Project Proposal

- due **Tuesday, 2/1 @ 11:59pm**
- one page **maximum** stating:
  - student names
  - problem
  - data you will use
  - draft of proposed solution (can be **updated later**)
  - experiments you will run (can be **updated later**)
  - references (you can use an **additional** page for this)
- send me pdf by email (mvasconcelos@eng.ucsd.edu) with:
  - Subject: Group X Proposal, where X is the group number in this list
  - cc to all group members

This assignment is worth 5% of your class grade. **If you submit the proposal in time and make a serious attempt at addressing the bullet points above, you will get full score. I'm not, at this point, grading projects on their merits.** I will look at the proposal and give you some feedback. This will be mostly on issues that I think may become serious obstacles and you need to consider urgently. For example, if I find the problem you propose to be outside the scope of the class, that you may not be able to find data to train the methods you are proposing, etc. Note that if I say "OK", it just means that I see no such problems. It does not mean that you will receive an A just by doing what you proposed. I see these proposals more as a "direction to where the project is going." The projects themselves will be evaluated at the end of the quarter, according to the guidelines published.

1. **Hussain**, Tanvir; **Lewis**, Cameron; **Villamar**, Sandra
2. **Dong**, Meng; **Long**, Jianzhi; **Wen**, Bo; **Zhang**, Haochen
3. **Chen**, Yuzhao; **Li**, Zonghuan; **Song**, Yuze; **Yan**, Ge
4. **Li**, Jiayuan; **Xiao**, Nan; **Yu**, Nancy; **Zhou**, Pei
5. **Li**, Zheng; **Tao**, Jianyu; **Yang**, Fengqi
6. **Bian**, Xintong; **Jiang**, Yufan; **Wu**, Qiyao
7. **Chen**, Yongxing; **Yao**, Yanzhi; **Zhang**, Canwei
8. **Nukala**, Kishore; **Pulleti**, Sai; **Vaidyula**, Srikar
9. **Baluja**, Michael; **Cao**, Fangning; **Huff**, Mikael; **Shen**, Xuyang
10. **Arun**, Aditya; **Long**, Heyang; **Peng**, Haonan
11. **Cowin**, Samuel; **Hanna**, Aaron; **Liao**, Albert; **Mandadi**, Sumega
12. **Jia**, Yichen; **Jiang**, Zhiyun; **Li**, Zhuofan
13. **Dandu**, Murali; **Daru**, Srinivas; **Pamidi**, Sri
14. **He**, Bolin; **Huang**, Yen-Ting; **Wang**, Shi; **Wang**, Tzu-Kao
15. **Chen**, Luobin; **Feng**, Ruining; **Wu**, Ximei; **Xu**, Haoran
16. **Chen**, Rex; **Liang**, Youwei; **Zheng**, Xinran
17. **Aguilar**, Matthew; **Millhiser**, Jacob; **O'Boyle**, John; **Sharpless**, Will
18. **Wang**, Haoyu; **Wang**, Jiawei; **Zhang**, Yuwei
19. **Chen**, Yinbo; **Di**, Zonglin; **Mu**, Jiteng
20. **Chowdhury**, Debalina; **He**, Scott; **Ye**, Yiheng
21. **Lin**, Wei-Ru; **Ru**, Liyang; **Zhang**, Shaohua
22. **Bhavsar**, Shivad; **Blazej**, Christopher; **Bu**, Yinyan; **Liu**, Haozhe
23. **Chen**, Claire; **Hsieh**, Chia-Wei; **Lin**, Jui-Yu; **Tsai**, Ya-Chen
24. **Cheng**, Yu; **Yu**, Zhaowei; **Zaidi**, Ali
25. **Assadi**, Parsa; **Brugere**, Tristan; **Pathak**, Nikhil; **Zou**, Yuxin
26. **Candassamy**, Gokulakrishnan; **Dixit**, Rajeev; **Huang**, Joyce
27. **Kok**, Hong; **Wang**, Jacky; **Yan**, Yijia; **Yuan**, Zhouyuan
28. **Luan**, Zeting; **Yang**, Zheng
29. **Cuawenberghs**, Kalyani; **Mojtahed**, Hamed

ECE 271B – Winter 2022

Neural Networks

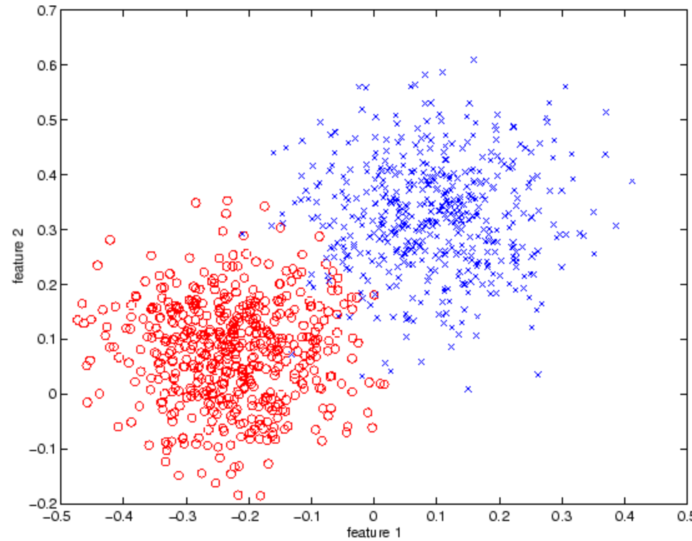
Disclaimer:
This class will be recorded
and made available to students asynchronously.

Manuela Vasconcelos

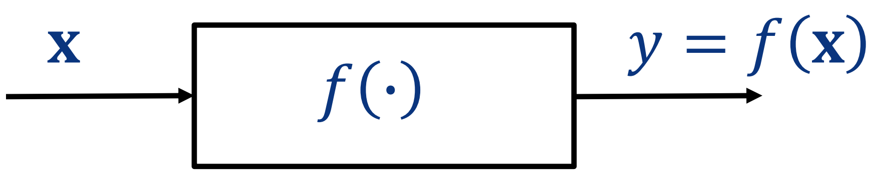
ECE Department, UCSD

Classification

- a **classification problem** has <u>two</u> types of variables
  - $\mathbf{x}$ – vector of observations (**features**) in the world
  - y – state (**class**) of the world
- e.g.
  - $\mathbf{x} \in \mathcal{X} \in \mathcal{R}^2 = (\text{fever}, \text{blood pressure})$
  - $\mathrm{y} \in \mathcal{Y} = \{\text{disease}, \text{no disease}\}$

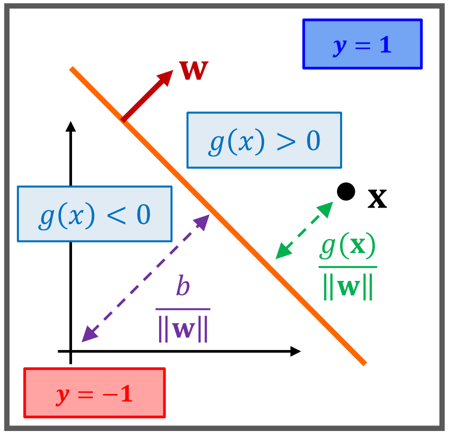


- $\mathbf{x}$, y related by (unknown) function



- **goal**: design a **classifier** $h\colon \mathcal{X} \longrightarrow \mathcal{Y}$ such that $h(\mathbf{x}) = f(\mathbf{x}), \forall \mathbf{x}$

The Perceptron



- classifier implements the linear decision rule

$$\boxed{h^*(\mathbf{x}) = \mathrm{sgn}[g(\mathbf{x})]} \quad \text{with} \quad \boxed{g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b}$$

- learning is formulated as an optimization problem
  - define set of errors

$$\boxed{E = \{\mathbf{x}_i | y_i(\mathbf{w}^T\mathbf{x}_i + b) \leq 0\}}$$

  - define the cost

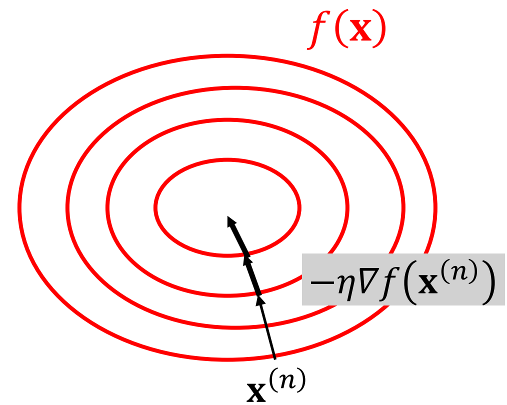
$$\boxed{J_P(\mathbf{w}, b) = -\sum_{i|\mathbf{x}_i \in E}^{n} y_i(\mathbf{w}^T\mathbf{x}_i + b)}$$

and minimize

Gradient Descent

- it is the **simplest** possible minimization technique
    - pick initial estimate $\mathbf{x}^{(0)}$
    - follow the negative gradient

$$\boxed{\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \eta \nabla f\left(\mathbf{x}^{(n)}\right)}$$



- usually, the gradient is a function of **entire** training set $\mathcal{D}$
- more efficient alternative is stochastic gradient descent
    - take the step immediately after each point
    - no guarantee this is a descent step but, **on average**, you follow the same direction after processing entire $\mathcal{D}$
    - very popular in learning, where $\mathcal{D}$ is **usually large**

Perceptron Learning

- for the Perceptron, this leads to:

$E = \{\mathbf{x}_i | y_i(\mathbf{w}^T\mathbf{x}_i + b) \le 0\}$

$$J_P(\mathbf{w}, b) = -\sum_{i|\mathbf{x}_i \in E}^{n} y_i(\mathbf{w}^T\mathbf{x}_i + b)$$

$$\frac{\partial J_P}{\partial \mathbf{w}} = -\sum_i y_i \mathbf{x}_i$$

$$\frac{\partial J_P}{\partial b} = -\sum_i y_i$$

gradient descent

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \eta \nabla f\left(\mathbf{x}^{(k)}\right)$$

for $\mathbf{x}_i \in E$,

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta y_i \mathbf{x}_i$$
$$b^{(k+1)} = b^{(k)} + \eta y_i$$

set $k = 0, \mathbf{w}_k = 0, b_k = 0$

set $R = \max_i \|\mathbf{x}_i\|$

do {

  for $i = 1{:}n$ {

    if $\underbrace{\boxed{y_i(\mathbf{w}_k{}^T\mathbf{x}_i + b_k) \le 0}}_{}$ then { $\mathbf{x}_i \in E$

- $\mathbf{w}_{k+1} = \mathbf{w}_k + \eta\, y_i\, \mathbf{x}_i$
- $b_{k+1} = b_k + \eta\, y_i\, R^2$
- $k = k + 1$

    }

  }

} until $y_i(\mathbf{w}^T\mathbf{x}_i + b_k) > 0, \forall i$ (no errors)

Perceptron Learning

- the interesting part is that this is guaranteed to **converge** in **finite time**

- **Theorem:** Let $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ and

$$R = \max_i \|\mathbf{x}_i\|.$$

If there is $(\mathbf{w}^*, b^*)$ such that $\|\mathbf{w}^*\| = 1$ and

$$y_i\left({\mathbf{w}^*}^T \mathbf{x}_i + b^*\right) > \gamma, \forall i,$$

then the Perceptron will find an error free hyper−plane in at most

$$\left(\frac{2R}{\gamma}\right)^2 \text{ iterations}$$

- the margin γ appears as a measure of the **difficulty** of the learning problem

Some History

- Minsky and Papert identified **serious** problems
  - there are very simple logic problems that the Perceptron cannot solve (e.g., Quiz#2, Prob. 1)

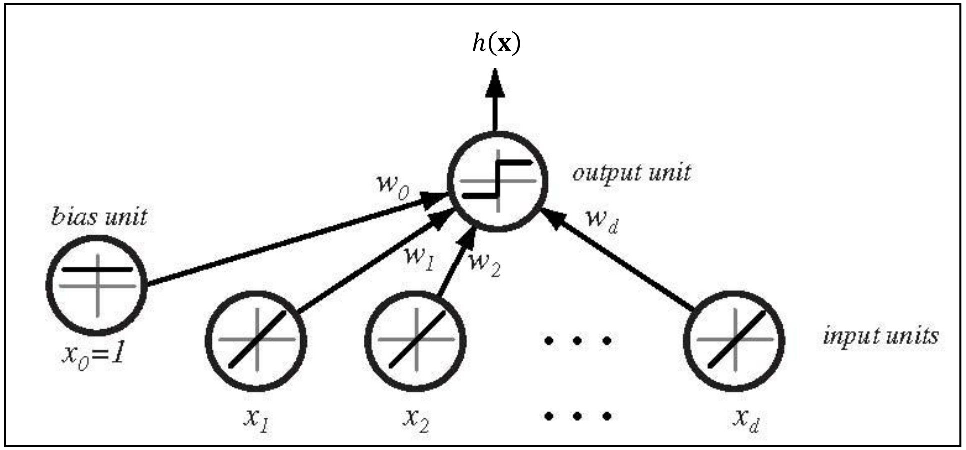
- later realized that these can be eliminated by relying on a **Multi–Layered Perceptron (MLP)** or **"neural network"**

- this is a **cascade of Perceptrons**, where

  - x_i are the input units
  - layer 1: $h_j(\mathbf{x}) = \mathrm{sgn}[\mathbf{w}_j^T \mathbf{x}]$
  - layer 2: $u(\mathbf{x}) = \mathrm{sgn}[\mathbf{w}^T h(\mathbf{x})]$

Graphical Representation

- the Perceptron is usually represented as



- input units: coordinates of $\mathbf{x}$
- weights: coordinates of $\mathbf{w}$
- homogeneous coordinates: $\mathbf{x} = (\mathbf{x}, 1)^T$

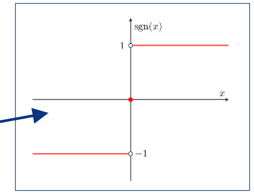
homogeneous coordinates in 2D:

$$\begin{aligned}\mathbf{w}^T\mathbf{x} + b &= w_1x_1 + w_2x_2 + b \\ &= (w_1, w_2, b)^T(x_1, x_2, 1) \\ &= \mathbf{w}^T\mathbf{x}\end{aligned}$$

$$h(\mathbf{x}) = \operatorname{sgn}\left(\sum_i w_i x_i + w_0\right) = \operatorname{sgn}(\mathbf{w}^T\mathbf{x})$$

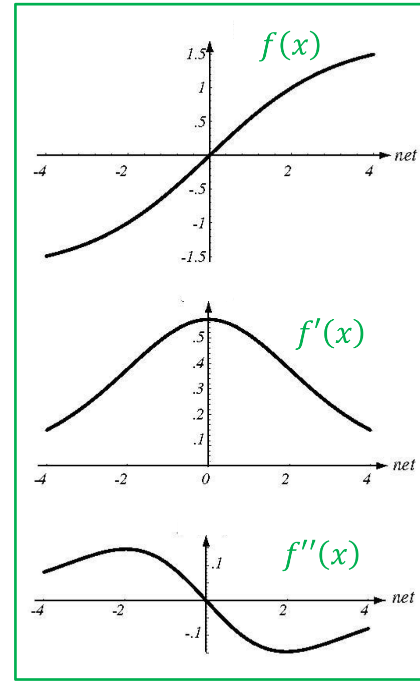
bias term
(what we have called b)

Non−Linearities



- the sgn(·) function is **problematic** in two ways:
  - no derivative at 0, non−smooth
- it **can be approximated** in various ways
  - e.g. by the hyperbolic tangent

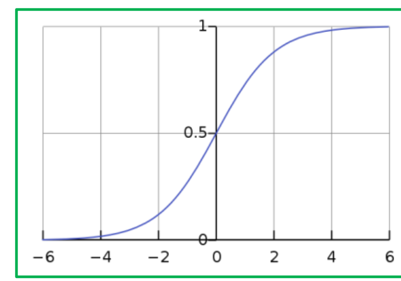
$$f(x) = \tanh(\sigma x) = \frac{e^{\sigma x} - e^{-\sigma x}}{e^{\sigma x} + e^{-\sigma x}}$$



  σ controls the approximation error, but has derivative everywhere, smooth

  - another **popular choice** is the sigmoid

$$f(x) = \frac{1}{1 + e^{-\sigma x}}$$



- The main difference between the two is the **output range**
  - $[-1,1]$ for the tanh
  - $[0,1]$ for the sigmoid

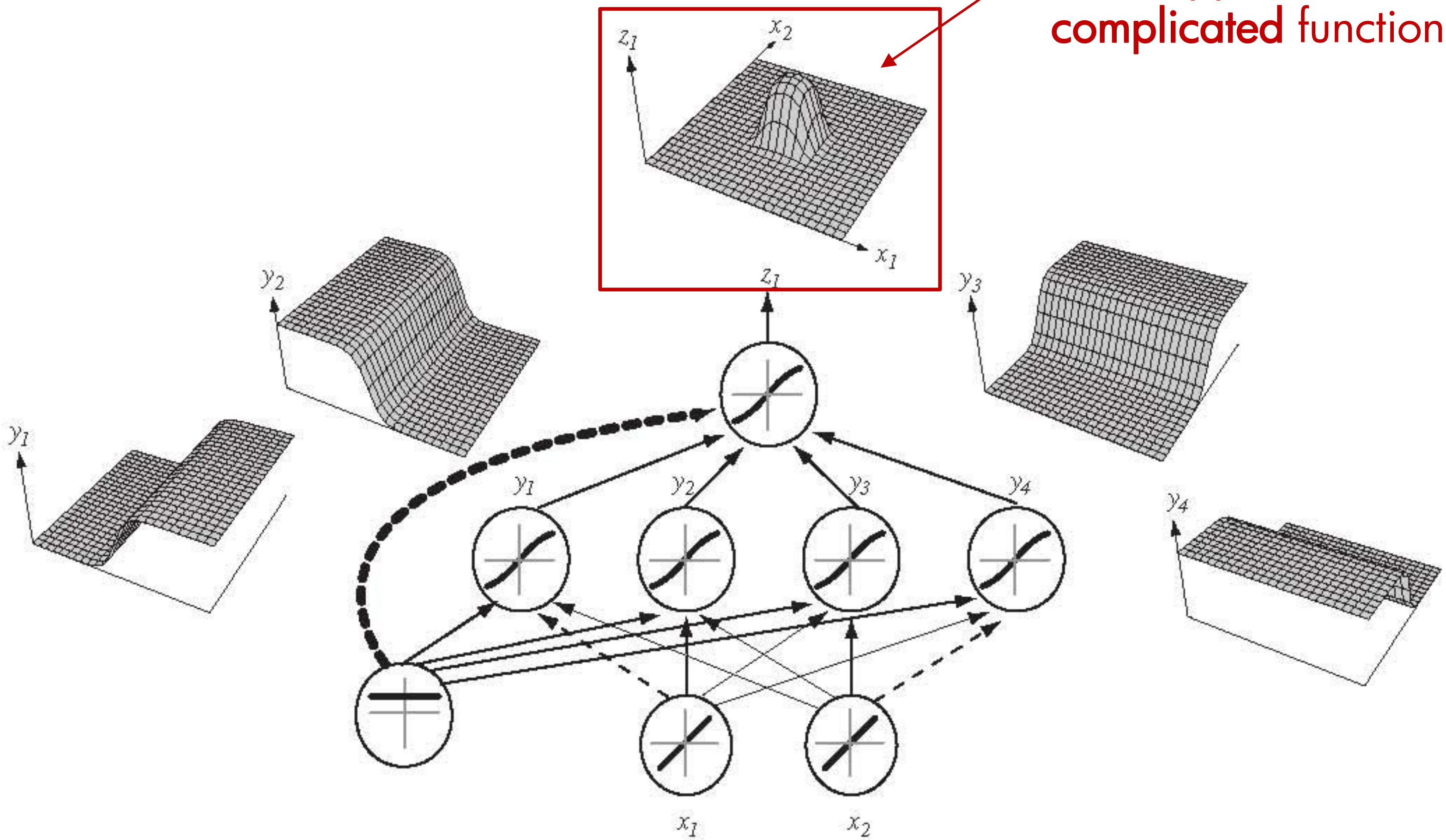
we will just refer to these as non−linearities, the exact form should be clear from context

- early **neural networks** were implemented with these functions

Neural Network

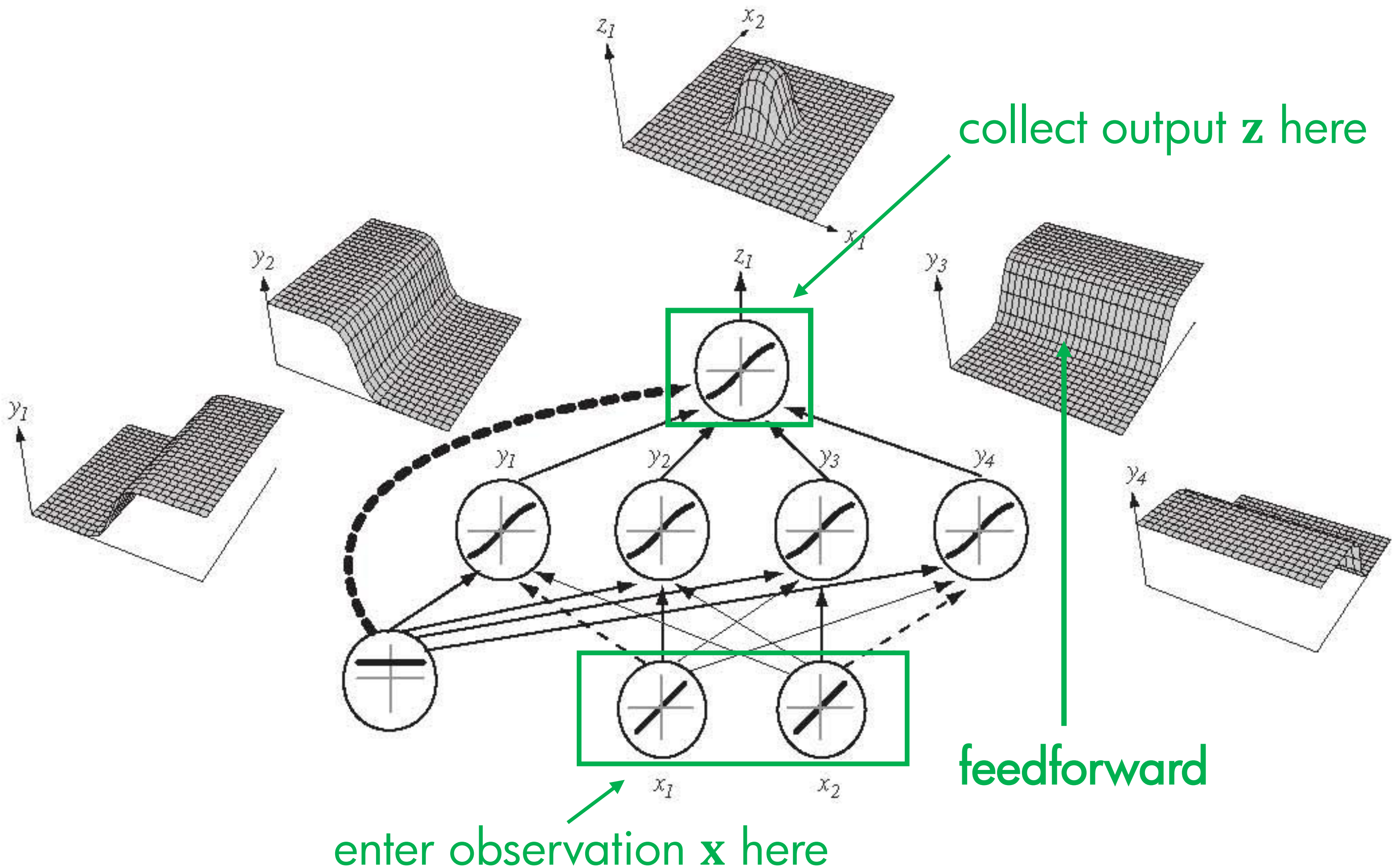
- the MLP as function approximation

even with just 2 layers,
it is possible
to approximate
complicated functions!



Two Modes of Operation

- normal mode, after training: feedforward

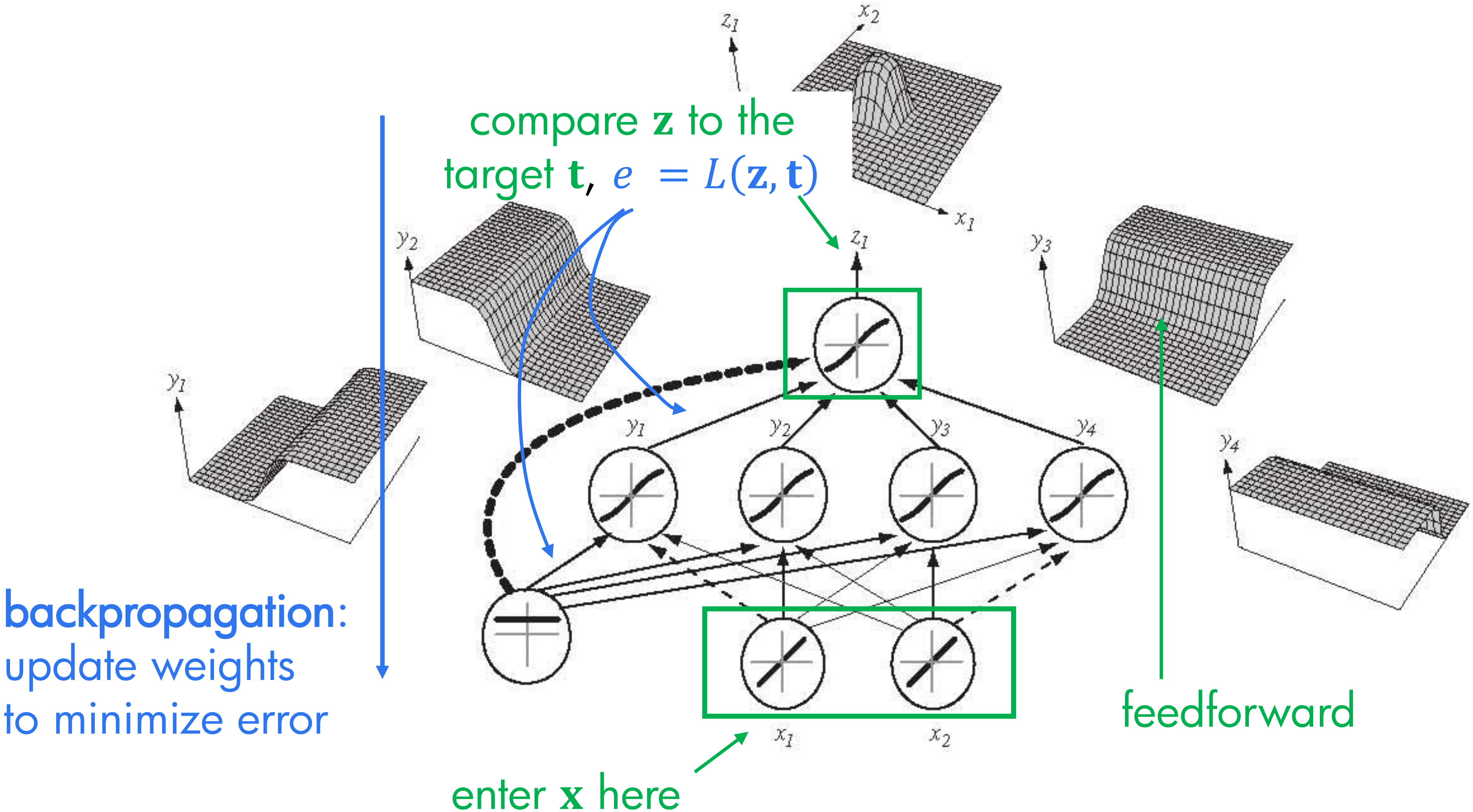


Two Modes of Operation

- training mode: backpropagation

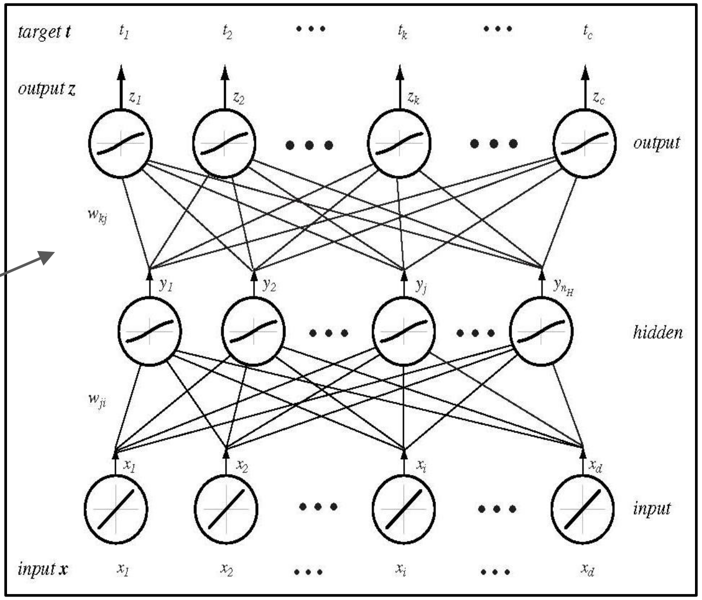
iterative:

- propagate $\mathbf{x}$ feedforward
- compute error e
- backpropagate to adjust weights



Network Variants

- The **details of the network** depend on what the task is
  - this determines
    the target (class label),
    the loss function, and
    the type of output units
  - usually, we have C outputs
    and the network looks like this
  - **at least three** important **tasks**
    - classification
      - pick one class out of C
        (e.g., digit recognition, pick one digit out of C – Quiz#2 Prob. 5)
    - multi–label classification
      - C outputs, each detects one class
    - regression
      - target in $\mathbb{R}^C$



Network Variants: Classification

- selects one class out of the C possible
- target is "one–hot–encoding" vector
  - $C - 1$ zero bits
  - one "hot" bit (set to 1) indicates the class
- loss function
  - cross–entropy

$$L(\mathbf{z}, \mathbf{t}) = -\sum_{k=1}^{C} t_k \log(z_k)$$

- output layer
  - $\mathbf{z}$ is computed with the softmax function
- you will work with this in the quiz (Quiz#2, Prob. 5)

Network Variants: Multi–Label Classification

- C binary classifiers, each detects one class
  - useful when the classes are <u>**not**</u> **exclusive** (**multi–label**)
  - an example is attribute classification
    - person is male, adult, has no beard, uses glasses, etc.
- target is a binary bit stream
  - one bit per attribute, detects its presence
  - multiple bits can be "hot" simultaneously
- loss function
  - cross–entropy:

$$L(\mathbf{z}, \mathbf{t}) = -\left[\sum_{k=1}^{C} t_k \log(z_k) + (1 - t_k) \log(1 - z_k)\right]$$

- output layer
  - C sigmoids

Network Variants: Regression

- target is continuous
- loss function
  - squared–error

$$L(\mathbf{z}, \mathbf{t}) = \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2 = \frac{1}{2} \sum_{k=1}^{C} [t_k - z_k]^2$$

- output layer
  - no non–linearities

- note that these choices are <u>not</u> **exclusive**
  - e.g., you can use the **squared–error** with the **multi–label classifier**
  - in what follows, we will <u>assume</u> this configuration
- in all cases, we get a **similar** <u>**backpropagation algorithm**</u>

Backpropagation

- is just gradient descent

- at the end of the day, the output $\mathbf{z}$ is just a "big function" of
  - input vector $\mathbf{x}$
  - weight matrix, which we can be represent by a "big" vector $\mathbf{W}$
  - e.g.

$$\mathbf{z} = s\left[\sum_j v_j\, s\left(\sum_i w_{ji} x_i\right)\right] = \mathbf{z}(\mathbf{x}; \mathbf{W}) \quad \text{with} \quad \mathbf{W} = (\mathbf{v}, \mathbf{w})$$

- **objective:** given a dataset $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{t}_1), \dots, (\mathbf{x}_n, \mathbf{t}_n)\}$, determine

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W}) \qquad J(\mathbf{W}) = \sum_{i=1}^{n} L\big(\mathbf{t}_i, \mathbf{z}(\mathbf{x}_i; \mathbf{W})\big) \qquad L(\mathbf{t}, \mathbf{z}) = \frac{1}{2}\sum_{k=1}^{C}[t_k - z_k]^2$$

squared–error with the multi–label classifier

Backpropagation

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

$$J(\mathbf{W}) = \sum_{i=1}^{n} L\big(\mathbf{t}_i, \mathbf{z}(\mathbf{x}_i; \mathbf{W})\big)$$

$$L(\mathbf{t}, \mathbf{z}) = \frac{1}{2} \sum_k [t_k - z_k]^2$$

- gradient descent
  - pick initial estimate $\mathbf{W}^{(0)}$
  - follow the negative gradient

$$\boxed{\mathbf{W}^{(n+1)} = \mathbf{W}^{(n)} - \eta \nabla J\big(\mathbf{W}^{(n)}\big)}$$

- stochastic gradient descent
  - take the step <u>**immediately**</u> after example $(\mathbf{x}_i, \mathbf{t}_i)$
    - pick initial estimate $\mathbf{W}^{(0)}$
    - follow the negative gradient

$$\boxed{\mathbf{W}^{(i+1)} = \mathbf{W}^{(i)} - \eta \nabla L\left(\mathbf{t}_i, \mathbf{z}\big(\mathbf{x}_i; \mathbf{W}^{(i)}\big)\right)}$$

Backpropagation

Ramelhart, David; Hinton, Geoffrey; Williams, Ronald; Learning Internal Representations by Backpropagating errors. In *Parallel Distributed Processing*, Volume 1, MIT Press, 1986.

- this is conceptually trivial, but computing the gradient of L looks **quite messy**
  - e.g.

$$z = s\left[\sum_j v_j\, s\left(\sum_i w_{ji} x_i\right)\right]$$

This is for 2 layers.
Imagine if you have 10 layers …
Modern NNs have more than 100!

  - what is $\partial z / \partial w_{13}$?

- it turns out that it is possible to do this easily by doing a **certain amount of book–keeping**

- the solution is the **backpropagation algorithm**, which is based on local updates

- the key to understanding it is to make the **right definitions**

In Detail

- notation:
  - input i: x_i
  - weight to hidden unit j: w_{ji}
  - hidden unit j:

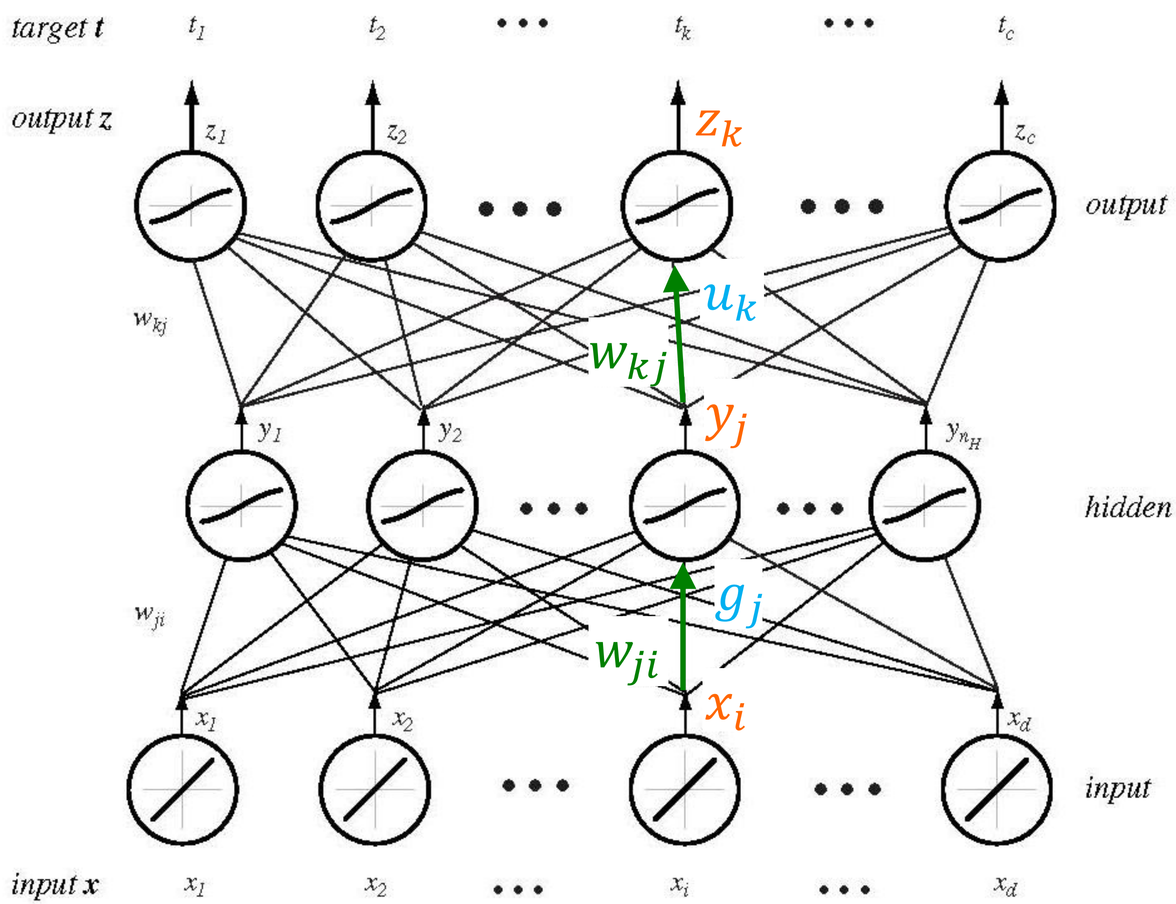
$$\boxed{g_j = \sum_i w_{ji} x_i}$$

$$\boxed{y_j = s[g_j]}$$

  - weight to output unit k: w_{kj}
  - output unit k:

$$\boxed{u_k = \sum_j w_{kj} y_j}$$

$$\boxed{z_k = s[u_k]}$$



before non−linearity

after non−linearity

Computing the Gradient of L

- $\boxed{\text{the key is the chain rule}}$
- the output layer is easy

$$\frac{\partial L}{\partial w_{kj}} = \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{kj}}$$
$$= -(t_k - z_k)s'[u_k]y_j$$

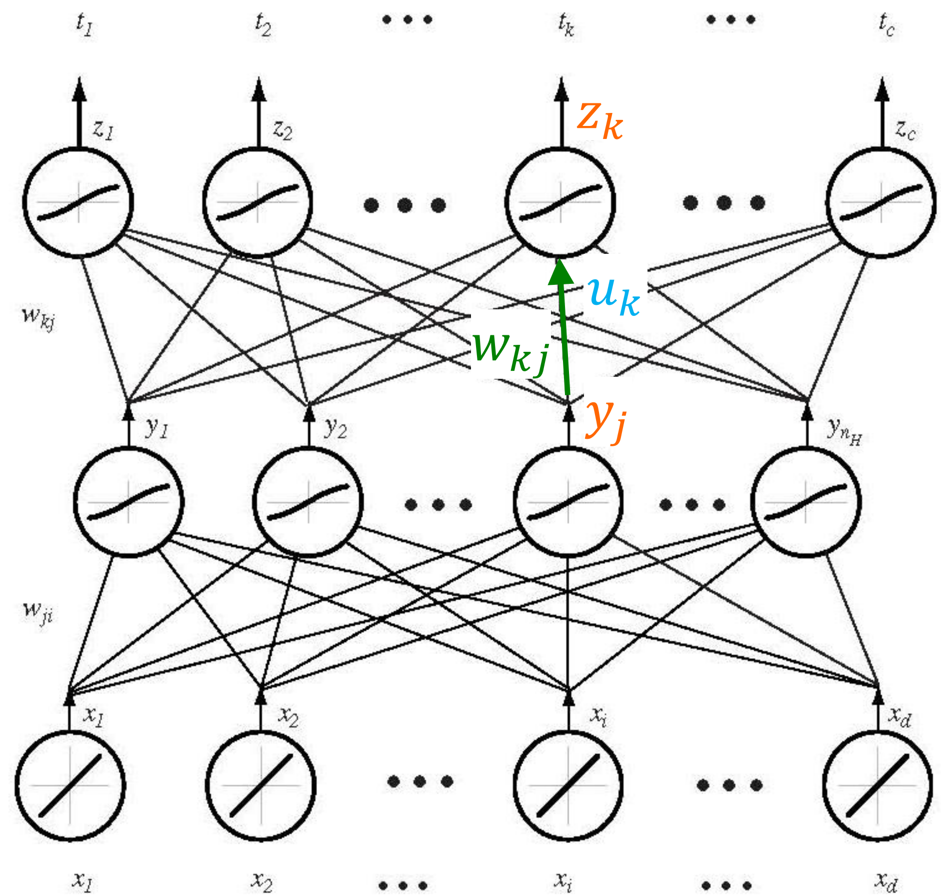
or

$$\boxed{\frac{\partial L}{\partial w_{kj}} = -\delta_k y_j} \qquad (*)$$

where

$$\delta_k = (t_k - z_k)s'[u_k] \qquad (**)$$

is the **sensitivity** of unit k



$$\boxed{L = \frac{1}{2}\sum_k [t_k - z_k]^2} \qquad \boxed{z_k = s[u_k]} \qquad \boxed{u_k = \sum_j w_{kj} y_j}$$

Computing the Gradient of L

$$\frac{\partial L}{\partial w_{kj}} = -\delta_k y_j \qquad (*)$$

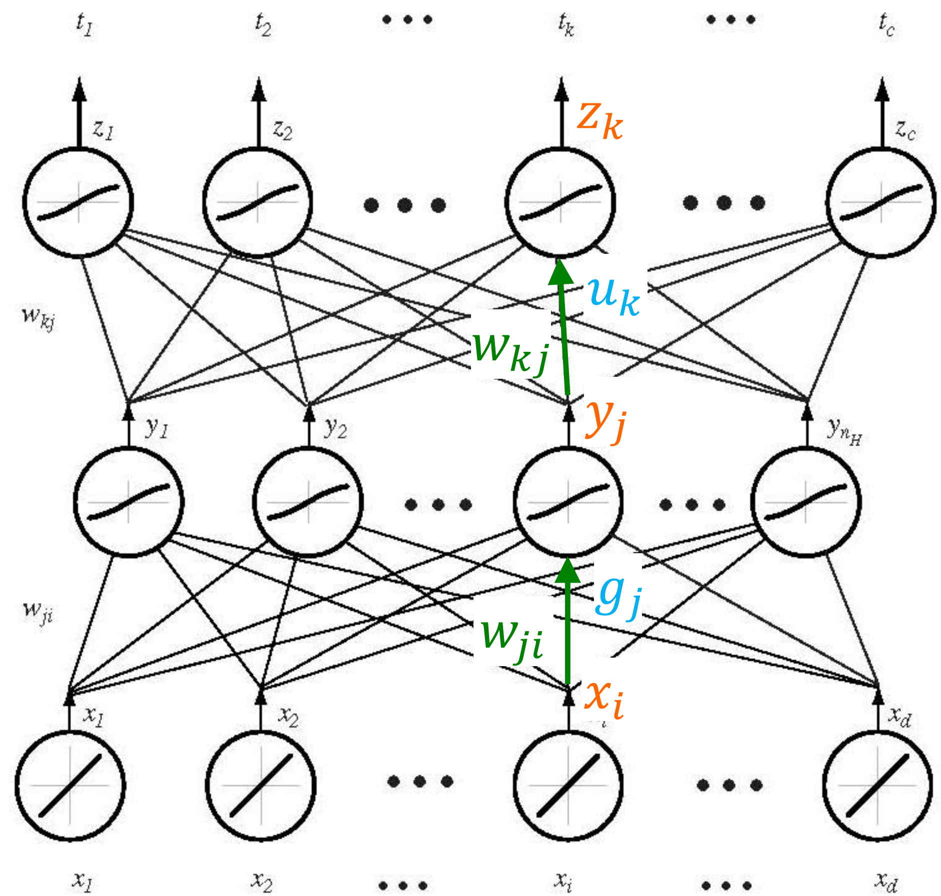
$$\delta_k = (t_k - z_k)s'[u_k] \qquad (**)$$

- for the **hidden layer**

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial g_j}\frac{\partial g_j}{\partial w_{ji}}$$

$$= \frac{\partial L}{\partial y_j} s'[g_j] x_i$$

- this is a little **more subtle** since L depends on y_j through all the z_k's



$$L = \frac{1}{2}\sum_k [t_k - z_k]^2$$

$$y_j = s[g_j]$$

$$g_j = \sum_j w_{ji} x_i$$

$$\frac{\partial L}{\partial w_{kj}} = -\delta_k y_j \qquad (*)$$

$$\delta_k = (t_k - z_k)s'[u_k] \qquad (**)$$

Computing the Gradient of L

- for the hidden layer

$$\frac{\partial L}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_k (t_k - z_k)^2\right]$$

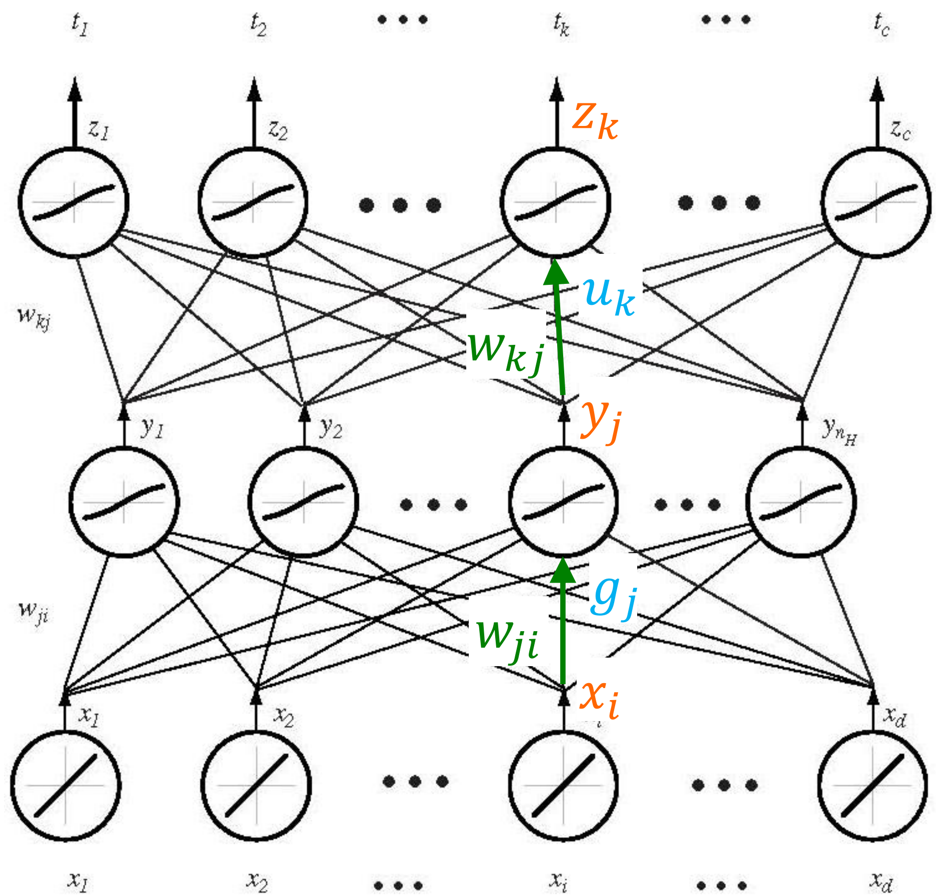
$$= -\sum_k (t_k - z_k)\frac{\partial z_k}{\partial y_j}$$

$$= -\sum_k (t_k - z_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial y_j}$$

$$= -\sum_k (t_k - z_k)\, s'[u_k]\, w_{kj}$$

- and, from $(**)$,

$$\frac{\partial L}{\partial y_j} = -\sum_k \delta_k w_{kj}$$



$$L = \frac{1}{2}\sum_k [t_k - z_k]^2$$

$$z_k = s[u_k]$$

$$u_k = \sum_j w_{kj} y_j$$

Computing the Gradient of L

$$\frac{\partial L}{\partial w_{kj}} = -\delta_k y_j \qquad (*)$$

$$\delta_k = (t_k - z_k) s'[u_k] \qquad (**)$$

- for the hidden layer
- overall

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial y_j} s'[g_j] x_i$$

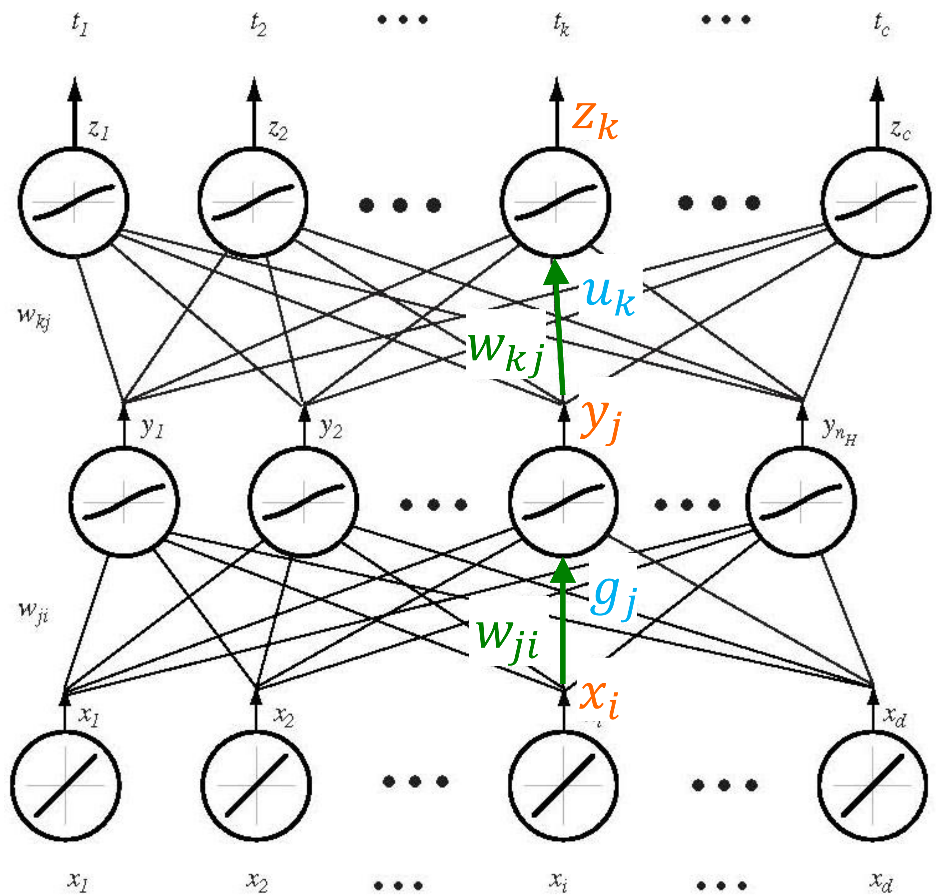
$$= -\left[\sum_k \delta_k w_{kj}\right] s'[g_j] x_i$$

- and by analogy with $(*)$

$$\boxed{\frac{\partial L}{\partial w_{ji}} = -\delta_j \, x_i}$$

with

$$\delta_j = \left[\sum_k \delta_k w_{kj}\right] s'[g_j]$$



In Summary

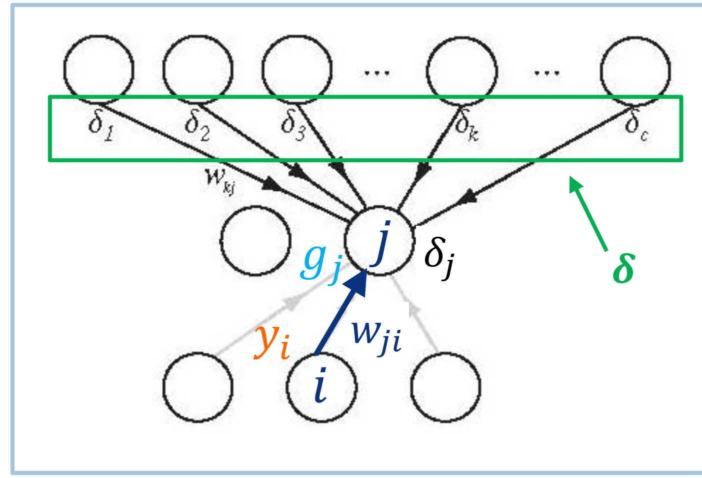
- for <u>any</u> pair (i, j)

$$\boxed{\frac{\partial L}{\partial w_{ji}} = -\delta_j \, y_i}$$

with

$$\boxed{\delta_j = (t_j - z_j) s'[u_j] \qquad \text{if } j \text{ is output}}$$

$$\boxed{\delta_j = \left[\sum_k \delta_k w_{kj}\right] s'[g_j] \quad \text{if } j \text{ is hidden}}$$



the **weight updates** are

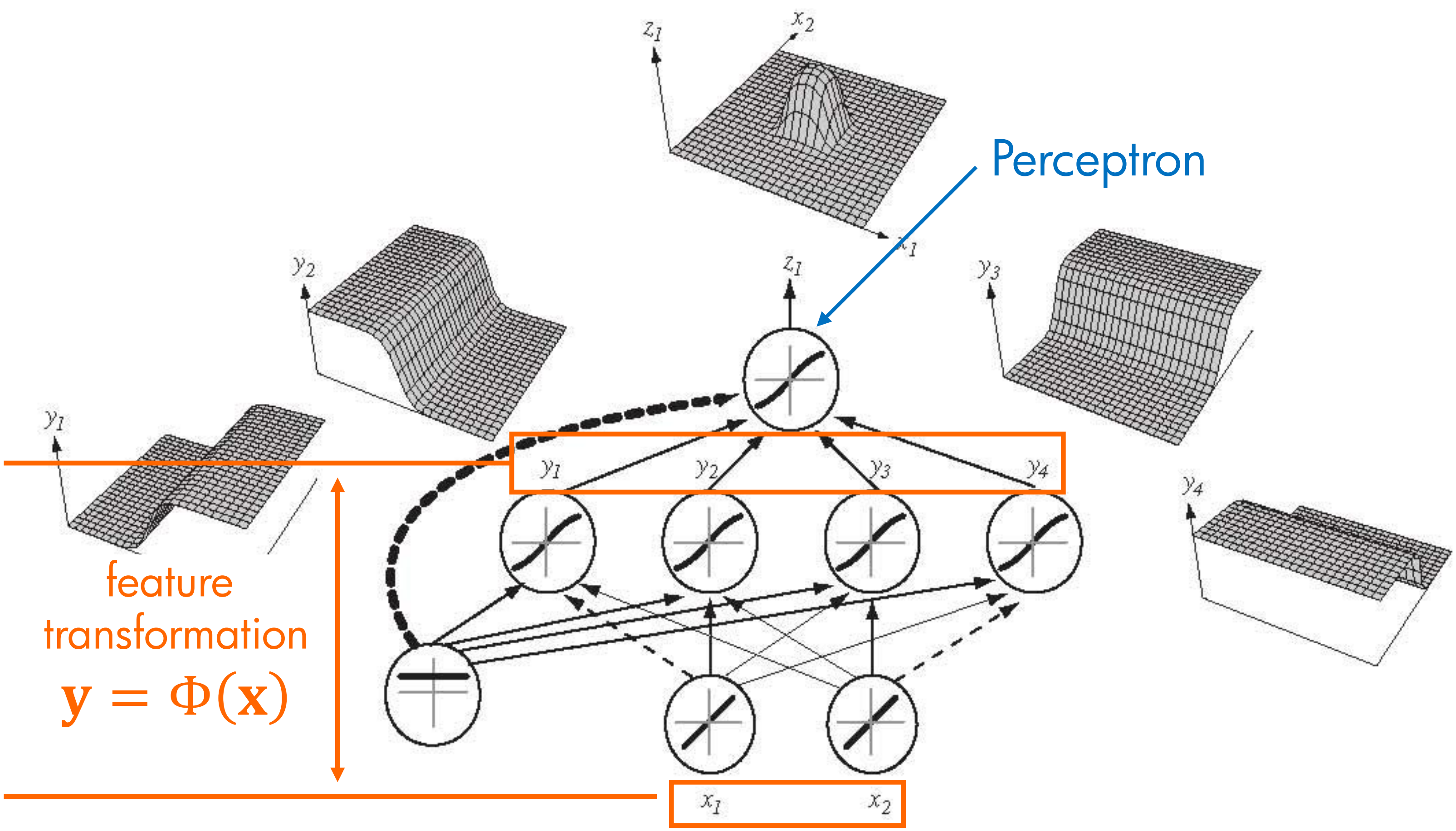
$$\boxed{w_{ji}^{(k+1)} = w_{ji}^{(k)} - \eta \frac{\partial L}{\partial w_{ji}}}$$

- the **error** is "**backpropagated**" by **local message passing!**

Feature Transformation

- MLP can be seen as:

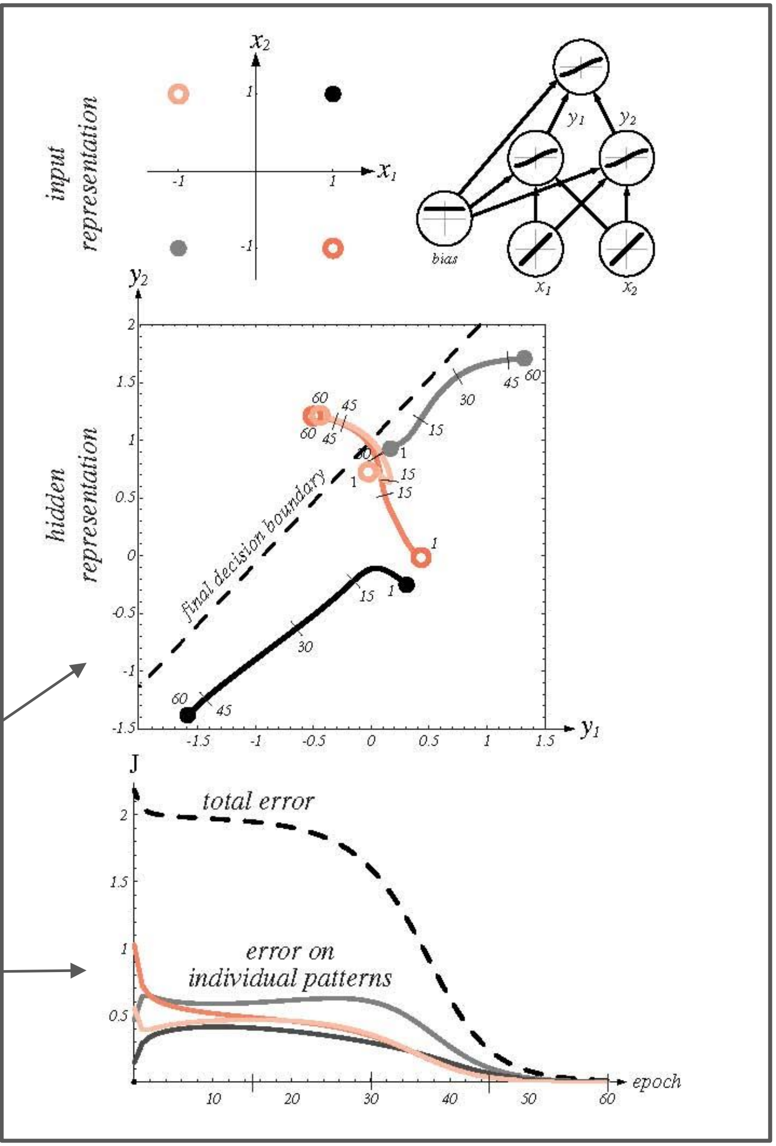
  non−linear feature transformation + linear discriminant



Feature Transformation

- feature transformation
  - searches for the space where the patterns become **separable**
- example
  - two–class problem
  - 2 – 1 network

    2 – 1
    2 hidden – 1 output
  - **non–linearly** separable on the space of x's
  - made **linearly** separable on the space of y's
  - the figure shows evolution of y's and the training error



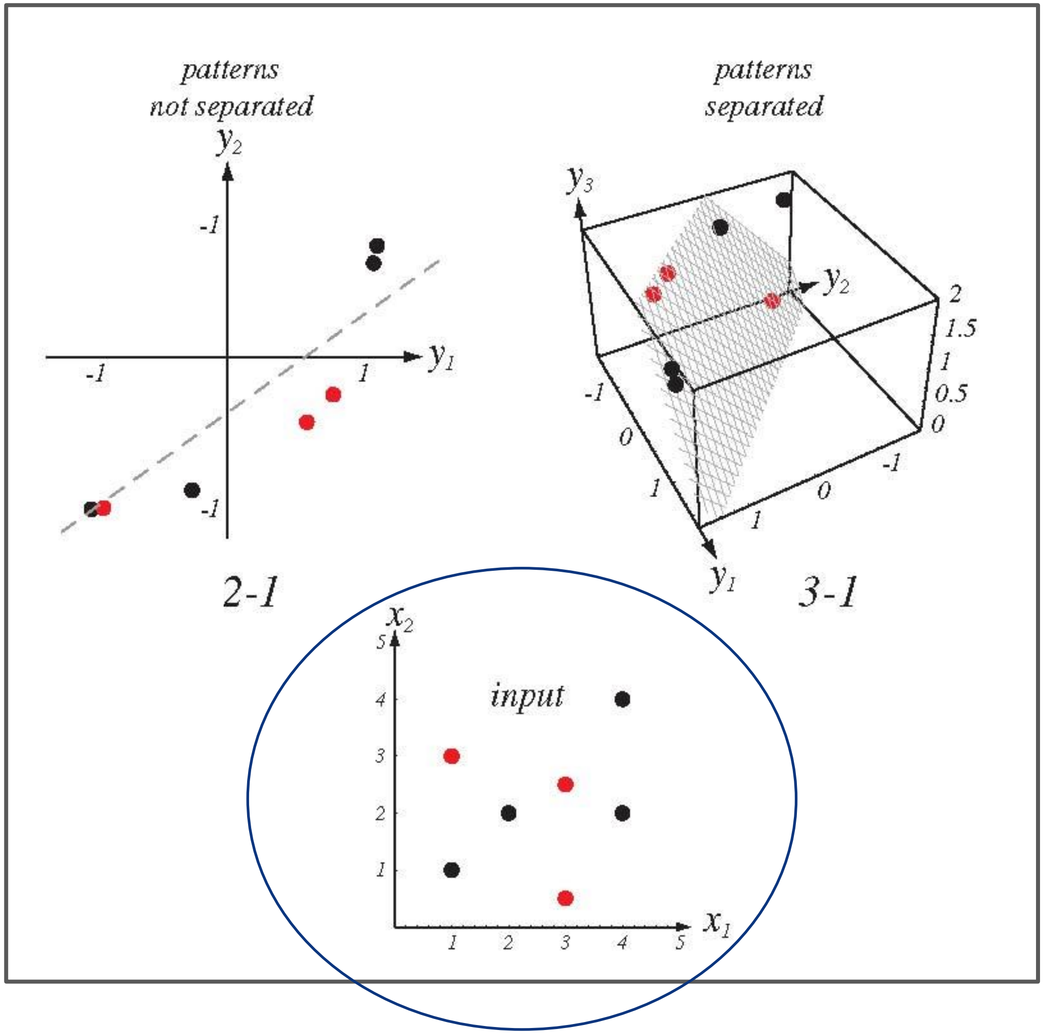
Feature Transformation

- Q: is separability always possible?

- A: not really, depends on the number of units

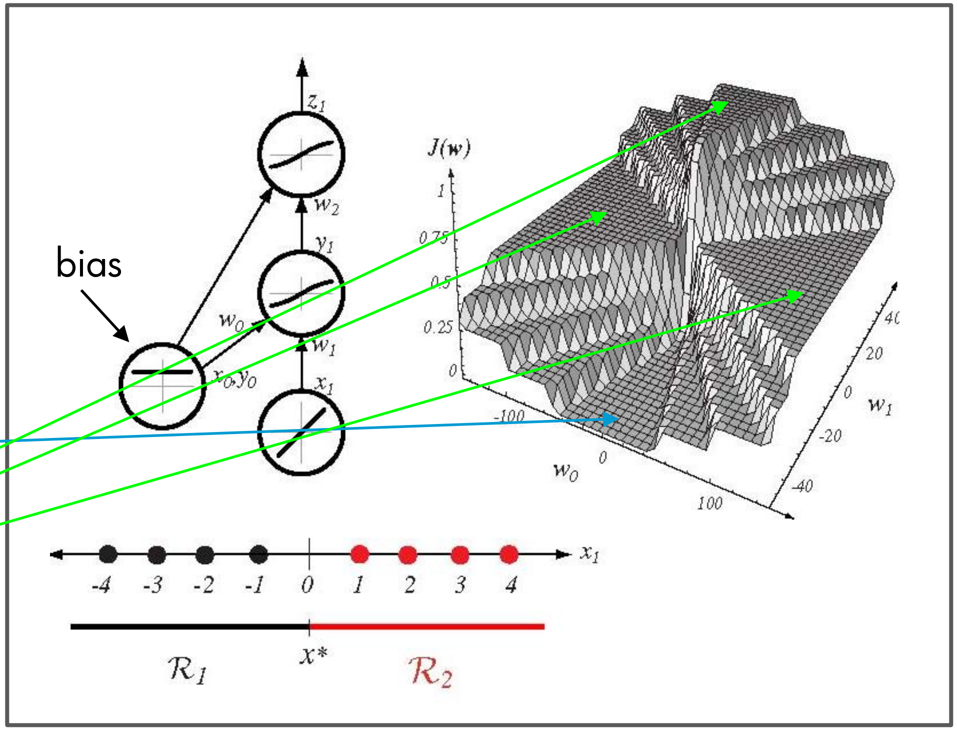
- example
  - two–class problem
  - 2–1 network is **not** enough
  - but 3–1 network **is**

- in practice
  - art–form
  - trial and error



Other Problems

- the optimization surface can be quite <u>nasty</u>
- <u>example</u>
  - scalar problem
  - 1–1 network
- cost has <u>**many**</u> "plateaus"
  - global optimal solution has no error
  - but gradient frequently close to zero
  - <u>slow</u> progress

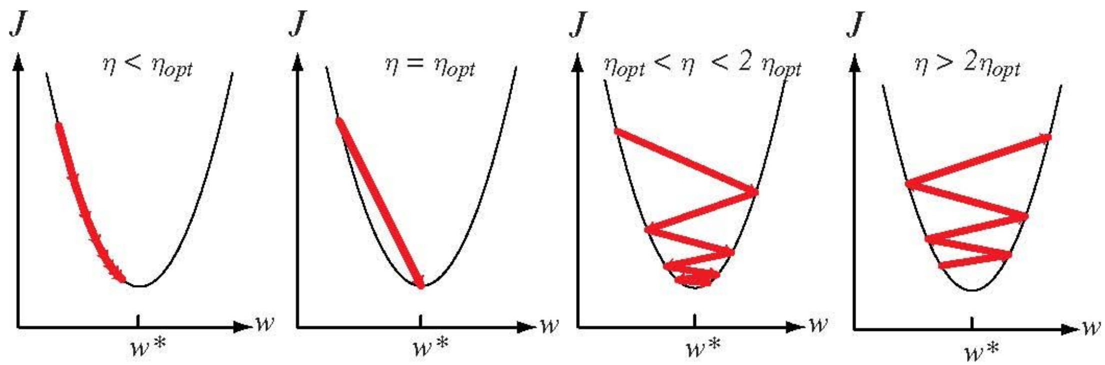


- in general: **one plateau per training example**
  - <u>improves</u> with more **examples**,
  - <u>degrades</u> with more **weights** (dimensions)

Other Problems

- how do we set the learning rate η?
  - if too small or too big, we will need various iterations
  - could even diverge
- **line search:**
  - pick $\eta^{(0)}$
  - compute $\mathbf{x}' = \mathbf{x}^{(n)} - \eta^{(0)} \nabla f\left(\mathbf{x}^{(n)}\right)$ and then $f(\mathbf{x}')$
  - if not good, make $\eta^{(k+1)} = \alpha\, \eta^{(k)}$ (with $\alpha < 1$) and repeat
  - until you get a minimum of $f(\mathbf{x})$

$$\boxed{\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \eta \nabla f\left(\mathbf{x}^{(n)}\right)}$$



Structural Risk Minimization

- what about **complexity penalties**, **overfitting**, and **all that**?

Recall

- SRM, in general:
    1. start from **nested** collection of families of functions $S_1 \subset \cdots \subset S_k$
    2. for each S_i, find the **set of parameters** that minimizes the empirical risk
    3. select the **function class** such that $R^* = \min_i \{R_{emp}^i + \Phi(h_i)\}$,
       where $\Phi(h)$ is a function of the complexity (VC dimension) of the family S_i

- can be done by
    1. family $S_i = \{\mathbf{MLPs}\text{ such that } \sum_{jk} w_{jk}^2 = \|\mathbf{W}\|^2 < \lambda_i\}$
    2. and **backpropagation** in this family

Structural Risk Minimization

- instead of

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

$$J(\mathbf{W}) = \sum_{i=1}^{n} L\big(\mathbf{t}_i, \mathbf{z}(\mathbf{x}_i; \mathbf{W})\big)$$

$$L(\mathbf{t}, \mathbf{z}) = \frac{1}{2}\sum_{k} [t_k - z_k]^2$$

- solve

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \textstyle\sum_{i=1}^{n} L\big(\mathbf{t}_i, \mathbf{z}(\mathbf{x}_i; \mathbf{W})\big) \quad \text{subject to} \quad \mathbf{W}^T\mathbf{W} < \lambda$$

- we will see that this is equivalent to

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \left\{ \sum_{i=1}^{n} L\big(\mathbf{t}_i, \mathbf{z}(\mathbf{x}_i; \mathbf{W})\big) + \frac{2\varepsilon}{\eta}\mathbf{W}^T\mathbf{W} \right\}$$

- re–working out backpropagation, this can be done by "shrinking"
  - **after each weight update**, do $\mathbf{W}^{new} = \mathbf{W}^{old}(1 - \varepsilon)$
  - this is known as weight decay and penalizes complex models

In Summary

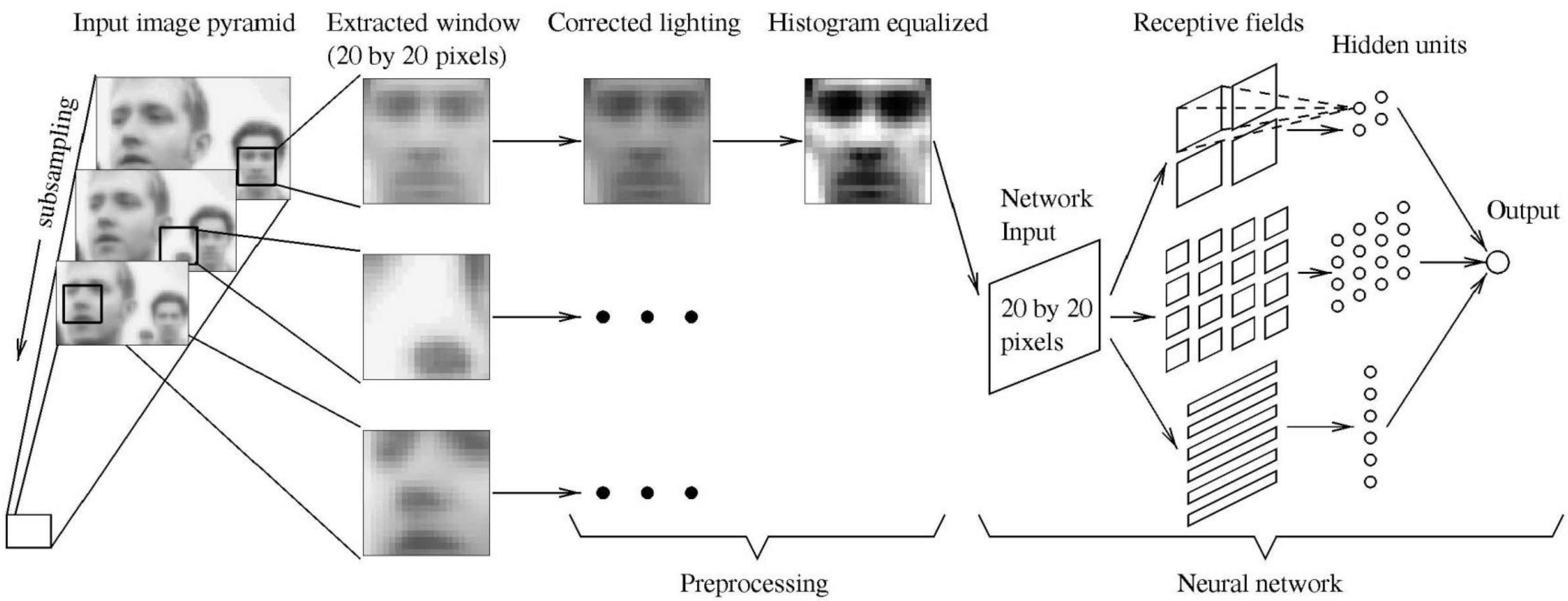
- this works, but requires **tunning** ε
- the **cost surface** is **nasty**
- one needs to try **different architectures**
- hence, training can be **painfully slow**
  - "weeks" is quite common
  - a good neural network may take years to train
- however, when you are finished it **tends to work well**
- the "original" examples
  - the Rowley and Kanade face detector
  - the LeCunn digit recognizer (see http://yann.lecun.com/exdb/lenet/index.html)

Rowley & Kanade

- neural network–based face detection

  Rowley, H.A., Baluja, S., Kanade, T., IEEE Transactions on PAMI, Volume: 20, Issue: 1, Jan 1998

- the face detector:



Results

A: 57/57/1
B: 4/2/0
C: 4/3/0
D: 1/1/1
E: 8/8/0
A: 2/2/0
B: 1/1/0
C: 3/1/2
D: 1/1/0
E: 1/1/0
F: 1/1/0
G: 1/1/1
H: 1/1/0
I: 1/1/1
J: 0/0/0
K: 1/1/0
L: 14/13/0
M: 1/1/0
N: 2/2/0
O: 1/1/0
P: 1/1/0
Q: 3/3/0
R: 1/1/0

Tricks (good for any learning algorithm)

- expand the training set to cover for most variation
  - a **more exhaustive** training set always produces better results than a less exhaustive one
  - if you can create interesting examples artificially, then by all means...
    - e.g., in vision: rotate, scale, translate

  - this is called data augmentation and works **independently** of what algorithm you are using

Tricks (good for <u>any</u> learning algorithm)

- where do I get negative examples?
  - finding a **good** negative example is difficult (e.g., what is a non–face?)
  - use the **classifier itself** to do it:

1. put together training set $\mathcal{D}_1$
2. train classifier C_k with training set $\mathcal{D}_k$
3. run on a dataset that has **no** positive examples
4. make $\mathcal{D}_{k+1}$= {examples classified as positive} ∪ $\mathcal{D}_k$
5. goto 2.

no–faces

  - this is called hard–negative mining
  - e.g., "close" non–face examples