# Neural Networks

Burr H. Settles
CS-540, UW-Madison
www.cs.wisc.edu/~cs540-1
Summer 2003



## Announcements (7/18)

- Homework #4 ("ML part 1") is out
  - Will be due Monday, 7/28
  - Homework #5 ("ML part 2") should be out on Monday, and due the same day
- Tournament results for Homework #2 should ready by Monday!



## Announcements (7/21)

- TAs have had a delay in HW#2 grading
  - Hopefully it will be done and the results will be in by tomorrow!
- Read 20.1-20.2 in *AI: A Modern Approach*
- HW#5 out this week
  - Due Monday 7/28 along with HW#4



## Neural Networks

- Neural networks (NNs) are AI models that try to mimic the brain in the way it stores knowledge and processes information
- Also known as:
  - Artificial Neural Networks (ANNs)
  - Connectionist Learning Models
    - As opposed the symbolic models, like decision trees
  - Parallel Distributed Processing (PDP) Models



## Neuroscience (1861-present)

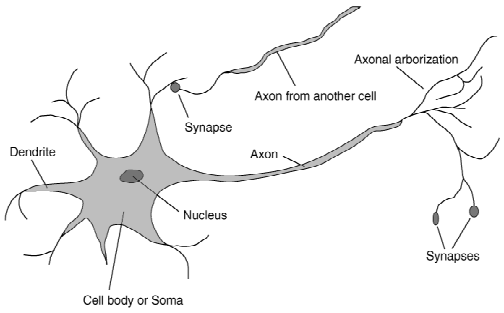
- Neuroscience is the study of the nervous system, particularly the functions of the brain
  - By the 19th century, it had been established that the brain played a central role in specific cognitive functions
  - Before that, people thought the heart or spleen might be the focus of cognitive activity
- Paul Broca jump-started the field with his studies of speech disorders: he isolated the speech center in the lower left hemisphere of the brain
  - Now called "Broca's Area"



## Neuroscience

- Special nerve cells called neurons had been theorized about by the late 1800s
  - At the turn of the 20th century, a staining method for actually viewing them was developed by Camillo Golgi
  - Santiago Ramon y Cajal used the staining technique to propose the structure of the nervous system
  - Golgi & Cajal shared the Nobel prize in 1906, though they had differing views:
    - Gogli thought brain's functions were carried out in the medium
    - Cajal theorized about a connectionist "neuronal doctrine"

## Neuronal Structure






## Neuronal Communication

- Neurons propagate information by "firing," or sending electrochemical signals along the axon
  - Axons can be 1 to 100 centimeters long!
- Synapses connect the axon of one neuron to the dendrites of up to 100,000 other neurons
  - The synapses function as signal amplifiers or repressors

∗ *If enough energy flows into a neuron from all of its synapses/dendrites, then it will fire, too, sending a message along its axon to other neurons*



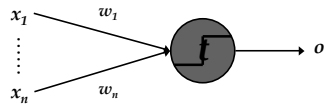
## Simulated Neurons

- We can create a mathematical approximation to the nature of neuronal communication:
  - Represent a "neuron" as a Boolean function
  - Each neuron can have an output capacity of either +1 (fire) or 0 (don't fire… sometimes use -1)
  - Each also has a set of inputs (i.e. other neurons, +1/0), each with an associated ±weight (i.e. synapse)
  - The neuron can compute a weighted sum over all the inputs and compare it to some threshold *t*
  - If the sum is ≥ *t*, then output +1 (fire), otherwise 0



## Perceptrons

- A perceptron is a simulated *neuron* that takes the agent's *percepts* (e.g. feature vector) as inputs an maps them to the appropriate output value:



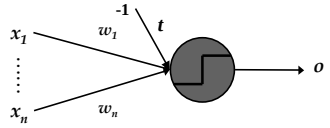

The output, *o* is the result of some activation function *g(in)*, where *in* is the weighted sum of the inputs ($x_1 \ldots x_n$). Right now, *g(in)* is a simple threshold or "step" function



## Perceptrons

- Really, the threshold *t* is just another weight (called the bias):

$$(w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) \geq t$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) - t \geq 0$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) + (t \times -1) \geq 0$$






## Perceptron Learning

- A perceptron learns is by adjusting its weights in order to minimize the error on the training set
- To start off, consider updating the value for a single weight on a single example *x* with the perceptron learning rule:
  - $w_i \leftarrow w_i + \Delta w_i \quad ; \qquad \Delta w_i = \alpha(true - o)x_i$
  - Where $\alpha$ is the learning rate, a value in the range [0,1], *true* is the true function value for the example, and *o* is the perceptron's output (so (*true* – *o*) is the error)

Note: the notation used in the new version of *AI: A Modern Approach* is really messy, and riddled with typos… so my notation will differ from the textbook

## Perceptron Learning

- Now that we can update a single weight on one example, we want to update *all* weights so that we minimize error on the *whole* training set
  - So this is really an optimization search in weight space, which is the hypothesis space for perceptrons
- For reasons we won't get into, it's actually useful to minimize the squared error over the whole set:
  - $E[w] \equiv \frac{1}{2} \Sigma_d (true_d - o_d)^2$
  - Where $E[w]$ is the sum of squared errors for the weight vector $w$, and $d$ ranges over examples in the training set

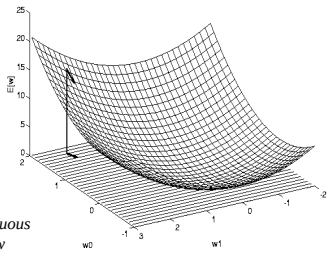
13

## Gradient Descent

Recall that minimization problems are called "gradient descent" tasks

If we have a perceptron with 2 weights, we want to find the pair of weights (i.e. point in 2D weight space) where $E[w]$ is the lowest

*But the weights are continuous values, so how do we know how much to change them?*




14

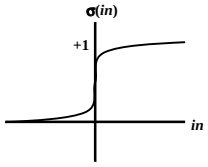
## Gradient Descent

- Solution: use calculus
  - Compute the magnitude and direction of the gradient of the error surface by using a partial derivative:
    - $\nabla E[w] \equiv [\delta E/\delta w_0, \delta E/\delta w_1, \delta E/\delta w_n]$
  - We want to update each weight $w_i$ by $\Delta w_i$ :
    - $\Delta w_i = -\alpha[\delta E/\delta w_i]$
    - $\Delta w_i = -\alpha \times -x_i \times (true - o) \times g'(in)$
    - $\Delta w_i = \alpha \times x_i \times (true - o) \times g'(in)$
  - And if we combine this with our weight update rule, we get the following complete perceptron training rule:
    - $w_i \leftarrow w_i + \alpha \times x_i \times (true - o) \times g'(in)$
    - This makes sense: if $(true - o)$ is positive, the weight should be increased for positive inputs $x_i$, and decreased for negatives

15

## On Activation Functions

- Houston, we have a problem!
  - We're using a simple step function as our activation function $g(in)$
  - This isn't differentiable, so we can't compute $g'(in)$
  - To remedy this, we can use a sigmoid function, which is similar, but is continuous, differentiable, and easy to compute:




$$g(in) = \sigma(in) = 1/(1 + e^{-in})$$

$$g'(in) = \sigma'(in) = \sigma(in) \times (1 - \sigma(in))$$

16

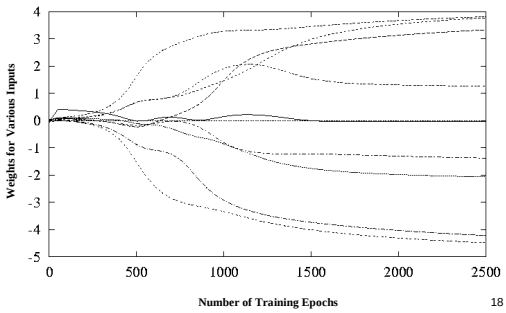
## Perceptron Training

- Initialize weights to small random values
- The perceptron training rule allows us to move to the point in weight space that minimizes squared error on the training set
- Recall that this is an optimization search, so after adjusting the weights we repeat the process with the new weights (each cycle is called an epoch)

✱ *We continue for a fixed number of epochs, or until the weights converge on an optimal set, or until they stop changing very much*

17

## Perceptron Training




18

## Perceptron Training

```
A = learning rate     // in range [0,1]
W = weight-vector     // initialize randomly near 0
repeat {
  for each example X in TRAINING_SET {
    IN = 0
    for each feature I in example X {
      IN += W[I] * X[I]     // weighted sum over X
    }
    OUT = sigmoid(IN)       // activation function
    ERR = true label of X – OUT
    for each weight index I in W {
      W[I] += A * X[I] * ERR * (OUT * (1 – OUT))
    }
} until converged   // or some other stopping criteria
```

19

## Perceptron Training

- Conceptually, the perceptron rule does this:
  - Compare the perceptron's output ($\sigma$) to what it *should* have been ( *f*, e.g. *true*), i.e. compute error
  - If the error is large, assign "blame" to the weight/input combinations that most influenced the wrong call, and raise/lower the weights accordingly
  - If the error is small, don't change them as much
- The key parameter is the learning rate $\alpha$:
  - If too small, learn slowly and convergence takes forever
  - If too large, can make changes that are too drastic

20

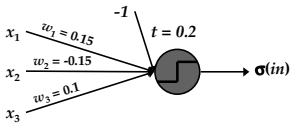
## Training in Practice

- The theoretically correct thing to do is batch mode training, where we compute $\nabla E[w]$ over the entire dataset and update weights accordingly
  - In practice, this is very slow and computationally expensive for one epoch, let alone until we converge
- In practice, we train in incremental mode, updating the weights one example at a time
  - If the learning rate $\alpha$ is low enough, we should converge to about the same weight vector

21

## Perceptron Training Example

- Consider this simple 3-input perceptron:




- Imagine we want to train this perceptron on the following dataset with a learning $\alpha$ rate = 0.5:

| | |
|---|---|
| **x = 001 f(x) = 0** | **x = 111 f(x) = 1** |
| **x = 110 f(x) = 1** | **x = 101 f(x) = 1** |
| **x = 000 f(x) = 0** | **x = 011 f(x) = 1** |

22

## Training Example: Epoch 1

$\alpha = 0.5$; $\Delta w_i = \alpha \times x_i \times error \times \sigma'(in)$

| *x* | *f(x)* | *in* | $\sigma(in)$ | $\sigma'(in)$ | *error* | $\Delta w_i$ | *t* | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|---|---|---|---|---|---|---|
| -- | -- | -- | -- | -- | -- | -- | 0.200 | 0.150 | -0.150 | 0.100 |
| 001 | 0 | -0.100 | 0.475 | 0.249 | -0.475 | -0.059 | 0.259 | 0.150 | -0.150 | 0.041 |
| 110 | 1 | -0.259 | 0.436 | 0.246 | 0.564 | 0.069 | 0.190 | 0.219 | -0.081 | 0.041 |
| 000 | 0 | -0.190 | 0.453 | 0.248 | -0.453 | -0.056 | 0.246 | 0.219 | -0.081 | 0.041 |
| 111 | 1 | -0.066 | 0.483 | 0.250 | 0.517 | 0.065 | 0.181 | 0.284 | -0.016 | 0.105 |
| 101 | 1 | 0.208 | 0.552 | 0.247 | 0.448 | 0.055 | 0.126 | 0.339 | -0.016 | 0.161 |
| 011 | 1 | 0.019 | 0.505 | 0.250 | 0.495 | 0.062 | 0.064 | 0.339 | 0.046 | 0.223 |
| | | | | | *Net Adjustments:* | | -0.136 | +0.189 | +0.196 | +0.123 |

**Average error$^2$ for this epoch: 0.244**

23

## Training Example: Epoch 2

$\alpha = 0.5$; $\Delta w_i = \alpha \times x_i \times error \times \sigma'(in)$

| *x* | *f(x)* | *in* | $\sigma(in)$ | $\sigma'(in)$ | *error* | $\Delta w_i$ | *t* | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|---|---|---|---|---|---|---|
| -- | -- | -- | -- | -- | -- | -- | 0.064 | 0.339 | 0.046 | 0.223 |
| 001 | 0 | 0.159 | 0.540 | 0.248 | -0.540 | -0.067 | 0.131 | 0.339 | 0.046 | 0.156 |
| 110 | 1 | 0.254 | 0.563 | 0.246 | 0.437 | 0.054 | 0.077 | 0.393 | 0.100 | 0.156 |
| 000 | 0 | -0.077 | 0.481 | 0.250 | -0.481 | -0.060 | 0.137 | 0.393 | 0.100 | 0.156 |
| 111 | 1 | 0.511 | 0.625 | 0.234 | 0.375 | 0.044 | 0.093 | 0.437 | 0.143 | 0.200 |
| 101 | 1 | 0.543 | 0.633 | 0.232 | 0.367 | 0.043 | 0.051 | 0.480 | 0.143 | 0.242 |
| 011 | 1 | 0.335 | 0.583 | 0.243 | 0.417 | 0.051 | 0.000 | 0.480 | 0.194 | 0.293 |
| | | | | | *Net Adjustments:* | | -0.064 | +0.141 | +0.148 | +0.070 |

**Average error$^2$ for this epoch: 0.193**
**Last epoch: 0.244**

24

## Training Example: Epoch 3

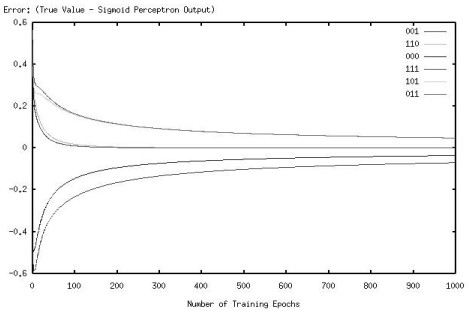
$\alpha = 0.5$; $\Delta w_i = \alpha \times x_i \times error \times \sigma'(in)$

| $x$ | $f(x)$ | $in$ | $\sigma(in)$ | $\sigma'(in)$ | $error$ | $\Delta w_i$ | $t$ | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|---|---|---|---|---|---|---|
| -- | -- | -- | -- | -- | -- | -- | 0.000 | 0.480 | 0.194 | 0.293 |
| 001 | 0 | 0.293 | 0.573 | 0.245 | -0.573 | -0.070 | 0.070 | 0.480 | 0.194 | 0.223 |
| 110 | 1 | 0.604 | 0.647 | 0.229 | 0.353 | 0.040 | 0.030 | 0.520 | 0.235 | 0.223 |
| 000 | 0 | -0.030 | 0.493 | 0.250 | -0.493 | -0.062 | 0.091 | 0.520 | 0.235 | 0.223 |
| 111 | 1 | 0.886 | 0.708 | 0.207 | 0.292 | 0.030 | 0.061 | 0.550 | 0.265 | 0.253 |
| 101 | 1 | 0.742 | 0.677 | 0.219 | 0.323 | 0.035 | 0.026 | 0.585 | 0.265 | 0.288 |
| 011 | 1 | 0.527 | 0.629 | 0.233 | 0.371 | 0.043 | -0.017 | 0.585 | 0.308 | 0.332 |
| | | | | | *Net Adjustments:* | | -0.017 | +0.105 | +0.114 | +0.039 |

**Average error² for this epoch: 0.171**
**Last epoch: 0.193**

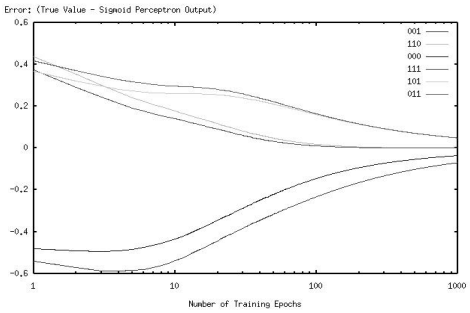
25

## Error Over Training Epochs




26

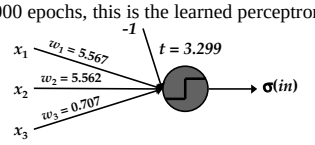
## Error Over Training Epochs




27

## Training Example Results

- After 1,000 epochs, this is the learned perceptron:




***Predictions on the training data:***

| | | | | | |
|---|---|---|---|---|---|
| $x$ = 001 | $f(x)$ = 0 | $\sigma(in)$ = 0.070 | $x$ = 111 | $f(x)$ = 1 | $\sigma(in)$ = 1.000 |
| $x$ = 110 | $f(x)$ = 1 | $\sigma(in)$ = 1.000 | $x$ = 101 | $f(x)$ = 1 | $\sigma(in)$ = 0.951 |
| $x$ = 000 | $f(x)$ = 0 | $\sigma(in)$ = 0.030 | $x$ = 011 | $f(x)$ = 1 | $\sigma(in)$ = 0.951 |

***And on some novel examples:***

| | | | | | |
|---|---|---|---|---|---|
| $x$ = 010 | $f(x)$ = ? | $\sigma(in)$ = 0.906 | $x$ = 100 | $f(x)$ = ? | $\sigma(in)$ = 0.906 |

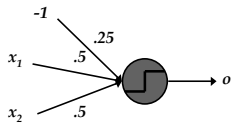
***Any ideas what function this might be?***

28

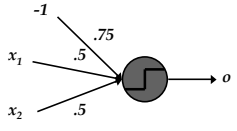
## Perceptrons and Logic

- Perceptrons can learn logical functions:

$x_1 \vee x_2$




$x_1 \wedge x_2$

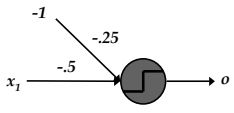



29

## Perceptrons and Logic

- Perceptrons can learn logical functions:

$\neg x_1$




$x_1 \otimes x_2$

*A perceptron cannot represent the XOR function! Why not??*

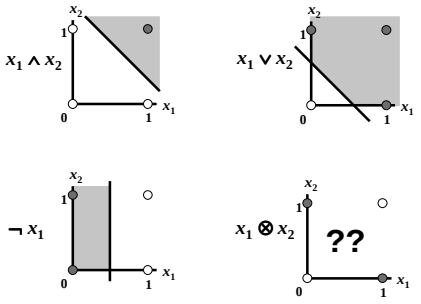
30

## Linear Separability

- Consider a perceptron with two inputs and and a threshold (bias):
  - The perceptron fires if $w_1x_1 + w_2x_2 - t \geq 0$
  - Recall the weights for the "and" perceptron:
    - $0.5x_1 + 0.5x_2 - 0.75 \geq 0$
  - This is really the equation for a line!
    - $x_2 \geq -x_1 + 1.5$ (in slope-intercept form)
- The activation threshold for a perceptron is actually a linearly separable "hyperplane" in the space of inputs

31

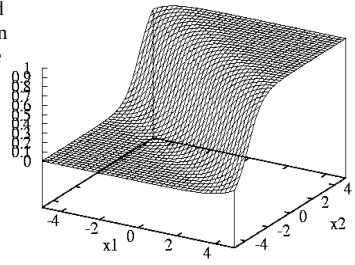
## Linear Separability




32

## Linear Separability

Using the sigmoid activation function achieves the same general effect, sliding the sigmoid surface across the linear hyperplane…




33

## The Need for a Network

- Clearly a single perceptron is limited compared to the expressiveness of a decision tree or *k*-NN
  - They can handle XOR, for example
- But remember: the brain is a network of neurons: the axon (output) is connected to the dendrites (inputs) of others through synapses (weights)
- By this analogy, we can create a multi-layer feed-forward neural network made up of perceptrons, which might learn more expressive functions

34

## Multi-Layer Networks

- The structure of a multi-layer network is fairly straightforward:
  - The input layer is the set of features (percepts)
  - Next is a hidden layer, which has an arbitrary number of perceptrons called hidden units that take the features (input layer) as inputs
  - The perceptron(s) in the output layer then takes the outputs of the hidden units as its inputs

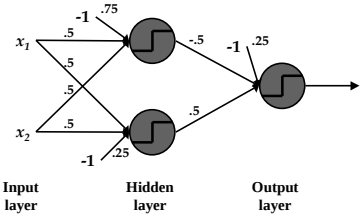
✱ *This is looking more like a model of the brain!*

35

## Multi-Layer Networks

- Here is a very simple multi-layer network that can handle the XOR function:

$x_1 \otimes x_2$



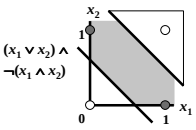

36

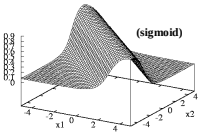
## Multi-Layer Networks

- This network is essentially equivalent to a more complex logical function:

  $(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$

- Which, represented graphically, is:

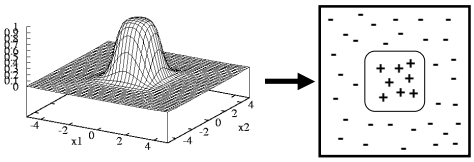



37

## Multi-Layer Networks

- We aren't limited to just *one* layer of hidden units, though, we could have even more, which will allow us to learn even more complex functions:



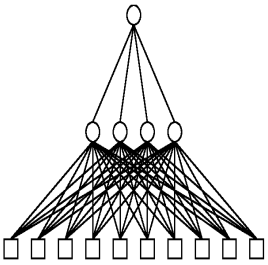

38

## Multi-Layer Networks

Multi-layer networks are called feed forward because the information is fed forward from the input layer (features) toward the output layer

For most problems, one layer of hidden units is sufficient

Such networks are also usually fully connected: every output from one layer is connected to every input of the next (but they don't need to be)



39

## Training Multi-Layer Networks

- Training multi-layer networks can be a bit complicated (the weight space is larger!)
  - The perceptron rule worked fine for a single unit that mapped input features to the final output value
  - But hidden units don't produce the final output
  - Output unit(s) take other perceptrons – not known feature values – as inputs
- The solution is to use the back-propagation algorithm, which is an intuitive extension of the perceptron training algorithm

40

## Back-Propagation (BP)

- BP generalizes the perceptron rule:
  - Gradient-descent search to minimize error on the training data (again, usually in iterative mode)
  - In the forward pass, features are fed forward to the output layer where error is calculated
  - Then, in the backward pass:
    - Update weights from hidden layer to output layer as usual: $\Delta w_{ho} = \alpha \times x_h \times ERR_o$ ; where $ERR_o = g'(in_o) \times (true - g(in_o))$
    - Update weights from input layer to the hidden layer: $\Delta w_{ih} = \alpha \times x_i \times ERR_h$ ; where $ERR_h = g'(in_h) \times \Sigma_o (w_{ho} \times ERR_o)$
- More complete version of the algorithm on p.746 of *AIMA* or section 4.5.2 of *Maching Learning*

41

## Problems with BP

- Because BP is a gradient descent (hill-climbing) search, it suffers from the same problems:
  - Doesn't necessarily find the globally best weight vector
    - Convergence is determined by the starting point (randomly initialized weights)
    - If $\alpha$ is set too large, can "bounce" right over the global minimum into a local minimum
- To deal with these problems:
  - Usually initialize weights close to 0
  - Can repeat training with multiple random restarts

42

## Non-Boolean Features

- So far, the networks we've described only take Boolean features [1,0] as inputs
- To hand discrete-valued features, we can create a unique input for each feature-value pair
  - e.g. Outlook = {Sunny, Overcast, Rainy} would be converted into 3 Boolean inputs
  - For classification purposes, the observed value's input is set to 1, the others are 0
- Continuous features can be left alone and fed through the network as real numbers
  - The weights will figure out what to do with them!

43

## Handling Multiple Labels

- Similarly, the networks so far have been for Boolean classification functions
- To handle multi-label classification tasks we can simply create extra output units:
  - Each unit corresponds to one label (e.g. animal/vegetable/mineral)
  - For classification, the unit with the highest output is the "winner," and the network assigns the corresponding label to that example
  - For training purposes, the "true" label's output unit is set to 1, and the others are set to 0

* *This variant on networks has been applied with wide success to several multi-class problems*

44

## Regression and Neural Nets

- Are neural networks very well suited for regression (i.e. real-valued function) tasks?
- We currently use the $\sigma$ function to allow the perceptrons to behave like classifiers, but we *could* just output the weighted sum of their inputs directly
  - Since this is a linear function, the *g'(in)* factor in training is 1

* *Neural networks can learn regression problems better than decision trees or* k*-nearest neighbors (though training can be slower than a standard network)*

45

## Expressiveness of Neural Nets

- Classification problems
  - Any Boolean function can be represented in a neural network with just 2 layers
  - But might require an exponential number of hidden units (in terms of input features)
- Regression problems
  - Perceptrons can learn any linear function
  - G. Cybenko, "Continuous valued neural networks with two hidden layers are sufficient," Tech. Report, 1988
    - 2-layer networks with enough hidden units can learn any bounded, continuous (e.g. polynomial) function
    - 3-layer networks can learn any function. Period.

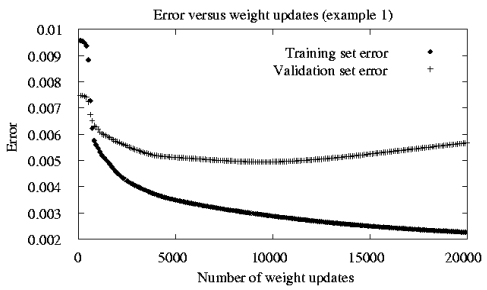
46

## Overfitting in Neural Nets

- Again, as with all machine learning algorithms, there is a risk of overfitting the training data
  - Neural nets with lots of hidden units are particularly prone to overfit, because the model is so expressive!
- Recall that the network ultimately "converges," within some $\varepsilon$ of change
  - If we keep cycling through epochs ad infinitum, we end up memorizing the training examples
- But how do we know when to stop?

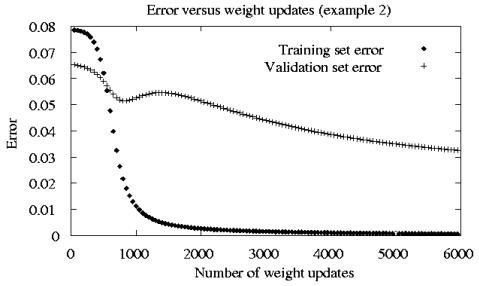
47

## Overfitting in Neural Nets




48

## Overfitting in Neural Nets




49

## Overfitting in Neural Nets

- As usual, we can use a tuning set to avoid overfitting in neural nets
  - We can train several candidate structures and use the tuning set to find one that's appropriately expressive
- More common: given a network structure, use early stopping by evaluating the network on the tuning set after each epoch
  - Stop when performance begins to dip on the tuning set
  - Sometimes allow a fixed number of epochs beyond the dip… just in case it goes back up

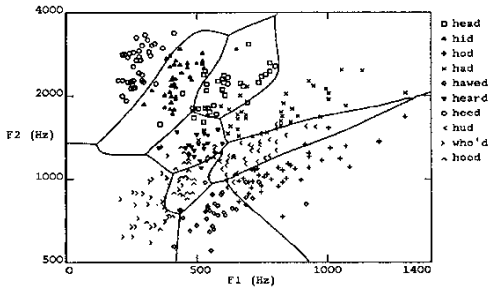
50

## Neural Network Applications

- W. Huang & R. Lippmann, "Neural net and traditional classifiers," *Neural Information Processing Systems*, 1988
- Used a simple network to disambiguate between vowel phoneme sounds in "h — d" words
  - Only 2 features (percepts) obtained from spectral analysis of recorded data
  - 7 hidden units in 1 layer (fully connected)
  - 10 outputs (words: *head, hid, who'd, hood*, etc…)

51

## Neural Network Applications




52

## Neural Network Applications

- Facial pose recognition
  - Section 4.7 of *Machine Learning* by T. Michell
    - All this image data is available at **www.cs.cmu.edu/~tom/faces.html**
- Used 30×32 pixel images of faces pointing left, straight, right, and up
  - Each of the 30×32 = 960 pixels was a continuous feature (grayscale value)
  - 3 hidden units
  - 4 output units (poses)

53

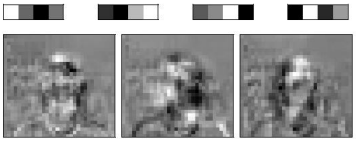
## Neural Network Applications

Example images:

left straight right up

Network weights (dark–, light+) after 100 epochs of training on 260 examples

Achieved 90% accuracy



hidden #1 hidden #2 hidden #3

54

## Neural Network Applications

- D. Pomerleau, "Knowledge-based training of artificial neural networks for autonomous robot driving," *Robot Learning*, 1993
- That's right, folks: ALVINN (Autonomous Land Vehicle In a Neural Network)
  - Takes 30×32 pixel input camera images
  - Only 1 hidden layer: 4 hidden units
  - 30 output units (discrete steering wheel direction)

55

## Issues with Neural Nets

- Neural networks are very powerful and can approximate any function, but they do have drawbacks
  - Many weights to learn: training can take a while
  - Sensitive to structure, initial weights, and learning rate
- Once the network has learned a hypothesis function, *what does it mean?*
  - Neural networks are connectionist models, thus not as comprehensible as decision trees, which are symbolic

56

## Understanding Neural Nets

- M. Craven & J. Shavlik, "Extracting Tree-Structured Representations of Trained Networks," *Advances in Neural Information Processing Systems*, MIT Press, 1996
  - "Trepan" Algorithm: extract decision trees from Neural Nets to better understand what they learned

| Problem Dataset | % Accuracy | | | Fidelity to NN |
|---|---|---|---|---|
| | Network | ID2/3 | Trepan | |
| **Heart** | 84.5 | 74.6 | 81.8 | 94.1 |
| **Promoters** | 90.6 | 83.5 | 87.6 | 85.7 |
| **Protein-coding** | 94.1 | 90.9 | 91.4 | 92.4 |
| **Voting** | 92.2 | 87.8 | 90.8 | 95.9 |

57

## Summary

- Perceptrons are mathematical models of neurons (brain cells)
  - Learn linearly separable functions
  - Insufficiently expressive for many problems
- Neural Networks are machine learning models that have multiple layers of perceptrons
  - Trained using back-propagation, a gradient descent search through weight space (NN hypothesis space)
  - Sufficiently expressive for any classification or regression task, also quite robust to noise

58

## Summary

- Many applications:
  - Speech processing, driving, face/handwriting recognition, backgammon, checkers, etc.
- Disadvantages:
  - Overly expressive: prone to overfitting
  - Difficult to design appropriate structure
  - Many parameters to estimate: slow training
  - Hill-climbing can get stuck in local optima
  - Poor comprehensibility

59