CS 401 Artificial Intelligence

FAST-NU

Dr. Kashif Zafar

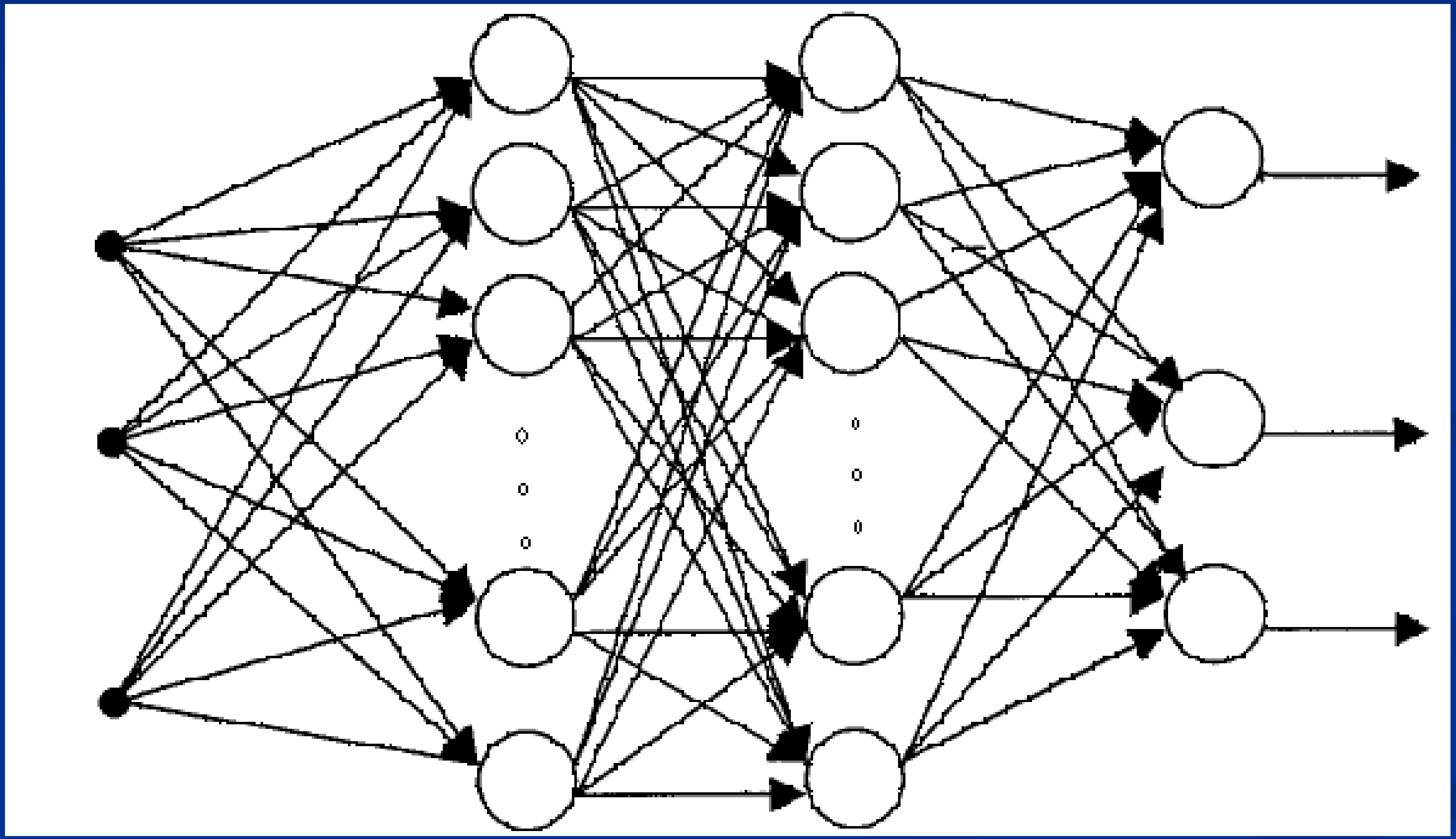
Lecture 13

November 23, 2021

Department of Computer Science
National University of Computers & Emerging Sciences
Lahore.

NEURAL NETWORKS

Multi Layer Perceptron: Architecture & Forward Pass



Input Units **Hidden Units** **Output Units**

NEURAL NETWORKS

Backpropagation Algorithm

- **Set up the architecture & initialize the weights of the network**
- **Apply the training pairs (input-output vectors) from the training set, one by one**
- **For each training pair, calculate the output of the network**
- **Calculate the error between actual output & desired output**
- **Propagate the error backwards & adjust the weights in such a way that minimizes the error**
- **Repeat the above steps for each pair in the training set until the error for the set is lower than the required minimum error**

NEURAL NETWORKS

Multi Layer Perceptron:

Training by Backpropagation Algorithm

Let E = accumulative error over a data set. It is a function of network weights

$$E = \sum_{\text{training samples}} \sum_j (d_j - O_j)^2$$

d_j is the desired output of node j and O_j is the actual output

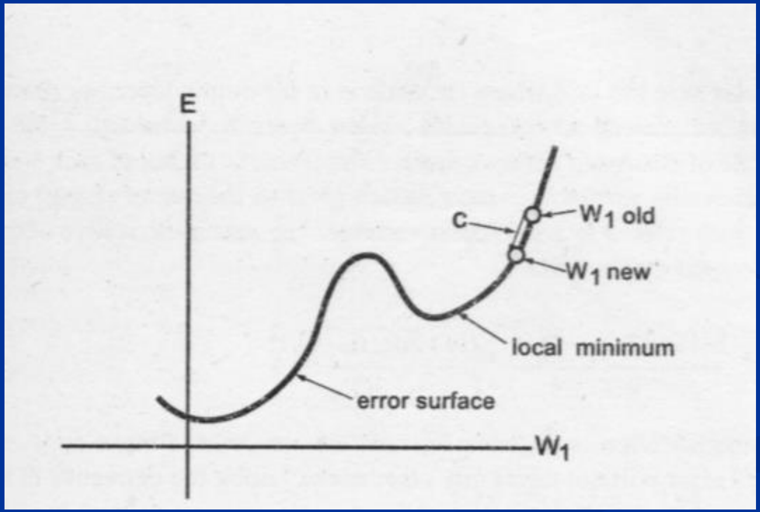
The error is squared so that the positive and negative errors may not cancel each other out during summation

NEURAL NETWORKS

Multi Layer Perceptron:

Training by Backpropagation Algorithm

Each weight configuration can be represented by a point on an error surface

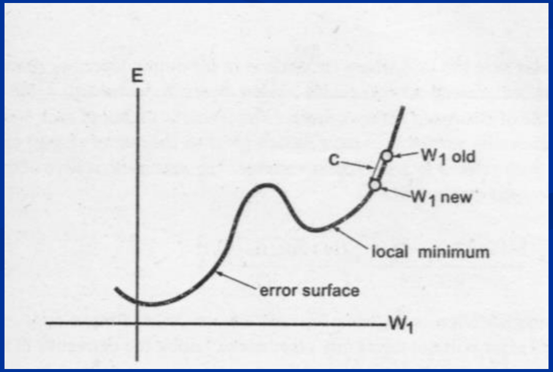


NEURAL NETWORKS

Multi Layer Perceptron: Training

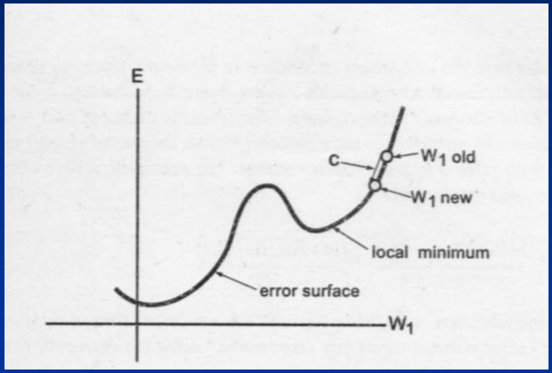
Starting from a random weight configuration, we want our training algorithm to move in the direction where error is reduced more rapidly

Delta rule attempts to minimize the local error and uses the derivative of the error to find the slope of the error space in the region local to a particular point



NEURAL NETWORKS

Multi Layer Perceptron: Training



Delta rule uses gradient descent:

$$\Delta w_{ij} = -c\ (\partial Error / \partial w_{ij})$$

Let current weight be 4
Then $\partial Error/ \partial w_{ij} = 9.5 – 9.0/5 – 3 = 0.25$
The new weight will be $w_{new} = w_{old} – c*0.25 = 3.875$
if c = 0.5 (c is the learning rate)

If the error curve had been downward, then

$$\partial Error/ \partial w_{ij} = 9.0 – 9.5/5 – 3 = -0.25$$

The new weight will be

$$w_{new} = w_{old} – c*-0.25 = 4.125$$

NEURAL NETWORKS

Multi Layer Perceptron: Training

Delta rule:

$$\Delta w_{ij} = -c\ (\partial Error/\ \partial w_{ij})$$

If the learning constant "c" is large (more than 0.5), weights move quickly to optimal value but there is a risk of overshooting the minimum or oscillation around optimum weights

If "c" is small, the training is less prone to these problems but system does not learn quickly; also the algorithm may get stuck in local minima

NEURAL NETWORKS

Multi Layer Perceptron: Training

The weights are updated incrementally, following the presentation of each training example

This corresponds to a stochastic approximation to gradient descent

To obtain the true gradient of Error, one would consider all of the training examples before altering the weight values

The stochastic approximation avoids costly computations per weight update

NEURAL NETWORKS

Multi Layer Perceptron: Training of Output Layer Weights

Randomly set the weights
Present first training input vector to the network
Calculate the outputs of all neurons
The inputs to the last layer of neurons would be the output of 2nd last layer
We calculate the Error of all the output neurons and now we wish to change the weights of an output neuron "j" so that its error reduces

We use Delta rule:

$$\Delta w_{ij} = -c\,(\partial Error / \partial w_{ij})$$

NEURAL NETWORKS

Multi Layer Perceptron: Training of Output Layer Weights

The equation $\partial Error / \partial w_{ij}$ means that we want the rate of change of network error as a function of the change in one of weights of an output node j

Since for our current training sample

$$Error = \sum_j (d_j - O_j)^2$$

Where O_j is itself a function of other variables (including w_{ij}), therefore we use partial derivatives (this gives us the rate of change of a multi-variable function w.r.t a particular variable)

NEURAL NETWORKS

Multi Layer Perceptron: Training of Output Layer Weights

To calculate this quantity we use chain rule
The Error is only indirectly dependent on w_{ij}, but it is directly dependent on variable O_j

$$\partial Error/ \partial w_{ij} = (\partial Error/ \partial O_j) \;.\; (\partial O_j / \partial w_{ij})$$

$\partial Error/ \partial O_j$ = rate of change of error w.r.t output of node j
Now $\partial Error/ \partial O_j = \sum_j (d_j - O_j)^2 / \partial O_j = -2(d_j - O_j)$

For $\partial O_j / \partial w_{ij}$ we have $(\partial O_j / \partial act_j)\,(\partial act_j / \partial w_{ij})$
$(\partial O_j / \partial act_j) = (\partial f(act)_j / \partial act_j) = f'(act_j)$
$(\partial act_j / \partial w_{ij}) = (\partial \sum_i x_i w_{ij} / \partial w_{ij}) = x_i$
Hence $\Delta w_{ij} = -c\,(\partial Error/ \partial w_{ij}) = -c[-(d_j - O_j) \,.\, f'(act_j) \,.\, x_i]$

Multi Layer Perceptron: Training of Hidden Layer Weights

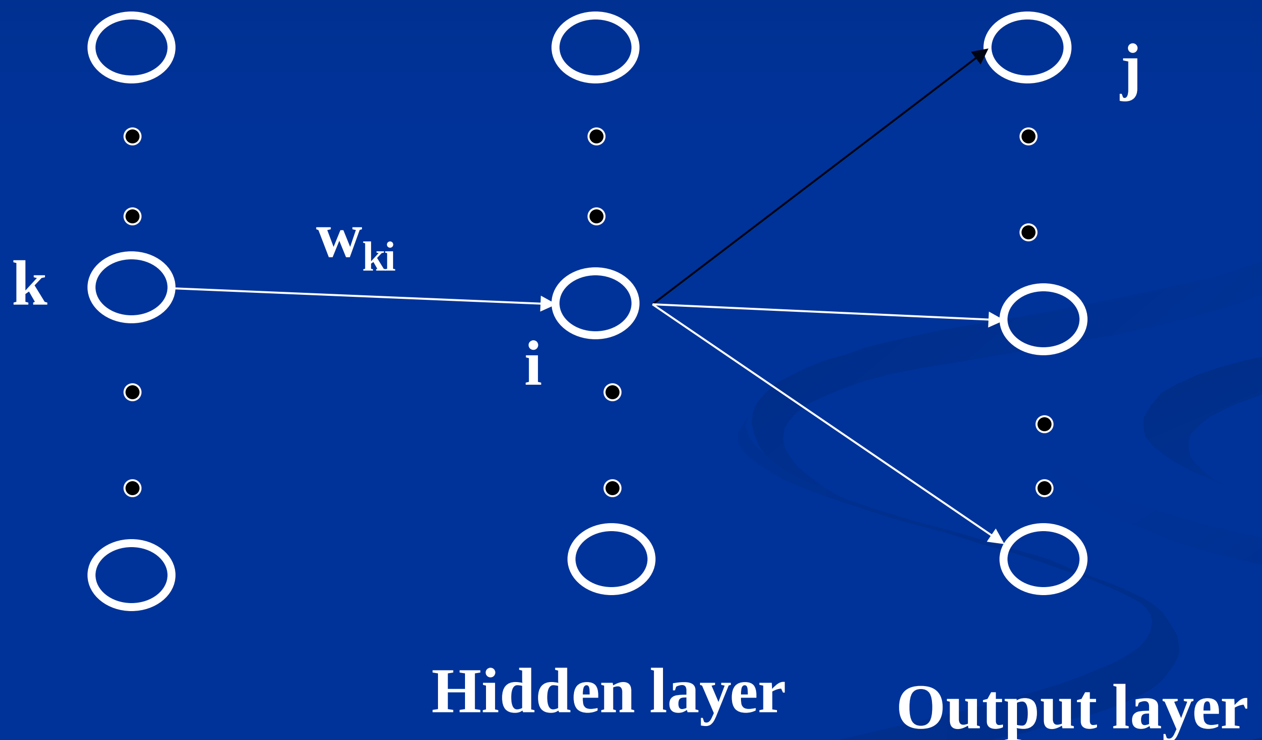
The formula for hidden layer weights update is different

because

the training examples provide target values only for the network outputs, and no target values are directly available to indicate the error of hidden unit's values

NEURAL NETWORKS

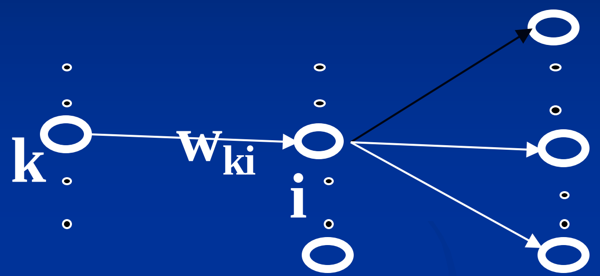
Multi Layer Perceptron: Training of Hidden Layer Weights



NEURAL NETWORKS

Multi Layer Perceptron: Training of Hidden Layer Weights

Adjustment of k^th weight of node "i"



$$\Delta w_{ki} = -c\ (\partial Error / \partial w_{ki})$$

Since Error is not a direct function of weight w_{ki}, therefore we use chain rule

$$\partial Error / \partial w_{ki} = (\partial Error / \partial O_i)\ .\ (\partial O_i / \partial w_{ki})$$

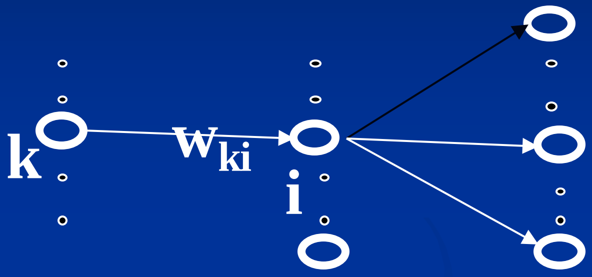
$\partial Error / \partial O_i$ = rate of change of error w.r.t output of node i

$$= \partial \textstyle\sum_j Error_j / \partial O_i$$

NEURAL NETWORKS

Multi Layer Perceptron: Training of Hidden Layer Weights

Since each $Error_j$ is independent of other $Error_j$ (each has its own independent weight set)



Hence

$$\partial \sum_j Error_j / \partial O_i = \sum_j (\partial Error_j / \partial O_i)$$

Again use chain rule we have

$$= \sum_j [(\partial Error_j / \partial act_j) \cdot (\partial act_j / \partial O_i)]$$

NEURAL NETWORKS

Multi Layer Perceptron: Training of Hidden Layer Weights

$\partial \sum_j \text{Error}_j / \partial O_i = \sum_j [(\partial \text{Error}_j / \partial act_j) \,.\, (\partial act_j / \partial O_i)]$

$\partial \text{Error}_j / \partial act_j = (\partial \text{Error}_j / \partial O_j)\, (\partial O_j / \partial act_j)$

where $\partial \text{Error}_j / \partial O_j = \partial (d_j - O_j)^2 / \partial O_j = -2(d_j - O_j)$

and $\partial O_j / \partial act_j = \partial f(act_j) / \partial act_j = f'(act_j)$

$(\partial act_j / \partial O_i) = (\partial \sum x_i w_{ij} / \partial O_i)$

Since $O_i = x_i$

hence $\partial act_j / \partial O_i = w_{ij}$

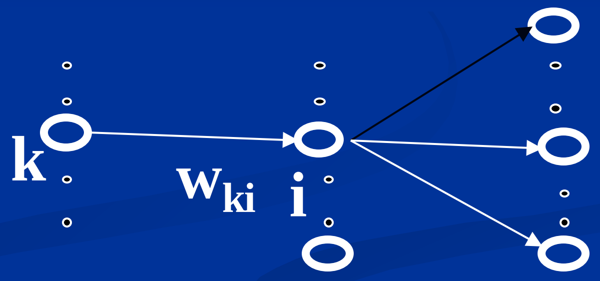
NEURAL NETWORKS

Multi Layer Perceptron: Training of Hidden Layer Weights

So we started with

$\partial \text{Error}/\partial w_{ki} = (\partial \text{Error}/\partial O_i) \cdot (\partial O_i / \partial w_{ki})$

and we have determined the first part



For the 2nd part

$\partial O_i / \partial w_{ki} = (\partial O_i / \partial act_i)(\partial act_i / \partial w_{ki})$

$(\partial act_i / \partial w_{ki}) = (\partial \sum_k x_k w_{ki} / \partial w_{ki}) = x_k$

$(\partial O_i / \partial act_i) = (\partial f(act)_i / \partial act_i) = f'(act)_i$

Hence $\Delta w_{ki} = -c\, (\partial \text{Error}/\partial w_{ki})$

$= -c[-2\sum_j \{(d_j - O_j)\, f'(act_j)\, w_{ij}\} f'(act)_i\, x_k]$

NEURAL NETWORKS

Multi Layer Perceptron: Training

A typical activation function is logistic function (which is a type of sigmoidal function)

$$f(act) = 1/(1 + e^{-\lambda act})$$

If value of λ (squashing parameter) is large we have a unit step function, if it is small we have almost a straight line between two saturation limits

$$f'(act) = f(act)(1 - f(act))$$

NEURAL NETWORKS

Multi Layer Perceptron: Training

This approach is called "gradient descent learning"

Requirement of this approach is that the activation function must be differentiable (i.e. continuous)

The number of input and output neurons are fixed

But the selection of number of hidden layers and the number of neurons in the hidden layers is done by trial and error

NEURAL NETWORKS

Multi Layer Perceptron: Training

The gradient descent is not guaranteed to converge to the global optimum

The algorithm we have discussed is the incremental gradient descent (or stochastic gradient descent) version of the Backpropagation

NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

Images of 20 different people

32 images per person

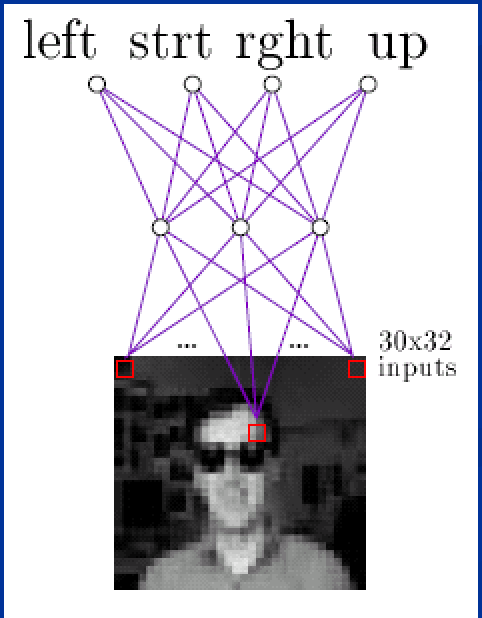
With varying expressions (happy, sad, angry, neutral)
and
looking in various directions (left, right, straight, up)
and
with and without sunglasses

Grayscale images (intensity between 0 to 255) and
size (resolution) of 120 x 128 pixels

NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

Typical input images



NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

An ANN can be trained on any one of a variety of target functions using this image data, e.g.

- **identity of a person**
- **direction in which person is looking**
- **gender of the person**
- **whether or not they are wearing sunglasses**

NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

Design Choices:

Separate the data into

training (260 images) and test sets (364 images)

Input Encoding

- **30 x 32 pixel image**
- **A coarse resolution of the 120 x 128 pixel image**
- **Every 4 x 4 pixels are replaced by their mean value**
- **The pixel intensity is linearly scaled from 0 to 1 so that inputs, hidden units and output units have the same range**

NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

Design Choices:

Output Encoding

- **Learning Task: Direction in which person is looking**
- **Only one neuron could have been used with outputs 0.2, 0.4, 0.6, and 0.8 to encode the four possible values**
- **But we use 4 output neurons, so that measure of confidence in the ANN's decision can be obtained**
- **Output vector:**
  1 for true & 0 for false; e.g. [1, 0, 0, 0]

NEURAL NETWORKS

Multi Layer Perceptron: Face Recognition Example

Design Choices:

Network Structure

- **How many Layers?**
  Usually one hidden layer is enough
- **How many units in the hidden layer**
  More than necessary units result in over-fitting
  Less units result in failure of training
  Trial & error: Start with a number and prune the units with the help of a cross-validation set

Reading Assignment & References

Chapter 4 of Tom M. Mitchell "Machine Learning"

http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/systems/nevprop/np.c

Derivation of Backpropagation Weight Updates

$$E(\vec{w}) = \frac{1}{2} \underbrace{\sum_{d \in D}}_{\text{Training Example}} \underbrace{\sum_{k \in outputs} (t_{kd} - o_{kd})^2}_{\text{Sum over all network output units}}$$

$$\underbrace{E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2}_{\text{Stochastic version of gradient descent}}$$

(Summing over all network output units for each training example)

x_{ji} = ith input to unit j

w_{ji} = weight associated with ith input to unit j

net_j = $\sum_i w_{ji} x_{ji}$ [Weighted sum of inputs for unit j]

o_j = the output computed by unit j

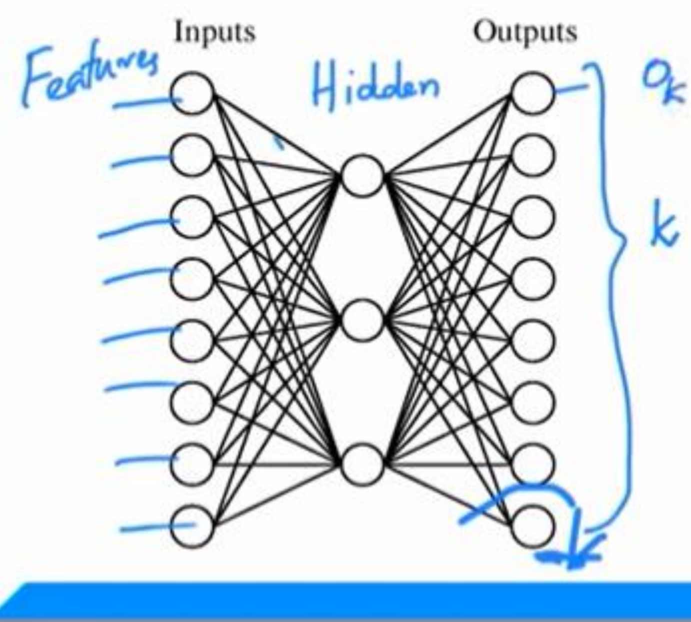
t_j = target output for unit j

σ = sigmoid function

outputs = the set of units in the final layer of the network.

Downstream(j) = the set of units whose immediate inputs include the output of unit j

Learning Hidden Layer Representations



Derivation of Backpropagation Weight Updates

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \underbrace{\sum_{k \in outputs} (t_{kd} - o_{kd})^2}_{\text{SUM OVER ALL NETWORK OUTPUT UNITS}}$$

$d \in D$ ← Training Example.

$$\underbrace{E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2}_{\text{STOCHASTIC VERSION OF GRADIENT DESCENT}}$$

(SUMMING OVER ALL NETWORK OUTPUT UNITS FOR EACH TRAINING EXAMPLE)

x_{ji} = ith INPUT TO UNIT j

w_{ji} = weight associated with ith input to unit j

net_j = $\sum_i w_{ji} x_{ji}$ [WEIGHTED SUM OF INPUTS FOR UNIT j]

o_j = the output computed by unit j

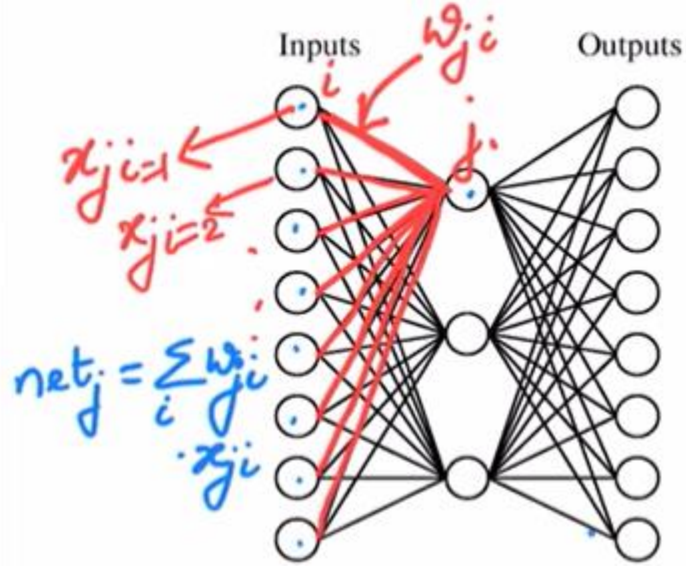
t_j = target output for unit j

σ = sigmoid function

outputs = the set of units in the final layer of the network.

Downstream(j) = the set of units whose immediate inputs include the output of unit j

Learning Hidden Layer Representations



Derivation of Backpropagation Weight Updates

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \underbrace{\sum_{k \in outputs}}_{\text{Sum over all network output units}} (t_{kd} - o_{kd})^2$$

Training Example.

$$\underbrace{E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2}_{\text{Stochastic version of gradient descent}}$$

(Summing over all network output units for each training example)

x_{ji} = ith input to unit j

w_{ji} = weight associated with ith input to unit j

net_j = $\sum_i w_{ji} x_{ji}$ [Weighted sum of inputs for unit j]

o_j = the output computed by unit j

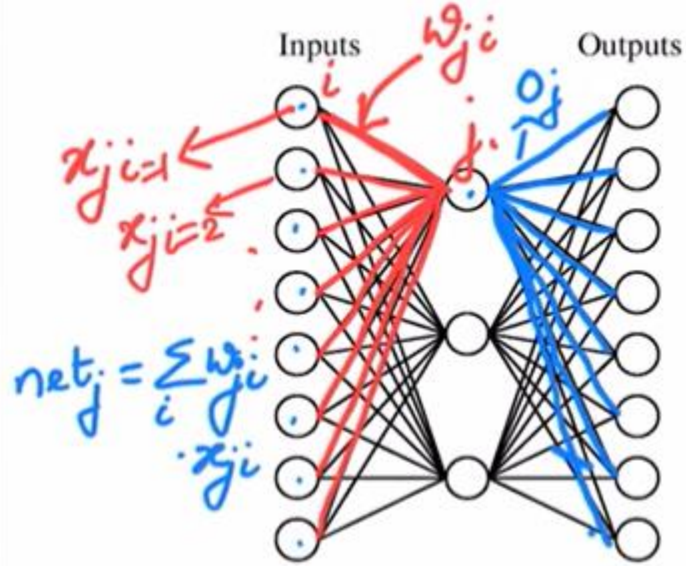
t_j = target output for unit j

σ = sigmoid function

outputs = the set of units in the final layer of the network.

Downstream(j) = the set of units whose immediate inputs include the output of unit j

Learning Hidden Layer Representations



$o_j = \sigma(ne$

Derivation of Backpropagation Weight Updates

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \underbrace{\sum_{k \in outputs} (t_{kd} - o_{kd})^2}_{\text{Sum over all network output units}}$$

$d \in D$: Training Example.

$$\underbrace{E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2}_{\text{Stochastic version of gradient descent}}$$

(Summing over all network output units for each training example)

x_{ji} = ith input to unit j

w_{ji} = weight associated with ith input to unit j

net_j = $\sum_i w_{ji} x_{ji}$ [Weighted sum of inputs for unit j]

o_j = the output computed by unit j

t_j = target output for unit j

σ = sigmoid function

outputs = the set of units in the final layer of the network.

Downstream(j) = the set of units whose immediate inputs include the output of unit j

Learning Hidden Layer Representations