

# CS 5600/6600/7890: Lecture 3

## Artificial Neural Networks (ANNs): Backpropagation and Gradient Descent

Vladimir Kulyukin  
Department of Computer Science  
Utah State University

# Outline

Review

Designing Neural Networks

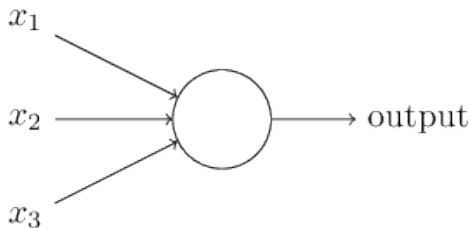
Training Neural Networks

Bird's Eye View on Backpropagation

Gradient Descent

# Perceptron

A perceptron takes several binary inputs  $x_1, x_2, \dots, x_n$  and produces a single binary output.



# Perceptron's Output

Connections from the inputs to the perceptron are governed by weights  $w_1, w_2, \dots, w_n$ .

Weights are real numbers expressing the importance of the respective inputs to the output.

The neuron's output, 0 or 1, is determined by whether the weighted sum  $\sum_j w_j x_j$  is  $\leq$  or  $>$  than some threshold  $\theta$ .

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \theta \\ 1 & \text{if } \sum_j w_j x_j > \theta. \end{cases}$$

# Computational Universality of Perceptrons

It is possible to simulate NAND gates with perceptrons. Since NAND gates can simulate any boolean function, perceptrons can also simulate any boolean function.

The big deal is the existence of *learning algorithms* that can automatically tune the weights and biases in networks of artificial neurons. This makes such networks radically different from circuits.

# When Networks Learn Well

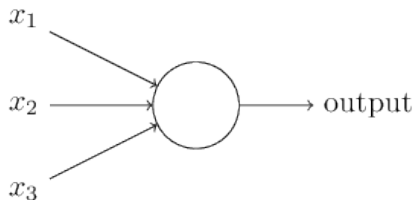
A network of perceptrons do not have the proportionality of weight/bias and output.

A new type of neuron, a *sigmoid* neuron, has been introduced to overcome this problem. Sigmoid neurons are similar to perceptrons but modified so that small changes cause only small changes in the output.

# Sigmoid Neuron

Sigmoid neurons are similar to perceptrons modified so that small changes in their weights and biases cause only a small change to their output.

A sigmoid neuron takes several inputs  $x_1, x_2, \dots, x_n$  but, unlike in a perceptron, these inputs are real numbers between 0 and 1. The output is also a real number between 0 and 1, not just 0 or 1, as with a perceptron.



# Sigmoid Neuron's Output

The output of a sigmoid neuron with inputs the  $x_1, x_2, \dots, x_n$ , the weights  $w_1, w_2, \dots, w_n$ , and the bias  $b$  is

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

or

$$\sigma(w \cdot x + b) = \frac{1}{1 + \exp(-\sum_j w_j x_j - b)},$$

where  $z = w \cdot x + b$ .



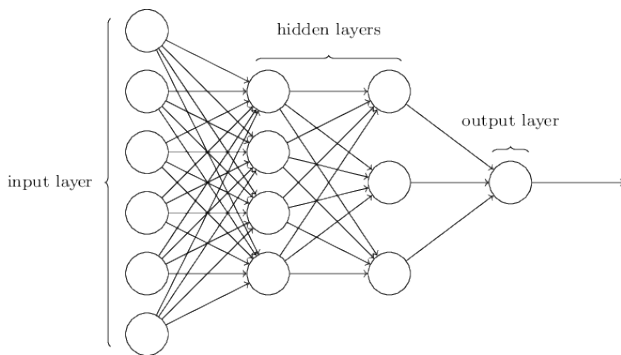
# What Do Sigmoid Neurons Buy Us?

The use of sigmoid neurons makes it much more likely that a small change in a weight/bias causes only a small change in output. We can use this fact in modify the weights/biases to get our network to behave more smoothly.

The function  $\sigma$  turns sigmoid neurons into smoothed perceptrons.

# Layers

Neural networks consists of layers. Each layer consists of neurons (either perceptrons or sigmoids). There can be arbitrarily many layers stacked on each other. The leftmost layer is called the *input* layer. The rightmost layer is called the *output* layer. The layers in between the input layer and the output layer, if there are any, are called the *hidden* layers.



# Synapses and Neurons

The links between neurons, sometimes called *synapses*, indicate which neurons input their outputs to which other neurons. These links have weights.

The job of a neuron is to take the inputs from the synapses and apply its activation function to them.

The job of a synapse is to take the value from the input neuron, multiply the value by its weight, and output the result into the output neuron.

# Parameters vs. Hyperparameters

The neural network differ from each other in terms of their layers, types of neurons, synapses, and activation functions. These are called *hyperparameters*. Hyperparameters cannot be changed.

The parameters of a neural network are weights and biases that can be manipulated and/or learned.

The neural networks also differ from each other by their learning procedures.

# Outline

Review

Designing Neural Networks

Training Neural Networks

Bird's Eye View on Backpropagation

Gradient Descent

# Key Ideas of Neural Computation

Neural networks use different types of neurons such as perceptrons and sigmoid neurons.

Neural networks use two fundamental algorithms: backpropagation and gradient descent.

There are some best practices that typically (but not always) improve neural network performance.

# Designing Input, Output, and Hidden Layers

The design of input and output layers is often straightforward. For example, if you're classifying  $32 \times 32$  grayscale images, then it makes sense to have  $32 \times 32 = 1024$  input neurons, each of which is a number between 0 and 255. If you're classifying  $64 \times 64$  grayscale images, you should have  $64 \times 64 = 4,096$  neurons.

If you're recognizing images, it makes sense to have 10 output nodes - one for each digit. If you're recognizing the presence/absence of a car in an image - 1 or 2 output neurons are enough.

There are no accepted rules for designing hidden layers. There are some heuristics that ANN researchers use but none of them are universally accepted.

The term *deep learning* refers to neural networks with many hidden layers.

# Building an ANN: Training/Testing Data

Let's suppose that we want to predict how the years of study and years of job experience correlate with our salaries.

We've decided that we'll build a  $2 \times 3 \times 1$  sigmoid network. In other words, 2 sigmoid neurons in the input layer (years of study and years of job experience), 3 sigmoid neurons in the hidden layers, and 1 sigmoid neuron in the output layer.

**Question:** What kind of data do we need to train/test this ANN?



## Building an ANN: Training/Testing Data

**Answer:** We need LOTS of 3-tuples  $(x_1, x_2, x_3)$ , where  $x_1$  is a person's years of study,  $x_2$  is the person's years of experience, and  $x_3$  is the person's salary.

# Building an ANN: Feedforward

The input to the neural network will be  $(x_1, x_2)$  of each  $(x_1, x_2, x_3)$ . Let's call it  $X$ . Let's refer to the ANN's output as  $\hat{Y}$ .

For each  $X = (x_1, x_2)$ , we want to compare how far the output of our network,  $\hat{Y}$ , is from  $x_3$ .  $x_3$  is called *ground truth*.

This is a classical example of a *feedforward* ANN where activation spreads from the input layer to the output layer.

# Building an ANN: Initializing Weights

There is one more question we need to answer before coding - how do we initialize weights?

We can use `numpy.random.randn(x, y)` that creates `x, y` numpy arrays of normally distributed random numbers with a mean of 0 and a variance of 1.

```
>>> import numpy as np
>>> np.random.randn(2, 2)
array([[ 0.35120796,  0.19578908],
       [-0.20131598,  0.06917832]])
>>> np.random.randn(2, 3)
array([[ 0.92847923,  0.10751104,  0.64730707],
       [ 0.81078793,  0.52196304,  0.59270607]])
```

# Building an ANN: Initializing Network

```
import numpy as np

def build_my_nn():
    # 1. seed random number generator
    np.random.seed(1)
    # 2. initiate 1st synapse matrix
    W1 = np.random.randn(2, 3)
    # 3. initiate 2nd synapse matrix
    W2 = np.random.randn(3, 1)
    # 4. return 2-tuple of synapse matrices
    return W1, W2
```

# Sigmoid Activation Function and its Derivative

Here is the sigmoid function, its derivative and derivative approximation:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) = \frac{e^{-x}}{(1 + e^{-x})^2}.$$

$$\frac{d}{dx}\sigma(x) \approx x(1 - x).$$

# Building an ANN: Sigmoid Activation Function

```
def sigmoid(x, deriv=False):  
    if (deriv == True):  
        #return x * (1 - x)  
        return np.exp(-x)/((1 + np.exp(-x))**2)  
    return 1 / (1 + np.exp(-x))
```

If you use the real derivative, beware of this warning.

```
Warning (from warnings module):
```

```
    return 1 / (1 + np.exp(-x))
```

```
RuntimeWarning: overflow encountered in exp
```

# Building an ANN: From Input to Output

How do we compute  $\hat{Y}$  from  $X = (x_1, x_2)$ ? This process of pushing the input through the network from the input to the output layer is called *feedforward*.

Let's adopt this notation.

1.  $W^{(1)}$  is the matrix of weights of the synapses from layer 1 (input) to layer 2 (hidden);
2.  $f(z)$  is the activation function;
3.  $W^{(2)}$  is the matrix of weight of the synapses from layer 2 (hidden) to layer 3 (output).

# Building an ANN: From Input to Output

Feedforward in our  $2 \times 3 \times 1$  network is defined by the following four equations:

1.  $Z^{(2)} = XW^{(1)}$ ;  $Z^{(2)}$  is the input to layer 2;
2.  $a^{(2)} = f(Z^{(2)})$ ;  $a^{(2)}$  is the output of layer 2;
3.  $Z^{(3)} = a^{(2)}W^{(2)}$ ;  $Z^{(3)}$  is the input to output layer (layer 3);
4.  $\hat{Y} = f(Z^{(3)})$ ;  $\hat{Y}$  is the output of output layer (layer 3).

where  $W^{(1)}$  is the matrix of synapse weights from layer 1 (input) to layer 2 (hidden),  $f(z)$  is the activation function, and  $W^{(2)}$  is the matrix of synapse weights from layer 2 (hidden) to layer 3 (output).



# Building an ANN: Feedforward: Equations 1 and 2

Equations 1 and 2:

$$Z^{(2)} = XW^{(1)}.$$

$$a^{(2)} = f(Z^{(2)}).$$

Here is Python:

```
Z2 = np.dot(X, W1)
a2 = sigmoid(Z2)
```

# Building an ANN: Feedforward: Equations 3 and 4

Equations 3 and 4:

$$Z^{(3)} = a^{(2)} W^{(2)}.$$

$$\hat{Y} = f(Z^{(3)}).$$

Here is Python:

```
Z3 = np.dot(a2, W2)
yHat = sigmoid(Z3)
```

# Building an ANN: Complete Feedforward

```
## equation 1
Z2 = np.dot(X, W1)
## equation 2
a2 = sigmoid(Z2)
## equation 3
Z3 = np.dot(a2, W2)
## equation 4
yHat = sigmoid(Z3)
```

# Outline

Review

Designing Neural Networks

Training Neural Networks

Bird's Eye View on Backpropagation

Gradient Descent

# Error/Cost Function

We have to decide how far our output  $\hat{y}$  is from the ground truth  $y$ . We can use this simple formula.

$$\hat{y}_{err} = y - \hat{y}.$$

If negative weights cause numerical instability, we can use

$$\hat{y}_{err} = |y - \hat{y}|$$

or

$$\hat{y}_{err} = (y - \hat{y})^2.$$

# Quadratic Cost Function or Mean Squared Error

A more common way to measure how far the actual output of the network  $a$  is from the desired output  $y(x)$  is to use the quadratic cost function, aka mean squared error (MSE):

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

The aim of ANN training is to minimize the cost  $C(w, b)$ .

# Question

We can change the weights of our ANN by adjusting one weight at a time. What do you think of this approach?

## Answer: Curse of Dimensionality

Not a good approach! Too expensive for large networks. This is known as the **curse of dimensionality**. We have to be smarter.



# Backpropagation

Backpropagation is an algorithm of adjusting synapse weights on the basis of the output error, i.e.,  $\hat{y}_{err}$ , not one weight at a time but layer by layer.

In our network, we have to compute the adjustments for 2 layers, the output layer and the hidden layer.

The adjustment for the output layer is used to adjust  $W^{(2)}$ . The adjustment for the hidden layer is used to adjust  $W^{(1)}$ .

# Backpropagation: Adjustment for Output Layer

The adjustment for the output layer is used to adjust  $W^{(2)}$ .

$$\hat{y}_{err} = y - \hat{y}$$

$$\hat{y}_{\Delta} = \hat{y}_{err} \cdot \sigma'(\hat{y})$$

In Python:

```
yHat_error = y - yHat  
yHat_delta = yHat_error * sigmoid(yHat, deriv=True)
```

# Backpropagation: Adjustment for Hidden Layer

The adjustment for the hidden layer is used to adjust  $W^{(1)}$ .

$$a_{err}^{(2)} = \hat{y}_{\Delta} \cdot (W^{(2)})^T$$

$$a_{\Delta}^{(2)} = \hat{a}_{err}^{(2)} \cdot \sigma'(a^{(2)}).$$

In Python:

```
a2_error = yHat_delta.dot(W2.T)
a2_delta = a2_error * sigmoid(a2, deriv=True)
```

## Backpropagation: Adjusting $W^{(1)}$ and $W^{(2)}$

$$W^{(2)} = W^{(2)} + (a^{(2)})^T \cdot \hat{y}_\Delta.$$

$$W^{(1)} = W^{(1)} + X^T \cdot a_\Delta^{(2)}.$$

In Python:

```
W2 += a2.T.dot(yHat_delta)
W1 += X.T.dot(a2_delta)
```

# Feedforward and Backprop: Training 3-layer ANN

```
W1, W2 = build_my_nn()
numIters = 500
for j in range(numIters):
    # Feedforward
    Z2 = np.dot(X, W1)
    a2 = sigmoid(Z2)
    Z3 = np.dot(a2, W2)
    yHat = sigmoid(Z3)
    # Backprop
    yHat_error = y - yHat
    yHat_delta = yHat_error * sigmoid(yHat, deriv=True)
    a2_error = yHat_delta.dot(W2.T)
    a2_delta = a2_error * sigmoid(a2, deriv=True)
    W2 += a2.T.dot(yHat_delta)
    W1 += X.T.dot(a2_delta)
```

# Outline

Review

Designing Neural Networks

Training Neural Networks

Bird's Eye View on Backpropagation

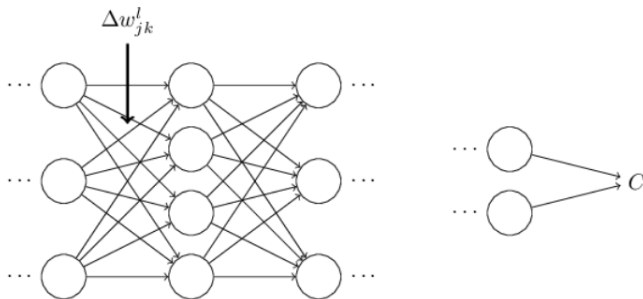
Gradient Descent

## Synapse Weight Notation: $w_{jk}^l$

$w_{jk}^l$  is the weight from neuron  $k$  in layer  $(l - 1)$  to neuron  $j$  in layer  $l$ .

## Weight Change: $\Delta w_{jk}^l$

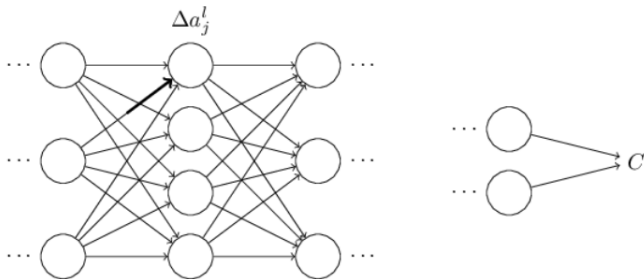
Let's assume that we'll make a change  $\Delta w_{jk}^l$  in weight  $w_{jk}^l$ :





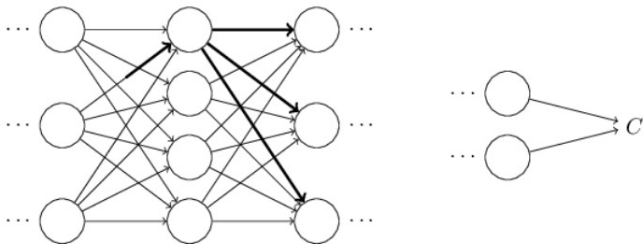
## Activation Change: $\Delta a_j^l$

The change in weight  $\Delta w_{jk}^l$  causes a change in the output activation in the corresponding neuron  $\Delta a_j^l$ :



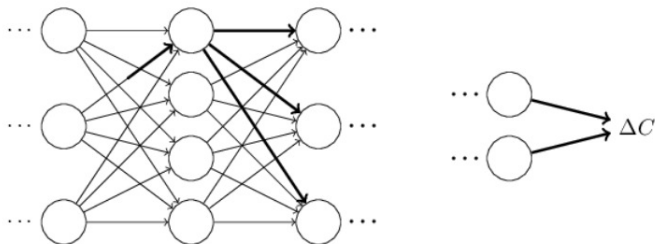
# Change in All Activations

The change in activation in one layer causes a change in all activations in the next layer:



# Change in All Activations

The change in all activations in one layer will percolate all the way to the final layer and the cost function:



# Change in Cost Function

The change in all activations in one layer will percolate all the way to the final layer and the cost function:

$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

# Change in Activation

The change  $\Delta w_{jk}^l$  causes  $\Delta a_j^l$ . This change is

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

# Change in Activations

The change in activation  $\Delta a_j^l$  causes changes in all activations at layer  $(l + 1)$ . In particular, for neuron  $q$  in layer  $(l + 1)$ :

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l.$$

Let's substitute

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

in the above equation to obtain:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

# A Single Activation Path

Let's take a path of activations  $a_j^l, a_q^{l+1}, \dots, a_n^{L-1}, a_m^L$ . Then

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

# Activation Along All Activation Paths

The total change in  $C$  must take into account all activation paths.  
Thus,

$$\Delta C \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$



# Derivation of Weight Gradient

We have

$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l$$

and

$$\Delta C \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

Hence,

$$\frac{\partial C}{\partial w_{jk}^l} \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}.$$

# Outline

Review

Designing Neural Networks

Training Neural Networks

Bird's Eye View on Backpropagation

Gradient Descent

# Ball Analogy

We would like to find the point where  $C$  achieves its global minimum.

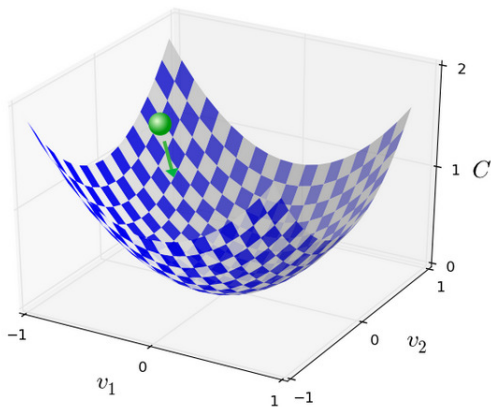
We can think of an optimization function, such as  $C$ , as a valley and imagine a ball rolling down the slope of the valley.

Theoretically, the ball will eventually roll down to the bottom of the valley. We can randomly choose a starting point for this imaginary ball and make the ball roll down and hope that the ball will reach the actual bottom.

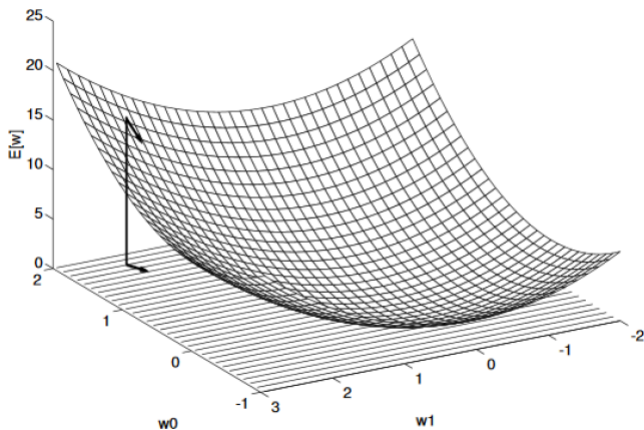
Let's investigate how we can do it in 2D.

# The Law of Ball Motion

We keep moving this imaginary ball over and over and decreasing  $C$  until we reach a global minimum.



## Another Look at Hypothesis Search Space



# Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(t - o)x_i$$

where

- ▶  $t = c(\vec{x})$  is target value
- ▶  $o$  is perceptron output
- ▶  $\eta$  is small constant (e.g., 0.1) called *learning rate*

# Gradient Descent

Consider a simple *linear unit*, where

$$o = w_0 + w_1x_1 + \cdots + w_nx_n$$

We will learn  $w_i$ 's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where  $D$  is set of training examples.

# Gradient Descent Rule

Gradient:

$$\nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$



# Derivation of Gradient Descent

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d) (-x_{i,d})\end{aligned}$$

where  $x_{i,d}$  is the input component  $x_i$  for training example  $d$ .

# Derivation of Gradient Descent

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

# Gradient Descent Algorithm

GRADIENT-DESCENT(*training\_examples*,  $\eta$ )

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where  $\vec{x}$  is the vector of input values, and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).*

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $\langle \vec{x}, t \rangle$  in *training\_examples*, Do
    - \* Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - \* For each linear unit weight  $w_i$ , Do
$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$
  - For each linear unit weight  $w_i$ , Do
$$w_i \leftarrow w_i + \Delta w_i$$

# Incremental Gradient Descent Algorithm

---

**Batch mode** Gradient Descent:

Do until satisfied

1. Compute the gradient  $\nabla E_D[\vec{w}]$
2.  $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

---

**Incremental mode** Gradient Descent:

Do until satisfied

- For each training example  $d$  in  $D$ 
  1. Compute the gradient  $\nabla E_d[\vec{w}]$
  2.  $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

---

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

*Incremental Gradient Descent* can approximate  
*Batch Gradient Descent* arbitrarily closely if  $\eta$   
made small enough

# Incremental Gradient Descent Training Rule

$$w_i = w_i + \eta(t - o)x_i$$

# Conclusions

Gradient descent algorithm is a computational method of changing the position  $v$  to find a minimum of the function  $C$  by using the equations on the previous slide.

Caveat: The update rule doesn't always work - there are cases when the global minimum of  $C$  is not reached. In practice, gradient descent works well to help ANNs to learn.

# References

1. T.M. Mitchell. *Machine Learning*.
2. M. Neilsen. *Neural Networks and Deep Learning*.