CSE/NB 528

Lecture 13: Supervised Learning

(Chapter 8)

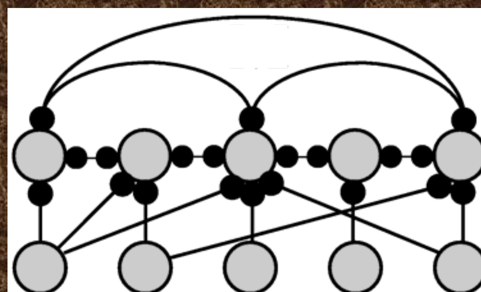
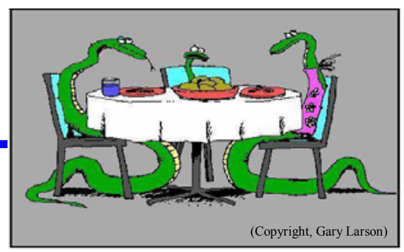


Image from http://clasdean.la.asu.edu/news/images/ubep2001/neuron3.jpg

Lecture figures are from Dayan & Abbott's book

http://people.brandeis.edu/~abbott/book/index.html

What's on the menu today?



(Copyright, Gary Larson)

"Oh, brother! . . . Not hamsters again!"

- Unsupervised Learning
  - Sparse Coding and ICA
- Supervised Learning
  - Why supervised learning?
    - Classification
    - Function Approximation
  - Perceptrons & Learning Rule
  - Linear Separability: Minsky-Papert deliver the bad news
  - Multilayer networks to the rescue
  - Function Approximation
  - Backpropagating (errors)

Unsupervised Learning: Sparse Coding and ICA

- Suppose input **u** is represented by linear superposition of causes $v_1, v_2, \ldots, v_k$ and "features" $\mathbf{g}_i$:

$$\mathbf{u} = \sum_i \mathbf{g}_i v_i = G\mathbf{v}$$

- Problem: For a set of inputs **u**, estimate causes v_i for each **u** and learn feature vectors $\mathbf{g}_i$ (also called basis vectors/filters)
- Idea: Find **v** and G that minimize reconstruction errors:

$$E = \frac{1}{2}|\mathbf{u} - \sum_i \mathbf{g}_i v_i|^2 = \frac{1}{2}(\mathbf{u} - G\mathbf{v})^T(\mathbf{u} - G\mathbf{v})$$

Probabilistic Interpretation

- E is the same as the negative log likelihood of data
  - ➩ Likelihood = Gaussian with mean $G\mathbf{v}$ and covariance I

$$p[\mathbf{u} \mid \mathbf{v}; G] = N(\mathbf{u}; G\mathbf{v}, I)$$

$$E = -\ln p[\mathbf{u} \mid \mathbf{v}; G] = \frac{1}{2}(\mathbf{u} - G\mathbf{v})^T(\mathbf{u} - G\mathbf{v}) + C$$

- Find **v** and G that maximize:

$$F(\mathbf{v}, G) = \langle \ln p[\mathbf{v}, \mathbf{u}; G] \rangle$$ Joint probability of **v** and **u**

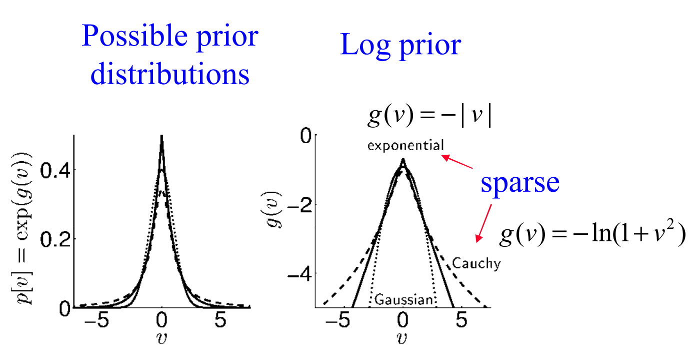
$$= \langle \ln p[\mathbf{u} \mid \mathbf{v}; G] + \ln p[\mathbf{v}; G] \rangle$$

Prior for causes (what should this be?)

What do we know about the causes **v**?

- We would like the causes to be *independent*
  - If cause A and cause B always occur together, then perhaps they should be treated as a single cause AB?
- Examples:
  - Image: Composed of several independent edges
  - Sound: Composed of independent spectral components
  - Objects: Composed of several independent parts
- Idea 1: We would like: $p[\mathbf{v};G] = \prod_a p[v_a;G]$
- Idea 2: If causes are independent, only a few of them will be active for any input → v_a will be 0 most of the time but high for certain inputs → sparse distribution for $p[v_a;G]$

Prior Distributions for Causes



$$p[\mathbf{v};G] \propto \prod_a \exp(g(v_a))$$

Finding the optimal **v** and *G*

- Want to maximize:

$$F(\mathbf{v},G) = \langle \ln p[\mathbf{u} \mid \mathbf{v};G] + \ln p[\mathbf{v};G] \rangle$$

$$= \left\langle -\frac{1}{2}(\mathbf{u}-G\mathbf{v})^T(\mathbf{u}-G\mathbf{v}) + \sum_a g(v_a) \right\rangle + K$$

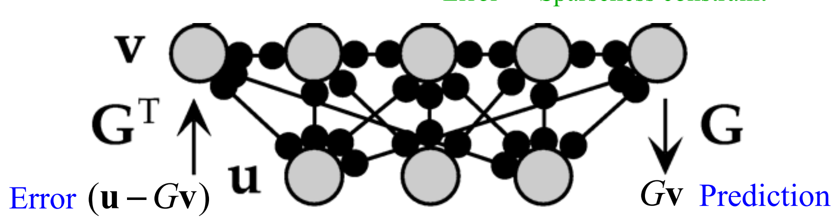
- Alternate between:
  1. Maximize *F* with respect to **v** keeping G fixed
     - Set d**v**/dt ∝ d*F*/d**v** ("gradient ascent/hill-climbing")
  2. Maximize *F* with respect to G, given the **v** above
     - Set d*G*/dt ∝ d*F*/d*G* ("gradient ascent/hill-climbing")

Network for Estimating **v** and Learning G

$$\tau \frac{d\mathbf{v}}{dt} = \frac{dF}{d\mathbf{v}} = G^T(\mathbf{u}-G\mathbf{v}) + g'(\mathbf{v})$$

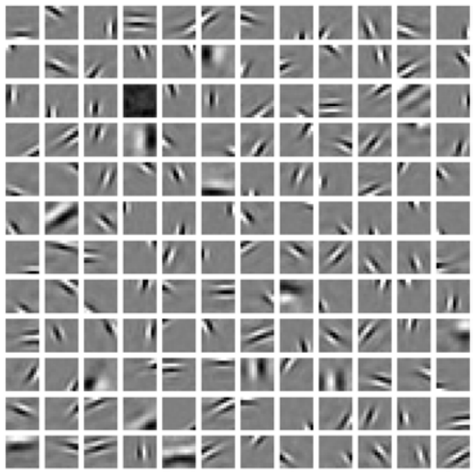
Firing rate dynamics

Error — Sparseness constraint



Learning rule $\tau_G \frac{dG}{dt} = \frac{dF}{dG} = (\mathbf{u}-G\mathbf{v})\mathbf{v}^T$ } Hebbian! (similar to Oja's rule)

Results of Learning G for Natural Images



Each square is a column $\mathbf{g}_i$ of G (obtained by collapsing rows of the square into a vector)

Almost all the $\mathbf{g}_i$ represent local edge features

Any image $\mathbf{u}$ can be expressed as:

$$\mathbf{u} = \sum_i \mathbf{g}_i v_i = G\mathbf{v}$$

What if there is a "teacher" telling you the desired output for each input?

Can you learn to generalize to novel inputs?

Supervised Learning

- Two Primary Tasks

1. Classification

- Inputs u_1, u_2, … and discrete classes C_1, C_2, …, C_k
- Training examples: (u_1, C_2), (u_2, C_7), etc.
- Learn the mapping from an arbitrary input to its class
- Example: Inputs = images, output classes = face, not a face

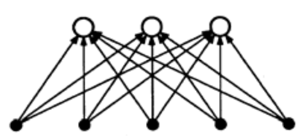
2. Function Approximation (regression)

- Inputs u_1, u_2, … and continuous outputs v_1, v_2, …
- Training examples: (input, desired output) pairs
- Learn to map an arbitrary input to its corresponding output
- Example: Highway driving
  Input = road image, output = steering angle

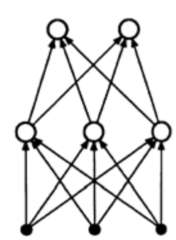
Classification using "Perceptrons"

- Fancy name for a type of layered feedforward networks
- Uses artificial neurons ("units") with binary inputs and outputs

Single-layer



Multilayer

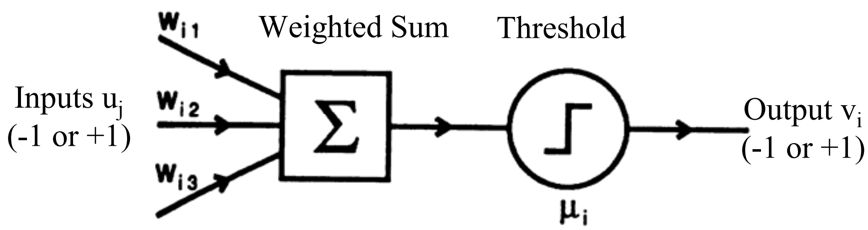


Perceptrons use "Threshold Units"

- Artificial neuron:
  - m binary inputs (-1 or 1) and 1 output (-1 or 1)
  - Synaptic weights w_{ij}
  - Threshold μ_i

$$v_i = \Theta(\sum_j w_{ij} u_j - \mu_i)$$

$\Theta(x) = 1$ if $x \geq 0$ and -1 if $x < 0$



What does a Perceptron compute?

- Consider a single-layer perceptron
  - Weighted sum forms a *linear hyperplane*

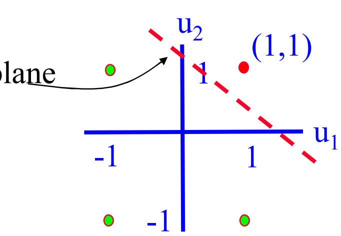
$$\sum_j w_{ij} u_j - \mu_i = 0$$

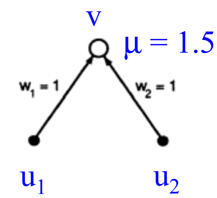
  - Everything *on one side* of hyperplane is in class 1 (output = +1) and everything *on other side* is class 2 (output = -1)
  - Any function that is linearly separable can be computed by a perceptron

Linear Separability

- Example: AND is linearly separable
  - ➪ a AND b = 1 if and only if a = 1 and b = 1

Linear hyperplane





Perceptron for AND

Perceptron Learning Rule

- Given inputs **u** and desired output v^d, adjust **w** as follows:

  1. Compute error signal $e = (v^d - v)$ where v is the current output

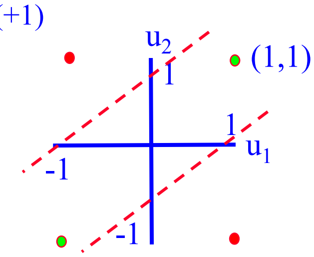
  2. Change weights according to the error e
     ⇒ For positive inputs, increase weights if error is positive and decrease if error is negative (opposite for negative inputs)

$$\mathbf{w} \rightarrow \mathbf{w} + \varepsilon(v^d - v)\mathbf{u}$$

$A \rightarrow B$ means replace A with B

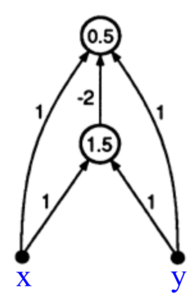
Linear Inseparability

- Single-layer perceptron with threshold units fails if classification task is not linearly separable
  - Example: XOR
  - a XOR b = 1 iff (a = -1, b = 1) or (a = 1, b = -1)
  - No single line can separate the "yes" (+1) outputs from the "no" (-1) outputs!



Solution in 1980s: Multilayer perceptrons

- Removes limitations of single-layer networks
  - Can solve XOR
- An example of a two-layer perceptron that computes XOR



- Output is +1 if and only if $x + y - 2\Theta(x + y - 1.5) - 0.5 > 0$

(Here, inputs x, y are assumed to be 0 or 1)

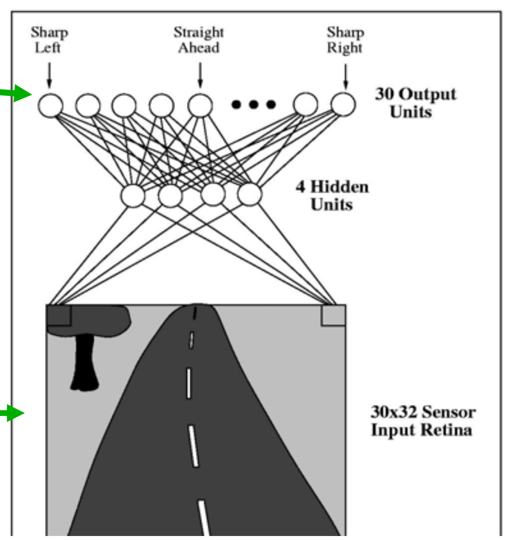
What if you want to approximate a continuous function?

Can a network learn to drive?

Example Network

Get steering angle from a human driver

Desired Output:
$\mathbf{d} = (d_1\ d_2\ \ldots\ d_{30})$

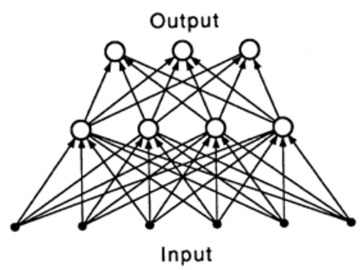


Get current camera image

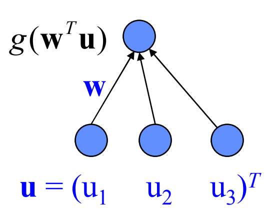
Input $\mathbf{u} = (u_1\ u_2\ \ldots\ u_{960})$ = image pixels

Function Approximation

- We want networks that can <u>learn a function</u>
  - Network maps real-valued inputs to real-valued outputs
  - Want to generalize to predict outputs for new inputs
  - <u>Idea</u>: Given input data, *minimize errors* between network's output and desired output by *adapting weights*



Sigmoidal Networks



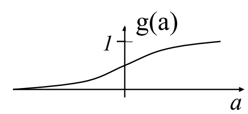
Output

Input nodes

The most common activation function:

Sigmoid function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



(non-linear "squashing" function)

Gradient-Descent Learning ("Hill-Climbing")

- Given training examples ($\mathbf{u}^m$, d^m) (m = 1, …, N), define an error function (cost function or "energy" function)

$$E(\mathbf{w}) = \frac{1}{2}\sum_m (d^m - v^m)^2 \qquad v^m = g(\mathbf{w}^T\mathbf{u}^m)$$

- Would like to change $\mathbf{w}$ so that $E(\mathbf{w})$ is minimized
  - ➩ Gradient Descent: Change $\mathbf{w}$ in proportion to $-dE/d\mathbf{w}$ (why?)

$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE}{d\mathbf{w}}$$

$$\frac{dE}{d\mathbf{w}} = -\sum_m (d^m - v^m)\frac{dv^m}{d\mathbf{w}} = -\sum_m (d^m - v^m) g'(\mathbf{w}^T\mathbf{u}^m)\mathbf{u}^m$$

"Stochastic" (or On-line) Gradient Descent

- What if the inputs only arrive one-by-one?
- Stochastic gradient descent approximates sum over all inputs with an "on-line" running sum:

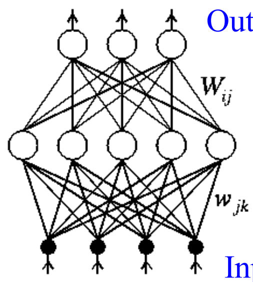
$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

$$\frac{dE_1}{d\mathbf{w}} = -\underbrace{(d^m - v^m)}_{\text{delta = error}} g'(\mathbf{w}^T\mathbf{u}^m)\mathbf{u}^m$$

Also known as the "delta rule" or "LMS rule"

But wait….

- Delta rule tells us how to modify the connections from input to output (one layer network)
  - One layer networks are not that interesting (remember XOR?)
- What if we have multiple layers?



Output $\mathbf{v} = (v_1\ v_2 \ldots v_J)^T$; Desired = $\mathbf{d}$

How do we adapt these "hidden" connections?

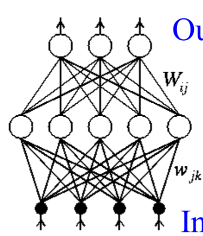
Input $\mathbf{u} = (u_1\ u_2 \ldots u_K)^T$

Let's Backpropagate (Errors)

- Backpropagation = gradient-descent learning for multilayer feedforward networks
- Idea: Propagate credit/blame for errors back to internal nodes
  - Use chain rule (from calculus) to change weights for internal "hidden" nodes

error = $(\mathbf{d} - \mathbf{v})$

Backpropagate this to correct all weights



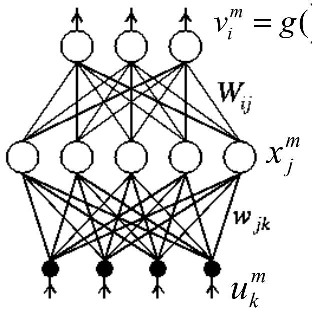
Output $\mathbf{v} = (v_1\ v_2 \ldots v_J)^T$

Delta rule

Backprop rule

Input $\mathbf{u} = (u_1\ u_2 \ldots u_K)^T$

Notation for Backprop



$$v_i^m = g\Big(\sum_j W_{ij}\, g\big(\underbrace{\sum_k w_{jk} u_k^m}_{}\big)\Big)$$

x_j^m

Find **W** and **w** that minimize total squared output error:

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_m \|\mathbf{d}^m - \mathbf{v}^m\|^2$$

$$= \frac{1}{2}\sum_{m,i} (d_i^m - v_i^m)^2$$

Backpropagation (for Math lovers' eyes only!)

- Learning rule for hidden-output connection weights:

$$W_{ij} \rightarrow W_{ij} - \varepsilon \frac{\partial E}{\partial W_{ij}}$$

$$\frac{dE}{dW_{ij}} = -\sum_m (d_i^m - v_i^m)\, g'\Big(\sum_j W_{ij} x_j^m\Big) x_j^m$$

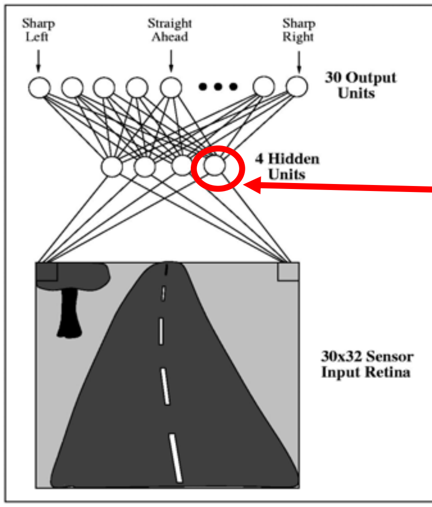
Delta rule

- **Backpropagation rule** for input-hidden connection weights:

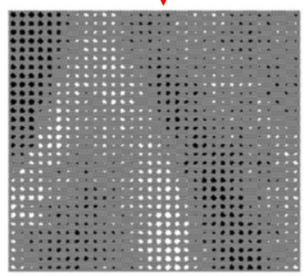
$$w_{jk} \rightarrow w_{jk} - \varepsilon \frac{\partial E}{\partial w_{jk}} \quad \text{But}: \frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial x_j^m} \cdot \frac{\partial x_j^m}{\partial w_{jk}} \quad \{\textit{chain rule}\}$$

$$\frac{dE}{dw_{jk}} = -\sum_{m,i} (d_i^m - v_i^m)\, g'\Big(\sum_j W_{ij} x_j^m\Big) W_{ij} \cdot g'\Big(\sum_k w_{jk} u_k^m\Big) u_k^m$$

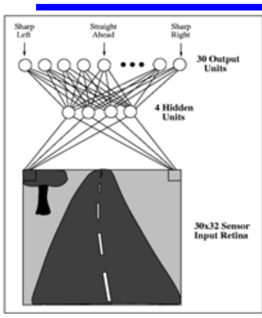
Learning to Drive using Backprop



One of the learned "road features" w_i



ALVINN (Autonomous Land Vehicle in a Neural Network)



CMU Navlab

Trained using human driver + camera images

After learning:

- Drove up to 70 mph on highway
- Up to 22 miles without intervention
- Drove cross-country largely autonomously

(Pomerleau, 1992)

Next Class: Reinforcement Learning

- Things to do:
  - Read Chapter 9
  - Finish Last Homework (due Thu, May 24)
  - Work on mini-project