Neural Networks: Basic Concepts

Mark Craven and David Page
Computer Sciences 760
Spring 2019

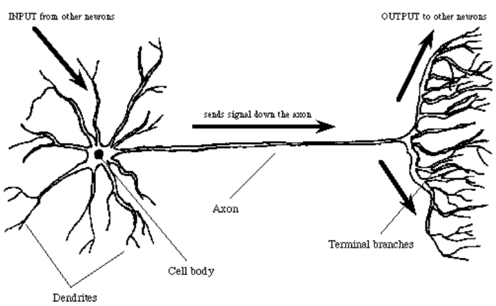
Goals for the lecture

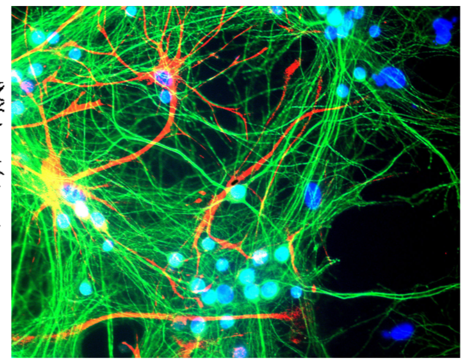
you should understand the following concepts

- perceptrons
- the perceptron training rule
- linear separability
- hidden units
- multilayer neural networks
- gradient descent
- stochastic (online) gradient descent
- activation functions
  - sigmoid, hyperbolic tangent, ReLU
- loss functions
  - squared error, cross entropy
- logistic regression

Neural networks

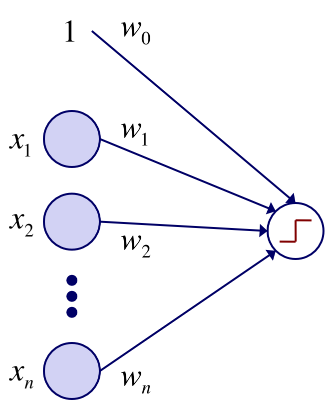
- a.k.a. *artificial neural networks*, *connectionist models*
- inspired by interconnected neurons in biological systems
  - simple processing units
  - each unit receives a number of real-valued inputs
  - each unit produces a single real-valued output





Perceptrons

[McCulloch & Pitts, 1943; Rosenblatt, 1959; Widrow & Hoff, 1960]

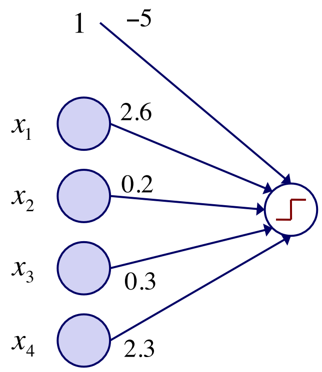


$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

input units:
represent given $\boldsymbol{x}$

output unit:
represents binary classification

Perceptron example



$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

features, class labels are represented numerically

$$\mathbf{x} = \langle 1,0,0,1 \rangle \qquad w_0 + \sum_{i=1}^{n} w_i x_i = -0.1 \qquad o = 0$$

Learning a perceptron: the perceptron training rule

1. randomly initialize weights

2. iterate through training instances until convergence

   2a. calculate the output for the given instance

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

   2b. update each weight

$$\Delta w_i = \eta \left(y - o \right) x_i$$

η is *learning rate*;
set to value << 1

$$w_i \leftarrow w_i + \Delta w_i$$

Representational power of perceptrons

perceptrons can represent only *linearly separable* concepts

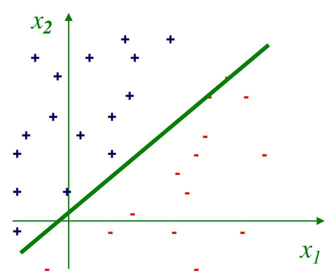
$$o = \begin{cases} 1 \text{ if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 \text{ otherwise} \end{cases}$$

decision boundary given by:

$$1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 > 0$$

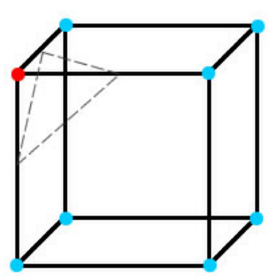
$$w_1 x_1 + w_2 x_2 = -w_0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$



Representational power of perceptrons

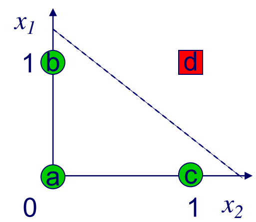
- in previous example, feature space was 2D so decision boundary was a line
- in higher dimensions, decision boundary is a hyperplane



Some linearly separable functions

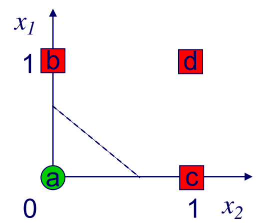
AND

	x_1	x_2	y
a	0	0	0
b	0	1	0
c	1	0	0
d	1	1	1



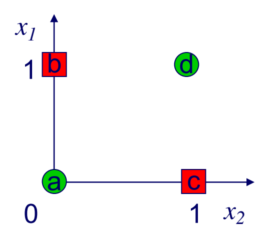
OR

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	1

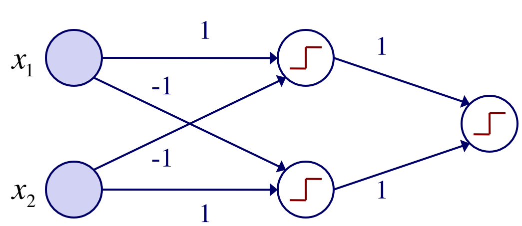


XOR is not linearly separable

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	0

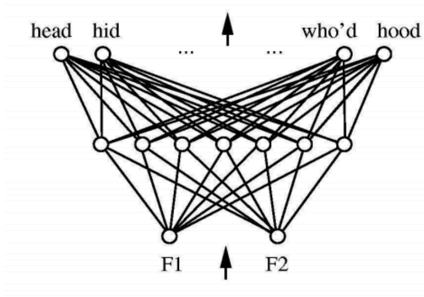


a multilayer perceptron
can represent XOR



assume $w_0 = 0$ for all nodes

Example multilayer neural network



output units

hidden units

input units

figure from Huang & Lippmann, *NIPS* 1988

input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context “h__d”

Decision regions of a multilayer neural network

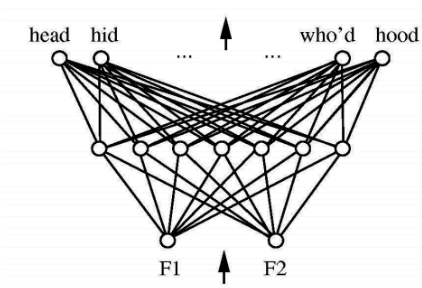
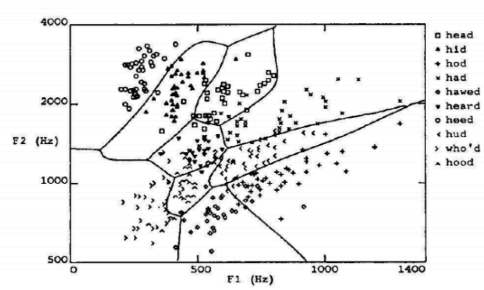


figure from Huang & Lippmann, *NIPS* 1988

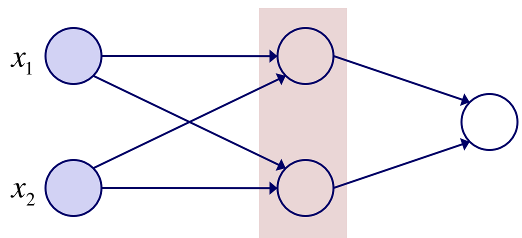


input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context “h__d”

Learning in multilayer networks

- work on neural nets fizzled in the 1960's
  - single layer networks had representational limitations (linear separability)
  - no effective methods for training multilayer networks



how to determine error signal for *hidden units?*

- revived again with the invention of *backpropagation* method [Rumelhart & McClelland, 1986; also Werbos, 1975]
  - key insight: require neural network to be differentiable; use *gradient descent*

Gradient descent general idea

- Specify a loss function that we want to optimize
- Specify a neural network in which the weights are differentiable w.r.t. the loss function
- Iteratively update the weights to minimize the loss function, using derivatives to guide each step

Gradient descent in weight space

Given a training set $D = \left\{(\boldsymbol{x}^{(1)}, y^{(1)}) \ldots (\boldsymbol{x}^{(m)}, y^{(m)})\right\}$ we can specify a *loss function* that is a function of our weight vector $\boldsymbol{w}$

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{d \in D} \left(y^{(d)} - o^{(d)} \right)^2$$

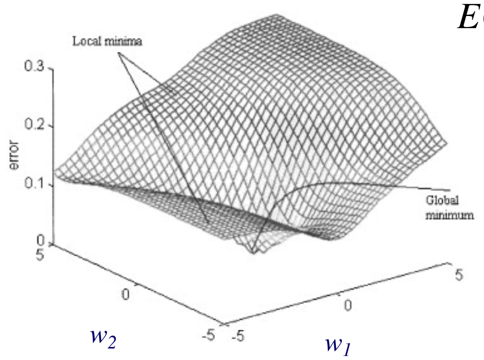


figure from Cho & Chow, *Neurocomputing* 1999

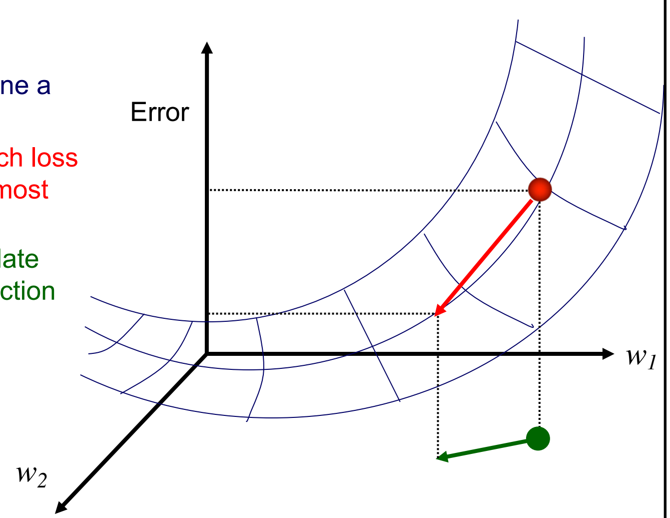
This loss function defines a surface over the model (i.e. weight) space

Gradient descent in weight space

gradient descent is an iterative process aimed at finding a minimum in the loss function

on each iteration

- current weights define a point in this space
- find direction in which loss function descends most steeply
- take a step (i.e. update weights) in that direction



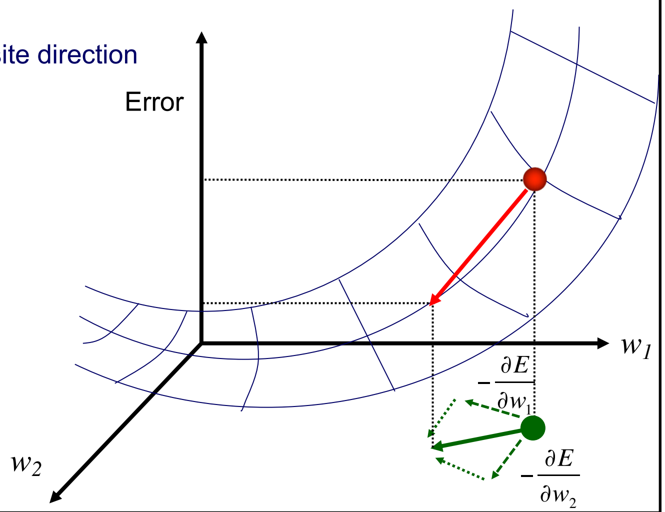
Gradient descent in weight space

calculate the gradient of E: $\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$

take a step in the opposite direction

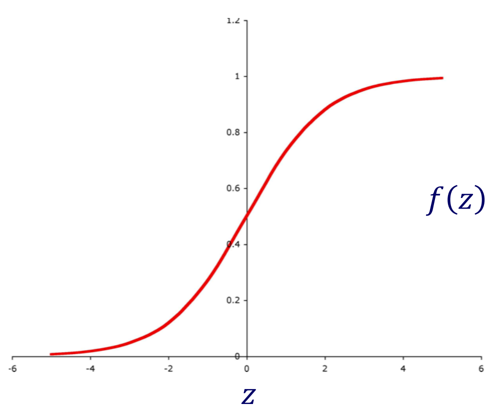
$$\Delta \mathbf{w} = -\eta \, \nabla E(\mathbf{w})$$

$$\Delta w_i = -\eta \, \frac{\partial E}{\partial w_i}$$



The sigmoid function

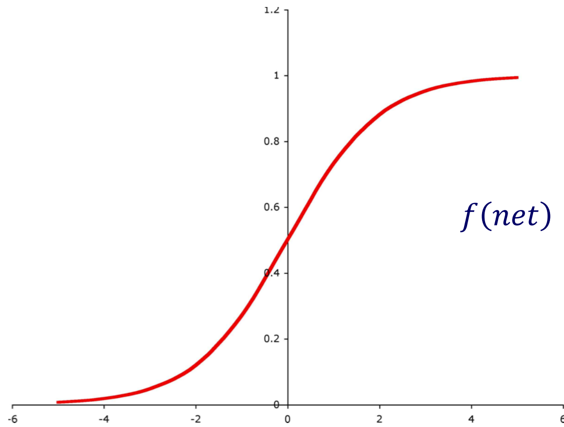
- to be able to differentiate E with respect to w_i , our network must represent a continuous function
- one choice is to use *sigmoid functions* instead of threshold functions in our hidden and output units



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

The sigmoid function

for the case of a single-layer network



$$f(net) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$

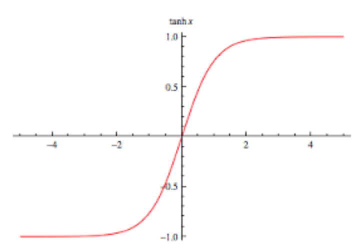
$$net = w_0 + \sum_i w_i x_i$$

Other activation functions

- the sigmoid is just one choice for an *activation function*
- there are others we can use including

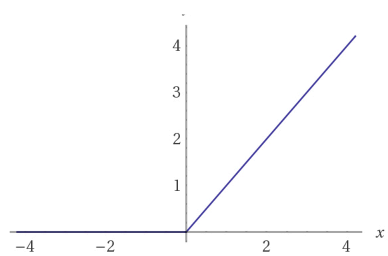
hyperbolic tangent

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$



rectified linear (ReLU)

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$



Other loss functions

- squared error is just one choice for an *loss function*
- there are others we can use including

cross entropy

$$E(\boldsymbol{w}) = \sum_{d \in D} -y^{(d)} \ln\left(o^{(d)}\right) - \left(1 - y^{(d)}\right) \ln\left(1 - o^{(d)}\right)$$

multiclass cross entropy

$$E(\boldsymbol{w}) = -\sum_{d \in D} \sum_{i=1}^{\#\, classes} y_i^{(d)} \ln\left(o_i^{(d)}\right)$$

Batch neural network training

given: network structure and a training set $D = \left\{ (\boldsymbol{x}^{(1)}, y^{(1)}) \ldots (\boldsymbol{x}^{(m)}, y^{(m)}) \right\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

initialize the error $E(\boldsymbol{w}) = 0$

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

increment the error $E(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$

calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E\left(\boldsymbol{w}\right)$$

Online vs. batch training

- Standard gradient descent (batch training): calculates error gradient for the entire training set, before taking a step in weight space
- *Stochastic gradient descent* (online training): calculates error gradient for a single instance (or a small set of instances, a *mini-batch*), then takes a step in weight space
  – much faster convergence
  – less susceptible to local minima

Online neural network training (stochastic gradient descent)

given: network structure and a training set $D = \left\{ (\boldsymbol{x}^{(1)}, y^{(1)}) \ldots (\boldsymbol{x}^{(m)}, y^{(m)}) \right\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

calculate the error $E(\boldsymbol{w}) = \frac{1}{2}\left(y^{(d)} - o^{(d)} \right)^2$

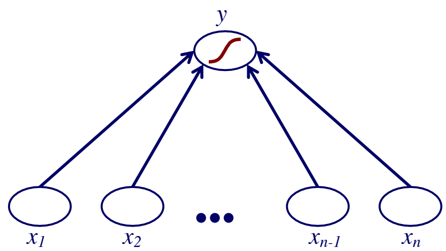
calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E\left(\boldsymbol{w}\right)$$

Logistic regression



- a single layer neural net with a sigmoid in which the weights are trained to minimize cross entropy

$$E(\boldsymbol{w}) = -\sum_{d \in D} \ln P\left(y^{(d)} | \boldsymbol{x}^{(d)}\right)$$

$$= \sum_{d \in D} -y^{(d)} \ln\left(o^{(d)}\right) - \left(1 - y^{(d)}\right) \ln\left(1 - o^{(d)}\right)$$

- the name is a misnomer since LR is used for classification