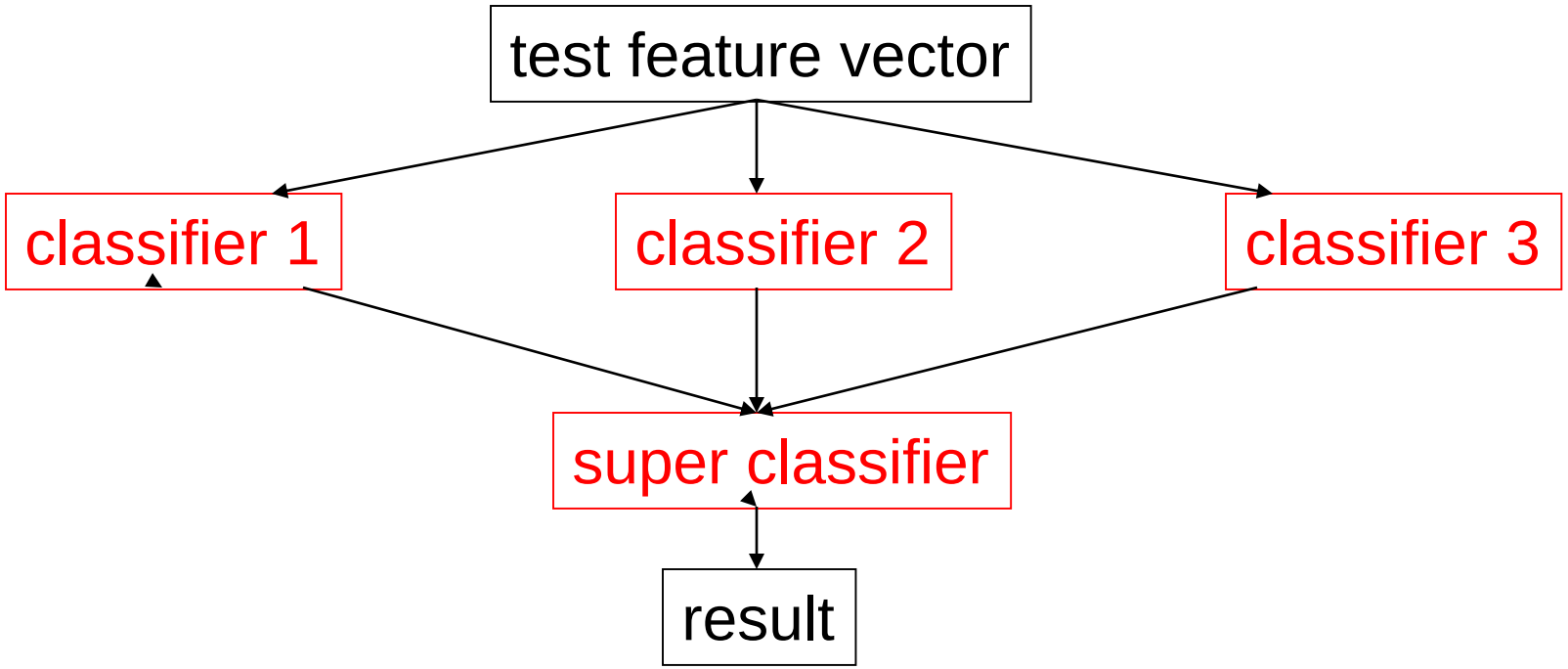
Ensembles

- An ensemble is a set of classifiers whose combined results give the final decision.



MODEL* ENSEMBLES

- Basic Idea
  - Instead of learning one model
  - Learn several and combine them

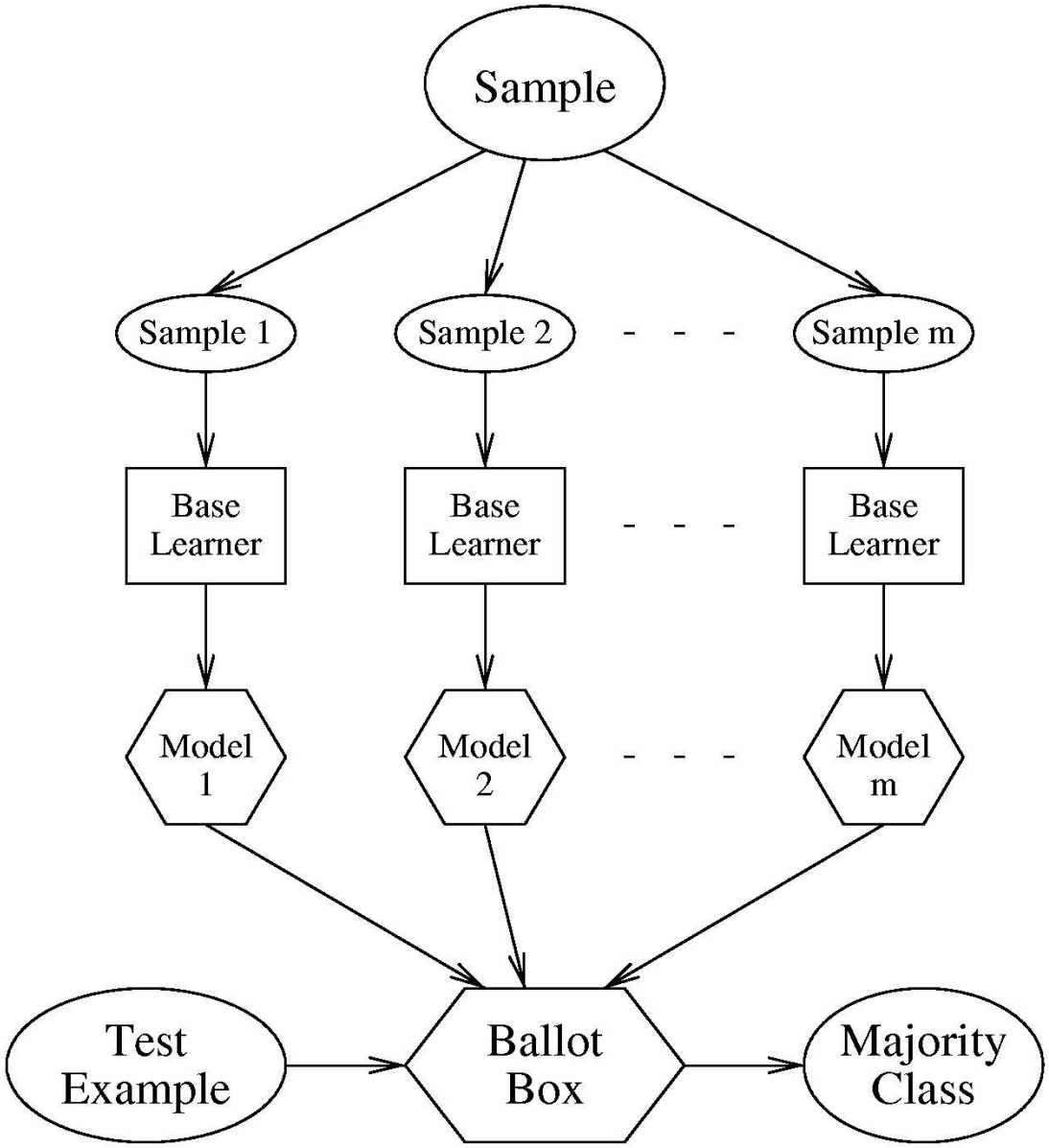
- Often this improves accuracy by a lot

- Many Methods
  - Bagging
  - Boosting
  - Stacking

*A model is the learned decision rule. It can be as simple as a hyperplane in n-space (ie. a line in 2D or plane in 3D) or in the form of a decision tree or other modern classifier.

Bagging

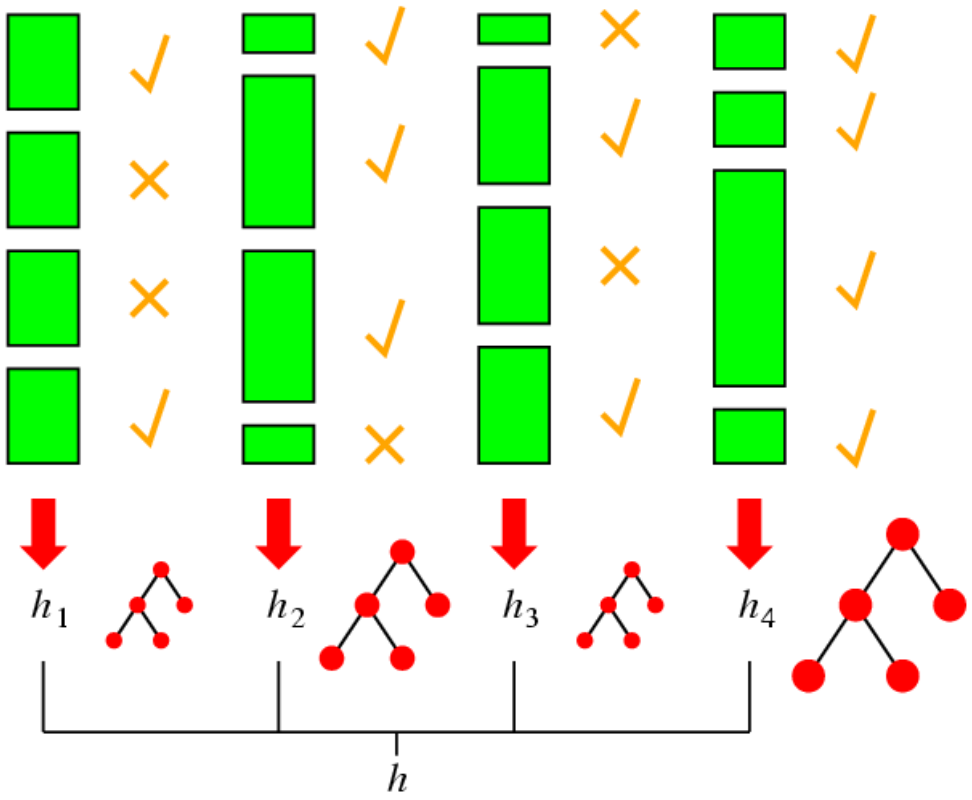
- Generate bootstrap replicates of the training set by sampling with replacement
- Learn one model on each replicate
- Combine by uniform voting


Sample
Sample 1
Sample 2
Sample m
Base Learner
Base Learner
Base Learner
Model 1
Model 2
Model m
Test Example
Ballot Box
Majority Class

Boosting

- Maintain a vector of weights for samples
- Initialize with uniform weights
- Loop
  - Apply learner to weighted samples
  - Increase weights of misclassified ones
- Combine models by weighted voting

Idea of Boosting



Boosting In More Detail
(Pedro Domingos' Algorithm)

1. Set all E weights to 1, and learn H1.
2. Repeat m times: increase the weights of misclassified Es, and learn H2,…Hm.
3. H1..Hm have "weighted majority" vote when classifying each test
   Weight(H)=accuracy of H on the training data

ADABoost

- ADABoost boosts the accuracy of the original learning algorithm.

- If the original learning algorithm does slightly better than 50% accuracy, ADABoost with a large enough number of classifiers is guaranteed to classify the training data perfectly.

ADABoost Weight Updating (from Fig 18.34 text)

/* First find the sum of the weights of the misclassified samples */

for j = 1 to N do /* go through training samples */

if h[m](x_j) <> y_j then error <- error + w_j

/* Now use the ratio of error to 1-error to change the weights of the correctly classified samples */

for j = 1 to N do

if h[m](x_j) = y_j then w[j] <- w[j] * error/(1-error)

Example

- Start with 4 samples of equal weight .25.
- Suppose 1 is misclassified. So error = .25.
- The ratio comes out .25/.75 = .33
- The correctly classified samples get weight of .25*.33 = .0825

.2500
.0825
.0825
.0825

What's wrong? What should we do?

We want them to add up to 1, not .4975.

Answer: To normalize, divide each one by their sum (.4975).

Sample Application: Insect Recognition

Using circular regions of interest selected by an interest operator, train a classifier to recognize the different classes of insects.

Boosting Comparison

- **ADTree classifier only** **(alternating decision tree)**

- Correctly Classified Instances 268 70.1571 %
- Incorrectly Classified Instances 114 29.8429 %
- Mean absolute error 0.3855
- Relative absolute error 77.2229 %

Classified as ->	Hesperperla	Doroneuria
Real Hesperperlas	167	28
Real Doroneuria	51	136

Boosting Comparison

AdaboostM1 with ADTree classifier

- Correctly Classified Instances 303 79.3194 %
- Incorrectly Classified Instances 79 20.6806 %
- Mean absolute error 0.2277
- Relative absolute error 45.6144 %

Classified as ->	Hesperperla	Doroneuria
Real Hesperperlas	167	28
Real Doroneuria	51	136

Boosting Comparison

- **RepTree classifier only (reduced error pruning)**

- Correctly Classified Instances 294 75.3846 %
- Incorrectly Classified Instances 96 24.6154 %
- Mean absolute error 0.3012
- Relative absolute error 60.606 %

Classified as ->	Hesperperla	Doroneuria
Real Hesperperlas	169	41
Real Doroneuria	55	125

Boosting Comparison

AdaboostM1 with RepTree classifier

- Correctly Classified Instances 324 83.0769 %
- Incorrectly Classified Instances 66 16.9231 %
- Mean absolute error 0.1978
- Relative absolute error 39.7848 %

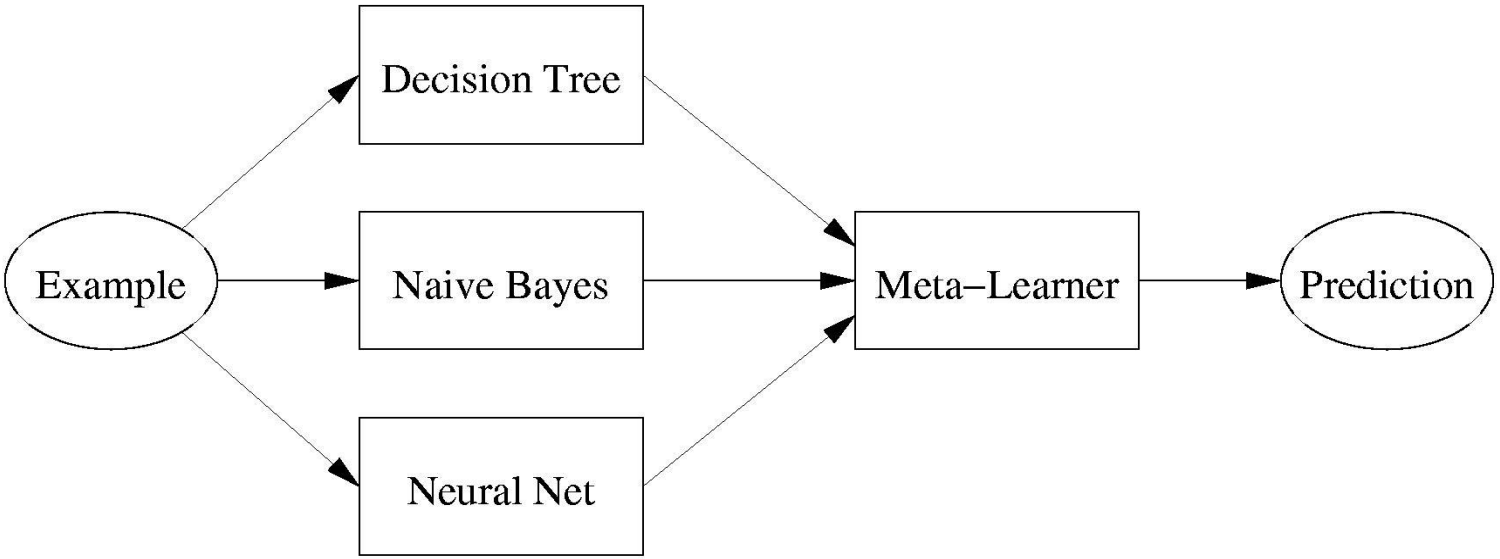
Classified as ->	Hesperperla	Doroneuria
Real Hesperperlas	180	30
Real Doroneuria	36	144

References

- **AdaboostM1**: Yoav Freund and Robert E. Schapire (1996). "Experiments with a new boosting algorithm". Proc International Conference on Machine Learning, pages 148-156, Morgan Kaufmann, San Francisco.
- **ADTree**: Freund, Y., Mason, L.: "The alternating decision tree learning algorithm". Proceeding of the Sixteenth International Conference on Machine Learning, Bled, Slovenia, (1999) 124-133.

Stacking

- Apply multiple base learners
  (e.g.: decision trees, naive Bayes, neural nets)
- Meta-learner: Inputs = Base learner predictions
- Training by leave-one-out cross-validation:
  Meta-L. inputs = Predictions on left-out examples



Random Forests

- Tree bagging creates decision trees using the bagging technique. The whole set of such trees (each trained on a random sample) is called a decision forest. The final prediction takes the average (or majority vote).

- Random forests differ in that they use a modified tree learning algorithm that selects, at each candidate split, a random subset of the features.

Back to Stone Flies

Random forest of 10 trees, each constructed while considering 7 random features.
Out of bag error: 0.2487. Time taken to build model: 0.14 seconds

Correctly Classified Instances	292	76.4398 % (81.4 with AdaBoost)
Incorrectly Classified Instances	90	23.5602 %
Kappa statistic	0.5272	
Mean absolute error	0.344	
Root mean squared error	0.4069	
Relative absolute error	68.9062 %	
Root relative squared error	81.2679 %	
Total Number of Instances	382	

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.69	0.164	0.801	0.69	0.741	0.848	cal
	0.836	0.31	0.738	0.836	0.784	0.848	dor
WAvg.	0.764	0.239	0.769	0.764	0.763	0.848	

```
  a   b   <-- classified as
129  58 |   a = cal
 32 163 |   b = dor
```

More on Learning

- Neural Nets
- Support Vectors Machines
- Unsupervised Learning (Clustering)
  - K-Means
  - Expectation-Maximization

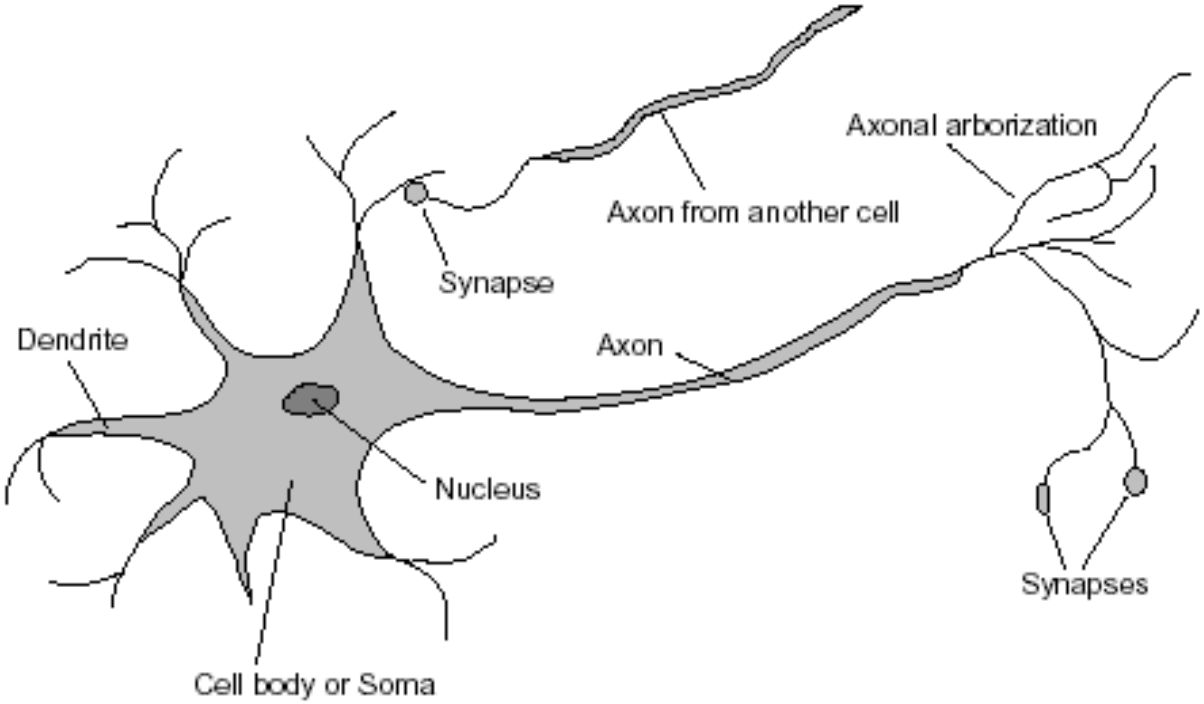
Neural Net Learning

- Motivated by studies of the brain.
- A network of "artificial neurons" that learns a function.
- Doesn't have clear decision rules like decision trees, but highly successful in many different applications. (e.g. face detection)
- We use them frequently in our research.
- I'll be using algorithms from

http://www.cs.mtu.edu/~nilufer/classes/cs4811/2016-spring/lecture-slides/cs4811-neural-net-algorithms.pdf

Brains

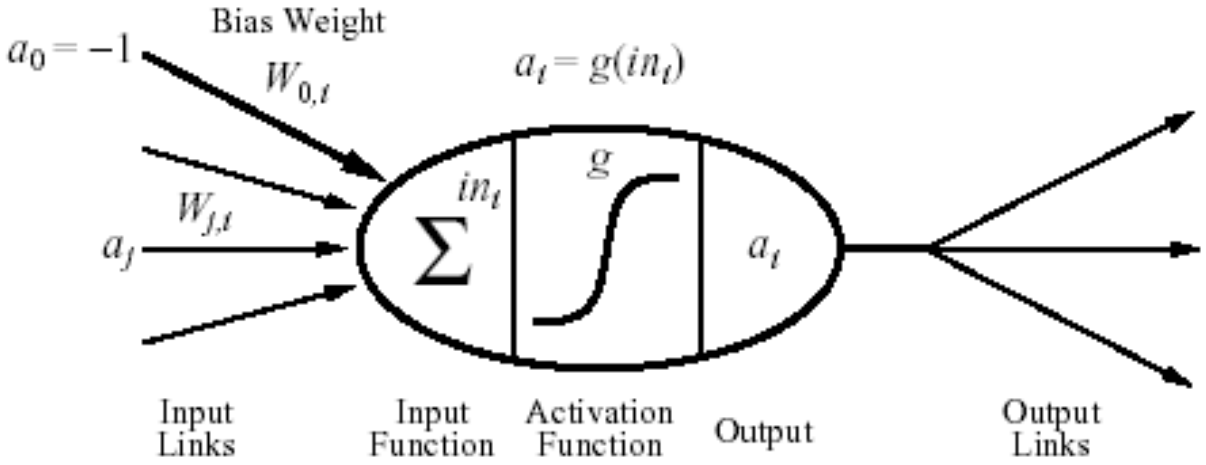
10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

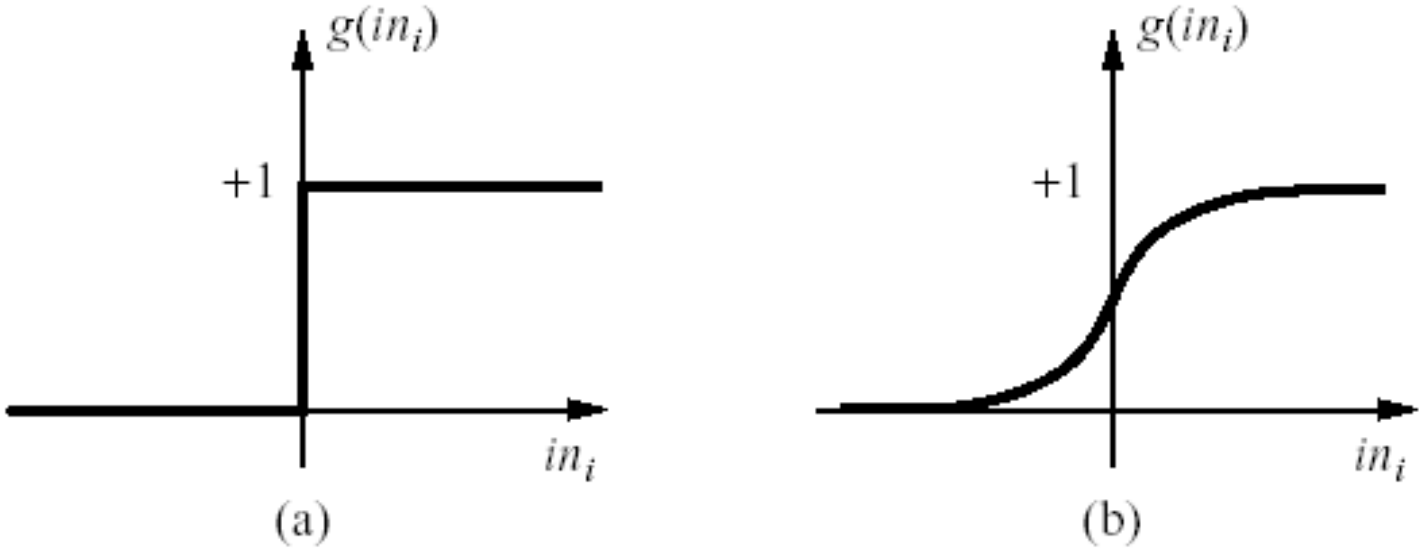
Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$



A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Activation functions

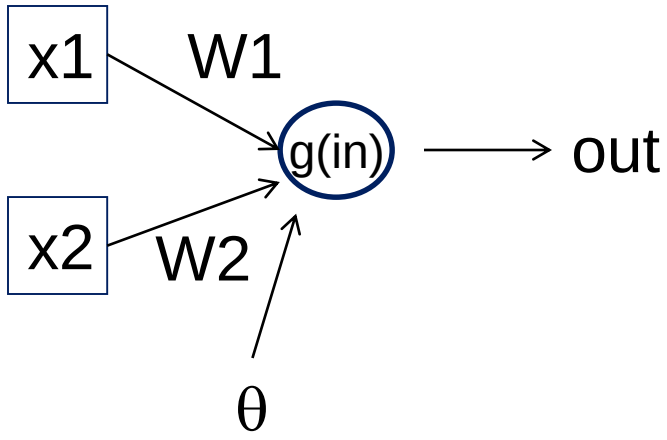


(a) is a step function or threshold function

(b) is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Simple Feed-Forward Perceptrons



$in = (\sum W_j x_j) + \theta$

out = g[in]

g is the activation function

It can be a step function:

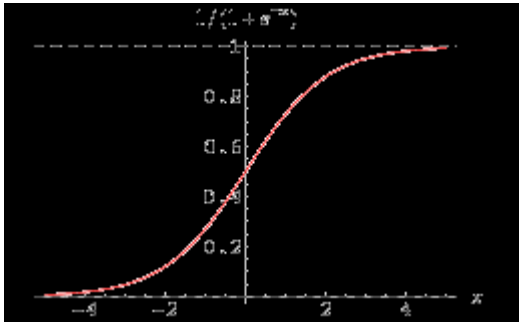
g(x) = 1 if x >=0 and

0 (or -1) else.

It can be a sigmoid function:

g(x) = 1/(1+exp(-x)).

The sigmoid function is differentiable and can be used in a gradient descent algorithm to update the weights.

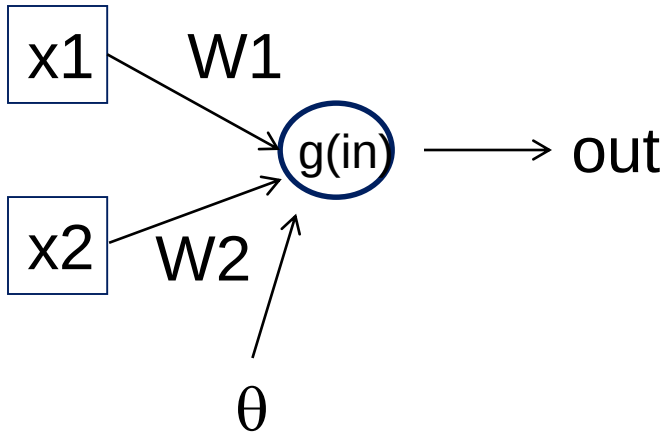


Gradient Descent

takes steps proportional to the negative of the gradient of a function to find its local minimum

- Let **X** be the inputs, y the class, **W** the weights
- $in = \sum W_j x_j$
- $Err = y - g(in)$
- $E = \frac{1}{2} Err^2$ is the squared error to minimize
- $\partial E/\partial W_j = Err * \partial Err/\partial W_j = Err * \partial/\partial W_j(g(in))(-1)$
- $= -Err * g'(in) * x_j$
- The update is $W_j \leftarrow W_j + \alpha * Err * g'(in) * x_j$
- α is called the learning rate.

Simple Feed-Forward Perceptrons



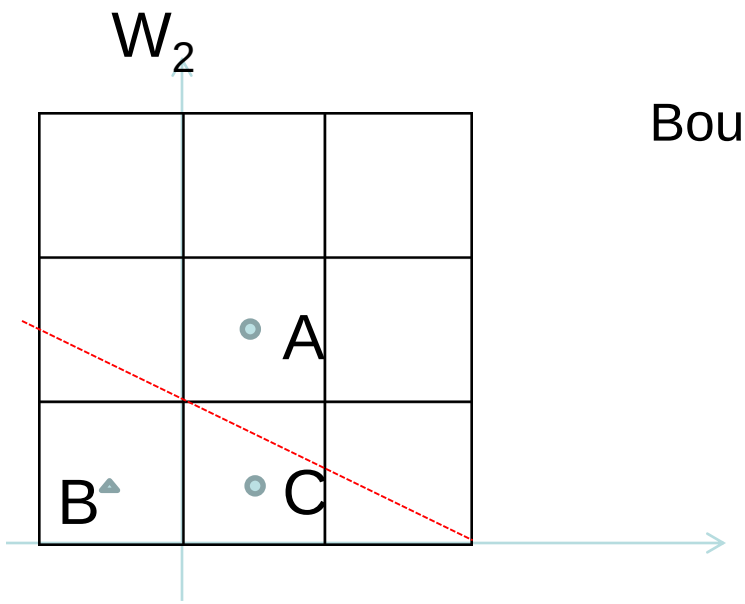
repeat
  for each e in examples do
    in = $(\sum W_j x_j) + \theta$
    Err = y[e] – g[in]
    $W_j = W_j + \alpha$ Err g'(in) x_j[e]
until done

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

Note1: when g is a step function, the g'(in) is removed.
Note2: later in back propagation, Err * g'(in) will be called Δ
Note3: We'll let g(x) = 1 if x >=0 else -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



wrong boundary

Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Learning

Examples:
A=[(.5,1.5),+1],
B=[(-.5,.5),-1],
C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

repeat
  for each e in examples do
    in = $(\sum W_j x_j) + \theta$
    Err = y[e] – g[in]
    $W_j = W_j + \alpha$ Err g'(in) x_j[e]
until done

A=[(.5,1.5),+1]
in = .5(1) + (1.5)(2) -2 = 1.5
g(in) = 1; Err = 0; NO CHANGE

B=[(-.5,.5),-1]
In = (-.5)(1) + (.5)(2) -2 = -1.5
g(in) = -1; Err = 0; NO CHANGE

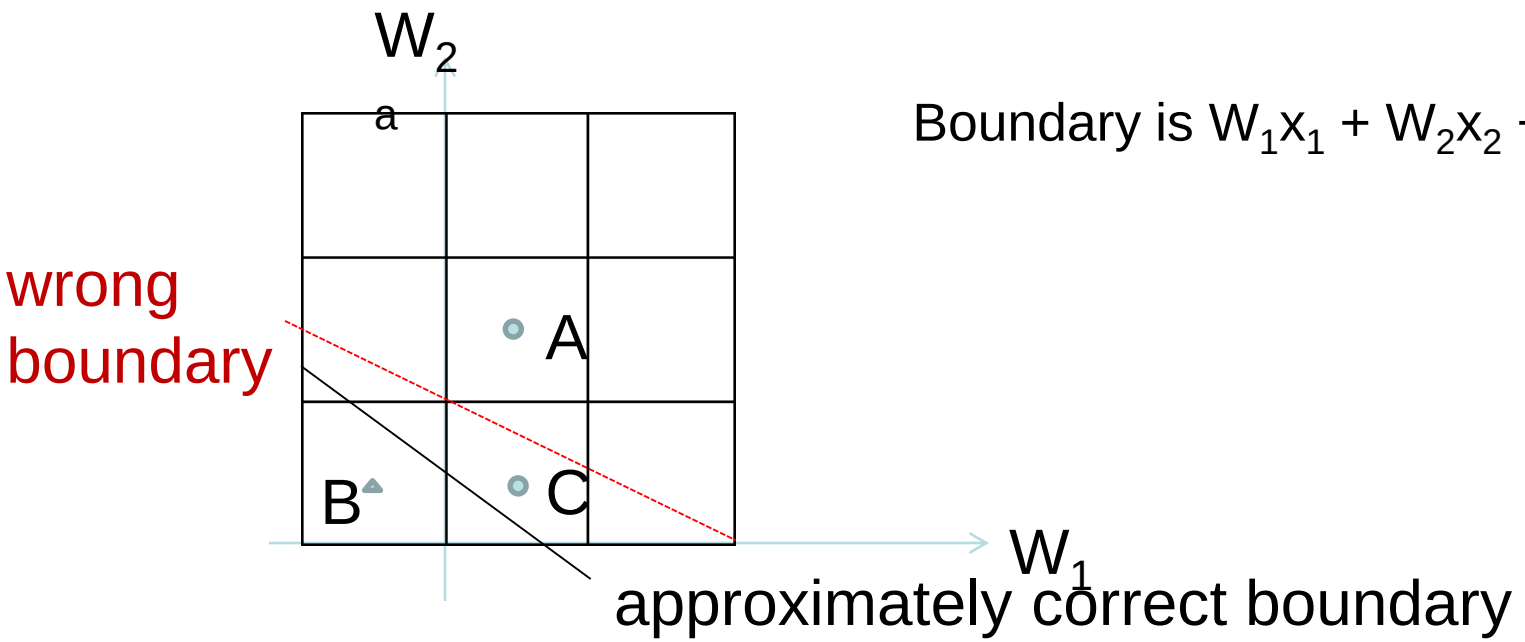
C=[(.5,.5),+1]
in = (.5)(1) + (.5)(2) – 2 = -.5
g(in) = -1; Err = 1-(-1)=2

Let α=.5

W1 <- W1 + .5(2) (.5) leaving out g'
  <- 1 + 1(.5) = 1.5
W2 <- W2 + .5(2) (.5)
  <- 2 + 1(.5) = 2.5
θ <- θ + .5(+1 – (-1))
θ <- -2 + .5(2) = -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Back Propagation

- Simple single layer networks with feed forward learning were not powerful enough.
- Could only produce simple linear classifiers.
- More powerful networks have multiple hidden layers.
- The learning algorithm is called back propagation, because it computes the error at the end and propagates it back through the weights of the network to the beginning.

The backpropagation algorithm (slightly different from text)

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
$w_{i,j} \leftarrow$ a small random number
repeat
for each example **(x,y)** in *examples* **do**
/* Propagate the inputs forward to compute the outputs. */
for each node i in the input layer **do** // Simply copy the input values.
$a_i \leftarrow x_i$
for $l = 2$ to L **do** // Feed the values forward.
for each node j in layer l **do**
$in_j \leftarrow \sum_i w_{i,j}\, a_i$
$a_j \leftarrow g(in_j)$
for each node j in the output layer **do** // Compute the error at the output.
$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
/* Propagate the deltas backward from output layer to input layer */
for $l = L - 1$ **to** 1 **do**
for each node i in layer l **do**
$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$ // "Blame" a node as much as its weig
/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.
until some stopping criterion is satisfied

return *network*

Let's break it into steps.

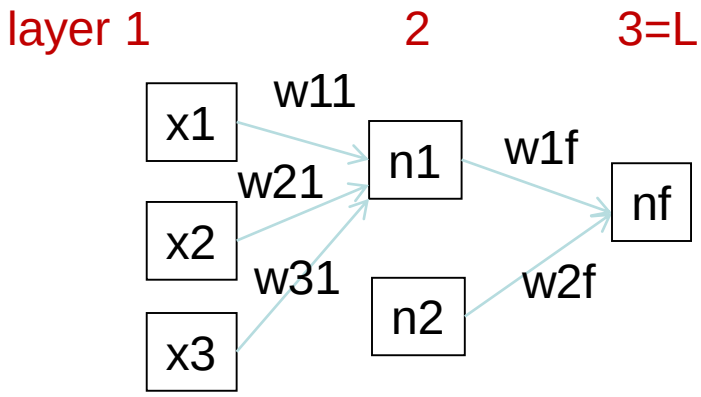
The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

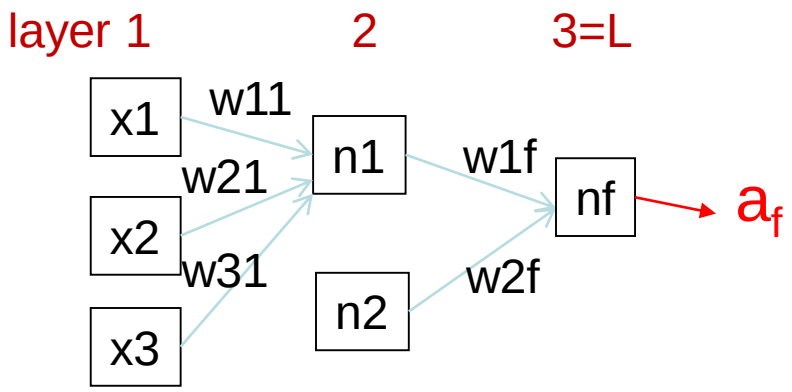
inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
$w_{i,j} \leftarrow$ a small random number



Forward Computation

repeat
  for each example **(x,y)** **in** *examples* **do**
    /* Propagate the inputs forward to compute the outputs. */
    for each node i in the input layer **do** // Simply copy the input values.
      $a_i \leftarrow x_i$
    for $l = 2$ to L **do** // Feed the values forward.
      for each node j in layer l **do**
        $in_j \leftarrow \sum_i w_{i,j}\, a_i$
        $a_j \leftarrow g(in_j)$

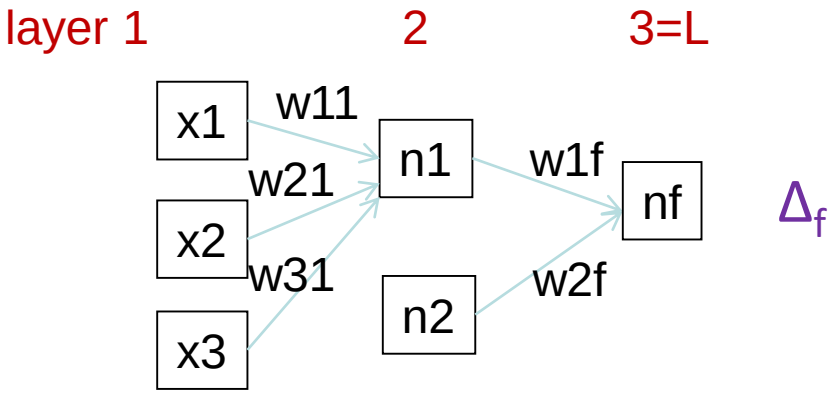


Backward Propagation 1

for each node j in the output layer **do** // Compute the error at the output.

$$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$$

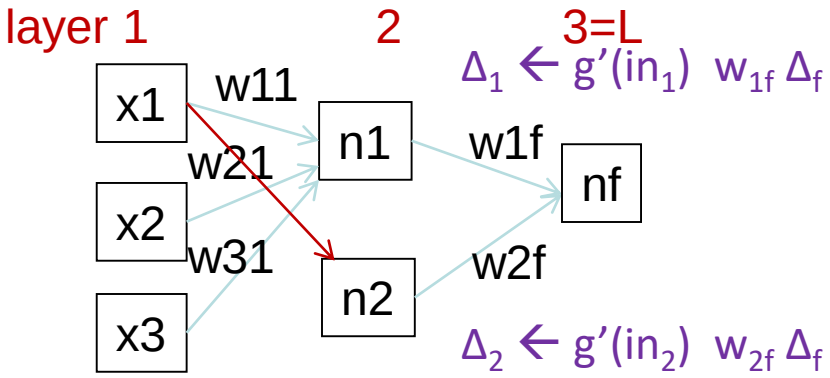
- Node nf is the only node in our output layer.
- Compute the error at that node and multiply by the
- derivative of the weighted input sum to get the change delta.



Backward Propagation 2

/* Propagate the deltas backward from output layer to input layer */
for $l = L - 1$ **to** 1 **do**
    for each node i in layer l **do**
        $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ // "Blame" a node as much as its weig

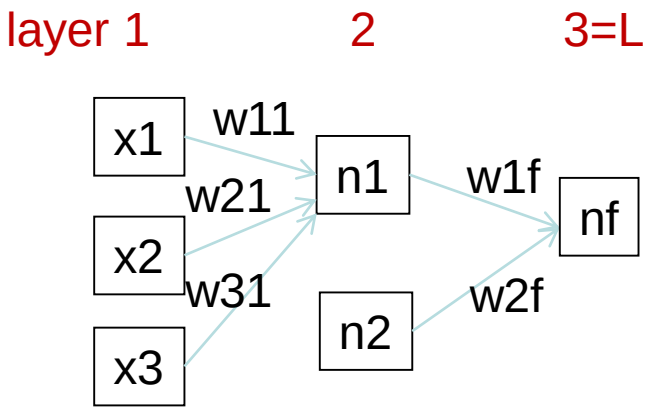
- At each of the other layers, the deltas use
  - the derivative of its input sum
  - the sum of its output weights
  - the delta computed for the output error



Backward Propagation 3

/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.

Now that all the deltas are defined, the weight updates just use them.



Back Propagation Summary

- Compute delta values for the output units using observed errors.
- Starting at the output-1 layer
  - repeat
    - propagate delta values back to previous layer
    - update weights between the two layers
  - till done with all layers
- This is done for all examples and multiple epochs, till convergence or enough iterations.

Time taken to build model: 16.2 seconds

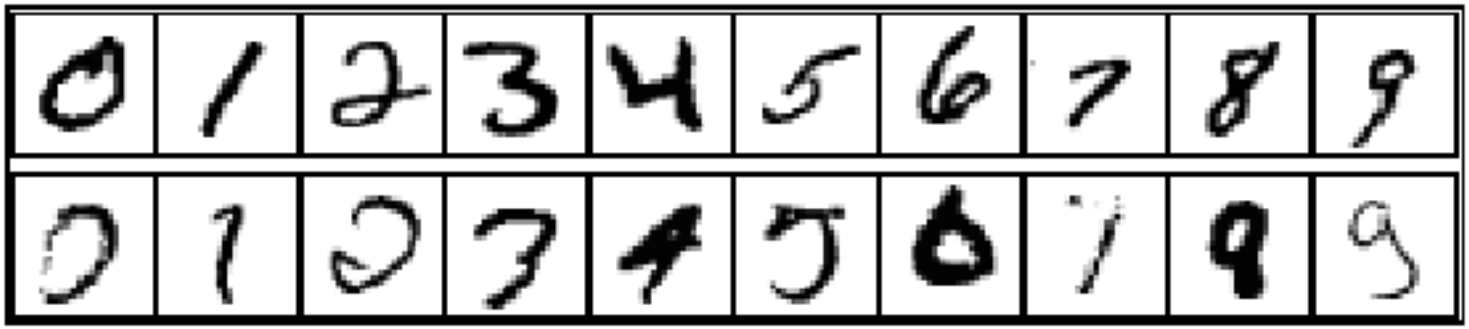
Correctly Classified Instances	307	80.3665 % (did not boost)
Incorrectly Classified Instances	75	19.6335 %
Kappa statistic	0.6056	
Mean absolute error	0.1982	
Root mean squared error	0.41	
Relative absolute error	39.7113 %	
Root relative squared error	81.9006 %	
Total Number of Instances	382	

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.706	0.103	0.868	0.706	0.779	0.872	cal
	0.897	0.294	0.761	0.897	0.824	0.872	dor
W Avg.	0.804	0.2	0.814	0.804	0.802	0.872	

=== Confusion Matrix ===

a	b	<-- classified as
132	55	a = cal
20	175	b = dor

Handwritten digit recognition



3-nearest-neighbor = 2.4% error

400–300–10 unit MLP = 1.6% error

LeNet: 768–192–30–10 unit MLP = 0.9% error

Current best (kernel machines, vision algorithms) $\approx$ 0.6% error