CS 4700:
Foundations of Artificial Intelligence

Carla P. Gomes / Bart Selman
gomes@cs.cornell.edu

Module:
Perceptron Learning
R&N 18.7

Derivation of a learning rule for Perceptrons Minimizing Squared Errors

Threshold perceptrons have some advantages , in particular

→ Simple learning algorithm that fits a threshold perceptron to any linearly separable training set.

Key idea: Learn by adjusting weights to reduce error on training set.

→ update weights repeatedly (epochs) for each example.

We' ll use:

→Sum of squared errors (e.g., used in linear regression), classical error measure

→Learning is an optimization search problem in weight space.

Derivation of a learning rule for Perceptrons Minimizing Squared Errors

Let S = {($\mathbf{x}_i$, y_i): i = 1, 2, ..., m} be a training set. (Note, **x** is a vector of inputs, and y is the vector of the true outputs.)

Let $h_{\mathbf{w}}$ be the perceptron classifier represented by the weight vector **w**.

Definition:

$$E(\mathbf{x}) = Squared\ Error(\mathbf{x}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2$$

Derivation of a learning rule for Perceptrons Minimizing Squared Errors

The squared error for a single training example with input **x** and true output y is:

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

Where $h_{\mathbf{w}}$ **(x)** is the output of the perceptron on the example and y is the true output value.

We can use the gradient descent to reduce the squared error by calculating the partial derivatives of E with respect to each weight.

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j \end{aligned}$$

Note: g' (in) derivative of the activation function. For sigmoid g' =g(1-g). For threshold perceptrons, where g' (n) is undefined, the original perceptron rule simply omitted it.

$$\frac{\partial E}{\partial W_j} = -Err \times g'(in) \times x_j$$

Gradient descent algorithm → we want to reduce , *E*, *f*or each weight w_i , change weight in direction of steepest descent:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

α learning rate

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

Intuitively:

Err = y – h_W(x) positive

output is too small → weights are increased for positive inputs and decreased for negative inputs.

Err = y – h_W(x) negative

→ opposite

Perceptron Learning: Intuition

Rule is intuitively correct!

Greedy Search:

Gradient descent through weight space!

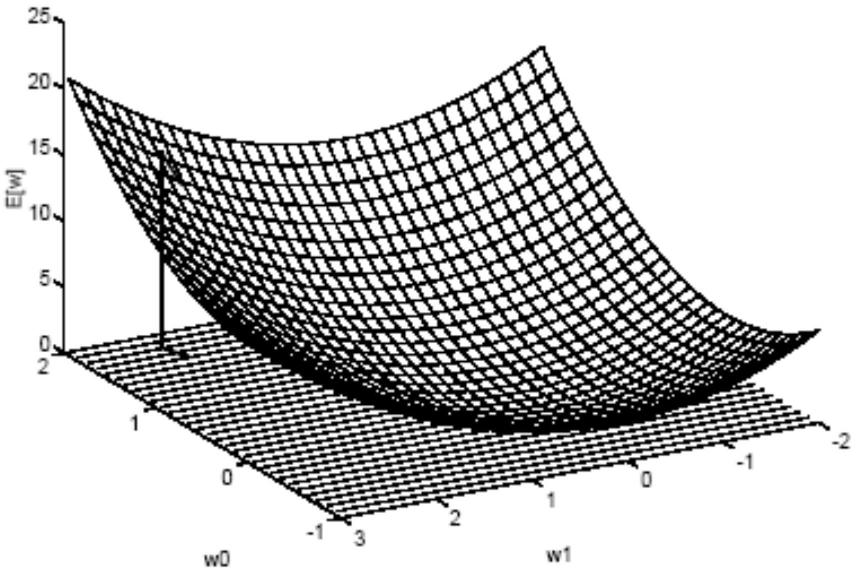
Surprising proof of convergence:

Weight space has no local minima!

With enough examples, it will find the target function!

(provide α not too large)

Gradient descent in weight space



From T. M. Mitchell, *Machine Learning*

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

Perceptron learning rule:

1. Start with random weights, $\mathbf{w} = (w_1, w_2, \ldots, w_n)$.
2. Select a training example $(\mathbf{x},y) \in S$.
3. Run the perceptron with input $\mathbf{x}$ and weights $\mathbf{w}$ to obtain g
4. Let α be the training rate (a user-set parameter).

$$\forall w_i, w_i \leftarrow w_i + \Delta w_i,$$

where

$$\Delta w_i = \alpha(y - g(in))g'(in)x_i$$

5. Go to 2.

Epoch → cycle through the examples

Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small.

The stochastic gradient method selects examples randomly from the training set rather than cycling through them.

Perceptron Learning: Gradient Descent Learning Algorithm

function PERCEPTRON-LEARNING(*examples*, *network*) **returns** a perceptron hypothesis
  inputs: *examples*, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
    network, a perceptron with weights $W_j,\ j = 0 \ldots n$, and activation function g

  repeat
    for each e **in** *examples* **do**
      $in \leftarrow \sum_{j=0}^{n} W_j\, x_j[e]$
      $Err \leftarrow y[e] - g(in)$
      $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(*network*)

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.