Course number: 80240743

Deep Learning

Xiaolin Hu (胡晓林) & Jun Zhu (朱军)
Dept. of Computer Science and Technology
Tsinghua University

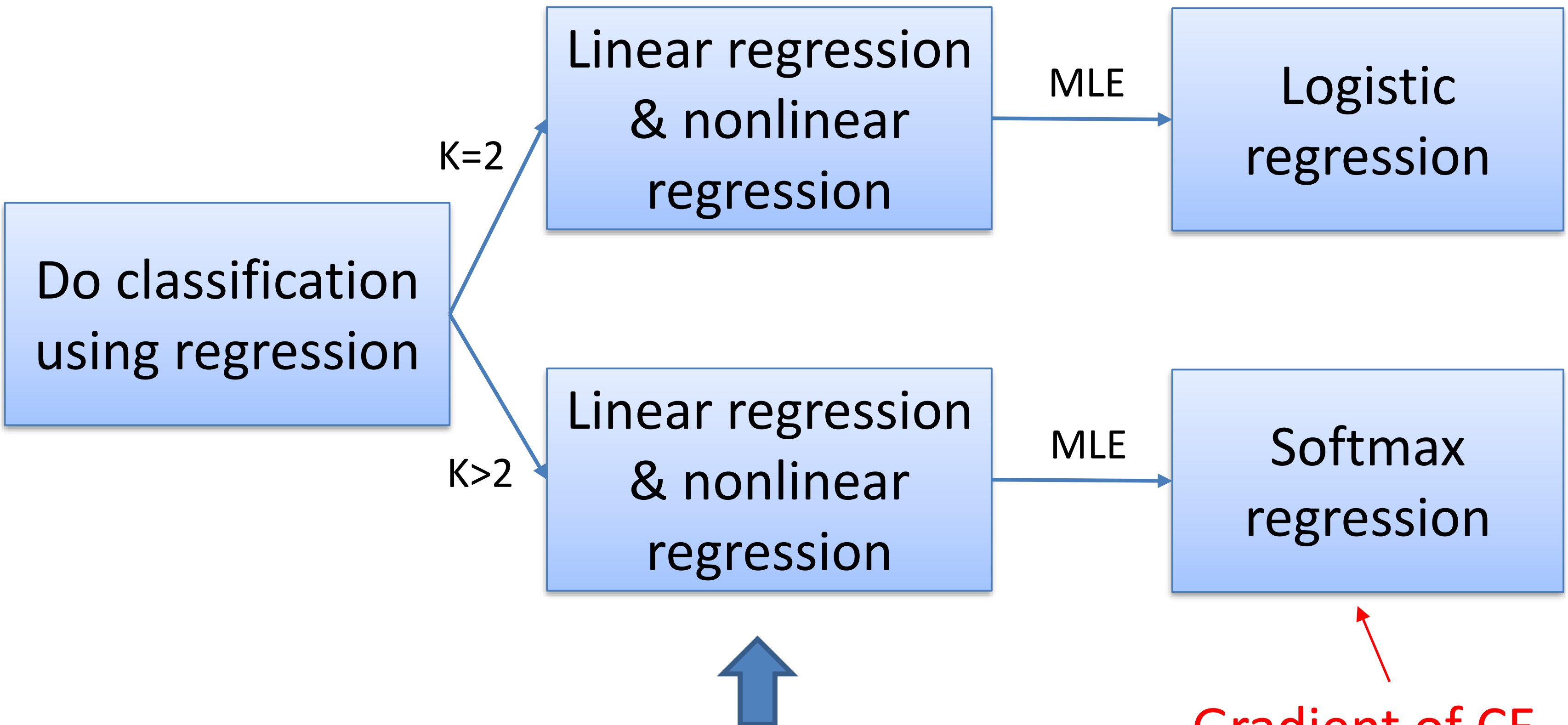
Lecture 3: Multi-layer Perceptron

Xiaolin Hu
Dept. of Computer Science and Technology
Tsinghua University

Outline

1. Regression and classification (cont’d)
2. Multi-layer perceptron
   – Feedforward calculation
   – Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

Recap



If the task is regression, then we stop here

Gradient of CE loss?

Cross-entropy error function

$$h_i^{(n)} = P(\mathrm{t}_i^{(n)} = 1 | \boldsymbol{x}^{(n)})$$

$$\begin{aligned}
E(\boldsymbol{\theta}) &= -\frac{1}{N} \ln P(\boldsymbol{t}^{(1)}, \ldots, \boldsymbol{t}^{(N)}) \\
&= -\frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{K} t_i^{(n)} \ln \frac{\exp(\boldsymbol{\theta}^{(i)\top} \boldsymbol{x}^{(n)})}{\sum_{j=1}^{K} \exp(\boldsymbol{\theta}^{(j)\top} \boldsymbol{x}^{(n)})} \\
&= -\frac{1}{N} \sum_{n=1}^{N} \boxed{t_q^{(n)} \ln h_q^{(n)}} \\
&= -\frac{1}{N} \sum_{n=1}^{N} \ln P(\mathrm{t}_q^{(n)} = 1 | \boldsymbol{x}^{(n)})
\end{aligned}$$

Suppose $\boldsymbol{x}^{(n)}$ belongs to the q-th class

- This function is called cross-entropy error function
- Also called CE loss function

"Cross-entropy" in general*

- The cross-entropy for two distributions p and q over a given set is defined as follows:

$$H(p,q) = E_p[-\log q] = \overset{\geq 0}{H(p)} + \overset{\geq 0}{D_{\mathrm{KL}}(p||q)}$$

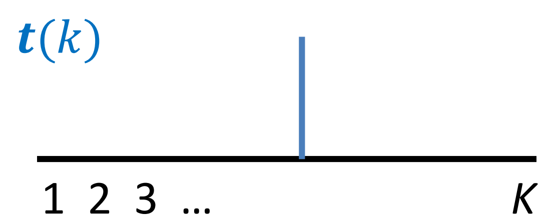
where $H(p)$ is the entropy of p and $D_{\mathrm{KL}}(p|q)$ is the Kullback–Leibler divergence of q from p

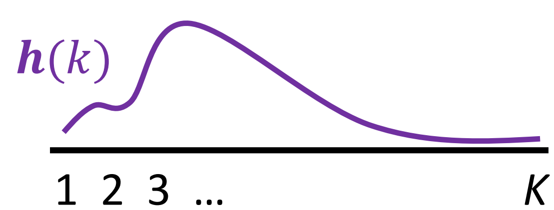
  - If p is fixed, then min CE is equivalent to min $D_{\mathrm{KL}}(p||q)$
  - For discrete p and q this means

$$H(p,q) = -\sum_x p(x) \log q(x)$$

  - For single label classification p is a one-hot vector $\boldsymbol{t}$

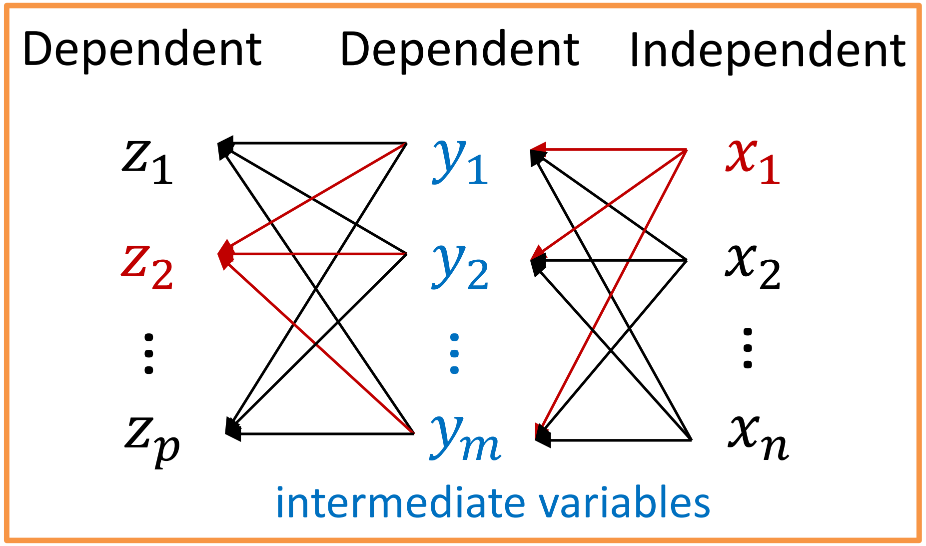
$$E^{(n)}(\boldsymbol{\theta}) = -\sum_{i=1}^{K} t_i^{(n)} \ln h_i^{(n)}$$





Recap: Derivative of two-step composition

- Independent variables $x_1, x_2, \dots, x_n$
- Each y_i is a function of $x_1, x_2, \dots, x_n$
- Each z_i is a function of $y_1, y_2, \dots, y_m$



What's partial derivative of z_i w.r.t. x_j?

$$\frac{\partial z_i}{\partial x_j} = \sum_{k=1}^{m} \frac{\partial z_i}{\partial y_k} \frac{\partial y_k}{\partial x_j}$$

Sum over the intermediate variables

for any $i \in \{1,2,\dots,p\}$ and $j \in \{1,2,\dots,n\}$

Calculate the gradient

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

where $h_i^{(n)} = P(\mathrm{t}_i^{(n)} = 1 | \boldsymbol{x}^{(n)}) = \dfrac{\exp(u_i^{(n)})}{\sum_{j=1}^{K}\exp(u_j^{(n)})}, \quad u_k^{(n)} = \boldsymbol{\theta}^{(k)\top}\boldsymbol{x}^{(n)}$

$$\frac{\partial E^{(n)}}{\partial \boldsymbol{\theta}^{(k)}} = \frac{\partial E^{(n)}}{\partial u_k^{(n)}}\frac{\partial u_k^{(n)}}{\partial \boldsymbol{\theta}^{(k)}} = \sum_{i=1}^{K} \frac{\partial E^{(n)}}{\partial h_i^{(n)}}\frac{\partial h_i^{(n)}}{\partial u_k^{(n)}}\frac{\partial u_k^{(n)}}{\partial \boldsymbol{\theta}^{(k)}}$$

Local sensitivity or local gradient

$$\frac{\partial E^{(n)}}{\partial h_i^{(n)}} = -t_i^{(n)}\frac{1}{h_i^{(n)}}$$

?

$$\frac{\partial u_k^{(n)}}{\partial \boldsymbol{\theta}^{(k)}} = \boldsymbol{x}^{(n)}$$

Calculate the gradient

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

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

If $k \neq i$, u_k appears only in the denominator

$$\frac{\partial h_i^{(n)}}{\partial u_k^{(n)}} =$$

If $k = i$, u_k appears in both numerator and denominator

$$\frac{\partial h_i^{(n)}}{\partial u_k^{(n)}} =$$

Calculate the gradient

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

where $h_i^{(n)} = P(\mathrm{t}_i^{(n)} = 1 | \boldsymbol{x}^{(n)}) = \dfrac{\exp(u_i^{(n)})}{\sum_{j=1}^{K} \exp(u_j^{(n)})}, \quad u_k^{(n)} = \boldsymbol{\theta}^{(k)\top}\boldsymbol{x}^{(n)}$

If $k \neq i$, u_k appears only in the denominator

$$\frac{\partial h_i^{(n)}}{\partial u_k^{(n)}} = -\frac{\exp(u_i^{(n)})\exp(u_k^{(n)})}{\left(\sum_j \exp(u_j^{(n)})\right)^2} = -h_k^{(n)} h_i^{(n)}$$

If $k = i$, u_k appears in both numerator and denominator

$$\frac{\partial h_i^{(n)}}{\partial u_k^{(n)}} = \frac{\exp(u_k^{(n)})}{\sum_j \exp(u_j^{(n)})} - \frac{\left(\exp(u_k^{(n)})\right)^2}{\left(\sum_j \exp(u_j^{(n)})\right)^2} = h_k^{(n)}(1 - h_k^{(n)})$$

Therefore $\dfrac{\partial h_i^{(n)}}{\partial u_k^{(n)}} = h_i^{(n)}(\Delta_{i,k} - h_k^{(n)})$ where $\Delta_{i,k} = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{else.} \end{cases}$

Calculate the gradient

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

where $h_i^{(n)} = P(\mathrm{t}_i^{(n)} = 1|\boldsymbol{x}^{(n)}) = \dfrac{\exp(u_i^{(n)})}{\sum_{j=1}^{K}\exp(u_j^{(n)})}, \quad u_k^{(n)} = \boldsymbol{\theta}^{(k)\top}\boldsymbol{x}^{(n)}$

$$\frac{\partial E^{(n)}}{\partial \boldsymbol{\theta}^{(k)}} = \sum_{i=1}^{K} \frac{\partial E^{(n)}}{\partial h_i^{(n)}} \frac{\partial h_i^{(n)}}{\partial u_k^{(n)}} \frac{\partial u_k^{(n)}}{\partial \boldsymbol{\theta}^{(k)}}$$

$$= \sum_{i=1}^{K} \left(-t_i^{(n)} \frac{1}{h_i^{(n)}}\right) \left(h_i^{(n)}(\Delta_{i,k} - h_k^{(n)})\right) \left(\boldsymbol{x}^{(n)}\right)$$

$$= -\left(\sum_{i=1}^{K} t_i^{(n)}\Delta_{i,k} - \underbrace{\sum_{i=1}^{K} t_i^{(n)}}_{=1} h_k^{(n)}\right)\boldsymbol{x}^{(n)}$$

$$= -\left(t_k^{(n)} - h_k^{(n)}\right)\boldsymbol{x}^{(n)}$$

Calculate the gradient

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

where $h_i^{(n)} = P(\mathrm{t}_i^{(n)} = 1|\boldsymbol{x}^{(n)}) = \dfrac{\exp(u_i^{(n)})}{\sum_{j=1}^{K}\exp(u_j^{(n)})}, \quad u_k^{(n)} = \boldsymbol{\theta}^{(k)\top}\boldsymbol{x}^{(n)}$

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

is the local gradient or local sensitivity.

Average over samples

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

$$= -\frac{1}{N}\sum_{n=1}^{N}\left(t_k^{(n)} - P(t_k^{(n)} = 1|\boldsymbol{x}^{(n)})\right)\boldsymbol{x}^{(n)}$$

Vector-matrix form

- Note the definitions

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_{11} & \cdots & \theta_{1m} \\ \vdots & \vdots & \vdots \\ \theta_{K1} & \cdots & \theta_{Km} \end{pmatrix} \quad \frac{\partial E}{\partial \boldsymbol{\theta}} = \begin{pmatrix} \partial E/\partial\theta_{11} & \cdots & \partial E/\partial\theta_{1m} \\ \vdots & \vdots & \vdots \\ \partial E/\partial\theta_{K1} & \cdots & \partial E/\partial\theta_{Km} \end{pmatrix}$$

m: the number of inputs; K: the number of outputs

- Output: $\boldsymbol{f}(\boldsymbol{x})$ is the softmax function where $\boldsymbol{f}, \boldsymbol{b} \in R^K$, $\boldsymbol{x} \in R^m$
- The gradient of the cross-entropy error function

$$\nabla_{\boldsymbol{\theta}} E = \begin{pmatrix} (\partial E/\partial \boldsymbol{\theta}^{(1)})^\top \\ \vdots \\ (\partial E/\partial \boldsymbol{\theta}^{(K)})^\top \end{pmatrix} = \frac{1}{N} \sum_{n=1}^{N} \left(\boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)} \right) \left(\boldsymbol{x}^{(n)}\right)^\top \quad \in R^{K \times m}$$

Summary
MSE and CE

- Nonlinear regression (linear regression as a special case)
  - Output: $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{\theta}^\top \boldsymbol{x})$, where $\boldsymbol{h}$ could be any act function
  - MSE: $E = \frac{1}{N}\sum_{n=1}^{N} E^{(n)}, E^{(n)} = \frac{1}{2}\left\|\boldsymbol{h}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)}\right\|_2^2$
  - Gradient: $\nabla_{\boldsymbol{\theta}} E = \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)}\right) \odot \boldsymbol{f}'\left(\boldsymbol{x}^{(n)}\right)\left(\boldsymbol{x}^{(n)}\right)^\top$
- Softmax regression (logistic regression as a special case)
  - Output: $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{\theta}\boldsymbol{x})$, where $\boldsymbol{h}$ is the softmax function
  - Cross-entropy error: $E = \frac{1}{N}\sum_{n=1}^{N} E^{(n)}, E^{(n)} = -\left(\boldsymbol{t}^{(n)}\right)^\top \ln \boldsymbol{h}^{(n)}$
  - Gradient: $\nabla_{\boldsymbol{\theta}} E = \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)}\right)\left(\boldsymbol{x}^{(n)}\right)^\top$

Training and testing

- Calculate the gradient of the cross-entropy error function

$$\nabla_{\boldsymbol{\theta}} E = \frac{1}{N} \sum_{n=1}^{N} \left(\boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)} \right) \odot \left(\boldsymbol{x}^{(n)}\right)^{\mathsf{T}}$$

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

$$J(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \lambda \left|\left|\boldsymbol{\theta}\right|\right|^2 / 2$$

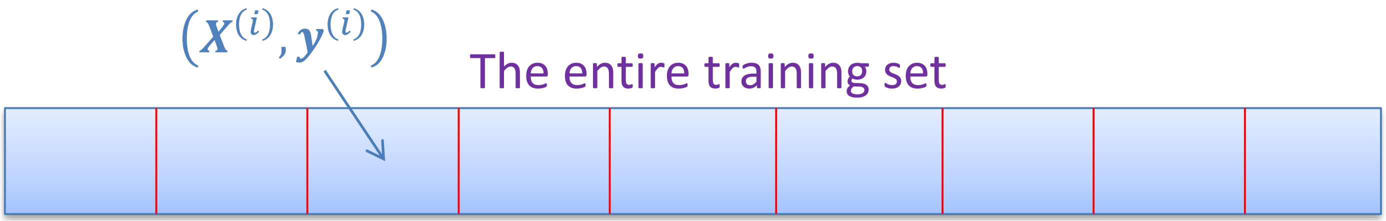
- Training: minimize the cost function with gradient $\nabla J(\boldsymbol{\theta})$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla J(\boldsymbol{\theta})$$

  where α is the learning rate

- Testing: find the maximum $P(\mathrm{t}_k = 1 | \boldsymbol{x})$ among k for a new input $\boldsymbol{x}$

Recall: Stochastic gradient decent



- Minimizing the cost function over the entire training set is computationally expensive
- We often decompose the training set into smaller subsets or minibatches and optimize the cost function defined over individual minibatches $(\boldsymbol{X}^{(i)}, \boldsymbol{y}^{(i)})$ and take the average

$$J(\boldsymbol{\theta}) = \frac{1}{N'} \sum_{i=1}^{N'} L(\boldsymbol{X}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})$$

$$\boldsymbol{g} = \frac{1}{N'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{N'} L(\boldsymbol{X}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{g}$$

- A total of N' minibatches
- The batchsize ranges from 1 to a few hundreds

Introducing bias

- So far we have assumed

$$h_k(\boldsymbol{x}) = P(\mathrm{t}_k = 1|\boldsymbol{x}) = \frac{\exp(u_k^{(n)})}{\sum_{j=1}^{K} \exp(u_j^{(n)})} \qquad u_k^{(n)} = \boldsymbol{\theta}^{(k)\top}\boldsymbol{x}^{(n)}$$

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

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

- It's easy to show that

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

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

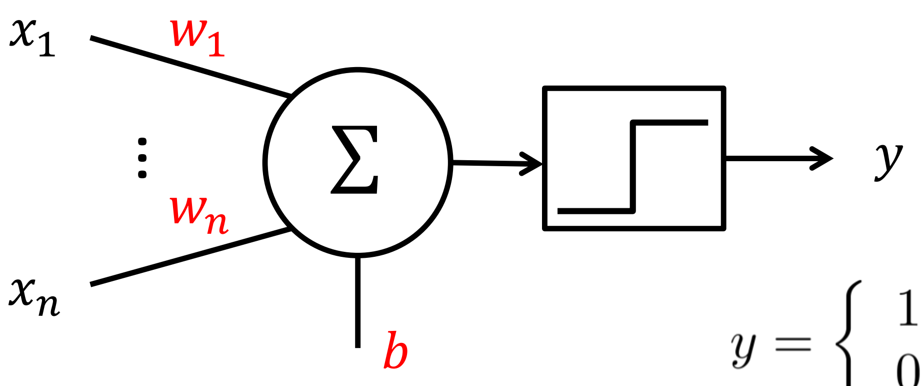
- Regularization is often applied on $\boldsymbol{W}$ only

$$J(\boldsymbol{W}, \boldsymbol{b}) = E(\boldsymbol{W}, \boldsymbol{b}) + \lambda\left|\left|\boldsymbol{W}\right|\right|^2/2$$

Outline

1. Regression and classification (cont’d)
2. Multi-layer perceptron
   – Feedforward calculation
   – Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

Recap: Perceptron



$$y = \begin{cases} 1 & \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ 0 & \text{else} \end{cases}$$

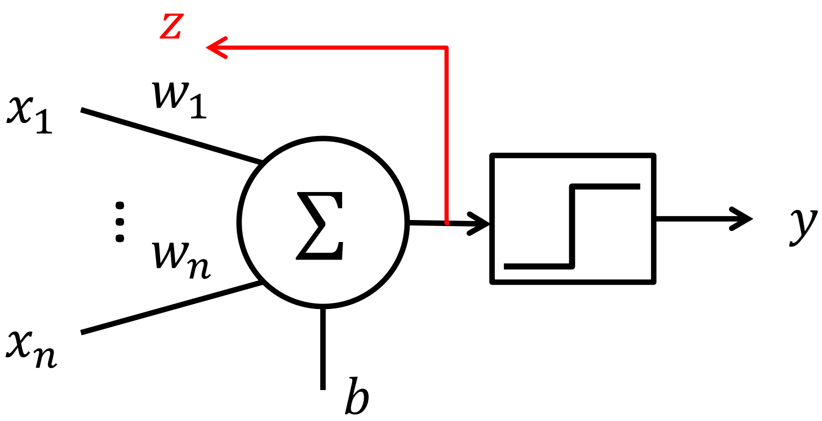
- For each data points $\boldsymbol{x}^{(j)} \in R^m$ and the corresponding labels $t^{(j)}$
  - Calculate the actual output $y^{(j)}$
  - Update the weights: $\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} + \eta\left(t^{(j)} - y^{(j)}\right)\boldsymbol{x}^{(j)}$;

$$b^{\text{new}} = b^{\text{old}} + \eta\left(t^{(j)} - y^{(j)}\right)$$

where $\eta > 0$ is the learning rate

- The decision boundary is a hyperplane: $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$

Recap: ADALINE



$$y = \begin{cases} 1 & \boldsymbol{w}^\top \boldsymbol{x} + b > 0.5 \\ 0 & \text{else} \end{cases}$$

Or

$$y = \begin{cases} 1 & \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ -1 & \text{else} \end{cases}$$

- Same architecture as Perceptron; different training algorithm
  - $z = \boldsymbol{w}^\top \boldsymbol{x} + b$ instead of y is used to adjust the weights and bias
- Minimize MSE $E = \frac{1}{N}\sum_j \left(t^{(j)} - z^{(j)}\right)^2$. The learning algorithm:

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} + \eta\left(t^{(j)} - z^{(j)}\right)\boldsymbol{x}^{(j)}$$
$$b^{\text{new}} = b^{\text{old}} + \eta\left(t^{(j)} - z^{(j)}\right)$$

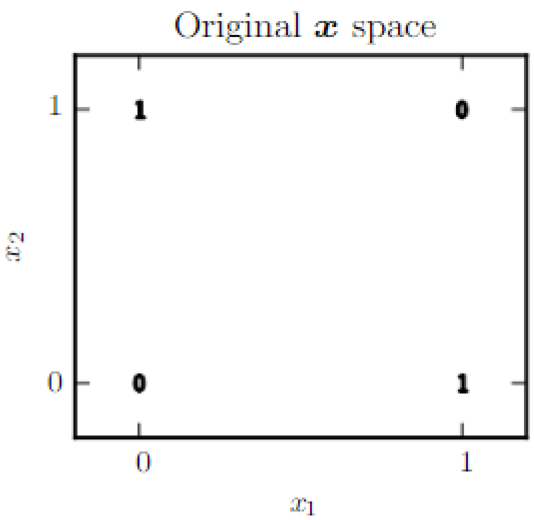
where $\eta > 0$ is the learning rate

- The decision boundary is a hyperplane: $\boldsymbol{w}^\top \boldsymbol{x} + b = 0.5$ or $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$

Solve XOR problem using ADALINE

- Boolean function:

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

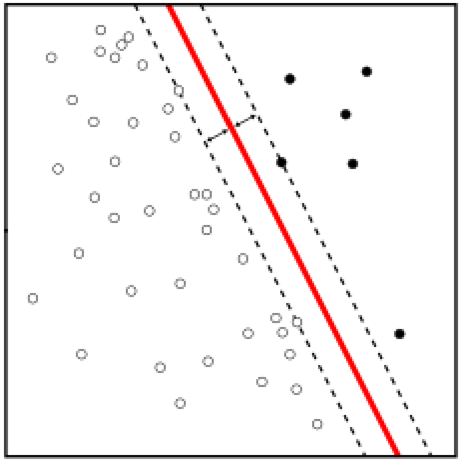


- Error function $E = \frac{1}{4}\sum_{j=1}^{4}\left(t^{(j)} - z^{(j)}\right)^2$ where $z^{(j)} = \boldsymbol{w}^\top \boldsymbol{x}^{(j)} + b$
- Let $\nabla_{\boldsymbol{w}} E = 0, \nabla_b E = 0$, then

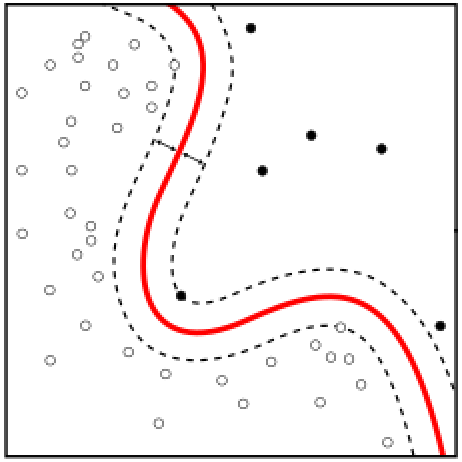
$$2w_1 + 2w_2 + 4b = 2$$
$$2w_1 + w_2 + 2b = 1 \quad \Rightarrow \quad \boldsymbol{w}^* = ?, b^* = ?$$
$$w_1 + 2w_2 + 2b = 1$$

Limitation

- Both Perceptron and ADALINE can only solve linearly separable classification problems



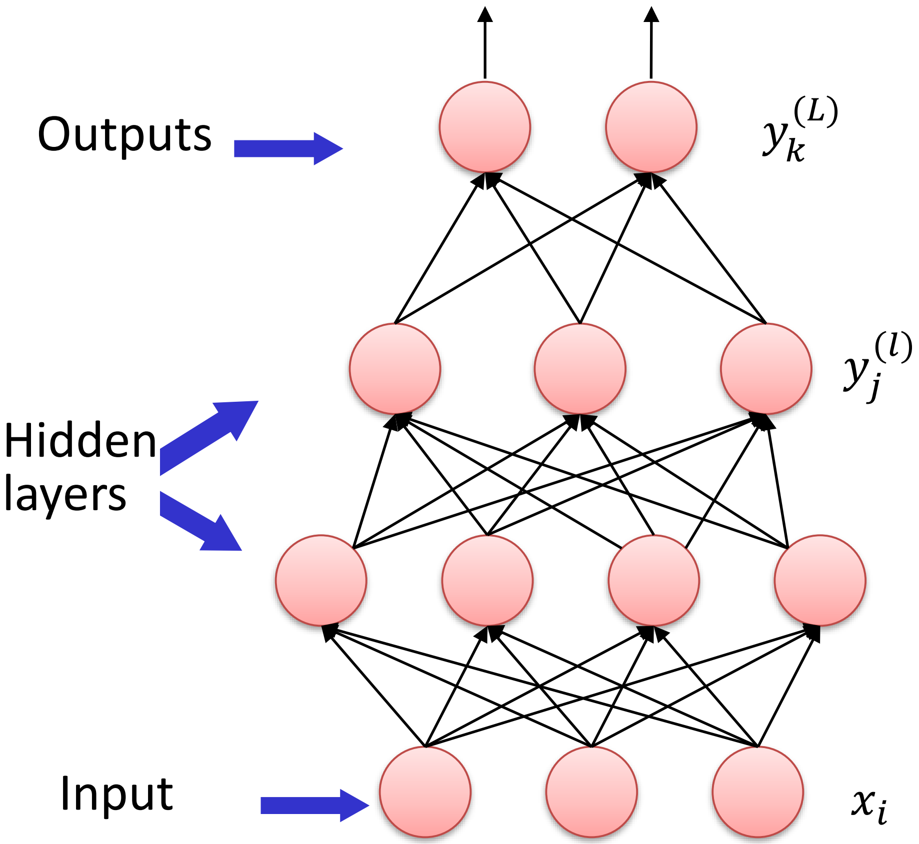
linearly separable



linearly non-separable

- This result discouraged the NN research in 1960s-1970s (1^{st} winter)
- If the problem is linearly non-separable, what should we do?

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 sigmoid function

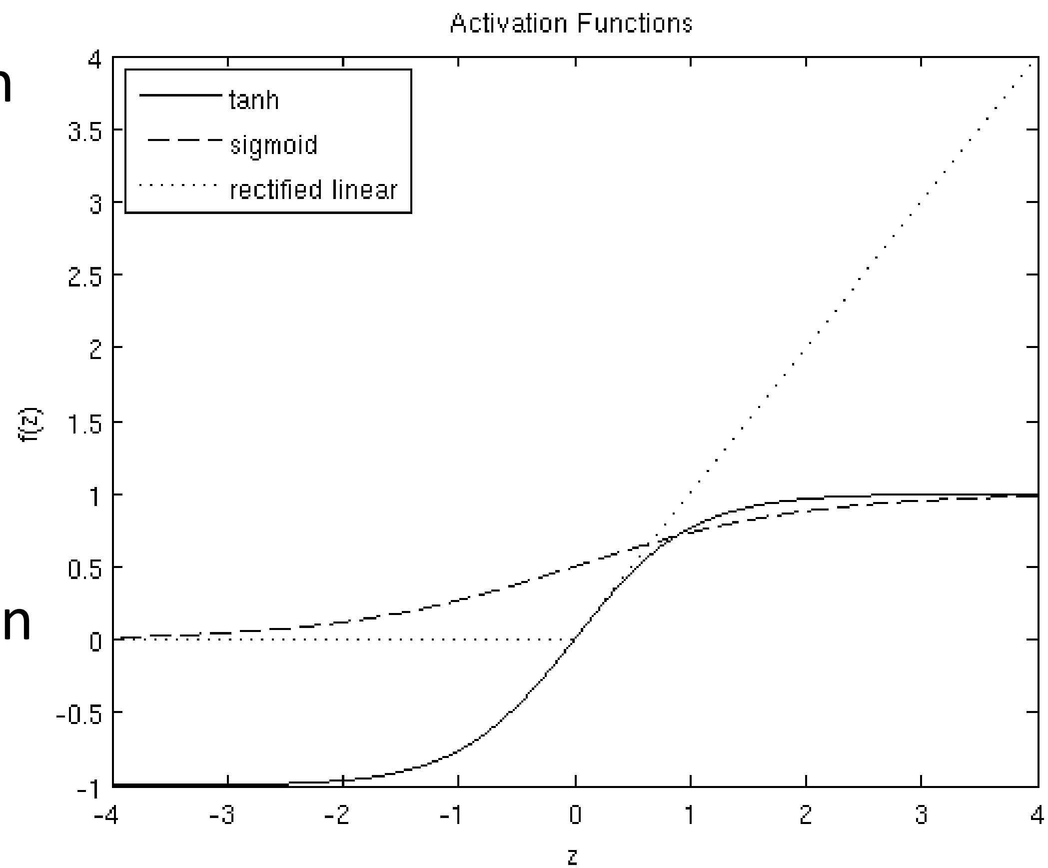
$$f(z) = \frac{1}{1 + \exp(-z)}$$

- 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

$$f(z) = \frac{1}{1 + \exp(-z)} \xrightarrow{\text{gradient}} f'(z) = f(z)(1 - f(z))$$

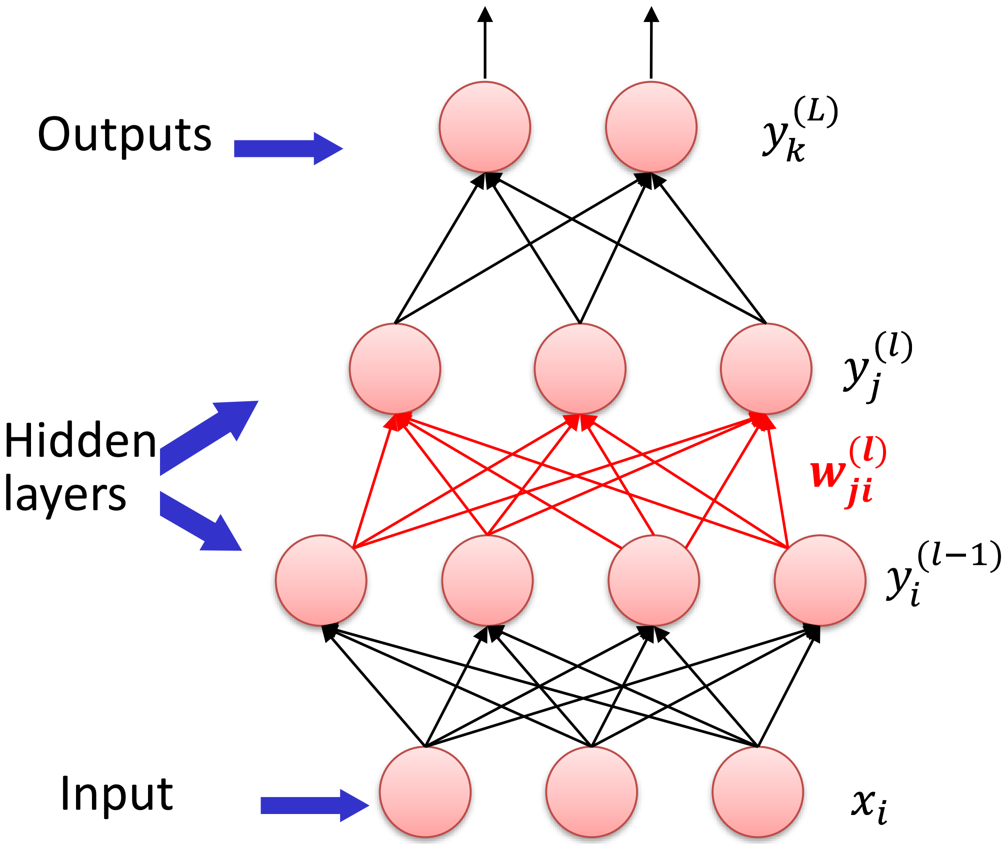
- Hyperbolic tangent function

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \xrightarrow{\text{gradient}} f'(z) = 1 - f(z)^2$$

- Rectified linear activation function (ReLU)

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

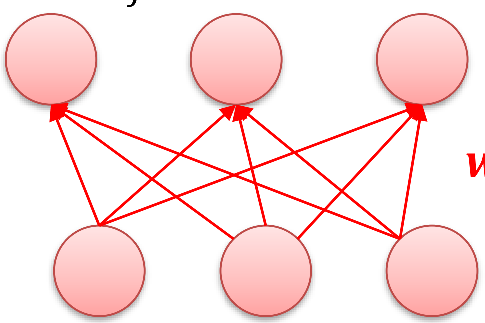
Forward pass



For clarity we don't separate the linear transformation and activation function

- 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
  $$y_j^{(l)} = f\left(u_j^{(l)}\right)$$
  where $f(\cdot)$ is activation function
  - Note $y^{(0)} = x$
- $l = L$ corresponds to the classification layer

Forward pass in the vector-matrix form



- If the previous layer has N neurons and the current layer has M neurons, define the weight matrix and bias vector as

$$\boldsymbol{W}^{(l)} = \begin{pmatrix} w_{11} & \cdots & w_{1N} \\ \vdots & \vdots & \vdots \\ w_{M1} & \cdots & w_{MN} \end{pmatrix} \boldsymbol{b}^{(l)} = \begin{pmatrix} b_1 \\ \vdots \\ b_M \end{pmatrix}$$

- Then for $l = 1, \dots, L-1$

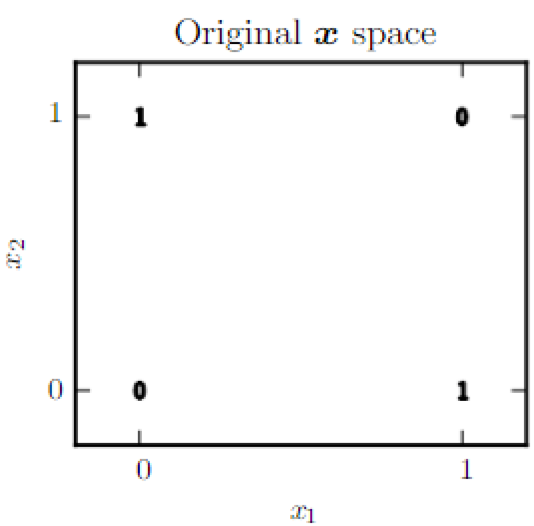
$$\boldsymbol{u}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)} \text{ and } \boldsymbol{y}^{(l)} = \boldsymbol{f}(\boldsymbol{u}^{(l)})$$

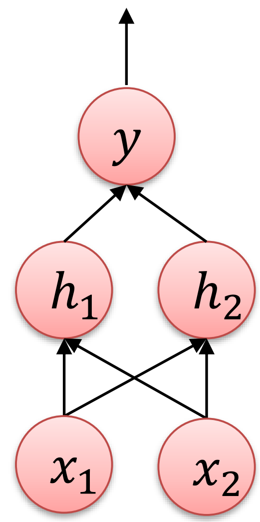
where $\boldsymbol{W}^{(l-1)} \in R^{M \times N}, \boldsymbol{b}^{(l)} \in R^M, u^{(l)}, \boldsymbol{y}^{(l)} \in R^M, \boldsymbol{y}^{(l-1)} \in R^N$

XOR problem revisited

- Boolean function:

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0





$$y = \mathrm{ReLU}(\boldsymbol{w}^\top \boldsymbol{h} + b)$$

$$\boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0$$

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})$$

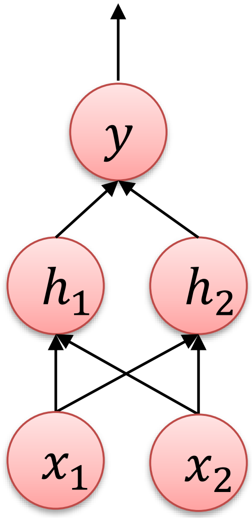
$$\boldsymbol{V} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

What are $\boldsymbol{h}^{(n)}$?

XOR problem revisited

- Boolean function:

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

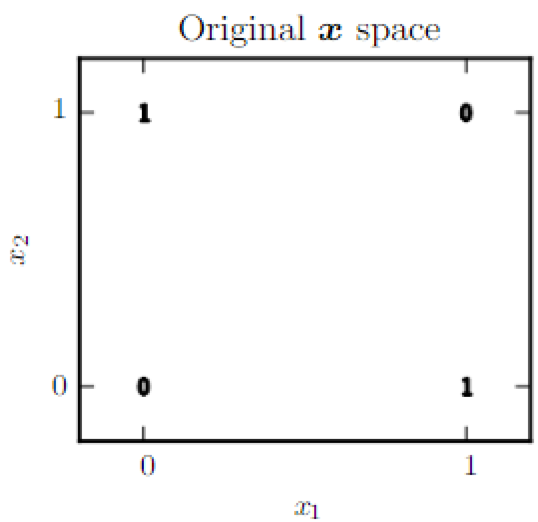


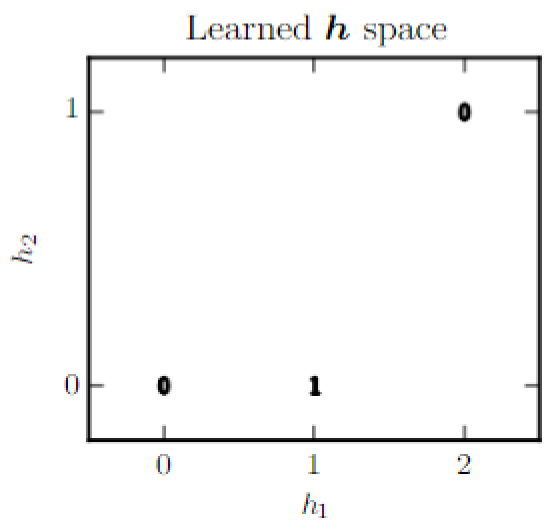
$$y = \mathrm{ReLU}(\boldsymbol{w}^\top \boldsymbol{h} + b)$$

$$\boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0$$

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})$$

$$\boldsymbol{V} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$





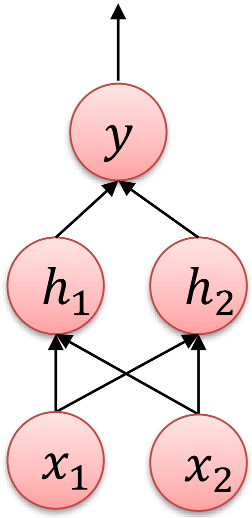
What are $\boldsymbol{h}^{(n)}$?

What are $y^{(n)}$?

XOR problem revisited

- Boolean function:

x_1	x_2	t	y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

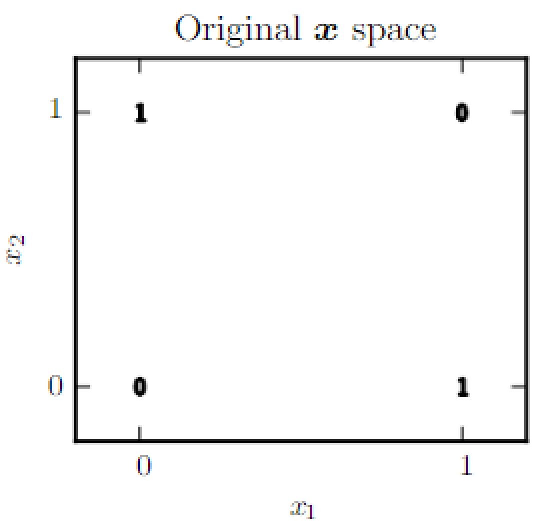


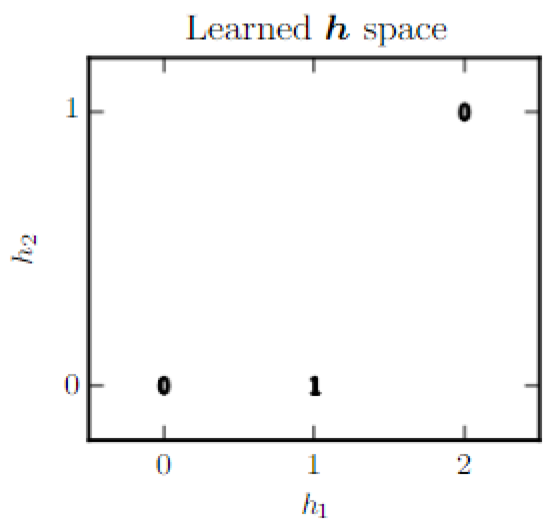
$$y = \mathrm{ReLU}(\boldsymbol{w}^\top \boldsymbol{h} + b)$$

$$\boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0$$

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})$$

$$\boldsymbol{V} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$





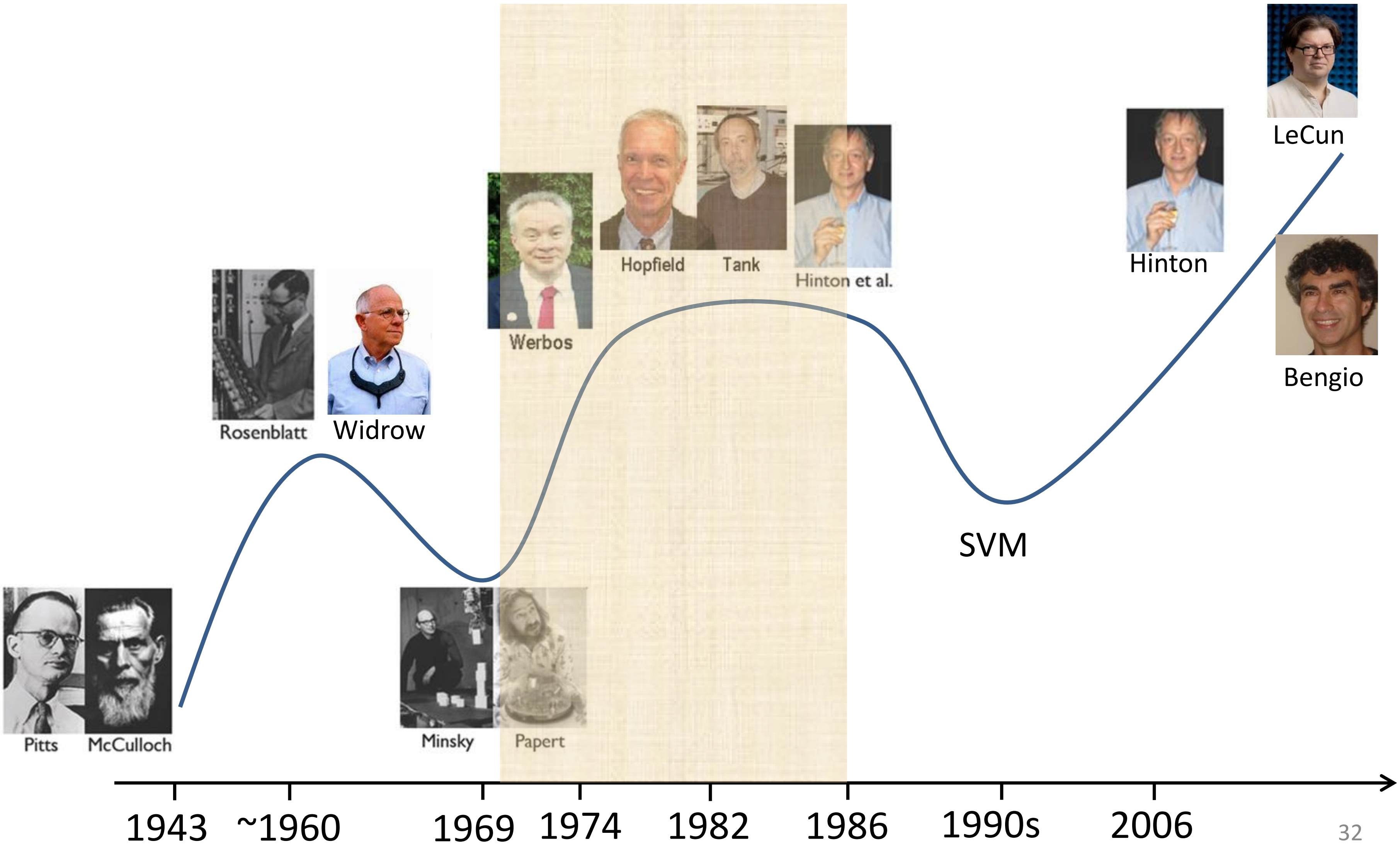
What are $\boldsymbol{h}^{(n)}$?

What are $y^{(n)}$?

Outline

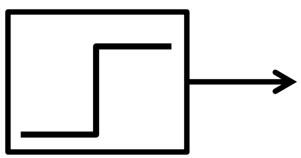
1. Regression and classification (cont'd)
2. Multi-layer perceptron
   - Feedforward calculation
   - Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

An efficient training algorithms was lacked



The main obstacles

- The activation function in the original Perceptron is the step function

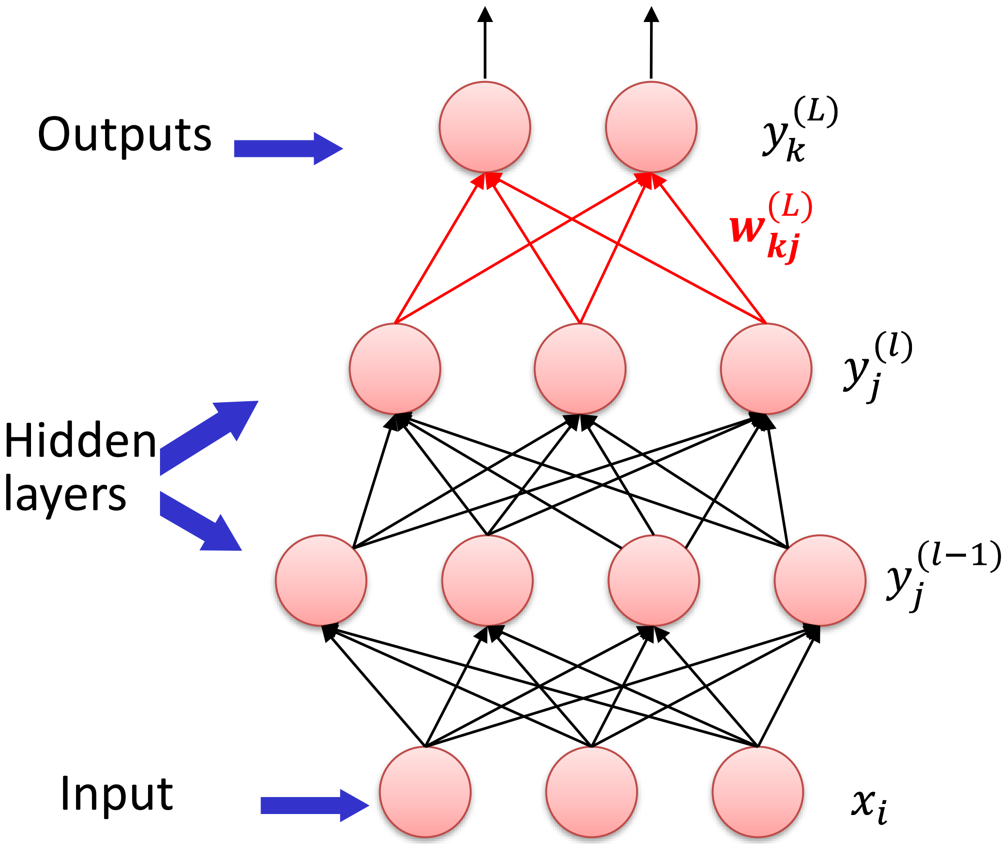


$$y = \begin{cases} 1 & \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ 0 & \text{else} \end{cases}$$

Could also be -1

- No optimization algorithms can deal with this function efficiently
- This was solved by introducing sigmoid functions in 1970s and 1980s

Consider the last layer



- Calculate the input to neuron k in the L-th layer
  $u_k^{(L)} = \sum_j w_{kj}^{(L)} y_j^{(L-1)} + b_k^{(L)}$
  and its output
  $$y_k^{(L)} = h\left(u_k^{(L)}\right)$$
  where $h(\cdot)$ is a nonlinear function
  - h can be sigmoid function, softmax function or other functions

For clarity we don't separate the linear transformation and activation 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

- Squared error or Euclidean loss

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

Is ReLU applicable?

- Cross-entropy error

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

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

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

Weight adjustment

- Weight adjustment

Learning rate

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

- 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

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

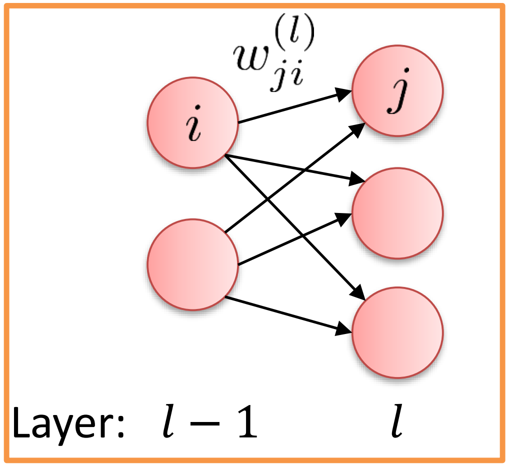
Gradient and local sensitivity

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

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

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

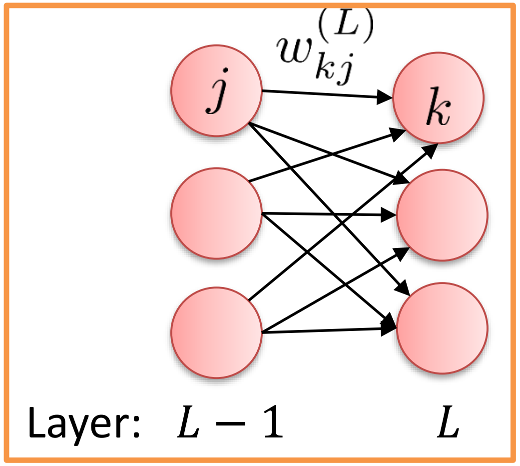
since $u_j^{(l)} = \sum_i w_{ji}^{(l)} f(u_i^{(l-1)}) + b_j^{(l)}$, where f is the activation function and $f(u_i^{(0)}) = x_i$.



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

Recall: Local sensitivity for MSE layer

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



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

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

where the activation function f can be

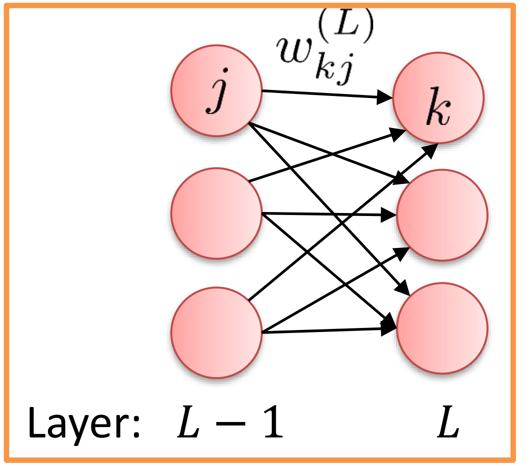
✓ logistic sigmoid ✓ tanh ✓ ReLU

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

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

Recall: local sensitivity for softmax layer

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



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

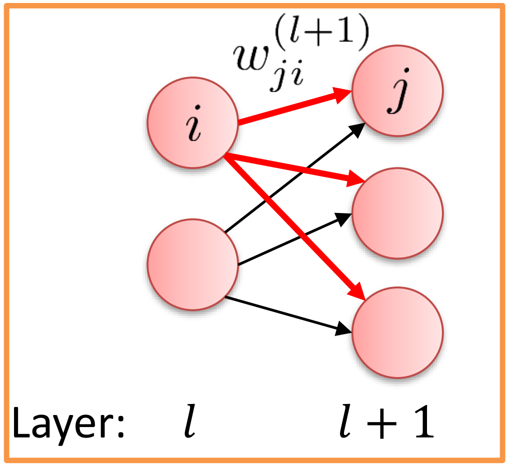
$$y_k^{(L)} \triangleq P(t_k = 1|\boxed{\boldsymbol{y}^{(L-1)}}) = \frac{\exp(\boldsymbol{w}_k^{(L)\top}\boldsymbol{y}^{(L-1)} + b_k^{(L)})}{\sum_{i=1}^{K}\exp(\boldsymbol{w}_i^{(L)\top}\boldsymbol{y}^{(L-1)} + b_i^{(L)})}$$

- 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



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

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

where f can be any activation function

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

Backpropagation in vector-matrix form

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

MSE: $\boldsymbol{\delta}^{(L)} = (\boldsymbol{y}^{(L)} - \boldsymbol{t}) \odot \boldsymbol{f}'(\boldsymbol{u}^{(L)})$ Cross-entropy Err: $\boldsymbol{\delta}^{(L)} = \boldsymbol{y}^{(L)} - \boldsymbol{t}$

where $\odot$ denotes element-wise multiplication

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

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

- Calculate the partial derivatives $1 \leq l \leq L$

Same dim as $\boldsymbol{W}^{(l)}$

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

for each sample n

- Update weights

avg over n

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

- The definition of $\boldsymbol{W}$ matrix

$$\boldsymbol{W} = \begin{pmatrix} w_{11} & \cdots & w_{1N} \\ \vdots & \vdots & \vdots \\ w_{M1} & \cdots & w_{MN} \end{pmatrix}$$

where M is the number of neurons in the current layer and N is the number of neurons in the previous layer

- The definition of matrix derivative

$$\frac{\partial E}{\partial \mathbf{W}} = \begin{pmatrix} \partial E/\partial w_{11} & \cdots & \partial E/\partial w_{1N} \\ \vdots & \vdots & \vdots \\ \partial E/\partial w_{M1} & \cdots & \partial E/\partial w_{MN} \end{pmatrix}$$

Gradient vanishing

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

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

  – For logistic function

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

  we have $f'(z) = f(z)\big(1 - f(z)\big)$ <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, $\boldsymbol{\delta}^{(l)}$ is smaller and smaller from L to 1. The gradient approaches zero in lower layers

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

ReLU function alleviates this effect

Implementation

- Run forward process
  - Calculate $\boldsymbol{f}\left(\boldsymbol{u}^{(l)}\right)$ and $\boldsymbol{f}'\left(\boldsymbol{u}^{(l)}\right)$ for $l = 1,2, \dots, L$
- Run backward process
  - Calculate $\boldsymbol{\delta}^{(l)}$ and $\partial E / \partial \boldsymbol{W}^{(l)}$,$\partial E / \partial \boldsymbol{b}^{(l)}$ for $l = L, L-1, \dots, 1$
- Update $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ for $l = 1,2, \dots, L$
- Modular programming
  - 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, output layer, etc.
  - Then you can design different structures of MLP by specifying the layer modules in a main file

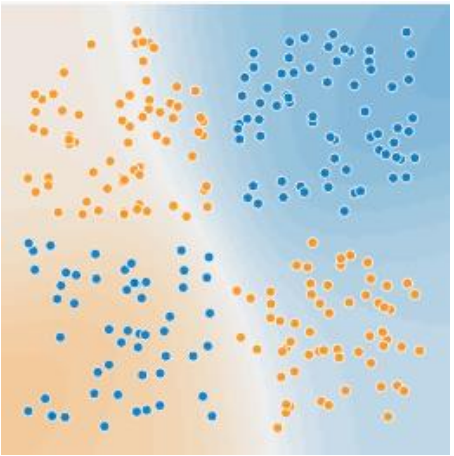
Summary of BP algorithm

	Forward	Backward
MSE output layer	$\boldsymbol{y}^{(L)} = \boldsymbol{f}\left(\boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L)}\right)$, where $\boldsymbol{f}$ is the act function	$\boldsymbol{\delta}^{(L)} = (\boldsymbol{y}^{(L)} - \boldsymbol{t}) \odot \boldsymbol{f}'(\boldsymbol{u}^{(L)})$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$
CE output layer	$\boldsymbol{y}^{(l)} = \boldsymbol{f}\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$, where $\boldsymbol{f}$ is the softmax function	$\boldsymbol{\delta}^{(L)} = \boldsymbol{y}^{(L)} - \boldsymbol{t}$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$
Hidden layer	$\boldsymbol{y}^{(l)} = \boldsymbol{f}\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$, where $\boldsymbol{f}$ is the act function	$\boldsymbol{\delta}^{(l)} = (\boldsymbol{W}^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)} \odot \boldsymbol{f}'(\boldsymbol{u}^{(l)})$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$

Typical activation functions: sigmoid, tanh, ReLU

Experiment 1: Classification of 2D points

- http://playground.tensorflow.org



Network setting

- Input: x1, x2
- Hidden layer: 1 layer, 4 neurons
- Use default values for other hyper-parameters

1. Run the training process
2. Change the learning rate to 1 and run
3. Change the learning rate back to 0.03, but use a regularization "L2" with rate 0.01, 0.1, or 1
4. Add a 2nd hidden layer with 2 neurons, and run

Experiment 1: Classification of 2D points

- http://playground.tensorflow.org

Set a suitable setting for solving this problem

- Change the hidden layers, input representation, learning rates, regularization…

Experiment 2: Classification of handwritten digits

https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

MNIST

- 60,000 training images and 10,000 test images
- 28x28 black and white images

Network setting

```
layer_defs = [];
layer_defs.push({type:'input', out_sx:24,
out_sy:24, out_depth:1});
layer_defs.push({type:'conv', sx:5, filters:8,
stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:16,
stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:3, stride:3});
layer_defs.push({type:'softmax',
num_classes:10});
```

Change the red part to:
layer_defs.push({type:'fc', num_neurons:32, activation:'relu'});

Outline

1. Regression and classification (cont’d)
2. Multi-layer perceptron
   – Feedforward calculation
   – Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

Motivation

	Forward	Backward
MSE output layer	$\boldsymbol{y}^{(L)} = \boldsymbol{f}\big(\boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L)}\big)$, where $\boldsymbol{f}$ is the act function	$\boldsymbol{\delta}^{(L)} = (\boldsymbol{y}^{(L)} - \boldsymbol{t}) \odot \boldsymbol{f}'(\boldsymbol{u}^{(L)})$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$
CE output layer	$\boldsymbol{y}^{(l)} = \boldsymbol{f}\big(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\big)$, where $\boldsymbol{f}$ is the softmax function	$\boldsymbol{\delta}^{(L)} = \boldsymbol{y}^{(L)} - \boldsymbol{t}$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$
Hidden layer	$\boldsymbol{y}^{(l)} = \boldsymbol{f}\big(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\big)$, where $\boldsymbol{f}$ is the act function	$\boldsymbol{\delta}^{(l)} = (\boldsymbol{W}^{(l+1)})^\top\boldsymbol{\delta}^{(l+1)} \odot \boldsymbol{f}'(\boldsymbol{u}^{(l)})$ $\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^\top$

Everytime when $\boldsymbol{f}$ changes, the forward and backward computations need change!

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 squared error layer $E^{(n)} = \frac{1}{2}\left|\left|\boldsymbol{f}\left(\boldsymbol{u}^{(L)}\right) - \boldsymbol{t}\right|\right|_2^2$

  can be decomposed into two layers

  – Activation layer: $y_k^{(L)} = f(u_k^{(L)})$, where f can be any function

  – Loss layer: $E^{(n)} = \frac{1}{2}\left|\left|\boldsymbol{y}^{(L)} - \boldsymbol{t}\right|\right|^2$

Question

- Consider the squared error function

$$E^{(n)} = \frac{1}{2}\left|\left|\boldsymbol{f}\left(\boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L)}\right) - \boldsymbol{t}\right|\right|_2^2$$

How many layers can be designed?

Question

- Consider the squared error function

$$E^{(n)} = \frac{1}{2}\left|\left|\boldsymbol{f}\left(\boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L)}\right) - \boldsymbol{t}\right|\right|_2^2$$

How many layers can be designed?

FC layer + activation layer + Euclidean loss layer

More flexible setting

- The cross-entropy error layer $E^{(n)} = -\sum_{k=1}^{K} t_k \ln f\left(u_k^{(L)}\right)$ 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

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

How many layers can be designed?

More flexible setting

- The cross-entropy error layer $E^{(n)} = -\sum_{k=1}^{K} t_k \ln f\left(u_k^{(L)}\right)$ 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

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

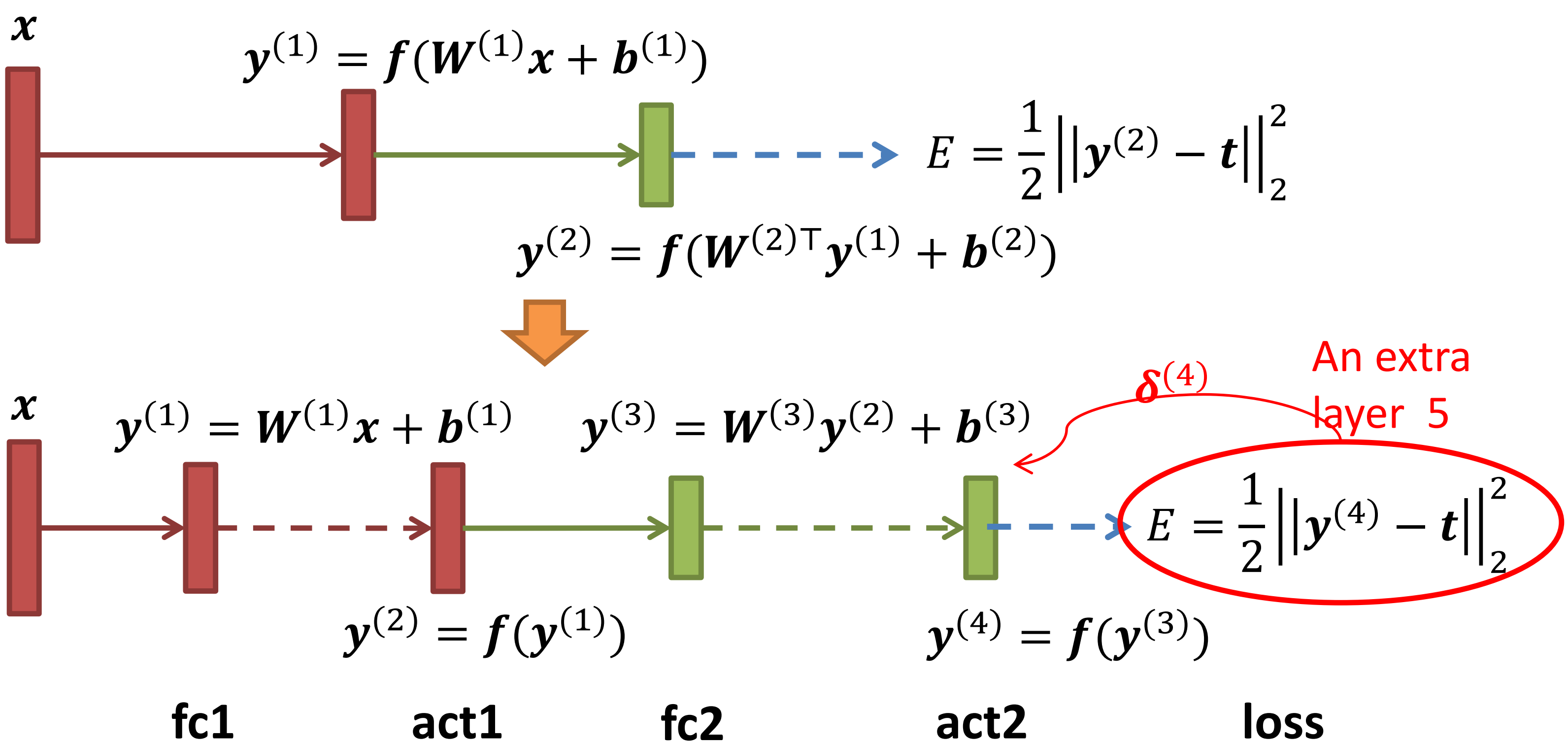
How many layers can be designed?

FC layer + softmax cross-entropy layer

Example 1

- An MLP with one hidden layer using the MSE loss

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

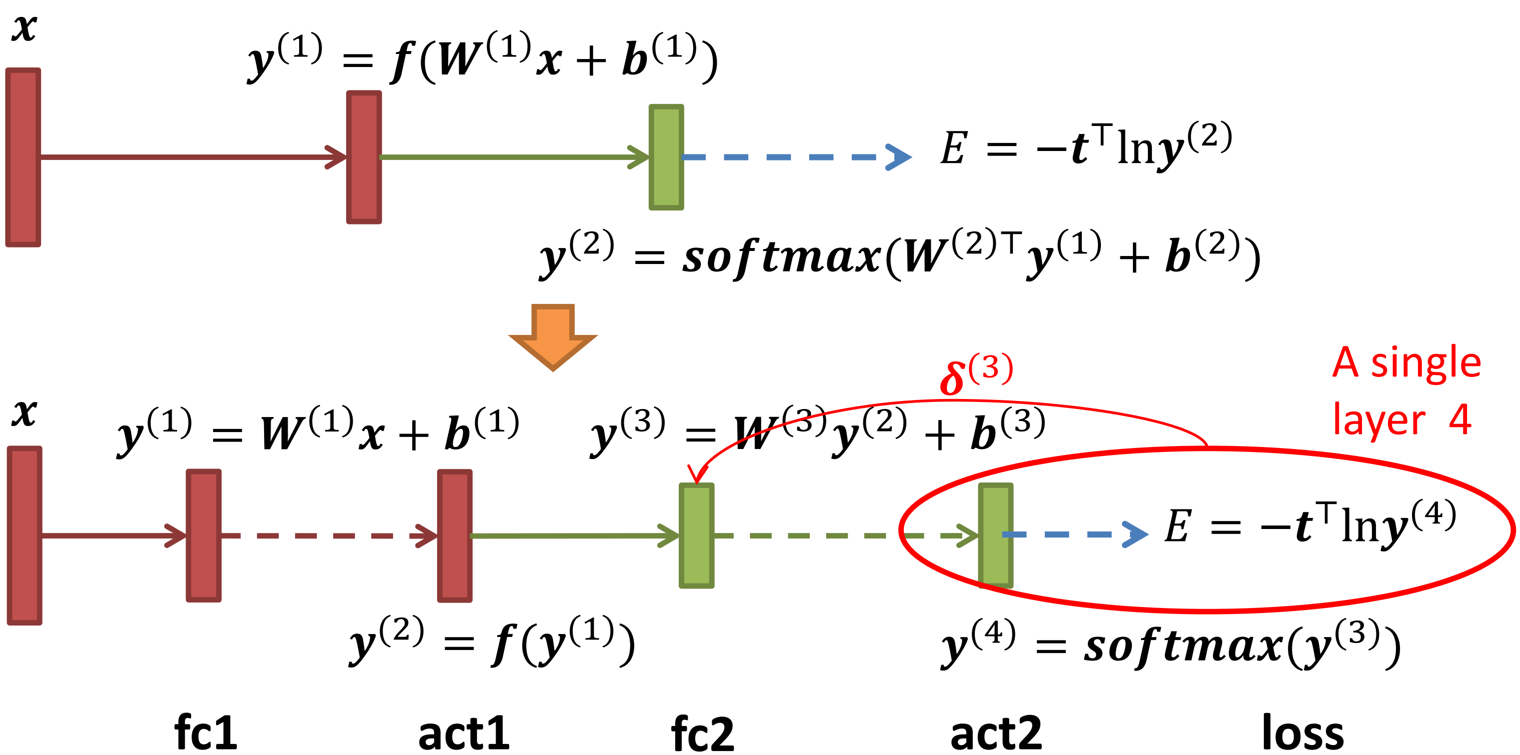


$\boldsymbol{u}^{(l)}$ and $\boldsymbol{y}^{(l)}$ are the same in every layer l

Example 2

- An MLP with one hidden layer using the CE loss

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



$\boldsymbol{u}^{(l)}$ and $\boldsymbol{y}^{(l)}$ are the same in every layer l

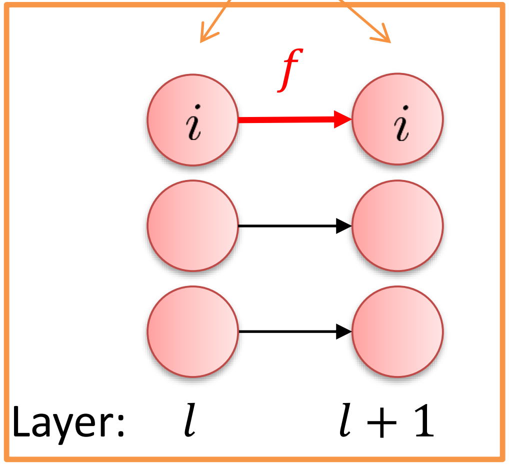
Exercise

- Derive the local sensitivity $\boldsymbol{\delta}$ and gradient $\partial E/\partial \boldsymbol{W}$ and $\partial E/\partial \boldsymbol{b}$ where applicable for
  - Euclidean loss layer: $E^{(n)} = \frac{1}{2}\left|\left|\boldsymbol{y}^{(L)} - \boldsymbol{t}\right|\right|^2$
    - Note that here we calculate $\boldsymbol{\delta}^{(L)}=\partial E^{(n)}/\partial \boldsymbol{y}^{(L)}$
  - Softmax-cross-entropy error layer $E^{(n)} = -\sum_{k=1}^{K} t_k \ln f\left(y_k^{(L)}\right)$
    - Note that here we calculate $\boldsymbol{\delta}^{(L-1)} = \partial E^{(n)}/\partial \boldsymbol{y}^{(L-1)}$
  - Fully connected layer:$y_j^{(l)} = \sum_i w_{ji}^{(l)} y_i^{(l-1)} + b_j^{(l)}$
  - Sigmoid layer: $y_j^{(l)} = f\left(y_j^{(l-1)}\right)$, where f is a sigmoid function
  - ReLU layer: $y_j^{(l)} = f\left(y_j^{(l-1)}\right)$, where f is a ReLU function

These layers are shown in the previous slides

Hint

No act fun anymore



- Suppose the $(l+1)$-th layer is a sigmoid activation layer:

$$y_i^{(l+1)} = f\left(y_i^{(l)}\right)$$

where f is the sigmoid function

- Neuron i in the l-th layer only affects neuron i in the $(l+1)$-th layer, therefore

Note that this layer doesn't have w and b

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

- Similarly, you can derive the results for other layers.

Outline

1. Regression and classification (cont’d)
2. Multi-layer perceptron
   – Feedforward calculation
   – Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

Weight initialization

$\boldsymbol{W}$ 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_{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 α 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 are constants.

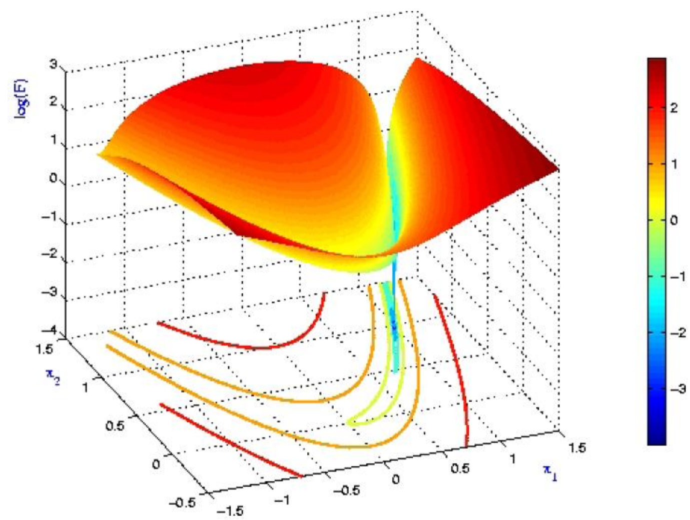
Order of training samples

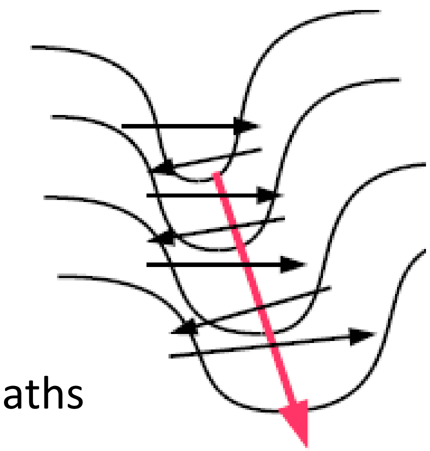
- If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence
- Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training.

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

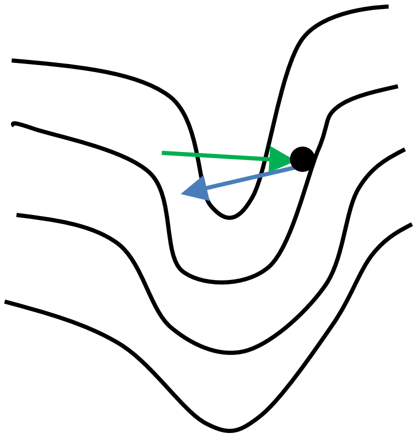
Momentum

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

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

  - $\boldsymbol{v}$ 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: γ is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher

An example

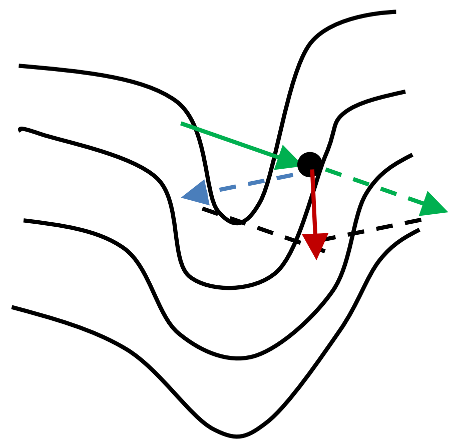


Let $\boldsymbol{g} = -\nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)}\right)$

Green: change of θ in the previous step

- Standard gradient decent:

$$\Delta\boldsymbol{\theta} = \alpha\boldsymbol{g}$$



- Gradient decent with momentum:

$$\Delta\boldsymbol{\theta} = \gamma\boldsymbol{v} + \alpha\boldsymbol{g}$$

This $\Delta\boldsymbol{\theta}$ is better aligned with the decreasing direction of the ravine

Outline

1. Regression and classification (cont'd)
2. Multi-layer perceptron
   – Feedforward calculation
   – Backward calculation
3. Layer decomposition
4. Training techniques-I
5. Summary

Summary of this lecture

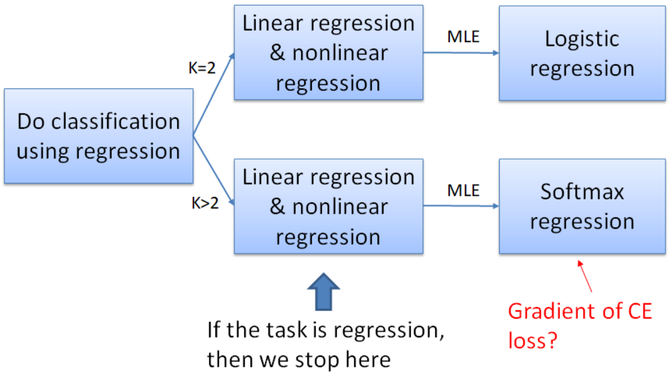
Knowledge

1. Regression and classification (cont'd)

$\boxed{\text{CE loss}}$

$$E^{(n)} = -\left(\boldsymbol{t}^{(n)}\right)^{\top} \ln \boldsymbol{h}^{(n)}$$

$$\nabla_{\boldsymbol{\theta}} E = \left(\boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right) - \boldsymbol{t}^{(n)}\right)\left(\boldsymbol{x}^{(n)}\right)^{\top}$$



2. Multi-layer perceptron
   - Forward calculation: for $l = 1, \dots, L$

$$\boldsymbol{u}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)} \text{ and } \boldsymbol{y}^{(l)} = \boldsymbol{f}\left(\boldsymbol{u}^{(l)}\right)$$

   - Backward calculation:

For $l = L$: $\boldsymbol{\delta}^{(L)} = (\boldsymbol{y}^{(L)} - \boldsymbol{t}) \odot \boldsymbol{f}'(\boldsymbol{u}^{(L)})$

or $\boldsymbol{\delta}^{(L)} = \boldsymbol{y}^{(L)} - \boldsymbol{t}$

For $l = L-1, \dots, 1$

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

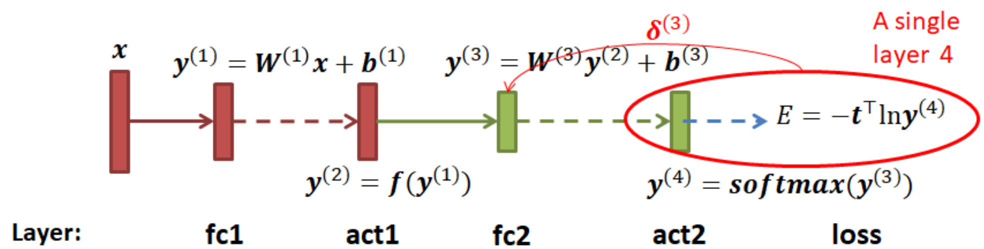
$$\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\boldsymbol{f}(\boldsymbol{u}^{(l-1)}))^{\top},$$

$$\frac{\partial E^{(n)}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

Summary of this lecture

Knowledge

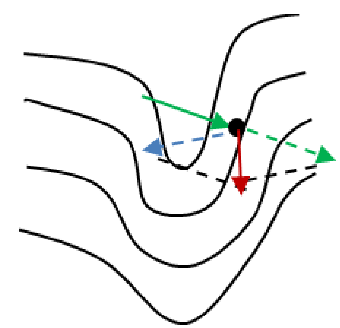
3. Layer decomposition



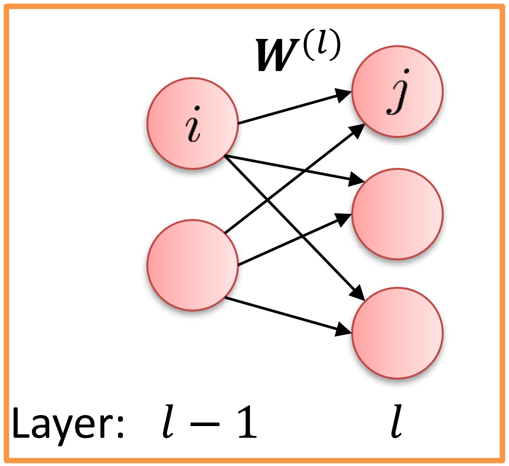
FC layer | sigmoid layer | ReLU layer | loss layer

4. Training techniques-I

Weight initialization | learning rate | order of training samples | momentum



Recommended reading



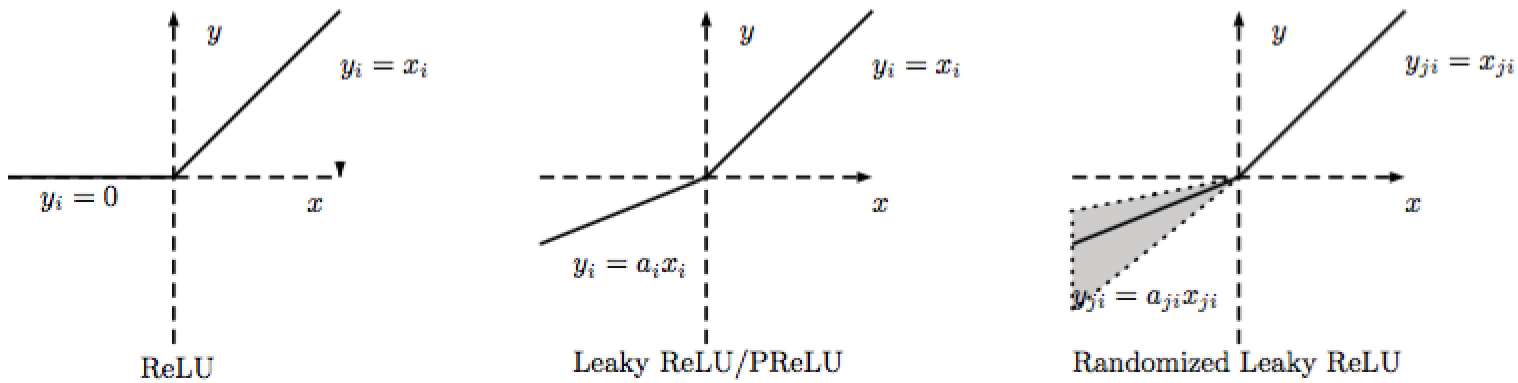
$$u^{(l)} = W^{(l)} y^{(l-1)} + b^{(l)}$$

$$\delta^{(l-1)} = \left(W^{(l)}\right)^{\top} \delta^{(l)}$$

- Liao, Leibo, Poggio (2015),
  How important is weight symmetry in backpropagation?
  AAAI
- Smith (2018)
  Cyclical learning rates for training neural networks
  arXiv:1506.01186v6

Recommended reading

- Variants of ReLU activation function



  - Xu, Wang, Chen, Li, Empirical Evaluation of Rectified Activations in Convolution Network, arXiv:1505.00853v2

- Other types of activation functions
  - Softplus: $f(x) = \log(e^x + 1)$
  - Softsign: $f(x) = \frac{x}{|x|+1}$