Note:

• Complete FP#3 on LDA and submit your code and results (screen shots/short doc report) via iLearn by midnight tomorrow!

• Project report due in three weeks
  – Read the assignment thoroughly
  – Late policy will apply.

• Project presentation in three weeks
  – Submit your slides two days prior (12/6, 5pm) by email
  – Read the assignments again to refresh your mem.
  – More details for presentation will be emailed soon
  – Presentation in alphabetical order

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)

\[ y = f(x, \theta) \]

- Margin Maximization
- Model Selection
- Regression and ML
History

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

Neuron Model

- McCulloch-Pitts Model

\[ y = f(z) = f(w_1x_1 + \cdots + w_jx_j + \cdots + w_Nx_N + \theta) \]
Transfer (Activation) Function

### Discrete

- **Hard Limit:** \( y = 0 \) if \( z < 0 \)
- \( y = 1 \) if \( z \geq 0 \)

### Continuous

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

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

Perceptron

- **A simple single-neuron network**
- Use the hard limit (threshold) transfer function
- Change the weight by an amount proportional to the difference between the desired output \( D_i \) and the actual output \( y_i \)

(Perceptron learning rule)

\[
\begin{align*}
w_{i+1} & = w_i + \Delta w_i \\
\Delta w_i & = \eta (D_i - y_i) x_i
\end{align*}
\]
Perceptron

- A simple single-neuron network
- Use the hard limit (threshold)
- Change the weight by an amount proportional to the difference between the desired output $D_i$ and the actual output $y_i$ (Perceptron learning rule)

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

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 …

Review: Regression

- Assume a regression model: $y = f(x;w) + e \sim N(0, \sigma^2)$
- We can fit a function $f(x;w)$ to data $(X_i, D_i)$ by …
- MLE: find $w$ that maximizes $P(Y|X,W) = N(f(x;w), \sigma^2)$
- LS: find $w$ that minimizes the sum-of-square errors

$w = \arg\min_w \sum_{i=1}^N (D_i - f(x_i;w))^2$

$
\left\{ \frac{\partial}{\partial w} \sum_i (D_i - f(x_i;w))^2 = 0 \right. \left. \right\}
$

- 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
  
  \[
  \text{step0: } w_{\text{old}} \leftarrow w_0 (\text{initialization})
  \]
  
  \[
  \text{step1: } w_{\text{new}} \leftarrow w_{\text{old}} - \eta \frac{\partial E(w)}{\partial w} \bigg|_{w=w_{\text{old}}}
  \]
  
  \[
  \text{step2: } w_{\text{old}} \leftarrow w_{\text{new}}
  \]

**Multivariate Gradient-Descent**

- Multivariate case: \( w = (w_1, \ldots, w_M) \)
  
  \[
  \text{step0: } w_{\text{old}} \leftarrow w_0 \text{ (initialization)}
  \]
  
  \[
  \text{step1: } w_{\text{new}} \leftarrow w_{\text{old}} - \eta \nabla E(w_{\text{old}})
  \]
  
  \[
  \text{step2: } w_{\text{old}} \leftarrow w_{\text{new}}
  \]

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

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

\[
w_j \leftarrow w_j - \eta \frac{\partial E(w)}{\partial w_j}
\]

where \( w_j \) is the \( j^{th} \) weights of \( w \) vector.
Simplest case: linear transfer

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

\[
\begin{align*}
& w = \arg\min_w \sum_{i=1}^N (D_i - w^T x_i)^2 \\
& E(w) = \sum_{i=1}^N (D_i - w^T x_i)^2
\end{align*}
\]

Gradient descent rule

\[
\begin{align*}
& w_j \leftarrow w_j - \eta \frac{\partial E(w)}{\partial w_j} \\
& E(w) = \sum_{i=1}^N (D_i - w^T x_i)^2
\end{align*}
\]

With the sum-of-square errors to be minimized

Simplest case: Do Calculus

\[
\begin{align*}
& w = \arg\min_w \sum_{i=1}^N (D_i - w^T x_i)^2 \\
& E(w) = \sum_{i=1}^N (D_i - w^T x_i)^2
\end{align*}
\]

\[
\begin{align*}
& \frac{\partial E(w)}{\partial w_j} = \sum_i 2(D_i - w^T x_i) \frac{\partial}{\partial w_j} (D_i - w^T x_i) \\
& = -2 \sum_i \delta_i \frac{\partial}{\partial w_j} w^T x_i; \hspace{0.5cm} \delta_i = D_i - w^T x_i \\
& = -2 \sum_i \delta_i \frac{\partial}{\partial w_j} \sum_j w_j x_j \\
& = -2 \sum_i \delta_i x_j
\end{align*}
\]

\[
\begin{align*}
& \delta_i \leftarrow D_i - w^T x_i \\
& w_j \leftarrow w_j + 2\eta \sum \delta_i x_i
\end{align*}
\]

This is actually the perceptron leaning rule!!!
Why Perceptron?

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

• 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!

\[ \delta_i \leftarrow y_i - w^T x_i \]
\[ w_j \leftarrow w_j + \eta \delta_i x_i \]

Batch / Online Learning Algorithm

Batch Algorithm: use all samples at once
1) Randomly initialize weights \( w_1, \ldots, w_m, w_b \)
2) Get supervised data set and append 1
3) For all training samples \( (x_1, y_1) \) to \( (x_N, y_N) \): accumulate
   error for each sample \( \delta_i \leftarrow y_i - w^T x_i \)
4) For all features \( (j=1 \text{ to } M) \): update each weight \( w_j \)
   by the delta rule \( w_j \leftarrow w_j + \eta \sum_i \delta_i x_i \)
5) Loop to (3) unless \( \sum \delta^2 \) 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
  \[ w_j \leftarrow w_j + \eta \delta_i x_i \]
Perceptron for Classification

What if all outputs are 0’s or 1’s?

- We can do a linear regression
- Do classification by threshold
  - 0 if \( y \leq 1/2 \)
  - 1 if \( y > 1/2 \)

- Any problem with this?

Problem is…

Least squares fit is useless

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

Solution:

Instead of \( y = w^T x \)
We fit \( y = g(w^T x) \)

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

So let’s fit a function (green) like this!!!
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
  \[ E(w) = \sum_{i=1}^{N} (D_i - g(w^t x_i))^2 \]

Learning Rule with Sigmoid

\[ y = g(w^t x)_N \]
\[ \frac{\partial E(w)}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial}{\partial w_j} (D_i - g(w^t x_i))^2 \]
\[ = \sum_{i=1}^{N} 2(D_i - g(w^t x_i)) \frac{\partial g(w^t x_i)}{\partial w_j} \]
\[ = -2 \sum_{i=1}^{N} (D_i - g(w^t x_i)) g'(w^t x_i) \frac{\partial}{\partial w_j} \sum_j w_j x_{ij} \]
\[ = -2 \sum_{i=1}^{N} \delta_i g(w^t x_i)(1 - g(w^t x_i)) x_{ij} \]

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

where
\[ \delta_i = D_i - g_i \]
\[ g_i = g(w^t x_i) \]
Limitation of Perceptron

- Perceptron provides a linear discriminant function

![Perceptron examples]

Perceptron cannot learn to classify this case...

Multi-Layer Perceptron (MLP)

The class of functions representable by perceptrons is limited

\[ \text{Out}(x) = g(w^T x) = g\left( \sum w_j x_j \right) \]

Use a wider representation!

\[ \text{Out}(x) = g\left( \sum W_j g\left( \sum w_{jk} x_k \right) \right) \]

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

Copyright © 2001, 2003, Andrew W. Moore
Neural Networks: Slide 51
MLP: one-hidden layer net

\[ v_1 = g\left( \sum_{k=1}^{N_{\text{INS}}} w_{1k} x_k \right) \]
\[ v_2 = g\left( \sum_{k=1}^{N_{\text{INS}}} w_{2k} x_k \right) \]
\[ v_3 = g\left( \sum_{k=1}^{N_{\text{INS}}} w_{3k} x_k \right) \]
\[ \text{Out} = g\left( \sum_{k=1}^{N_{\text{INS}}} w_{4k} v_k \right) \]

Backpropagation Algorithm

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

Find a set of weights \( \{W_j\}, \{w_{jk}\} \)
to minimize
\[ \sum_i \left( y_i - \text{Out}(x_i) \right)^2 \]
by gradient descent.

That’s it! Iterative steepest descent!!!
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
- How to use
  - **Phase 1**: Calculate sum-of-square errors (squared differences between the desired $<D_i>$ and actual network outputs $<y_i>$)
  - **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
  - **Momentum**: use past information
  - **Newton’s Method**: use quadratic form with 2nd derivative
  - **Conjugate Gradient**: quadratic assumption w/ only 1st 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
  – Enjoy thanksgiving break!
  – Last lectures on the state of the art.
  – No in-class exercises.