Note:

- **Final Session**: Complete FP#3 on LDA and submit your code and results (screen shots/short doc report) via Canvas **by midnight tomorrow**!
- Project report due in two weeks (Plus 2 days)
- Project presentation in two weeks
  - Submit your slides two days prior (**5/11, 5pm**) by **email**
  - **More information in Canvas now. Please read it.**

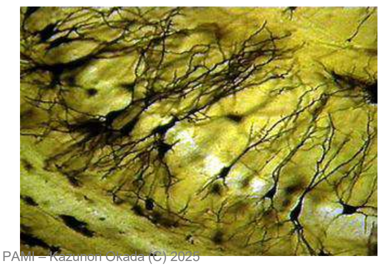
Artificial Neural Network

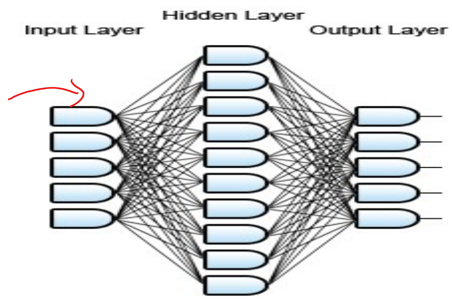
CSC 872

Pattern Analysis and Machine Intelligence

Artificial Neural Network (ANN)

- An information processing paradigm inspired by **biological nervous system** such as human **brain**
- Large number of highly interconnected **processing elements (neurons)** working together
- **Learn from examples** to adapt to new situation
- **Various connections/learning methods** for various applications

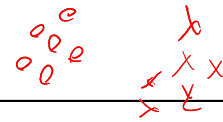




Application

- Pattern Classifications
  - Object & Speech recognition
  - Handwritten letter recognition
  - Credit scoring
- Control
  - Robot
  - Autonomous vehicle
- Time series modeling
  - S&P 500 Index prediction, LBS capital management, FL
  - Natural gas price, Northern Natural Gas, NE
  - Jury summoning prediction, Montgomery Courthouse, PA
- Optimization
  - Multiprocessor scheduling
  - VLSI placement
- Recent Apps Includes Self-Driving and Go-game etc

Basic Types



- Feed-forward Network
- Self-Organizing Map
- Hopfield Network
- Recurrent Network
- Stochastic Network
- Radial Basis Function Network
- Support Vector Machine
- Convolutional Neural Network (DL)

Margin Maximization

Model Selection

Vapnik

Basic Types

- Feed-forward Network **(maps input to outputs)**
- Self-Organizing Map
- Hopfield Network
- Recurrent Network
- Stochastic Network
- Radial Basis Function Network
- Support Vector Machine
- Convolutional Neural Network (DL)

$y = f(x, w)$

Regression & ML

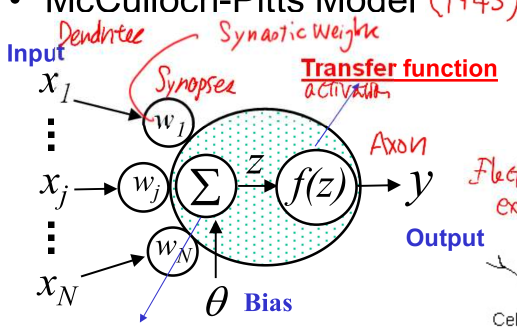
History

- **1943: McCulloch-Pitts Neuron Model**
- 1949: Hebbian Learning (Hebb)
- **1958: Perceptron (Rosenblatt)**
- 1969: Critique of Perceptron (Minsky) → Neuro Winter
- 1976: Adaptive Resonance Theory (ART) (Grossberg)
- 1982: Hopfield Network (Hopfield: associative)
- 1985: Boltzmann machine (Hinton/Sejnowski, simulated annealing)
- **1986: Multilayer Perceptron / Backpropagation (Rumelhart/McClelland)** 2nd Gen
- 1989: Self-Organizing Map (Kohonen)
- 1995: Support Vector Machine (Vapnik)
- 1995: AdaBoost (Freund, Schapire)
- Today: Deep learning 3rd Gen

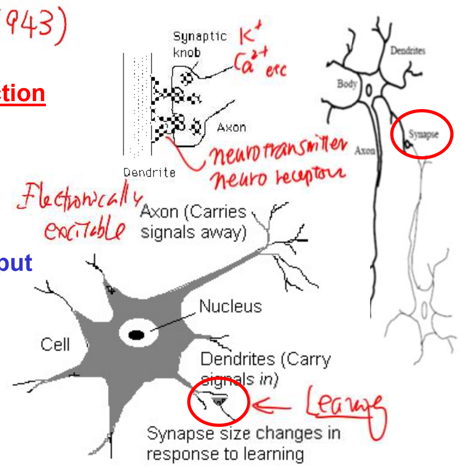
7

Neuron Model

- McCulloch-Pitts Model (1943)



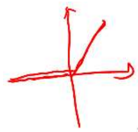
Weighted linear combination of inputs: $z = \Sigma_j^N w_j x_j + \theta$



$$y = f(z) = f(w_1x_1 + \cdots + w_jx_j + \cdots + w_Nx_N + \theta)$$

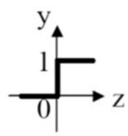
8

Transfer (Activation) Function

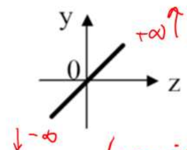


discrete | continuous

Hard Limit: y = 0 if z<0
y = 1 if z>=0



Linear:
y = z

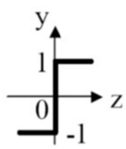


thresholding

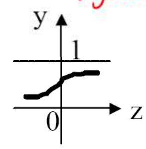
logistic regression

Squashing Function

Symmetrical Hard Limit: y = -1 if z<0
y = +1 if z>=0



Log-Sigmoid:
y = 1/(1+e^{-z})

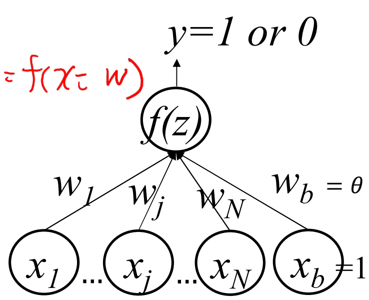


Bounded in [0 1]

$$y = f(z) = f(w_1 x_1 + \cdots + w_j x_j + \cdots + w_N x_N + \theta)$$

Perceptron (1958) $\{(x_i, D_i)\}_i$

- **A simple single-neuron network**
- Use the hard limit (threshold) transfer function $y_i = f(x_i\ w)$
- Change the weight by an amount proportional to the <u>difference between the desired output D_i and the actual output y_i</u>



(Perceptron learning rule)

$$w_{j+1} = w_j + \Delta w_j$$

$$\Delta w_j = \eta(\underbrace{D_i - y_i}_{\text{residual error = "delta"}})x_j$$

$$z = \sum_{j=1}^{N} w_j x_j + \theta$$

$$= \boldsymbol{w}^t \boldsymbol{x} + \theta$$

$$= [\theta \quad \boldsymbol{w}^t] \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix}$$

Perceptron

- **A simple single-neuron network**
- Use the hard limit (threshold)

y=1 or 0

How does it work?
How do we get the learning rule?
For what should we use this for?

Understand it as MLE=LS regression using Gradient Descent …

x_b=1

θ

$$\Delta w_i = \eta(D_i - y_i)x_i$$

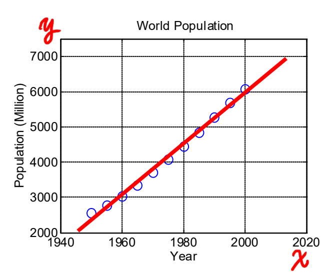
$$= [\theta \quad \boldsymbol{w}^t]\begin{bmatrix}1\\ \boldsymbol{x}\end{bmatrix}$$

Review: Regression

- Assume a regression model: ***y = f(x;w) + e ~ N(0,σ^2)***
- We can fit a function ***f(x;w)*** to data ***{(X$_i$,D$_i$)}*** by …
- **MLE**: find ***w*** that maximizes ***P(Y|X,W) = N(f(x;w),σ^2)***
- **LS**: find ***w*** that minimizes the sum-of-square errors

$$\boldsymbol{w} = \mathrm{argmin}_w \sum_{i=1}^{N} (D_i - f(\boldsymbol{x_i};\boldsymbol{w}))^2$$

$$\Leftrightarrow \frac{\partial}{\partial \boldsymbol{w}} \sum_i (D_i - f(\boldsymbol{x_i};\boldsymbol{w}))^2 = 0$$



- When ***f(x;w)*** is simple we have a closed-form solution for ***w***
- Otherwise we use **Gradient-Descent**

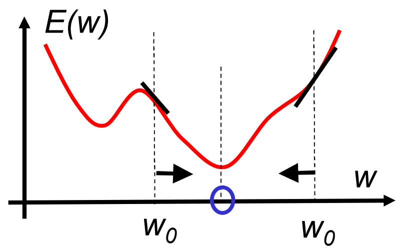
Review: Gradient-Descent

- Negative gradient as an iterative step

step0: $w_{old} \leftarrow w_0$(initialization)

step1: $w_{new} \leftarrow w_{old} - \eta \frac{\partial E(w)}{\partial w}\Big|_{w=w_{old}}$

step2: $w_{old} \leftarrow w_{new}$



Multivariate Gradient-Descent

- Multivariate case: $\boldsymbol{w} = (w_1,..,w_M)$

step0: $\boldsymbol{w}_{old} \leftarrow \boldsymbol{w}_0$ (initialization)

step1: $\boldsymbol{w}_{new} \leftarrow \boldsymbol{w}_{old} - \eta \nabla E(\boldsymbol{w}_{old})$

step2: $\boldsymbol{w}_{old} \leftarrow \boldsymbol{w}_{new}$

$$\nabla E(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} E(\boldsymbol{w}) \\ \frac{\partial}{\partial w_2} E(\boldsymbol{w}) \\ \vdots \\ \frac{\partial}{\partial w_M} E(\boldsymbol{w}) \end{bmatrix}$$

Gradient vector (points to the direction of steepest ascent!)

$$w_j \leftarrow w_j - \eta \frac{\partial}{\partial w_j} E(\boldsymbol{w})$$

where w_j is the j^{th} weights of $\boldsymbol{w}$ vector

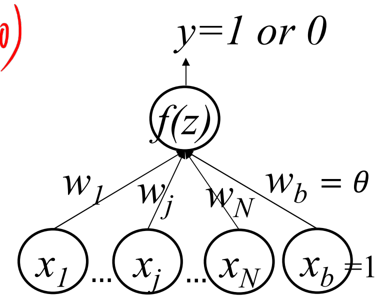
STOP: Simplest case: linear transfer

$y=z$

- Linear perceptron: $y = \boldsymbol{w}^T\boldsymbol{x}$ $(\theta=0)$
- **Same as linear regression!**
- MLE=LS: minimize the sum-of-square errors by gradient descent

$E(w)=$

$$\boldsymbol{w} = \mathrm{argmin}_w \sum_{i=1}^{N} (D_i - \boldsymbol{w}^t\boldsymbol{x}_i)^2$$



$$w_j \leftarrow w_j - \eta \frac{\partial E(\boldsymbol{w})}{\partial w_j}$$

Gradient descent rule

$$E(w) = \sum_{i=1}^{N} (D_i - \boldsymbol{w}^t\boldsymbol{x}_i)^2$$

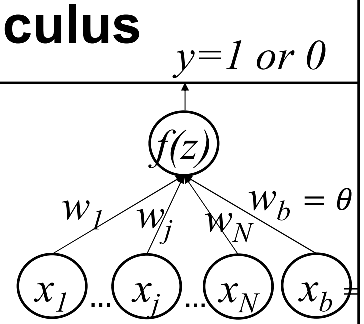
With the sum-of-square errors to be minimized

Simplest case: Do Calculus

$$\boldsymbol{w} = \mathrm{argmin}_w \sum_{i=1}^{N} (D_i - \boldsymbol{w}^t\boldsymbol{x}_i)^2$$

$$w_j \leftarrow w_j - \eta \frac{\partial E(\boldsymbol{w})}{\partial w_j}$$

$$E(w) = \sum_{i=1}^{N} (D_i - \boldsymbol{w}^t\boldsymbol{x}_i)^2$$



$$\frac{\partial E(\boldsymbol{w})}{\partial w_j} = \sum_i 2(D_i - \boldsymbol{w}^t\boldsymbol{x}_i)\frac{\partial}{\partial w_j}(D_i - \boldsymbol{w}^t\boldsymbol{x}_i)$$

$$= -2\sum_i \delta_i \frac{\partial}{\partial w_j}\boldsymbol{w}^t\boldsymbol{x}_i;\ \ \delta_i = D_i - \overset{y_i}{\boldsymbol{w}^t\boldsymbol{x}_i}$$

$w_1x_1 + w_2x_2 + \cdots + w_jx_j + \cdots + w_Nx_N$

$$= -2\sum_i \delta_i \frac{\partial}{\partial w_j}\sum_j w_j x_j$$

$\frac{\partial w_j x_j}{\partial w_j}$

$$= -2\sum_i \delta_i x_j$$

Delta: difference between desired and actual outputs

$$\delta_i \leftarrow D_i - \boldsymbol{w}^t\boldsymbol{x}_i$$

$$w_j \leftarrow w_j + 2\eta \Sigma_i\, \delta_i x_j$$

Neglect 2

This is actually the perceptron leaning rule!!!

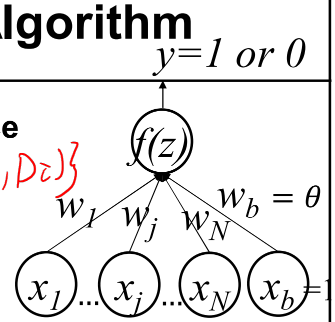
Why Perceptron?

- Perceptron learning rule is also known as
  - Delta rule
  - Windrow Hoff rule
  - LMS rule

$$\delta_i \leftarrow y_i - \boldsymbol{w}^t \boldsymbol{x}_i$$
$$w_j \leftarrow w_j + \eta \delta_i x_j$$

- But linear regression has a closed-form soln. Why GD?
- Advantage of iterative GD
  - Biologically more plausible
  - More easily parallelizable
  - Efficient when there are many feature attributes (large m)
  - When many feature attributes are used, it becomes difficult to do matrix inversion for the direct closed-form solution
- Disadvantage of iterative GD
  - Hard to choose good learning rate
  - You cannot be sure when GD stops (irregular run time speed)
  - Local minimum!

Batch / Online Learning Algorithm



MLE = LS soln!

Batch Algorithm: use all samples at once

1) Randomly initialize weights $w_1,..,w_m,w_b$ $\{(x_i, D_i)\}$
2) Get supervised data set and append 1
3) For all training samples (*i=1 to N*): accumulate error for each sample δ_i $\quad \delta_i \leftarrow y_i - \boldsymbol{w}^t \boldsymbol{x}_i$
4) For all features (j=1toM): update each weight w_j by the delta rule $\quad w_j \leftarrow w_j + \eta \Sigma_i \delta_i x_j$
5) Loop to (3) unless $\sum \delta_i^2$ stops improving

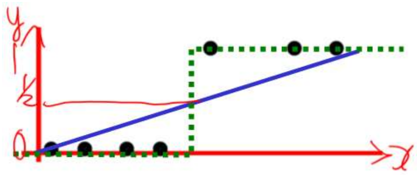
Online Algorithm: one sample at a time

- Each time you observe a sample (**x**, y)
- Update the weights with the error only from the sample

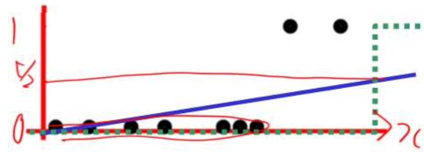
$$w_j \leftarrow w_j + \eta \delta_i x_j$$

Perceptron for Classification

What if all outputs are 0's or 1's ?



or



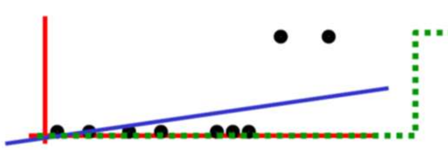
- We can do a linear regression
- Do classification by threshold
  - **0 if y ≤ ½**
  - **1 if y > ½**
- Any problem with this?

Blue = y

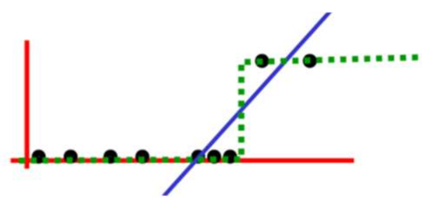
Green = classification

19

Problem is…



Least squares fit is useless

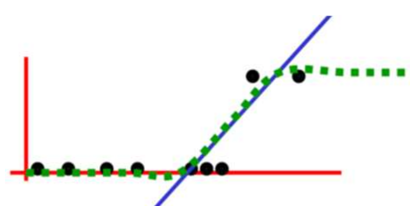


This is much better classification
but it is not a least squares fit

Solution:

Instead of $y = \boldsymbol{w}^T\boldsymbol{x}$
We fit $y = \boldsymbol{g}(\boldsymbol{w}^T\boldsymbol{x})$

Where $\boldsymbol{g}(z)$ is a squashing transfer function $g(z)\colon \mathbb{R} \rightarrow (0,1)$



So let's fit a function (green)
like this!!!

20

Perceptron with Sigmoid function

- Popular example of the squashing function

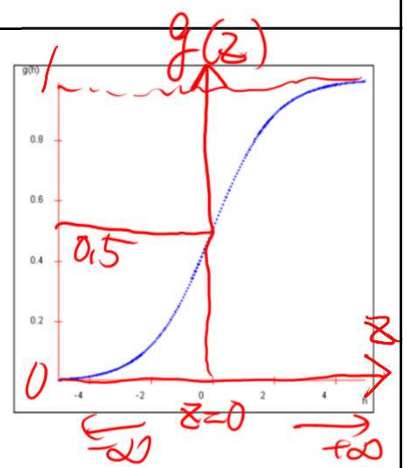
$$g(z) = \frac{1}{1+\exp(-z)}$$

- With nice property

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

- We want to find weights ***w*** that minimizes

$$E(w) = \sum_{i=1}^{N} (D_i - g(\boldsymbol{w}^t \boldsymbol{x}_i))^2$$



Learning Rule with Sigmoid

$$y = g(\boldsymbol{w}^t \boldsymbol{x})$$

$$\frac{\partial E(w)}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial}{\partial w_j}(D_i - g(w^t x_i))^2$$

$$= \sum_{i=1}^{N} 2\big(D_i - g(w^t x_i)\big) - \frac{\partial}{\partial w_j} g(w^t x_i)$$

$$= -2\sum_{i=1}^{N} \big(D_i - g(w^t x_i)\big) g'(w^t x_i) \frac{\partial}{\partial w_j} \sum_j w_j x_{ij}$$

x_{ij}

$$= -2\sum_{i=1}^{N} \delta_i g(w^t x_i)(1 - g(w^t x_i)) x_{ij}$$

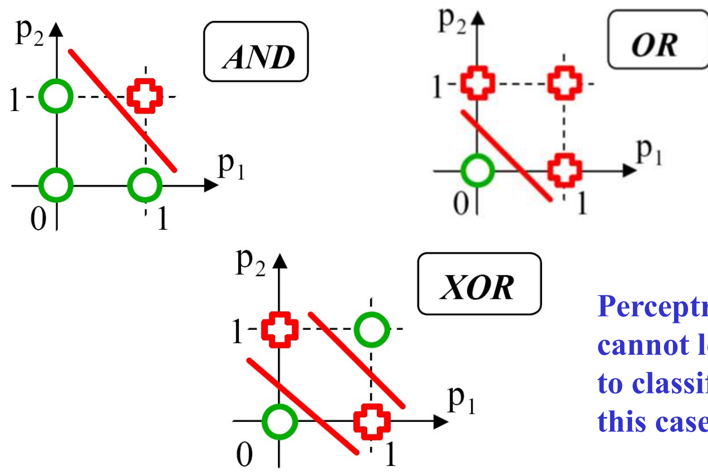
$$w_j \leftarrow w_j + \eta \sum_i \delta_i g_i (1 - g_i) x_{ij}$$

$$\delta_i = D_i - g_i$$

$$g_i = g(\boldsymbol{w}^t \boldsymbol{x}_i)$$

Limitation of Perceptron Minsky

- **Perceptron provides a linear discriminant function**



Perceptron cannot learn to classify this case…

23

Multi-Layer Perceptron (MLP)

The class of functions representable by perceptrons is limited

$$\text{Out}(x) = g(\mathbf{w}^T \mathbf{x}) = g\left(\sum_j w_j x_j\right)$$

Use a wider representation !

$$\text{Out}(x) = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_{jk}\right)\right)$$

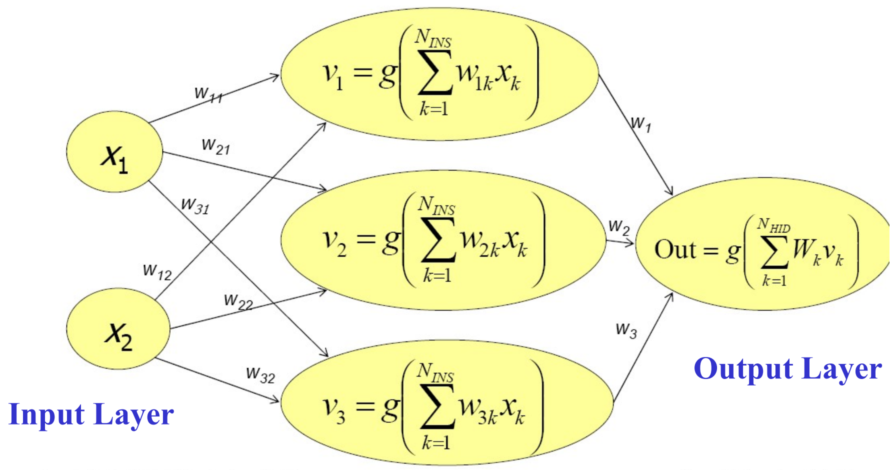
This is a nonlinear function
Of a linear combination
Of non linear functions
Of linear combinations of inputs

 Neural Networks: Slide 51

24

MLP: one-hidden layer net

$N_{INPUTS} = 2$ **Hidden Layer** $N_{HIDDEN} = 3$



Backpropagation Algorithm

$$\text{Out}(\text{x}) = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_k\right)\right)$$

Find a set of weights $\{W_j\},\{w_{jk}\}$

to minimize

$$\sum_i \left(y_i - \text{Out}(\text{x}_i)\right)^2$$

by gradient descent. = Iterative steepest descent!!!

That's it!
That's the backpropagation algorithm.

Backpropagation Learning Rule

- In any ANN book + MATLAB NN Toolkit
- How to actually derive from theory
  - Same as the regular GD but *E(w)* is now an indirect function of weights in the hidden layer(s)
  - Therefore use "**chain rule**" of calculus for deriving the update rules for weights in different (nested) layers

Phase 1

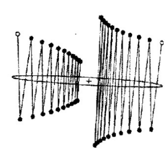
IN {W2} ← {W3} Out

Phase 2

- How to use
  - **Phase 1**: Calculate **sum-of-square errors** (squared differences between the desired <*Di*> and actual network outputs <*yi*>)
  - **Phase 2**: **Update weight from back to front** (hence backpropagation) by computing the partial derivatives using the chain rule

Backpropagation Issues

- **It is GD!** So it may converge at local minimum
- You must find right network topology and structure (number of hidden layers and nodes) by trial & error
- Setting the right learning rate is a subtle art!
  - **TOO SMALL**: it may take long time for convergence
  - **TOO LARGE**: it may diverge and/or oscillate!
  - This is a reason why we like iterative methods without learning rate (e.g., EM, Mean Shift)



- Many methods to make GD work better

Optimization

  - **Momentum**: use past information
  - **Newton's Method:** use quadratic form with 2^{nd} derivative
  - **Conjugate Gradient**: quadratic assumption w/ only 1^{st} derivative

Summary

- Artificial Neural Network
  - Types of Various ANNs
  - Neuron Model
  - Linear Perceptron
  - Delta Learning Rule
  - Sigmoid Perceptron
  - Limitation of Perceptron
  - Multi-Layer Perceptron
  - Back propagation

- Next: Deep Neural Networks
  - Last lectures.
  - No in-class exercises.