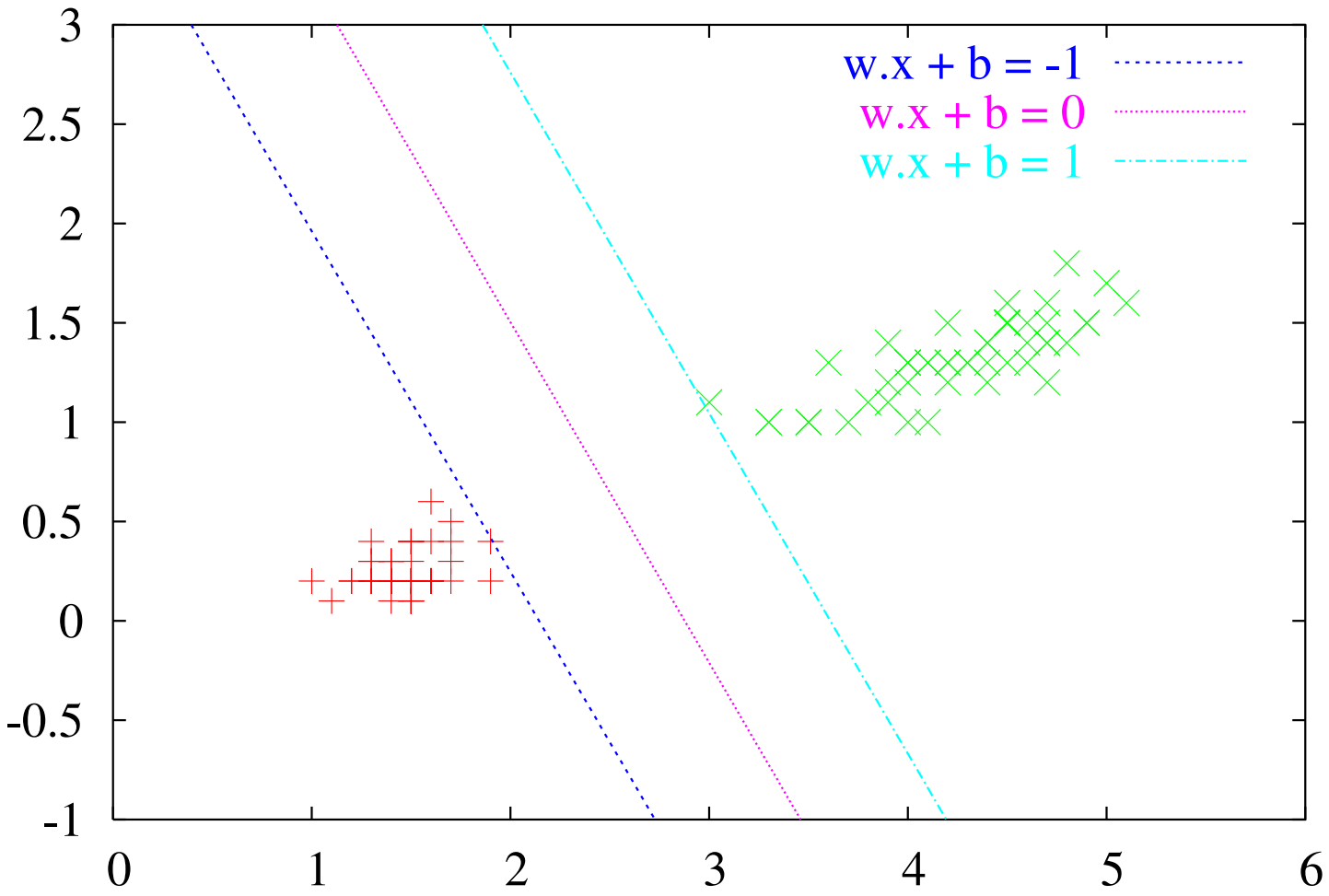
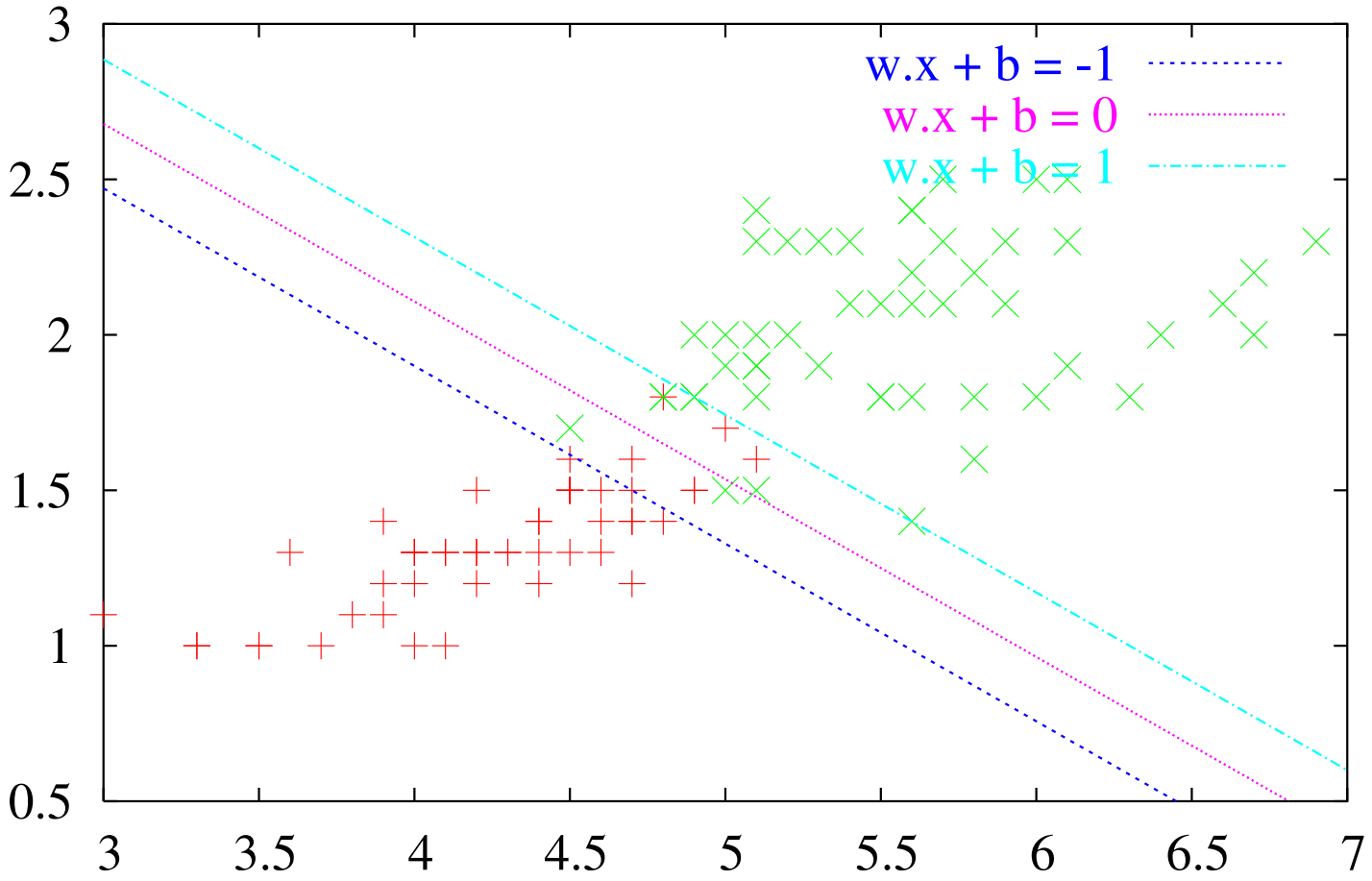
Learning, Part 2

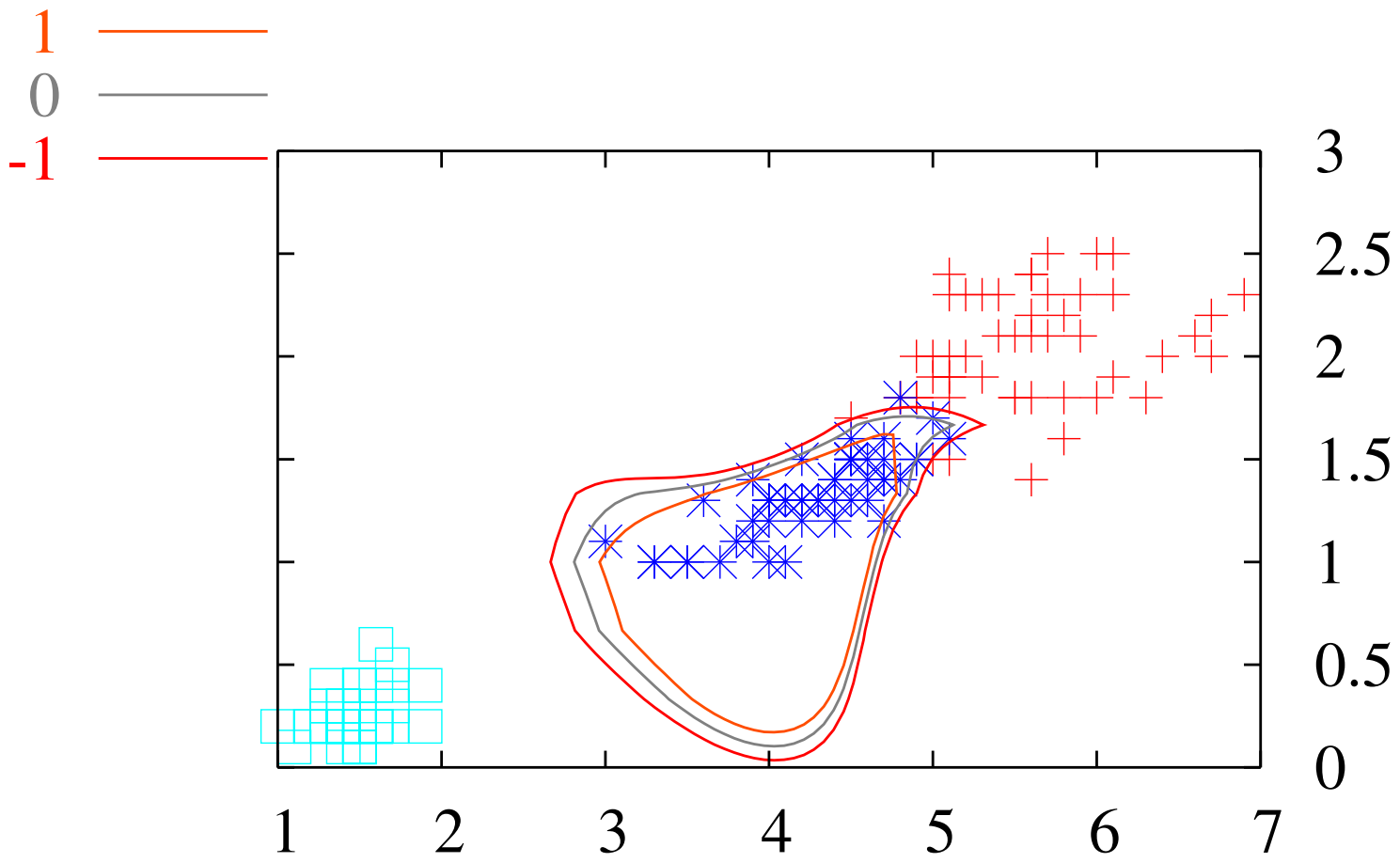
Zero Error Linear Decision Boundary



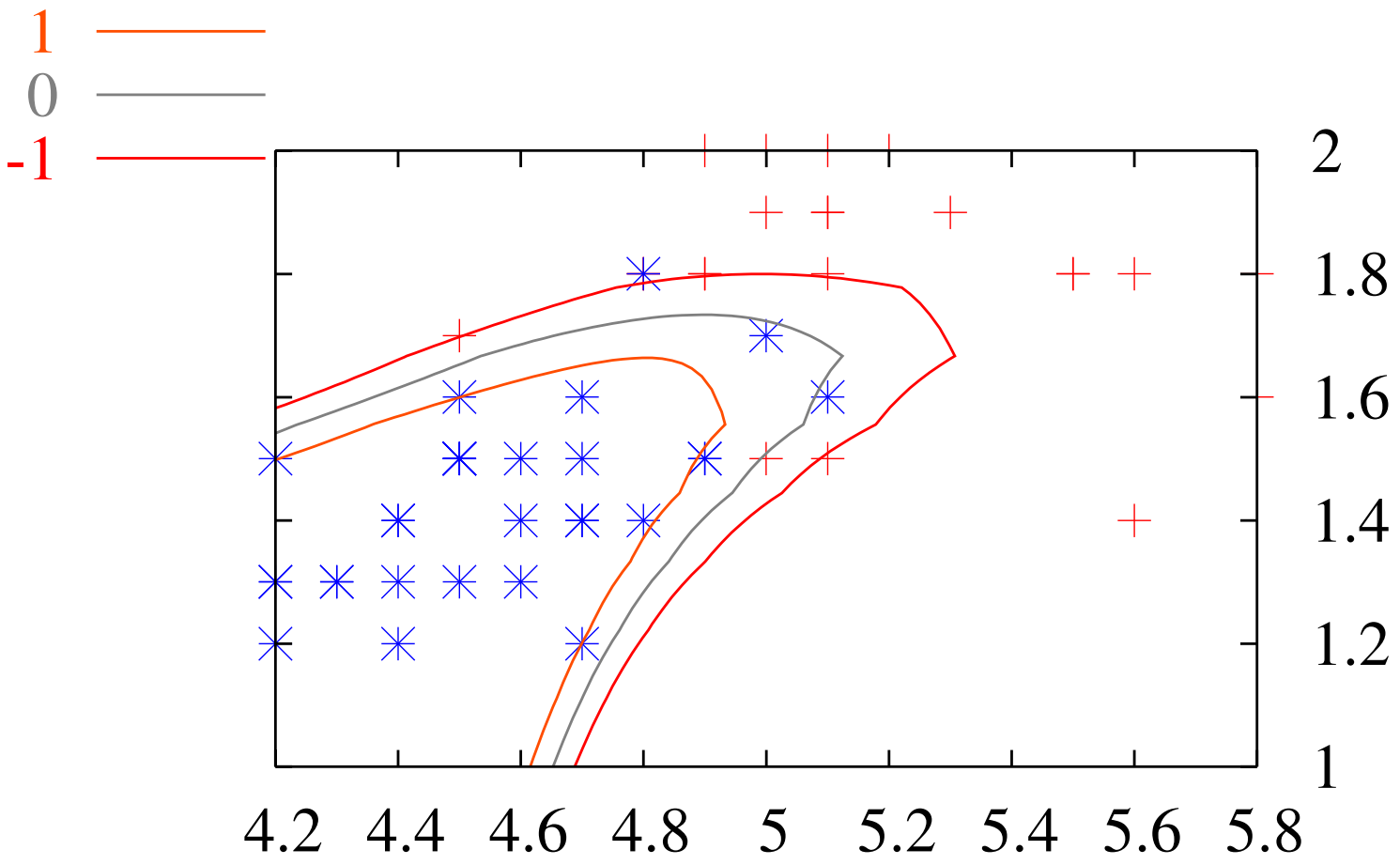
Low Error Linear Decision Boundary



Nonlinear Decision Boundary



Zoom on Nonlinear Decision Boundary



Motivation for Neural Networks

- Neural networks are inspired by neurons and their connections in the brain.
- An artificial neuron, called a *unit*, has inputs and an output.
- The output can be connected to other units.
- Typically, the output of a unit is computed by a linear function of its inputs passed through an activation function.
- Learning is adjusting weights to reduce error.
- Advantage: can learn non-linear functions.
- Disadvantage: more parameters and no guarantee of optimality.

Feedforward Neural Networks

- A *feed-forward* neural network is the most common type. Its units are organized as a directed acyclic graph.
- *Input units* simply output the values of the input features.
- *Hidden units* input the values of other units and produce outputs for other units.
- *Output units* produce predictions of output features.
- One standard model is an inital "layer" of input units, which feed into a single layer of hidden units, which feed into the output units.

Computing Outputs

Procedure *NNOutput*(e, H, O)

Inputs $e = \mathbf{x}$: the inputs of an example e

H: hidden units with activation function f_H and weights $\mathbf{w}_{H_j}$ for each hidden unit H_j

O: output unit with activation function f_O and weights $\mathbf{w}_O$

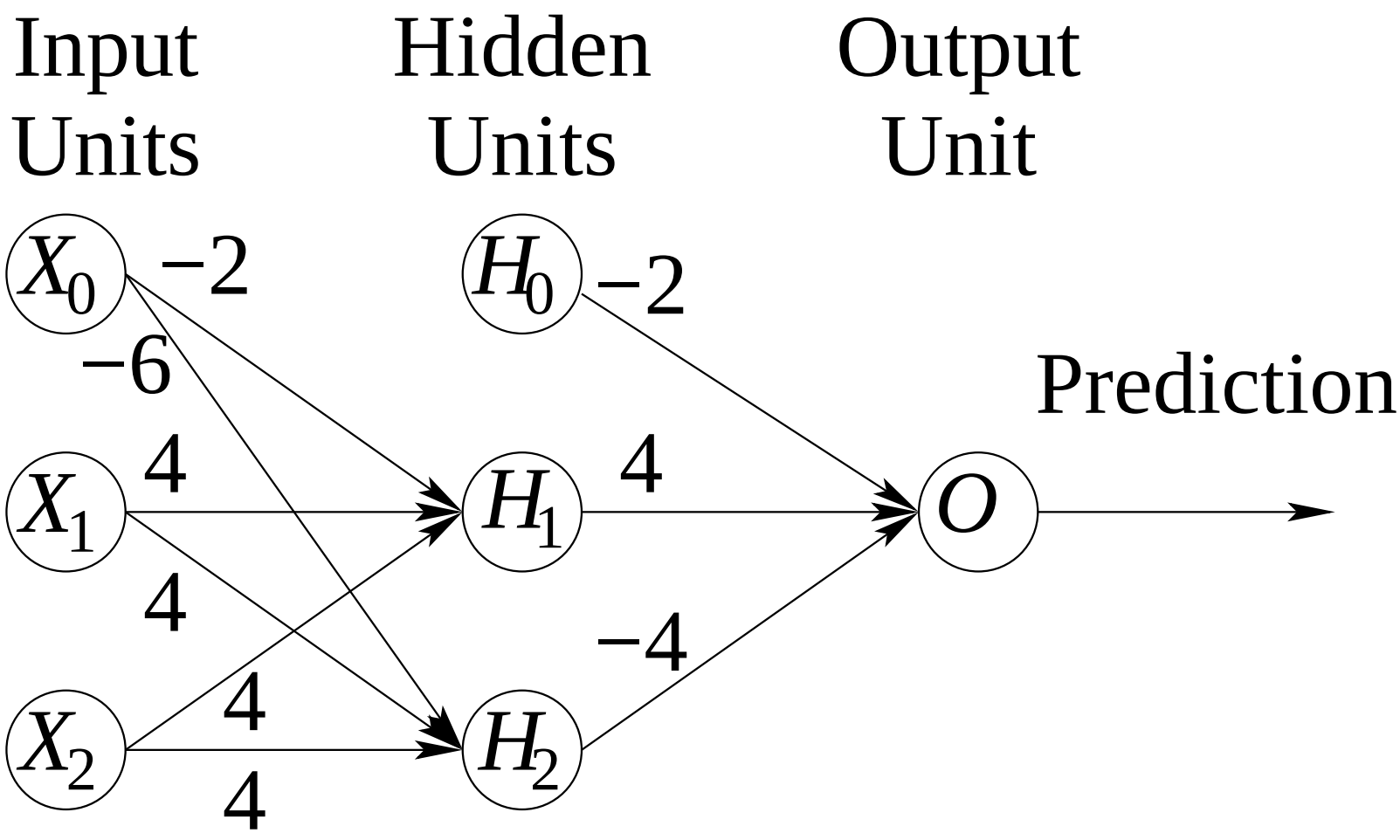
for each hidden unit H_j

$h_j \leftarrow f_H(\mathbf{w}_{H_j} \cdot \mathbf{x})$

$o \leftarrow f_O(\mathbf{w}_O \cdot \mathbf{h})$

return o

Xor Network



Computing for Xor

Suppose $f(z) = 1/(1 + e^{-z})$ is the activation function. X_0 and H_0 always output 1.

Y	X_0, X_1, X_2	H_1	H_2	O
0	1, 0, 0	$f(-2) \approx 0.12$	$f(-6) \approx 0.00$	$f(-1.53) \approx 0.18$
1	1, 0, 1	$f(2) \approx 0.88$	$f(-2) \approx 0.12$	$f(1.05) \approx 0.74$
1	1, 1, 0	$f(2) \approx 0.88$	$f(-2) \approx 0.12$	$f(1.05) \approx 0.74$
0	1, 1, 1	$f(6) \approx 1.00$	$f(2) \approx 0.88$	$f(-1.53) \approx 0.18$

For example for $X_0, X_1, X_2 = 1, 0, 1$, then

$H_1 = f(-2 * 1 + 4 * 0 + 4 * 1) = f(2) \approx 0.88$

$H_2 = f(-6 * 1 + 4 * 0 + 4 * 1) = f(-2) \approx 0.12$

$O \approx f(-2 + 4 * 0.88 - 4 * 0.12) \approx f(1.05) \approx 0.74$

0.74 is closer to 1 then 0, so predict 1, which $= Y$.

Learning by Backpropagation

Procedure *NNLearn*$(e, H, O, \mathbf{h}, o, \eta)$

Inputs $e = (\mathbf{x}, y)$: inputs and output of example e

H: hidden units with activation function f_H and weights $\mathbf{w}_{H_j}$ for each hidden unit H_j

O: output unit with activation function f_O and weights $\mathbf{w}_O$

$\mathbf{h}$, o: outputs of hidden and output units

η: learning rate

$\delta_O \leftarrow$ derivative of $\mathbf{w}_O \cdot \mathbf{h}$ wrt error

$\mathbf{w}_O \leftarrow \mathbf{w}_O - \eta\delta_O\mathbf{h}$

for each hidden unit H_j

$\delta_{H_j} \leftarrow \delta_O *$ derivative of $\mathbf{w}_{H_j} \cdot \mathbf{x}$ wrt $\mathbf{w}_O \cdot \mathbf{h}$

$\mathbf{w}_{H_j} \leftarrow \mathbf{w}_{H_j} - \eta\delta_{H_j}\mathbf{x}$

Comments on Backpropagation

- If squared error and $f(z) = 1/(1 + e^{-z})$
  $\delta_O \leftarrow o(1 - o)(o - y)$
  $\delta_{H_j} \leftarrow h_j(1 - h_j)(w_{O,j})\delta_O$
- *NNLearn* needs to be applied for many epochs (passes over the examples).
- The weights need to be initialized to small random values.
- Multiple runs might be needed to find good weights (and hyperparameters such as number of hidden units).
- More sophisticated gradient descent algorithms are much faster than backpropagation.

SVMs

- A *support vector machine* (SVM) assigns a weight α_i to each training example $(\mathbf{x}_i, y_i)$ ($\mathbf{x}_i$ is a vector of the values of the input features, and y_i is either -1 or 1).
- A [simplified] SVM computes a value on $\mathbf{x}$ by summing over all examples:

  $$h(\mathbf{x}) = \sum_i \alpha_i \, y_i \, k(\mathbf{x}, \mathbf{x}_i)$$

  where k is a *kernel function*.
- Learning is by optimizing the error function:

  minimize $\|h\|^2/2 + C \sum_i \max(0, 1 - y_i \, h(\mathbf{x}_i))$
  subject to $0 \leq \alpha_i \leq C$
  where $\|h\|$ is the size of h in kernel space

Kernel Functions

- The effect of a kernel function is to expand the number of features for linear classification.
- E.g., the kernel function $k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$ corresponds to a dot product of all the quadratic combinations of input features. If $\mathbf{x}_i = (1, a, b)$ and $\mathbf{x}_j = (1, c, d)$, then

$$(\mathbf{x}_i \cdot \mathbf{x}_j)^2 = (1 + ac + bd)^2$$
$$= 1 + 2ac + 2bd + a^2c^2 + 2acbd + b^2d^2 =$$
$$(1, \sqrt{2}a, \sqrt{2}b, a^2, \sqrt{2}ab, b^2) \cdot (1, \sqrt{2}c, \sqrt{2}d, c^2, \sqrt{2}cd, d^2)$$

- The gaussian kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$ is a common choice. [γ is an extra parameter to choose.]

Slow SVM Learning Algorithm

Procedure *SVMLearner*(X, Y, E, k, C)

Inputs X, Y: input features and target feature

E: set of training examples

k: kernel function

C: "soft margin" constant

initialize all example weights α_i to zero

repeat until termination

for each example $e = (\mathbf{x}_i, y_i) \in E$

$\widehat{y} \leftarrow \sum_j \alpha_j \, y_j \, k(\mathbf{x}_i, \mathbf{x}_j)$

if $y_i \widehat{y} < 1$ and $\alpha_i < C$ increase α_i

if $y_i \widehat{y} > 1$ and $\alpha_i > 0$ decrease α_i

return α

Comments on SVMs

- Each example with $\alpha_i > 0$ is called a *support vector*.
- The solution will have $\alpha_i = 0$ if $y_i\widehat{y} > 1$ and $\alpha_i = C$ if $y_i\widehat{y} < 1$ with inbetween values for $y_i\widehat{y} = 1$.
- An SVM has a global minimum with no local minima, though actual algorithms only get very close.
- Multiple runs might be needed to find the best kernel function and soft margin constant.
- More sophisticated SVM algorithms are much faster.

The Nearest Neighbor Algorithm

- The k-nearest neighbor algorithm classifies a test ex. by finding the k closest training exs., returning the most common class.
- Suppose 10% noise (best possible test error is 10%). With enough training exs., a test ex. will agree with its nearest neighbor with prob. $(.9)(.9) + (.1)(.1) = .82$ (both not noisy or both noisy) and disagree with prob. $(.9)(.1) + (.1)(.9) = .18$.
- 1-NN converges to less than twice optimal error (3-NN to less than 32% higher).
- Problems: distance measure, number of exs. needed, find NNs efficiently.

Ensemble Learning

- There are many algorithms for learning a single hypothesis.
- *Ensemble learning* will learn many hypotheses: run different algorithms or run the same algorithm on different training sets.
- *Bagging* runs a learning algorithm on repeated subsamples of the training set.
- If there are n examples, then a subsample of n examples is generated by sampling with replacement.
- On a test example, each hypothesis casts 1 vote for the class it predicts.

Boosting

- In *boosting*, the hypotheses are learned in sequence.
- Both hypotheses and examples have weights with different purposes.
- After each hypothesis is learned, its weight is based on its error rate, and the weights of the training examples (initially all equal) are also modified.
- On a test example, when each hypothesis predicts a class, its weight is the size of its vote. The ensemble predicts the class with the highest vote.

Example Boosting Algorithm

Procedure *AdaBoost*(E, A, t)
  Inputs E, A: examples and learning algorithm
    n: number of hypotheses to generate
  initialize example weights $\mathbf{w}$ to 1/number of exs.
  for i from 1 to n
    $h[i] \leftarrow A(E, \mathbf{w})$
    $\epsilon \leftarrow$ sum weights of exs. missed by $h[i]$
    for j from 1 to number of examples
      if $h[i]$ is correct on example j
        then $w[j] \leftarrow 0.5 * w[j]/(1-\epsilon)$
        else $w[j] \leftarrow 0.5 * w[j]/\epsilon$
    weight of $h[i] \leftarrow \log((1-\epsilon)/\epsilon)$
  return $h[1 \ldots n]$ and their weights

Example Run of AdaBoost

Using the 14 examples as a training set:

- Example weights are initialized to $1/14$.
- The hypothesis windy $=$ false $\leftrightarrow$ class $=$ pos is wrong on 5 of the 14 examples.
- The weights of the correctly classified examples are multiplied by $14/18$, and incorrectly classified by $14/10$. An example will have a weight of $1/18$ or $1/10$.
- This hypothesis has a weight of $\log(9/5)$.
- Note that after weight updating, the sum of the correctly classified examples equals the sum of the incorrectly classified examples.

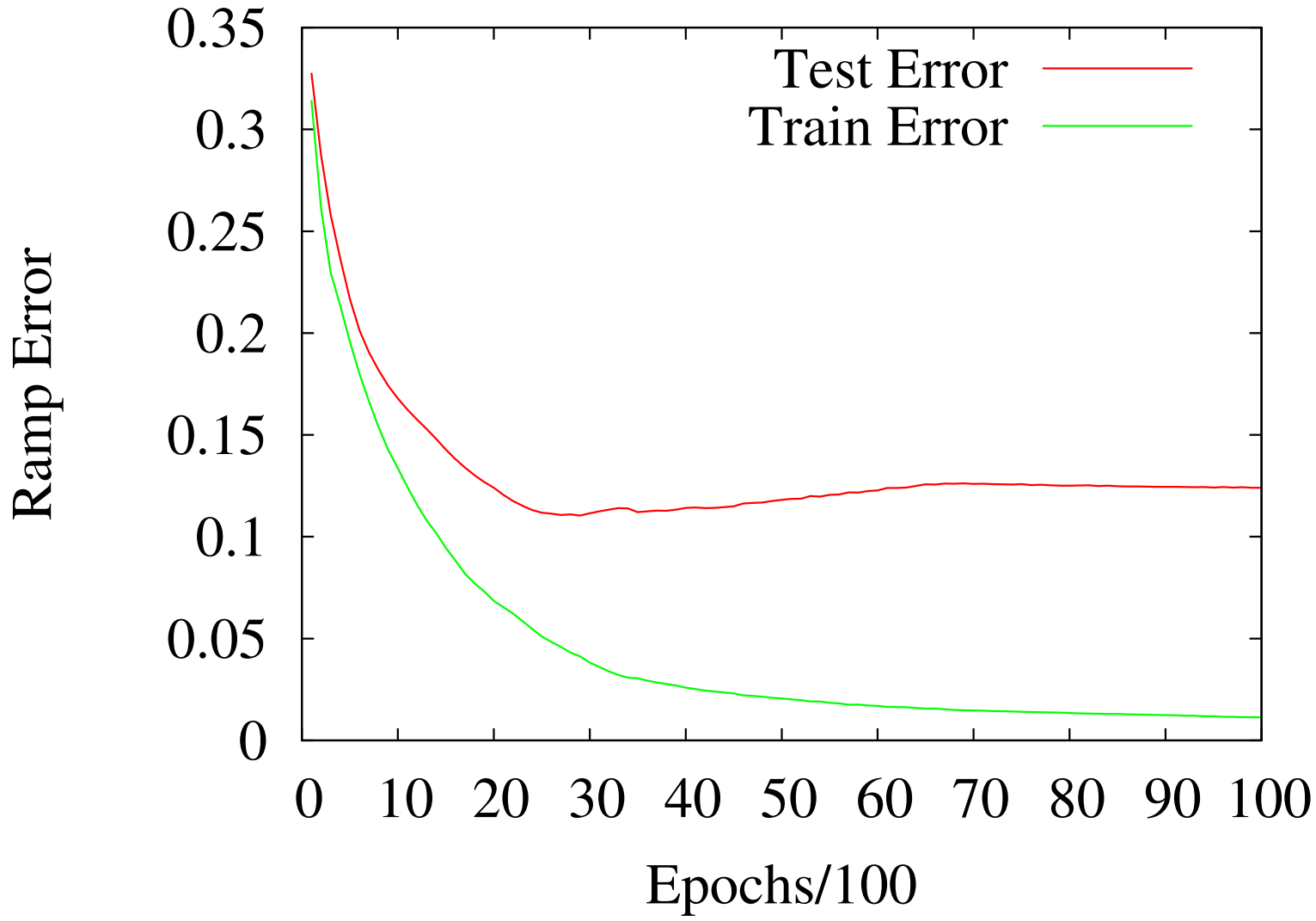
Example Run of AdaBoost, Continued

- The next hypothesis must be different from the previous one to have error less than $1/2$.
- Now the hypothesis outlook $=$ overcast $\leftrightarrow$ class $=$ pos has an error rate of $29/90$.
- The weights of the correctly classified examples are multiplied times $90/122$, and the incorrectly classified times $90/58$.
- This hypothesis has a weight of $\log(61/29)$.

Overfitting

- *Overfitting* is when the learner uses regularities that appear in the training set, but do not appear in the test set.
- The learning algorithm might fit noise in the training data (outliers, random fluctuations).
- The learning algorithm might fit to unique values in the training data (id numbers).
- The learning algorithm might "memorize" the training examples (more parameters than examples).
- This can be observed when running the algorithm longer does better on the training set, but does worse on the test set.

Overfitting Example



Validation Set

- A *validation set* is one way to avoid overfitting.
- The original data is divided into a training set, a validation set and a test set (more about test set later).
- As the learning algorithm runs on the training set, keep track of its error on the validation set. Use the algorithm settings that minimize validation error.

k-Fold Cross-Validation

- In *k-fold cross-validation*, the data is divided into a training set and a test set.
- The training set is evenly divided into k *folds*.
- Then k times: Train the learning algorithm on $k-1$ folds and validate on the remaining fold.
- Average over the results.

Regularization

- *Regularization* is a way to avoid overfitting.
- A learning model might have many parameters.
  - A NN with 10 hidden units has over 10 times more parameters than a linear model.
  - In natural language tasks, a parameter for every word (millions of words) or every word-word combination.
- If a linear model has as many or more parameters than examples, then the examples can be "memorized".
- Regularization modifies the error function so there is a penalty for larger parameters.

Regularization Example

- Suppose we want to minimize hinge loss over a set of examples. That is, find $\mathbf{w}$ minimizing:

  $\sum_i \max(0, 1 - y_i * (\mathbf{w} \cdot \mathbf{x}_i))$
- Perhaps this formulation leads to overfitting. To regularize this, we can add a term that penalizes large weights.

  $\lambda(\mathbf{w} \cdot \mathbf{w})/2 + \sum_i \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))$
- The weight update in the *LinearLearn* algorithm becomes:

  $\mathbf{w} \leftarrow \mathbf{w}(1 - \eta\lambda/m) + \eta\delta\mathbf{x}$

  where m is the number of examples.
- Use validation to find λ.

How to Use a Test Set

- Recall: the original data is divided into a training set, a validation set and a test set.
- Do not look at the test set.
- Do all the processing and analysis you want using the training and validation sets.
- Do not look at the test set.
- Select a hypothesis (or a few hypotheses) by training on the training and validation sets combined, using the best settings on the best algorithm(s).
- Evaluate hypothesis (or hypotheses) on the test set.

Can I Use the Test Set Again?

- In theory, **no**. By reusing the test set, you are in effect using the test set for training (searching for an algorithm that does well on the test set).
- In practice, a test set can be reused once or twice without getting into too much trouble, probably, maybe.
- Using many different datasets (and so many different test sets) from a variety of domains can be used to empirically compare algorithms.
- Moral: Prepare a second test set (maybe a third or fourth) so you have one in reserve after a possible initial failure.

Error Rate and Comparing Algorithms

- What is the true error rate?
  - If ϵ is the test error rate, and m examples in the test set ($m > 100$), then a 95% confidence interval is:

$$\epsilon \pm 1.96\sqrt{\frac{\epsilon\left(1-\epsilon\right)}{m-1}}$$

- Is algorithm A better than algorithm B?
  - Use the paired-difference test (next page assuming more than 100 examples in test set).

Paired-Difference Test

Let h_1 and h_2 be the two hypotheses to compare.
For each $(\mathbf{x}_i, y_i) \in E$:

$$\text{let } d_i = \begin{cases} 1 & \text{if } h_1(x_i) = y_i \neq h_2(x_i) \\ 0 & \text{if } h_1(x_i) = h_2(x_i) \\ -1 & \text{if } h_1(x_i) \neq y_i = h_2(x_i) \end{cases}$$

Calculate:

$$u = \frac{\sum_{i=1}^{m} d_i}{m} \qquad s^2 = \frac{\sum_{i=1}^{m} (d_i - u)^2}{m-1} \qquad z = \frac{u}{s/\sqrt{m}}$$

If $z \geq 1.96$, answer "h_1 is better than h_2".
If $z \leq -1.96$, answer "h_1 is worse than h_2".
The significance is 0.05.
If $|z| < 1.96$, then answer "I don't know".