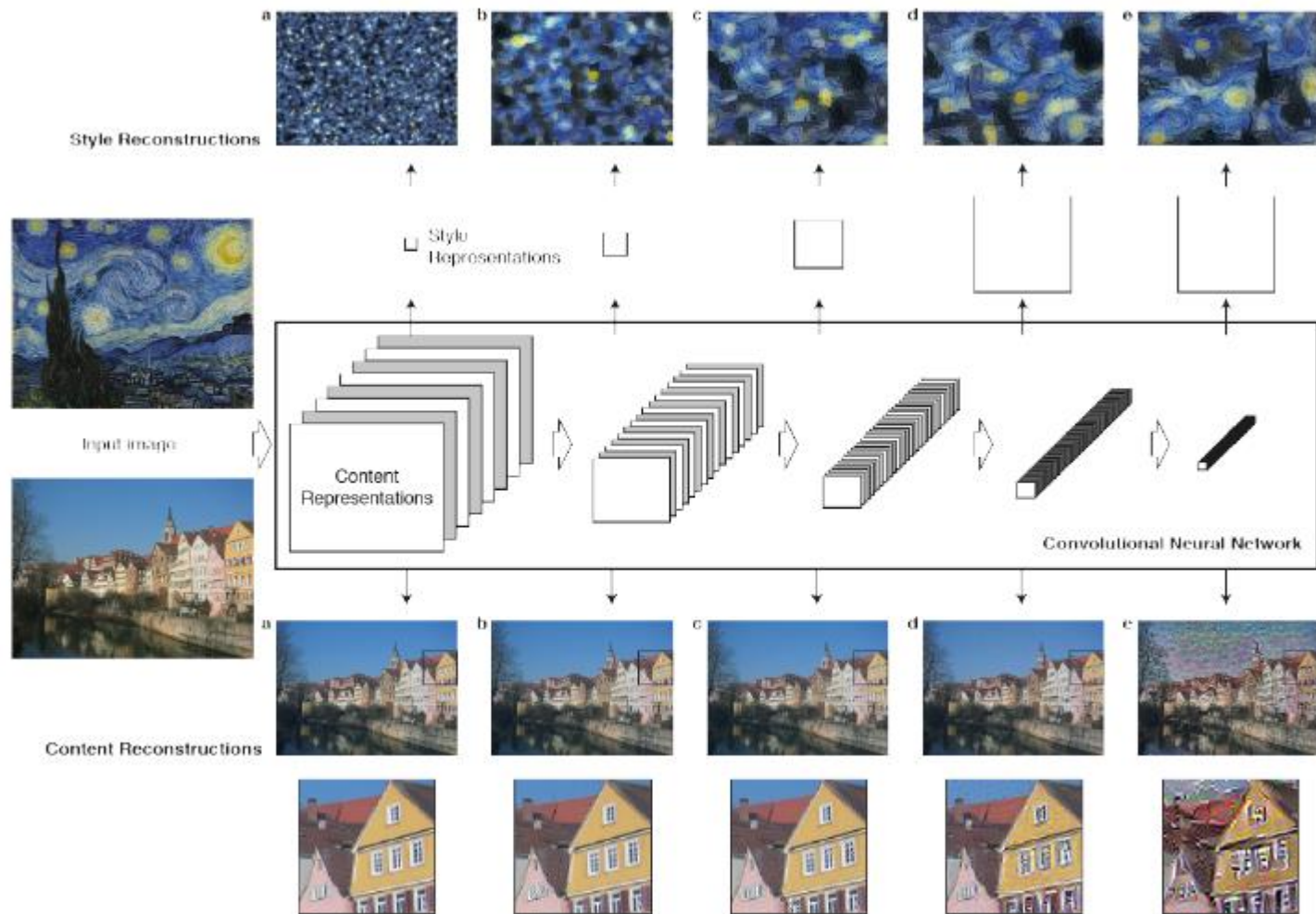
# A Neural Algorithm of Artistic Style



Gatys et al., 2015



# A Neural Algorithm of Artistic Style





# A Neural Algorithm of Artistic Style

- Content loss

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2$$

layer index
photo  
↓
↓

- Style loss

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

painting  
↓

where  $G$  and  $A$  are gram matrices, e.g.,

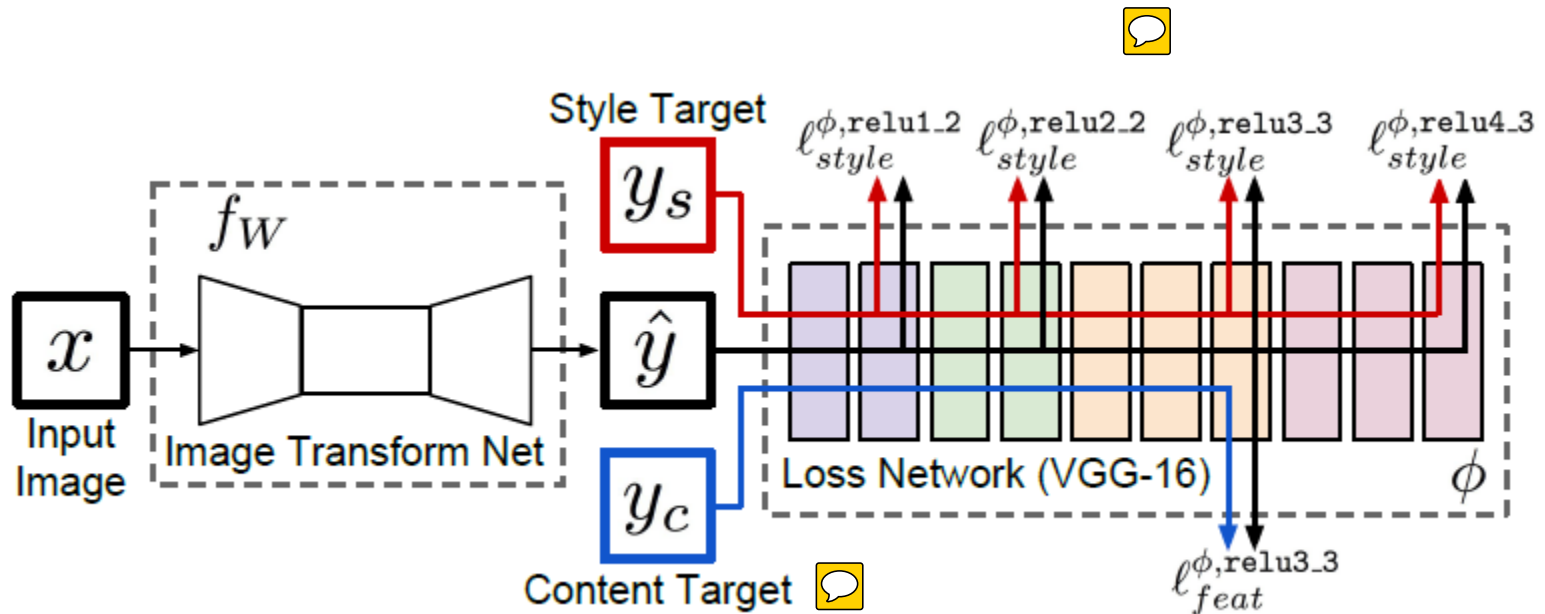
$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l$$

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

- Optimize the total loss w.r.t. pixels  $x$  (not  $w$ )

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

# Feedforward generation



$$\ell^{\phi, j}_{feat}(\hat{y}, y) = \frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y)\|_2^2$$

$$\ell^{\phi, j}_{style}(\hat{y}, y) = \|G_j^{\phi}(\hat{y}) - G_j^{\phi}(y)\|_F^2 \quad \text{where } G \text{ is the gram matrix}$$

Johnson, Alahi, Fei-Fei, ECCV 2016

# Results

Style  
*The Starry Night*,  
Vincent van Gogh,  
1889



Style  
*The Muse*,  
Pablo Picasso,  
1935

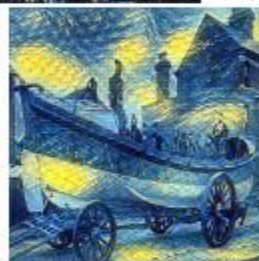
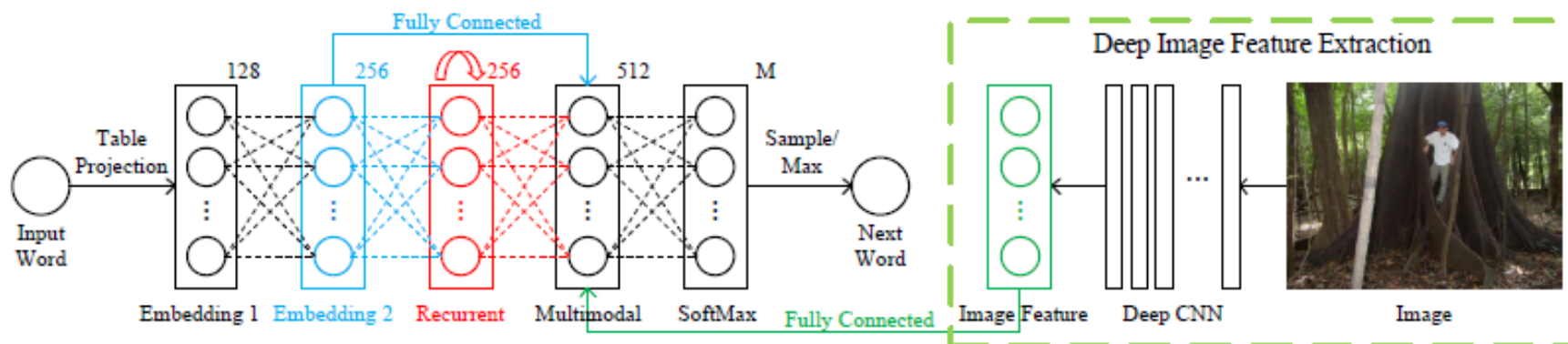


Image Size	Gatys <i>et al.</i> [11]			Ours	Speedup		
	100	300	500		100	300	500
$256 \times 256$	3.17	9.52s	15.86s	<b>0.015s</b>	212x	636x	1060x
$512 \times 512$	10.97	32.91s	54.85s	<b>0.05s</b>	205x	615x	1026x
$1024 \times 1024$	42.89	128.66s	214.44s	<b>0.21s</b>	208x	625x	1042x



# CNN + Recurrent neural network



Gen.

A square with burning street lamps and a street in the foreground;

Tourists are sitting at a long table with a white table cloth and are eating;

A dry landscape with green trees and bushes and light brown grass in the foreground and reddish-brown round rock domes and a blue sky in the background;

A blue sky in the background;

Mao et al., 2014

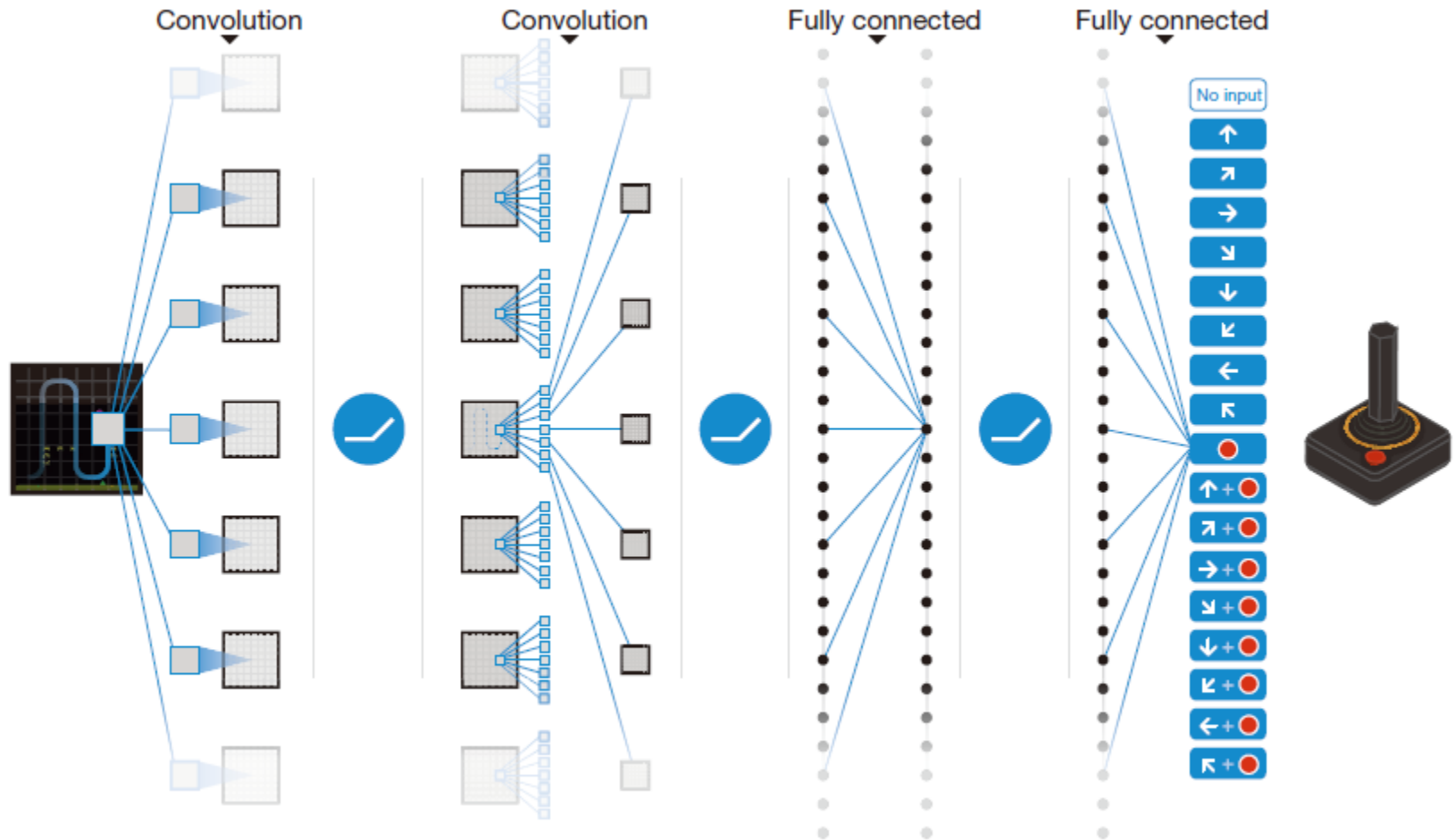
# CNN + reinforcement learning



- Atari 2600 platform offers 49 games
- Google's deep Q-network (DQN) performs the same as or better than the human expert in 29 games
- The same network, same learning algorithms



# DQN




Mnih et al., 2015

# Outline

- Regression and classification
- Multi-layer perceptron
- Convolutional neural network
- Applications
- Practical tricks

# Weight initialization

$W$  inputting to a neuron is drawn from a distribution:


- Gaussian 
  - a Gaussian distribution with zero mean and fixed std, e.g., 0.01
- Xavier
  - a distribution with zero mean and a specific std  $1/\sqrt{n_{\text{in}}}$  where  $n_{\text{in}}$  is the number of neurons feeding into the neuron
  - Gaussian distribution or uniform distribution is often used
- MSRA
  - a Gaussian distribution with zero mean and a specific std  $2/\sqrt{n_{\text{in}}}$



# Learning rate

- In SGD the learning rate  $\alpha$  is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update.
- Choosing the proper schedule
  - One standard method is to use a small enough **constant learning rate** that gives stable convergence in the initial epoch (full pass through the training set) or two of training and then halve the value of the learning rate as convergence slows down.
  - An even better approach is to 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.
  - Another commonly used schedule is to anneal the learning rate at each iteration  $t$  as  $\frac{a}{b+t}$  where  $a$  and  $b$  dictate the initial learning rate and when the annealing begins respectively.

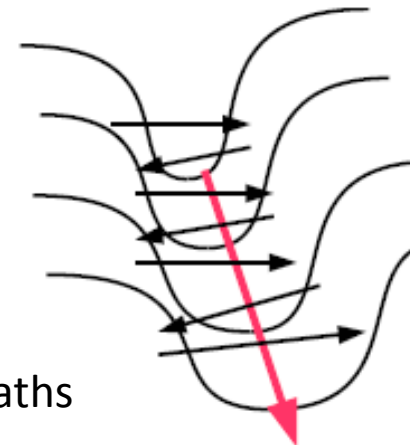
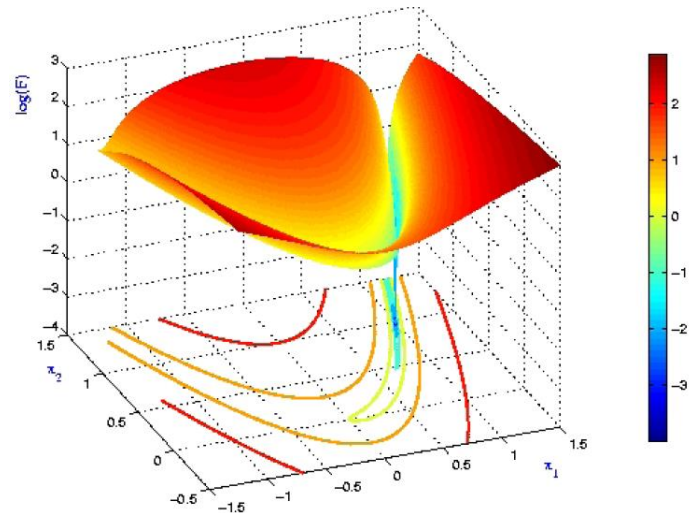
# Order of training samples

- 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

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$



Black arrows: gradient descent paths

# Momentum

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

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

- $v$  is the current velocity vector
- $\gamma \in (0,1]$  determines for how many iterations the previous gradients are incorporated into the current update.
- One strategy:  $\gamma$  is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher

# Problem 2: tensorflow experiments

- MLP for classifying MNIST handwritten digits
  - ① Construct an MLP with one hidden layer of 500 units using sigmoid activation function and crossentropy loss
  - ② Redo ① using ReLU activation function and compare the performance
- CNN for classifying MNIST handwritten digits
  - ① Image -> conv (32 5x5 filters) -> ReLU -> pooling (2x2 max) -> conv (64 5x5 filters) -> ReLU -> pooling (2x2 max) -> fully connected (1024 hidden units) -> ReLU -> fully connected (10 hidden units) -> softmax
- A code framework is provided

# Summary

- Regression and classification
  - Linear regression
  - Logistic regression
  - Softmax regression
  - SGD
- Multi-layer perceptron
  - FC layer, sigmoid layer, ReLU layer, loss layer
  - Backpropagation
- Convolutional neural network
  - Convolutional layer and pooling layer
- Applications
- Practical tricks
  - Weight initialization, learning rate, order of training samples, momentum