Perceptrons

Louis Oliphant
oliphant@cs.wisc.edu
CS 540 section 2
slides borrowed (with modifications) from Burr Settles

Announcements

- Review Session tonight at 4:30-5:30 CS 1325 (right here)
  - Come with questions
  - No lecture prepared
- Midterm tomorrow night 7:15-9:15 1240 CS
- HW 3 solution on line. Grading not done yet.

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

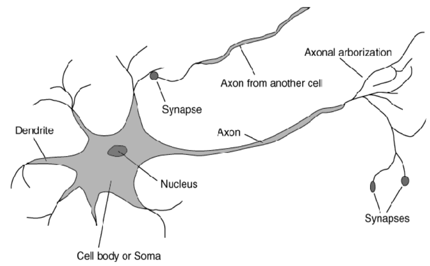
Neuroscience (1861-present)

- 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

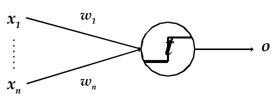
∗ *If enough energy flows into a neuron from all of its synapses/dendrites, then it will fire, too, sending a message along its axon to other neurons*

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 and maps them to the appropriate output value:



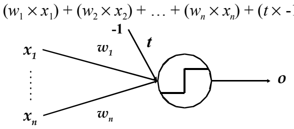
The output, o is the result of some activation function *g(in)*, where *in* is the weighted sum of the inputs ($x_1 \ldots x_n$). Right now, *g(in)* is a simple threshold or "step" function

Perceptrons – inference

$$o(x_{1,..}, x_n) = \begin{cases} 1\ \mathit{if}\ w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n > t \\ 0\ \mathit{otherwise} \end{cases}$$

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

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



Methods of Learning

- Perceptron Training Rule
- Delta Rule

Perceptron Learning

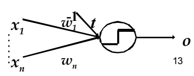
- A perceptron learns by adjusting its weights in order to minimize the error on the training set
- To start off, consider updating the value for a single weight on a single example x with the perceptron learning rule:
  - $w_i \leftarrow w_i + \Delta w_i$; $\Delta w_i = \alpha(true - o)x_i$
  - Where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

Using Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i \;; \qquad \Delta w_i = \alpha(true - o)x_i$$

- Suppose training example correctly classified
  - What is the change in weight, Δw_i ?
- What if training example incorrectly classified?
  - How will weights change?



Perceptron Training Rule

Proven to converge in a finite number of steps to weights that will correctly classify all training examples, *provided the training examples are linearly seperable.*

Gradient Descent and Delta Rule

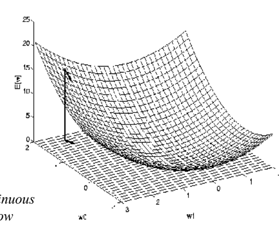
- works with unthresholded perceptron

$$o(x_1, .., x_n) = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n$$
$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- Delta rule converges toward a best-fit approximation to the target concept even when training examples are not linearly seperable
- Training error, for a given data set, is defined as
  - $E[w] \equiv \frac{1}{2} \sum_d (true_d - o_d)^2$
  - Where $E[w]$ is the sum of squared errors for the weight vector w, and d ranges over examples in the training set
  - This formulation of error makes a parabolic curve, and so has a global minimum.

Gradient Descent and Delta Rule

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



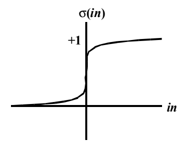
But the weights are continuous values, so how do we know how much to change them?

Gradient Descent and Delta Rule

- Find the gradient (partial derivatives):
  - $\nabla E[w] \equiv [\delta E/\delta w_0, \delta E/\delta w_1, \delta E/\delta w_n]$
- Update weights:
  - $w_i \leftarrow w_i + \Delta w_i$ and $\Delta w_i = -\alpha[\delta E/\delta w_i]$
  - Just need to calculate the partial derivative of the Error function
    - $\delta E/\delta w_i = \delta/\delta w_i(\frac{1}{2} \Sigma_d (true_d - o_d)^2)$
    - $\delta E/\delta w_i = \Sigma_d(true_d - o_d)(-x_{id})$
  - Putting it all together, this is called the Delta rule for training:
  - $\Delta w_i = \alpha\Sigma_d(true_d - o_d)(x_{id})$
  - Often this is rule is applied for each example instead of on the entire dataset
  - This makes sense: if (*true* – *o*) is positive, the weight should be increased for positive inputs x_i, and decreased for negatives

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)*
  - Using Delta rule will not work on thresholded perceptron
  - To remedy this, we can use a sigmoid function, which is similar, but is continuous, differentiable, and easy to compute:



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

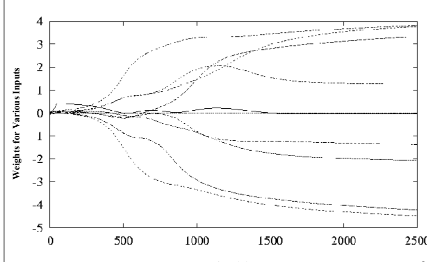
$$g'(in) = \sigma'(in) = \sigma(in) \times (1 - \sigma(in))$$

Perceptron Training

- Initialize weights to small random values
- The delta 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*

Perceptron Training



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
```


Perceptron Training

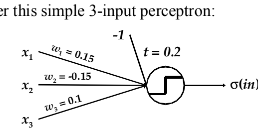
- Conceptually, the perceptron rule does this:
  - Compare the perceptron's output (σ) to what it *should* have been (*f*, e.g. *true*), i.e. compute error
  - If the error is large, assign "blame" to the weight/input combinations that most influenced the wrong call, and raise/lower the weights accordingly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

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 α is low enough, we should converge to about the same weight vector

Perceptron Training Example

- Consider this simple 3-input perceptron:



- Imagine we want to train this perceptron on the following dataset with a learning α 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**

Training Example: Epoch 1

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

x	$f(x)$	in	$\sigma(in)$	$\sigma'(in)$	$error$	Δw_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 error² for this epoch: 0.244

Training Example: Epoch 2

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

x	$f(x)$	in	$\sigma(in)$	$\sigma'(in)$	$error$	Δw_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 error² for this epoch: 0.193
Last epoch: 0.244

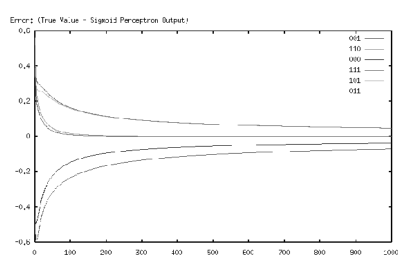
Training Example: Epoch 3

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

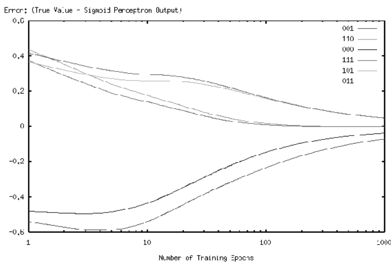
x	$f(x)$	in	$\sigma(in)$	$\sigma'(in)$	$error$	Δw_i	t	w_1	w_2	w_3
--	--	--	--	--	--	--	0.000	0.480	0.194	0.293
001	0	0.293	0.573	0.245	-0.573	-0.070	0.070	0.480	0.194	0.223
110	1	0.604	0.647	0.229	0.353	0.040	0.030	0.520	0.235	0.223
000	0	-0.030	0.493	0.250	-0.493	-0.062	0.091	0.520	0.235	0.223
111	1	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

Error Over Training Epochs

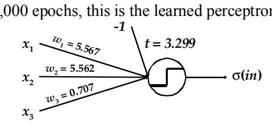


Error Over Training Epochs



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:

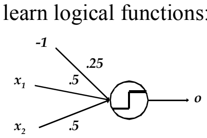
$x = 010$ $f(x) = ?$ $\sigma(in) = 0.906$	**$x = 100$ $f(x) = ?$ $\sigma(in) = 0.906$**

Any ideas what function this might be?

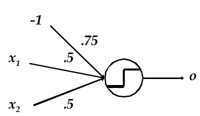
Perceptrons and Logic

- Perceptrons can learn logical functions:

$x_1 \vee x_2$



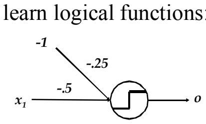
$x_1 \wedge x_2$



Perceptrons and Logic

- Perceptrons can learn logical functions:

$\neg x_1$



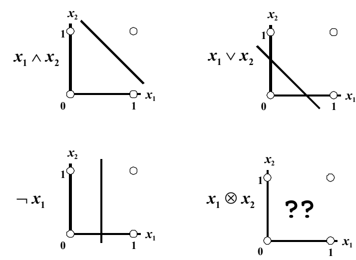
$x_1 \otimes x_2$

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

Linear Separability

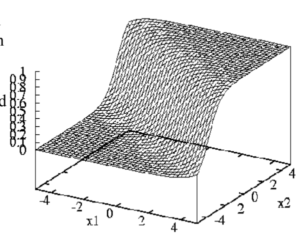
- Consider a perceptron with two inputs and and a threshold (bias):
  - The perceptron fires if $w_1x_1 + w_2x_2 - t \geq 0$
  - Recall the weights for the "and" perceptron:
    - $0.5x_1 + 0.5x_2 - 0.75 \geq 0$
  - This is really the equation for a line!
    - $x_2 \geq - x_1 + 1.5$ (in slope-intercept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

Linear Separability



Linear Separability

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



Summary

- Perceptrons are mathematical models of neurons (brain cells)
  - Learn linearly separable functions
  - Learn best-fit approximation of function is not linearly seperable
  - Insufficiently expressive for many problems