Artificial Neural Networks

Natasha Balac, Ph.D.

Paul Rodriguez, Ph.D.

Predictive Analytics Center of Excellence, Director

San Diego Supercomputer Center

University of California, San Diego

Artificial Neural Networks

- **Biological networks vs. Neural Networks**
- **Artificial Neuron**
- **Perceptron**
- **Back Propagation**
- **Hands-on Session**

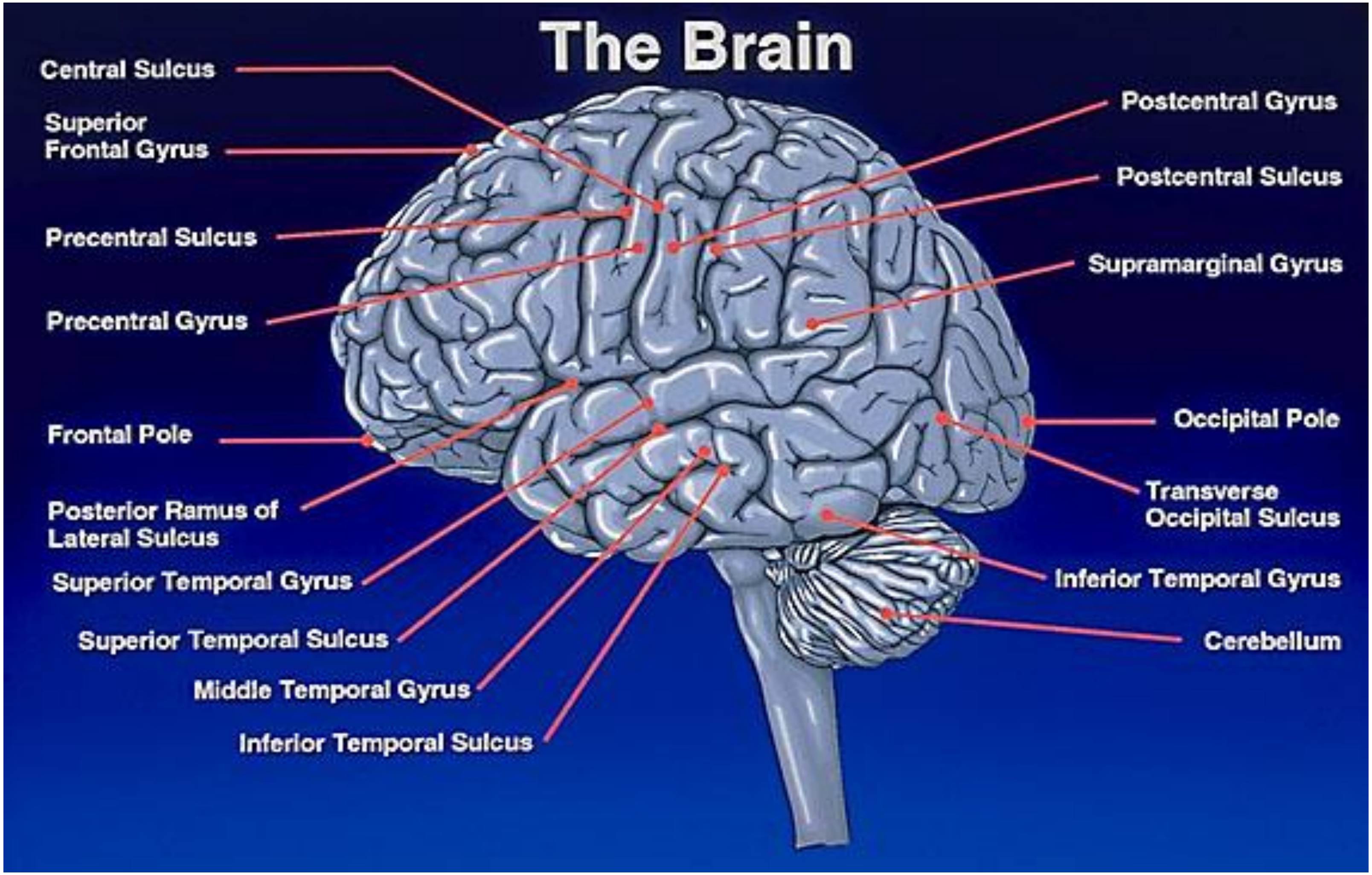
Brain vs. Serial Computer

- Our neurons do not transmit information very fast
- Even for a reflex that only involves three neurons, it may take several tenths of a second
- Somehow, we are able to process the information that it takes to drive a car using our relatively slow neurons
- Our nervous system does not process serially
- Human brain is an example of a *massively-parallel system*

Brain as an Information Processing System

- **It can learn (reorganize itself) from experience**
- **Partial recovery from damage is possible if healthy units can learn to take over the functions previously carried out by the damaged areas**
- **Performs massively parallel computations extremely efficiently**
- **It supports our intelligence and self-awareness**
  - How? Nobody knows yet

Human Brain Is Not Homogeneous



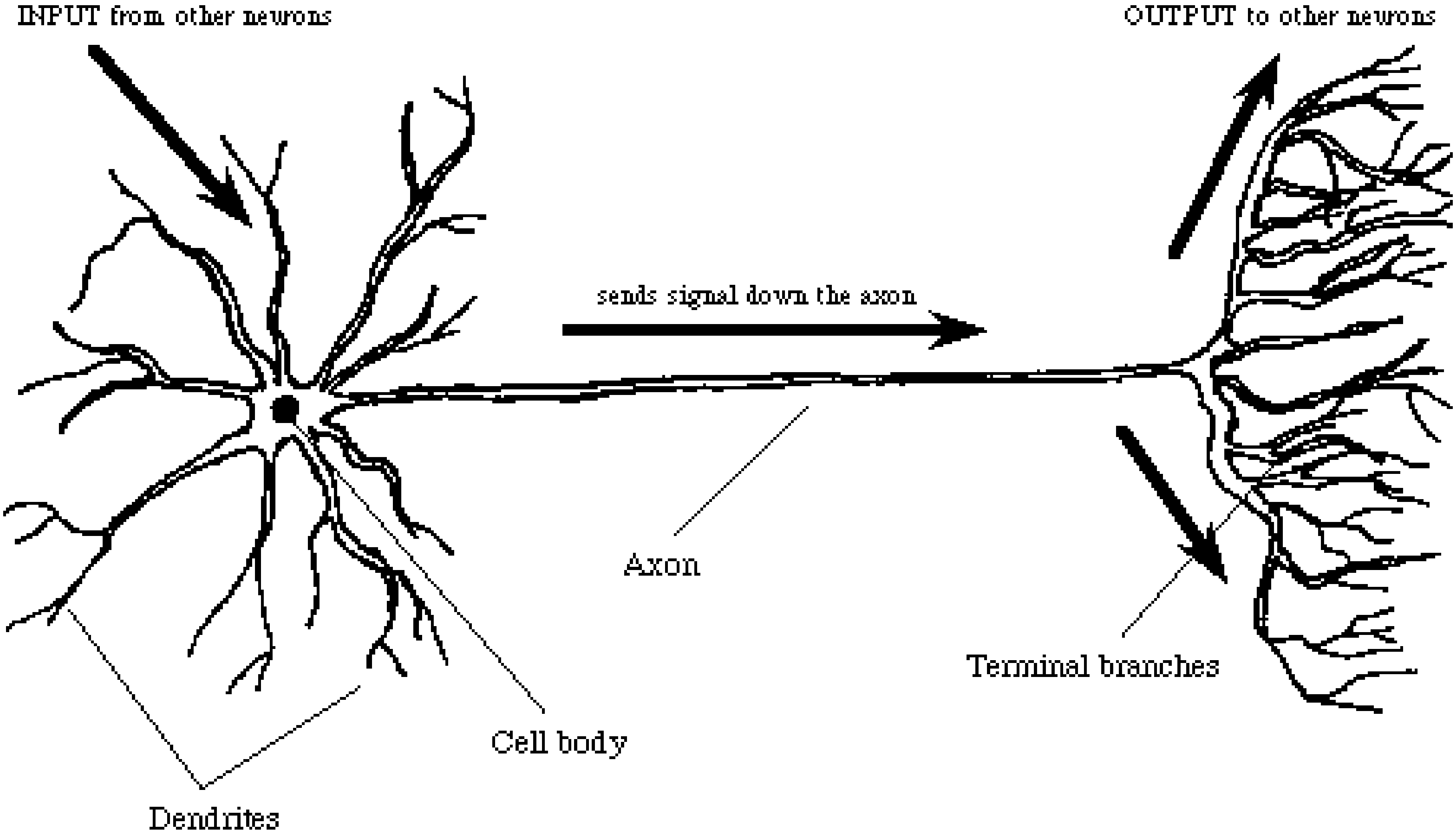
Human Brain

- Our brains are made up of about 100 billion tiny units called ***neurons***
- Each neuron is connected to thousands of other neurons and communicates with them via electrochemical signals
- Signals coming into the neuron are received via junctions called ***synapses***
- Synapses are located at the end of branches of the neuron cell called ***dendrites***

Human Brain

- **The neuron continuously receives signals from inputs and then performs a little bit of "magic"**
- **Neuron sums up the inputs to itself (in some way) and then, if the end result is greater than some threshold value, the neuron fires**
- **It generates a voltage and outputs a signal along an *axon***

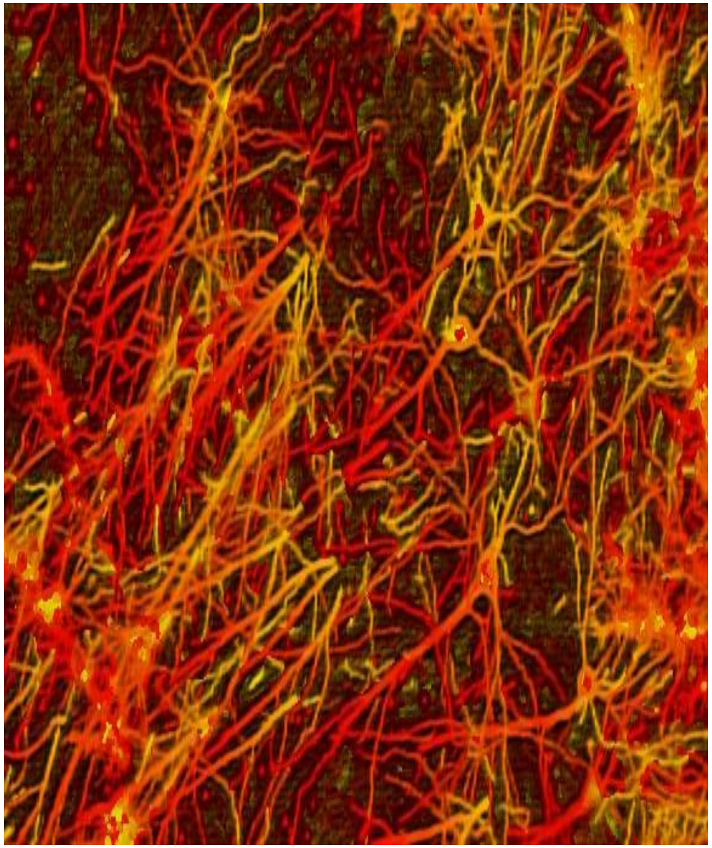
Neuron



INPUT from other neurons
OUTPUT to other neurons
sends signal down the axon
Axon
Terminal branches
Cell body
Dendrites

Neurons

- **In addition to these long-range connections, neurons also link up with many thousands of their neighbors**
- **To form very dense, complex local networks**



Brains Learn - Of Course! How?

- **One way brains learn is by altering the strengths of connections between neurons and by adding or deleting connections between neurons**
- **They learn "on-line"**
  - based on experience
  - and typically without the benefit of a teacher

A Simple Artificial Neuron

- **Basic computational element (model neuron) is often called a node or unit**
- **It receives input from some other units, or perhaps from an external source**
- **Each input has an associated weight *w*, which can be modified so as to model synaptic learning**

A Simple Artificial Neuron

- **The unit computes some function *f* of the weighted sum of its inputs:**

$$y_i = f\left(\sum_j w_{ij}\, y_j\right)$$

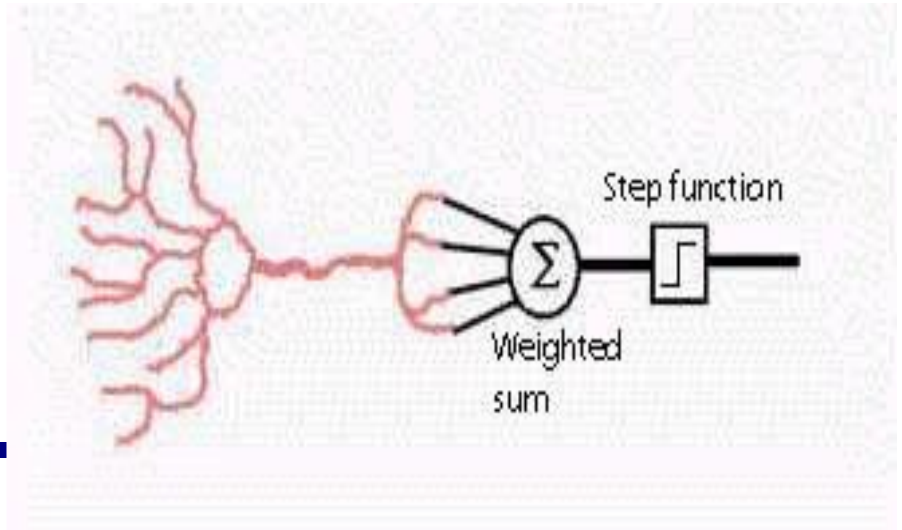
- **Its output, in turn, can serve as input to other units**

Neuron

- Each input into the neuron has its own weight
- Weight is a floating point number
  - will be adjust when training the network
- Weights in most neural nets can be both negative and positive
  - providing excitory or inhibitory influences to each input
- Each input enters the nucleus it's multiplied by its weight

Perceptron Activation

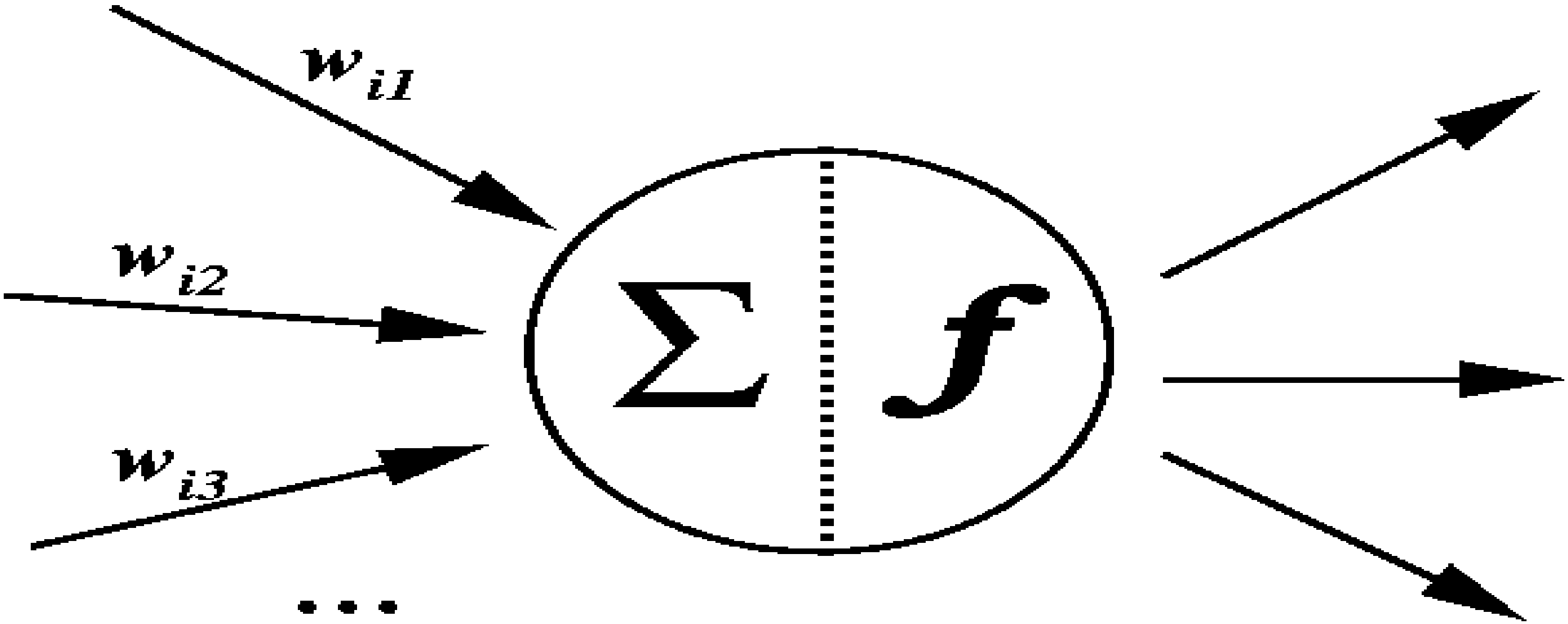
- **The nucleus then sums all these new input values which gives us the *activation***
  - **floating point number which can be negative or positive**
- **If the activation is greater than a threshold value**
  - **the neuron outputs a signal**
  - **If the activation is less than**
    - **1 the neuron outputs zero**
- **Called a *step* function**



Structure

- **A neuron can have any number of inputs 1-> n**
- **The inputs represented as: $x_1, x_2, x_3 \dots x_n$**
- **Corresponding input weights : $w_1, w_2, w_3 \dots w_n$**
- **Summation of the weights multiplied by the inputs: $x_1w_1 + x_2w_2 + x_3w_3 \dots + x_nw_n$**
  - **called activation value**
  - $a = x_1w_1+x_2w_2+x_3w_3\dots +x_nw_n$
    - $a=\Sigma\, w_ix_i$

Artificial Neuron



$$y_i = f(net_i)$$

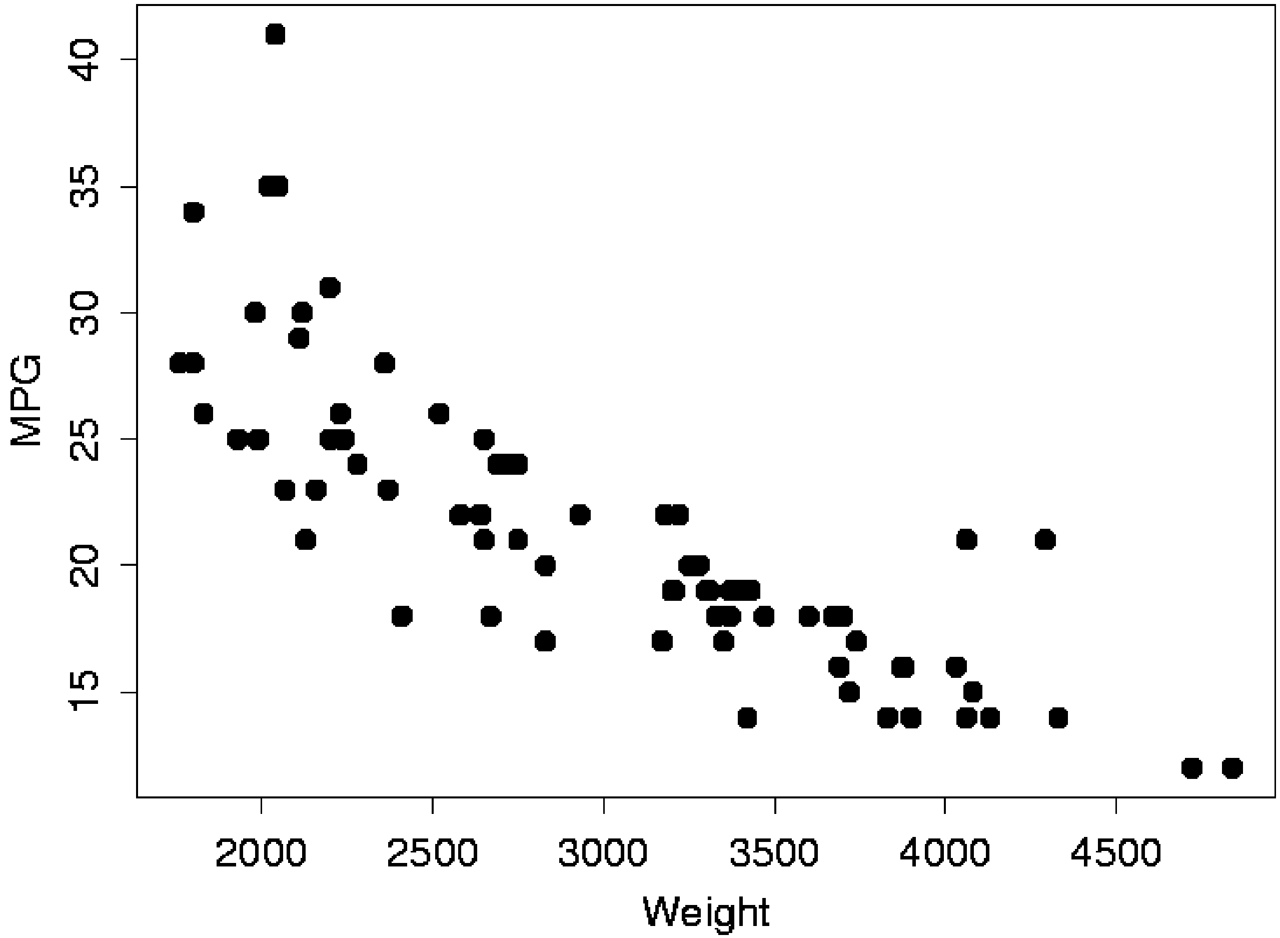
Activation Function

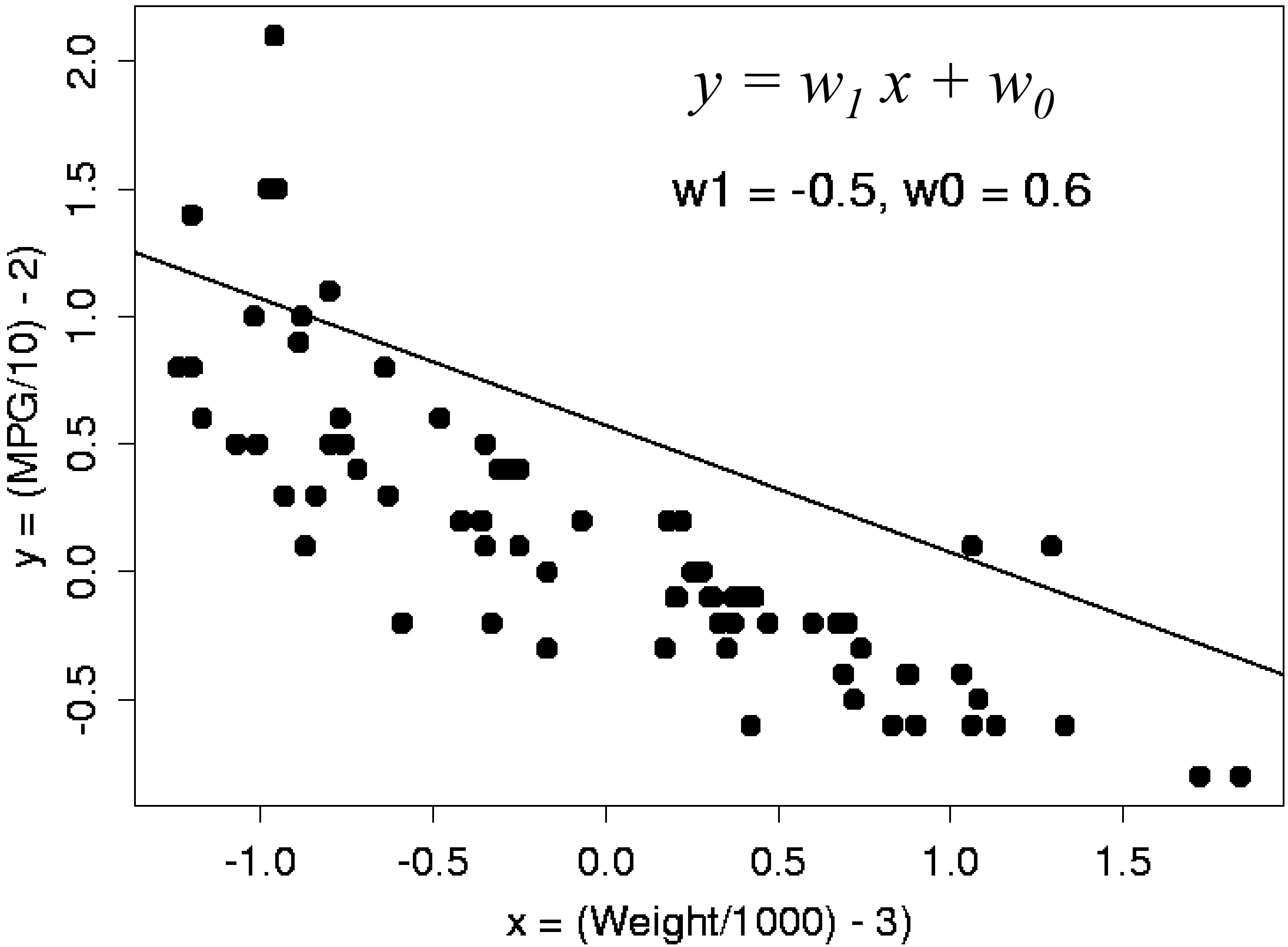
- Identity Function
- Step Function
- Logistic Function (Sigmoid)
- Symmetric Sigmoid
- Radial Basis Functions
- Derivatives

Example

- **Each dot in the figure provides information about the weight**
  - x-axis, units: U.S. pounds and fuel consumption
  - y-axis, units: miles per gallon for one of 74 cars
  - Clearly weight and fuel consumption are linked
    - heavier cars use more fuel

Car Weight and MPG Example





$y = w_1 x + w_0$
w1 = -0.5, w0 = 0.6
y = (MPG/10) - 2
x = (Weight/1000) - 3
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0

The Loss Function

- **In order to make precise what we mean by being a "good predictor”**
- **Define a loss (objective or error) function *E* over the model parameters**
- **Popular choice for *E* is the sum-squared error**

$$E = \frac{1}{2} \sum_{p} (t_p - y_p)^2$$

Minimizing the Loss

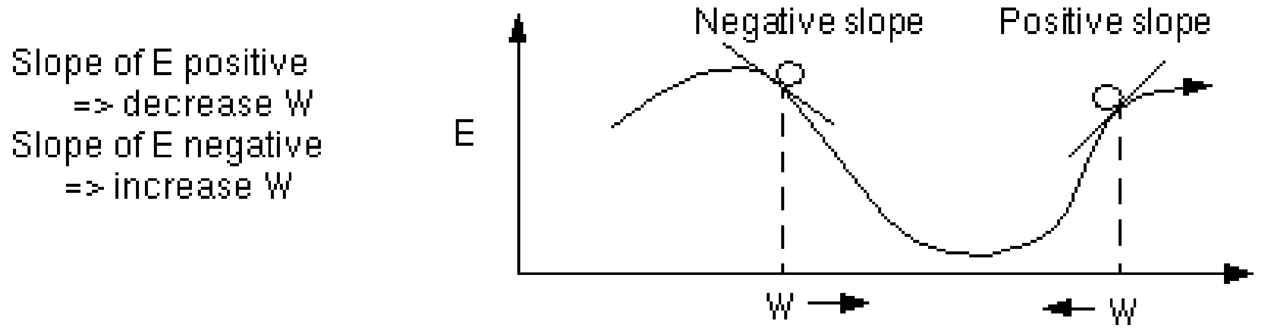
- **The loss function *E* provides an objective measure of predictive error for a specific choice of model parameters**
- **We can thus restate our goal of finding the best (linear) model as finding the values for the model parameters that minimize *E***

Gradient Descent

- **For linear models**
  - **linear regression** provides a direct way to compute these optimal model parameters
- **Does not generalize to nonlinear models**
- **Even though the solution cannot be calculated explicitly in that case**
  - the problem can still be solved by an iterative numerical technique called **gradient descent**

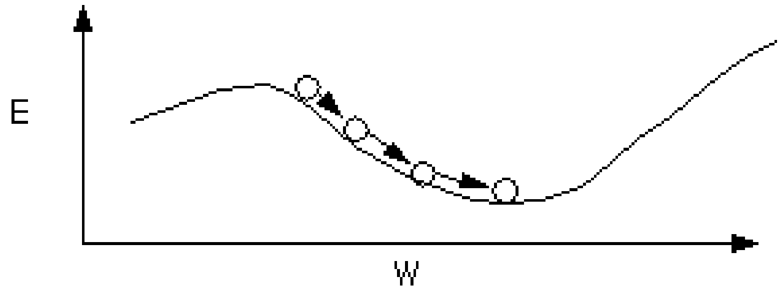
How does this work?

- **The gradient of E gives us the direction in which the loss function at the current setting of the *w* has the steepest slope**
- **In order to decrease *E* take a small step in the opposite direction**



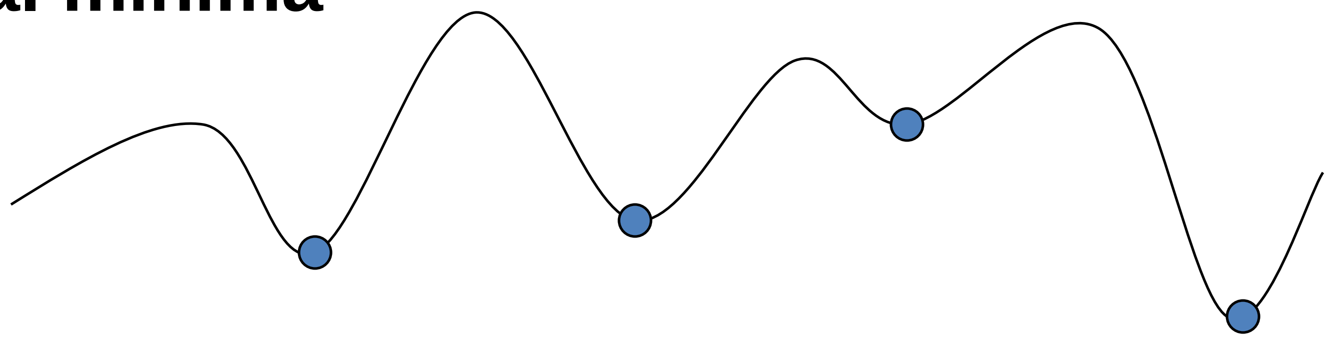
Gradient Descent

- **By repeating this over and over, we move "downhill" in *E* until reach a minimum where *G* = 0**
- **so that no further progress is possible**



Gradient Descent

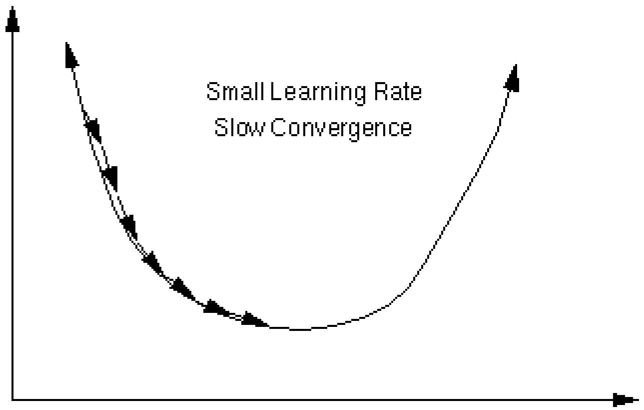
- **Similar to hill-climbing**
- **Can be problems knowing when to stop**
- **Local minima**



  - **can have multiple local minima (note: for perceptron, E(w) only has a single global minimum, so this is not a problem)**
  - **gradient descent goes to the closest local minimum:**
    - **solution: random restarts from multiple places in weight space**

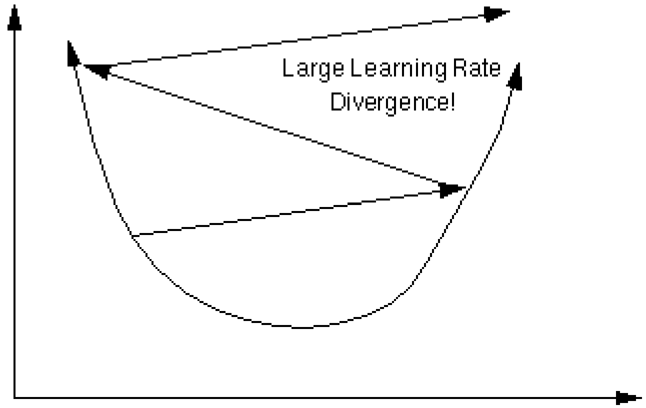
The Learning Rate

- **Important consideration is the learning rate μ**
  - determines by how much we change the weights *w* at each step
- **If μ is too small the algorithm will take a long time to converge**



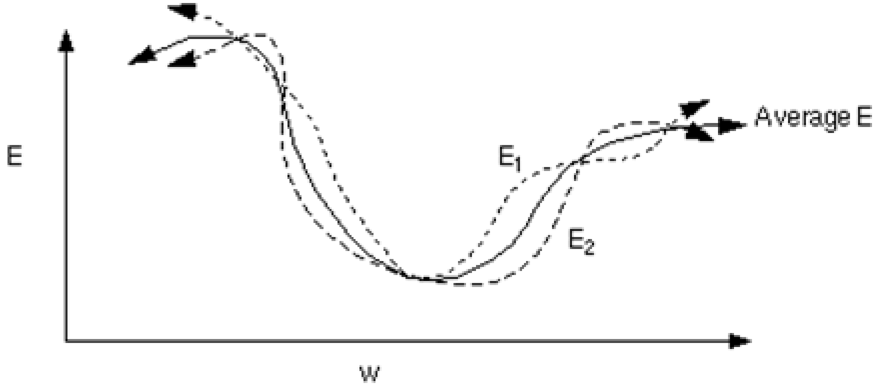
The Learning Rate

- **If μ is too large**
  - we may end up bouncing around the error surface out of control - the algorithm **diverges**
  - This usually ends with an overflow error in the computer's floating-point arithmetic

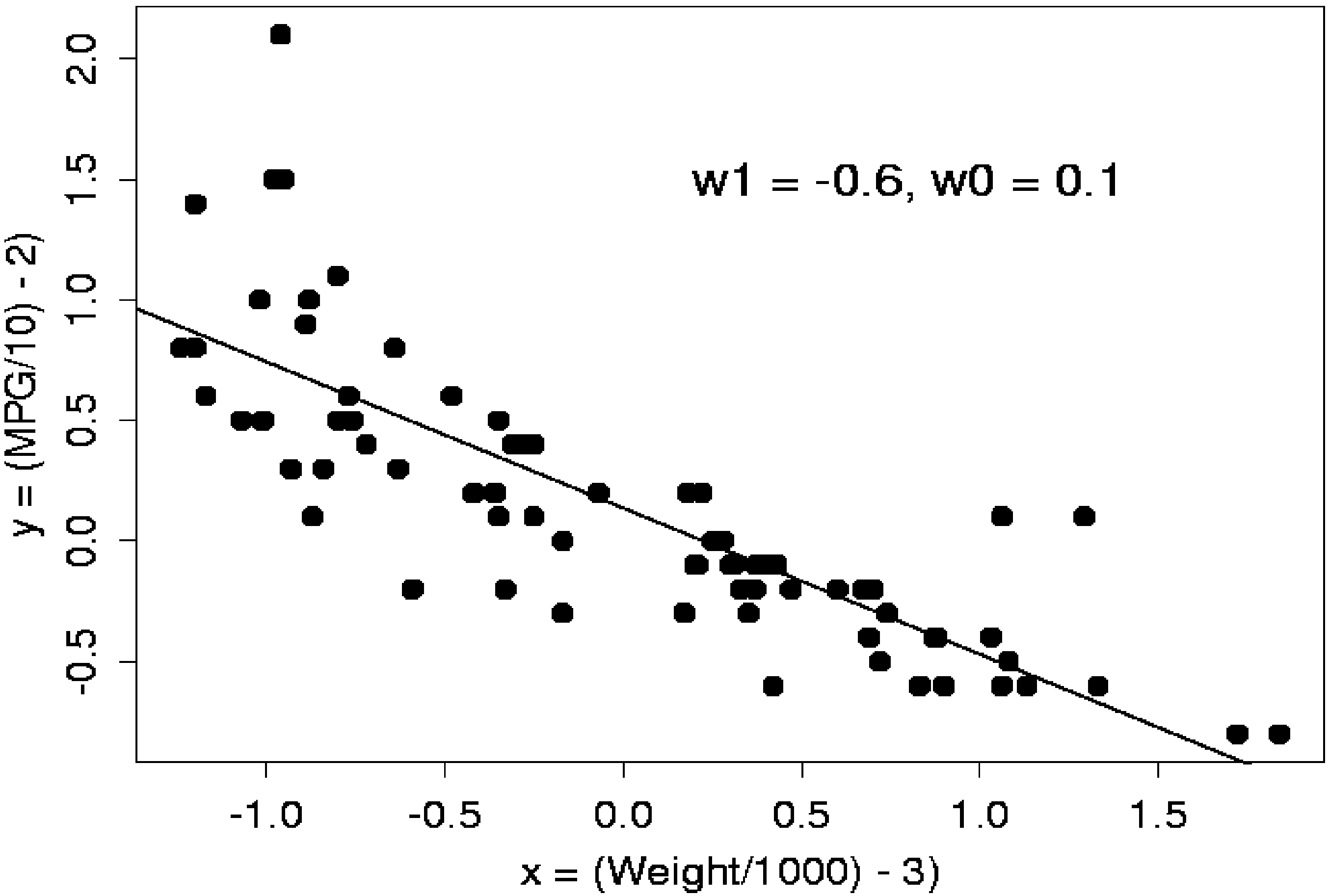


Stochastic Gradient Descent

- **Since the gradient for a single data point can be considered a noisy approximation to the overall gradient *G***
  - this is also called **stochastic** (noisy) gradient descent



Linear Model For Car Data Found By Gradient Decent



It's a Neural Network!

- **Linear model of equation *$y = w_1 x + w_0$* can be implemented by the simple neural network**
  - It consists of a
    - **bias** unit
    - an **input** unit
    - a linear **output** unit
- **The input unit makes external input *x* (the weight of a car) available to the network**
- **While the bias unit always has a constant output of 1**

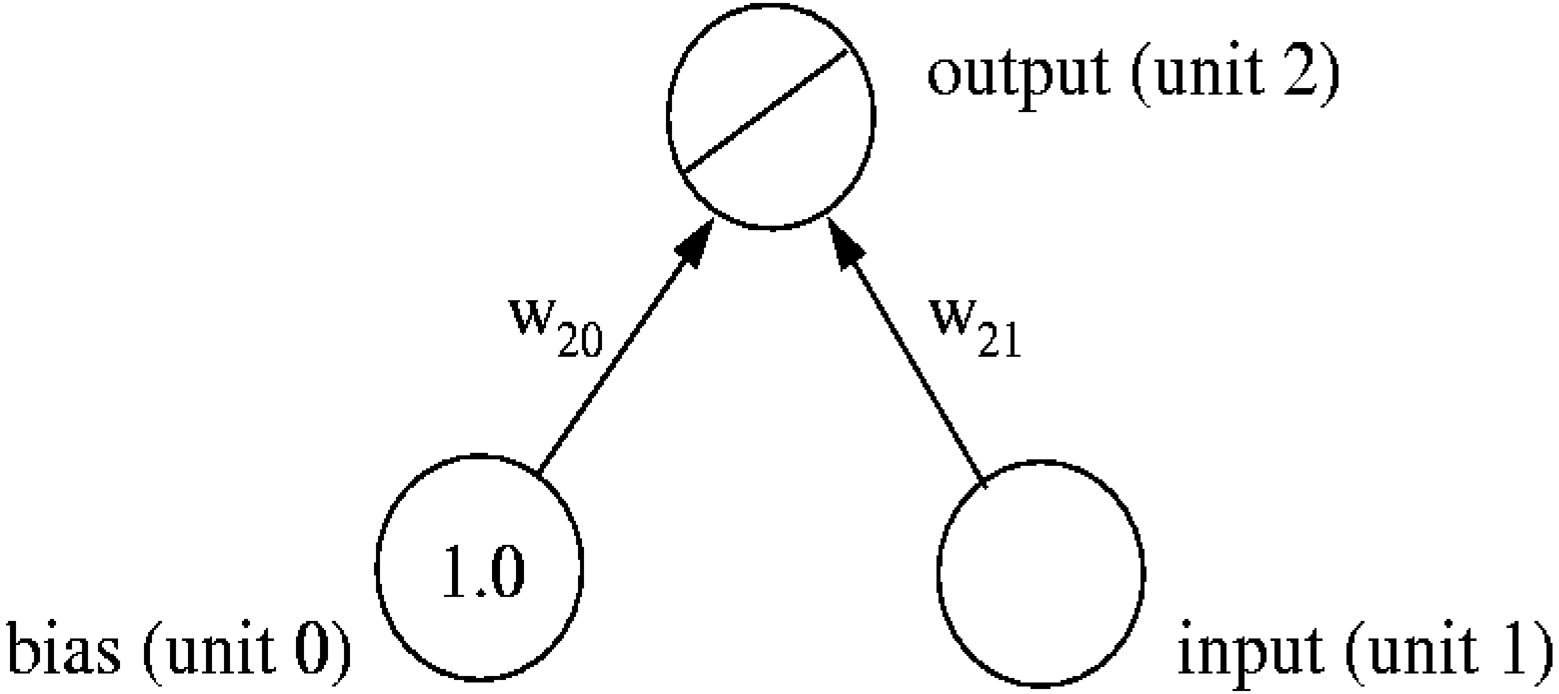
Simple Neural Network Example

- **The output unit computes the sum**
  - $y_2 = y_1\, w_{21} + 1.0\, w_{20}$
- **This is equivalent to the previous equation**

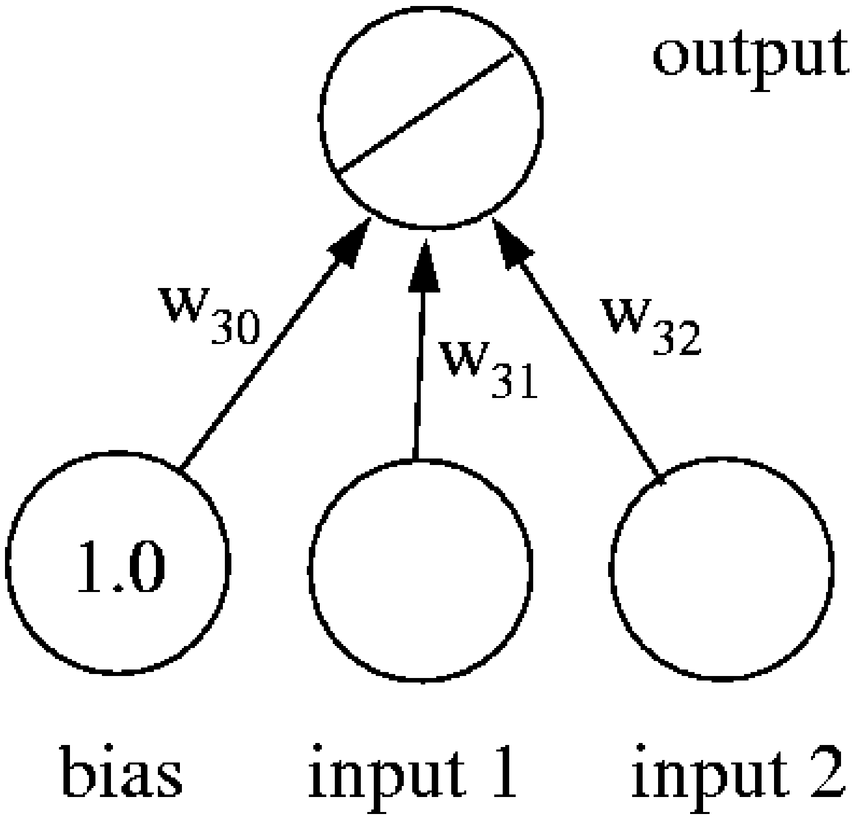
$$\boldsymbol{y = w_1\, x + w_0}$$

  - with w_{21} implementing the slope of the straight line
  - and w_{20} its intercept with the y1-axis

Simple Neural Network

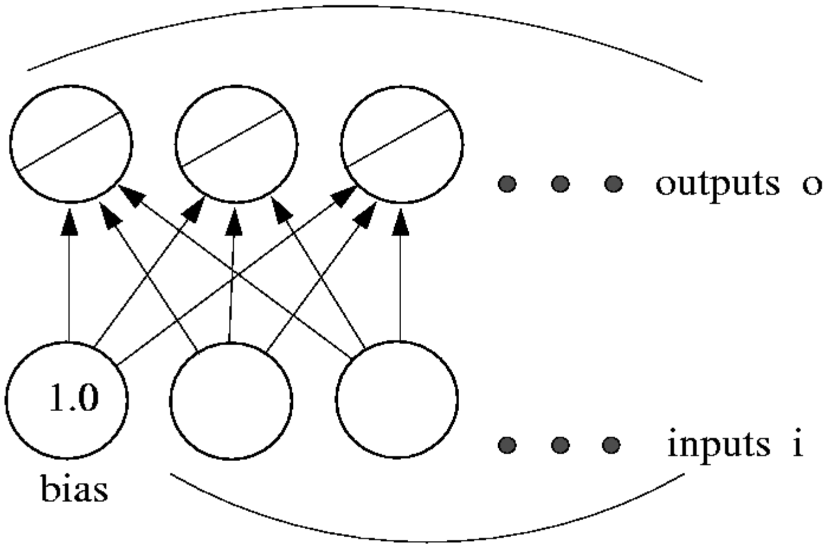


Multiple inputs



Network Structure

- **The network now has a typical layered structure**
  - a layer of input units (and the bias)
  - connected by a layer of weights to
  - a layer of output units



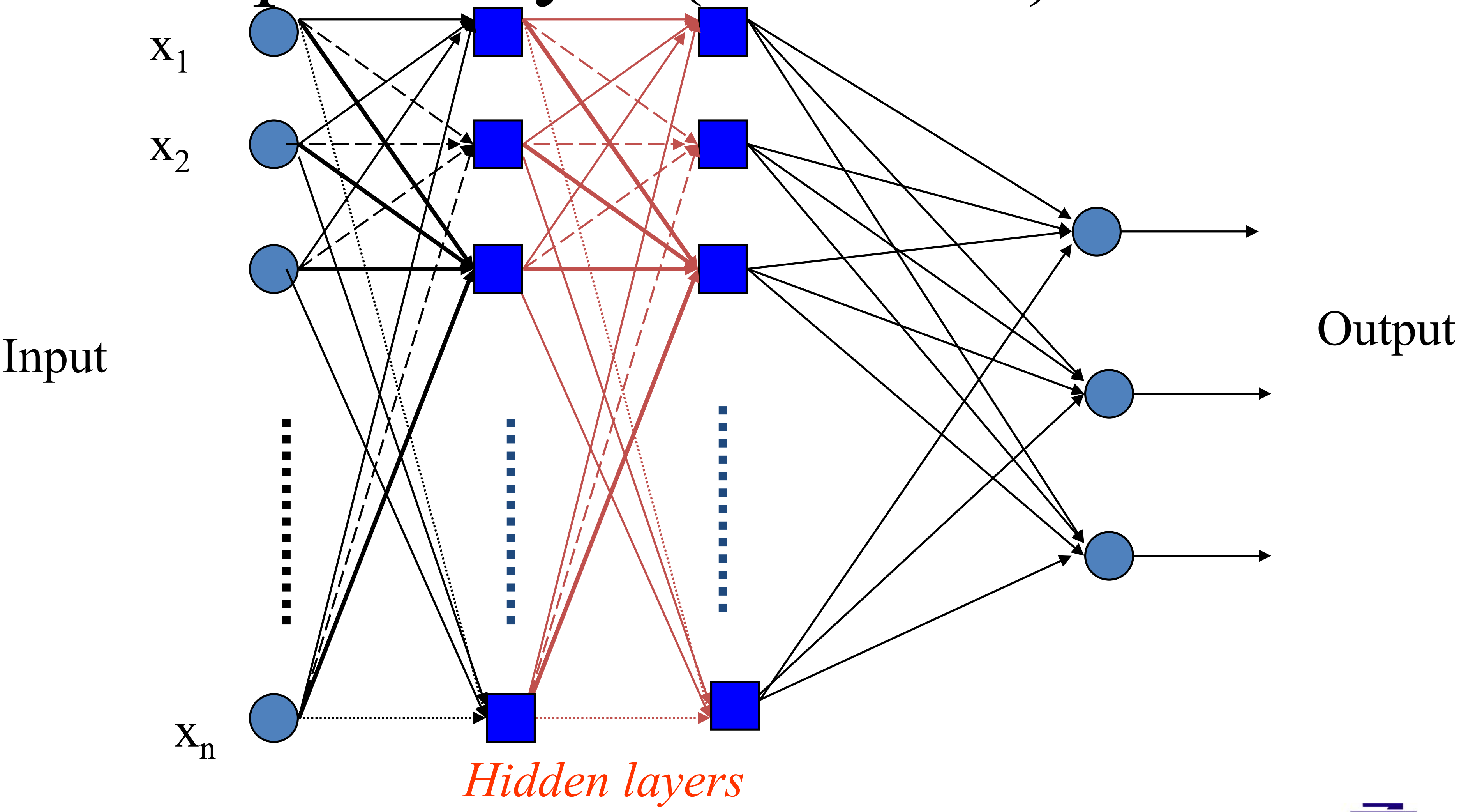
NN Definition

- **NN is a network of many simple processors ("units"), each possibly having a small amount of local memory**
- **The units are connected by communication channels ("connections") which usually carry numeric data of various kinds**
- **The units operate only on their local data and on the inputs they receive via the connections**

About Neural Networks

- **Most NNs have some sort of "training" rule**
  - weights of connections are adjusted based on data
- **NNs "learn" from examples**
- **NNs are capable of exhibiting capability for generalization beyond the training data**

Example:4-layer (2-hidden) network



General Artificial Neuron Model

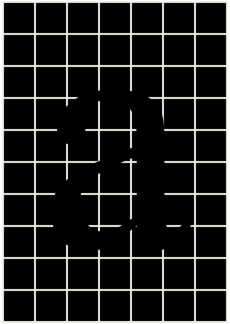
- Has five components, shown in the following list

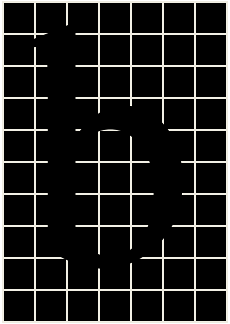
The subscript i indicates the i-th input or weight

1. A set of inputs, x_i
2. A set of weights, w_i
3. A bias, u
4. An activation function, f
5. Neuron output, y

Example: Handwritten Digit Classification

- First need a data set to learn from: sets of characters

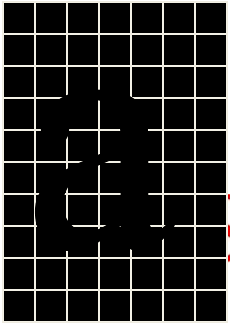


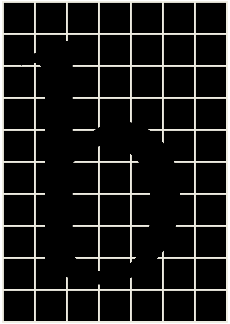


- How are they represented?
  - Input vector $\underline{x} = (x_1, \ldots, x_n)$ to the network
    - vector of ones and zeroes for each pixel according to whether it is black/white

Example: Hand-written Digit Classification

- Set of input vectors is our Training Set X which has already been classified into a's and b's
- Given a training set X, our goal is to tell if a new image is an a or b
    - Classify it into one of 2 classes C_1 or C_2
        - in general one of k classes C_1.. C_k





Generalization

Q. How do we tell if a new unseen image is an a or b?

A. Brute force: have a library of all possible images

There are:

256 x 256 pixels => $2^{256 \times 256} = 10^{158{,}000}$ images

Impossible! Typically have less than a few thousand images in training set

Generalization Problem

- System must be able to classify UNSEEN patterns from the patterns it has seen
  - I.e. Must be able to generalize from the data in the training set
- Intuition: biological neural networks do this well, so maybe artificial ones can do the same?
- As they are also shaped by experiences maybe we'll also learn about how the brain does it

What is backprop?

- **Short for "backpropagation of error“**
- **Method for computing the gradient of the error function with respect to the weights for a feed forward network**
- **Straightforward but elegant application of the chain rule (elementary calculus)**
- ***Backpropagation* or *backprop* often refers to a training method that uses backpropagation to compute the gradient**

Backpropagation Learning

- In backpropagation learning, every time an input vector of a training sample is presented, the output vector *o* is compared to the desired value *d*
- The comparison is done by calculating the squared difference of the two:

$$\mathrm{Err} = (d - o)^2$$

Backpropagation Learning

- **The value of Err tells us how far away we are from the desired value for a particular input**
- **The goal of backpropagation is to minimize the sum of Err for all the training samples**
  - **so that the network behaves in the most "desirable" way**
  - **Minimize:**

$$\Sigma\ Err=(d-o)^2$$

Training

- Once the neural network has been created it needs to be trained
- Initialize the neural net with random weights and then feed it a series of inputs
- For each input we check to see what its output is and adjust the weights accordingly so that whenever it sees a positive example it outputs 1 otherwise 0

Learning: The training process

- **Progressive adaptation of the synaptic connection values to let the NN learn the desired behavior**
  - **Feed the NN with an input from training data**
  - **Compare the NN's outputs with the training data's output**
  - **The differences are used to compute the error of the NN's response**

Learning:
The training process

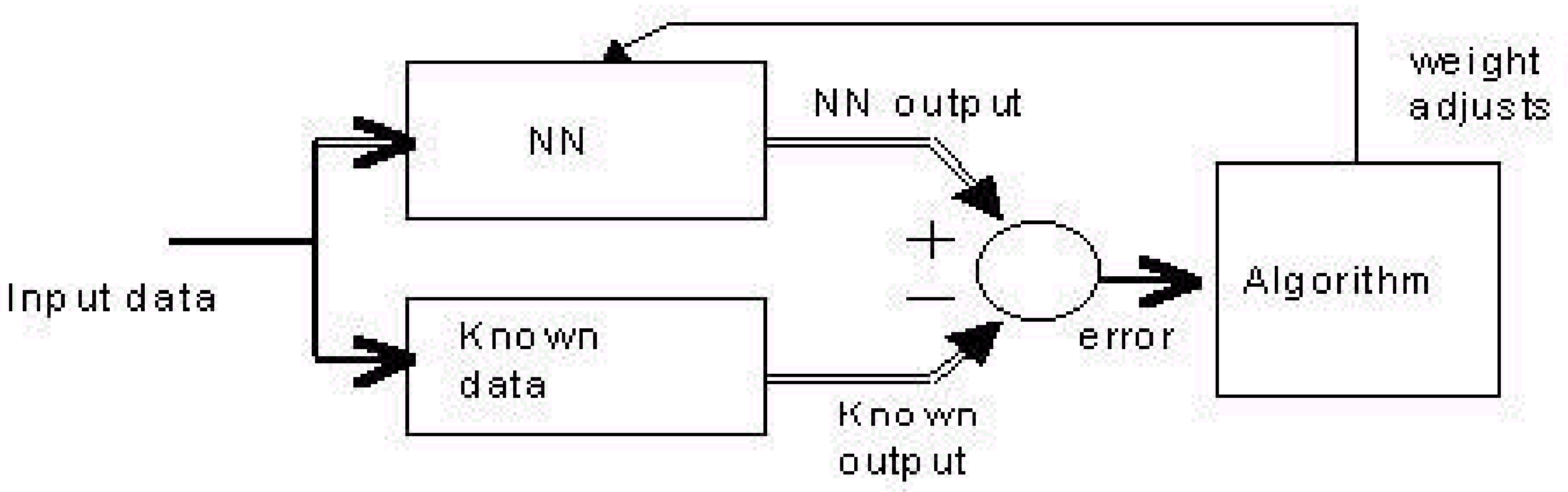
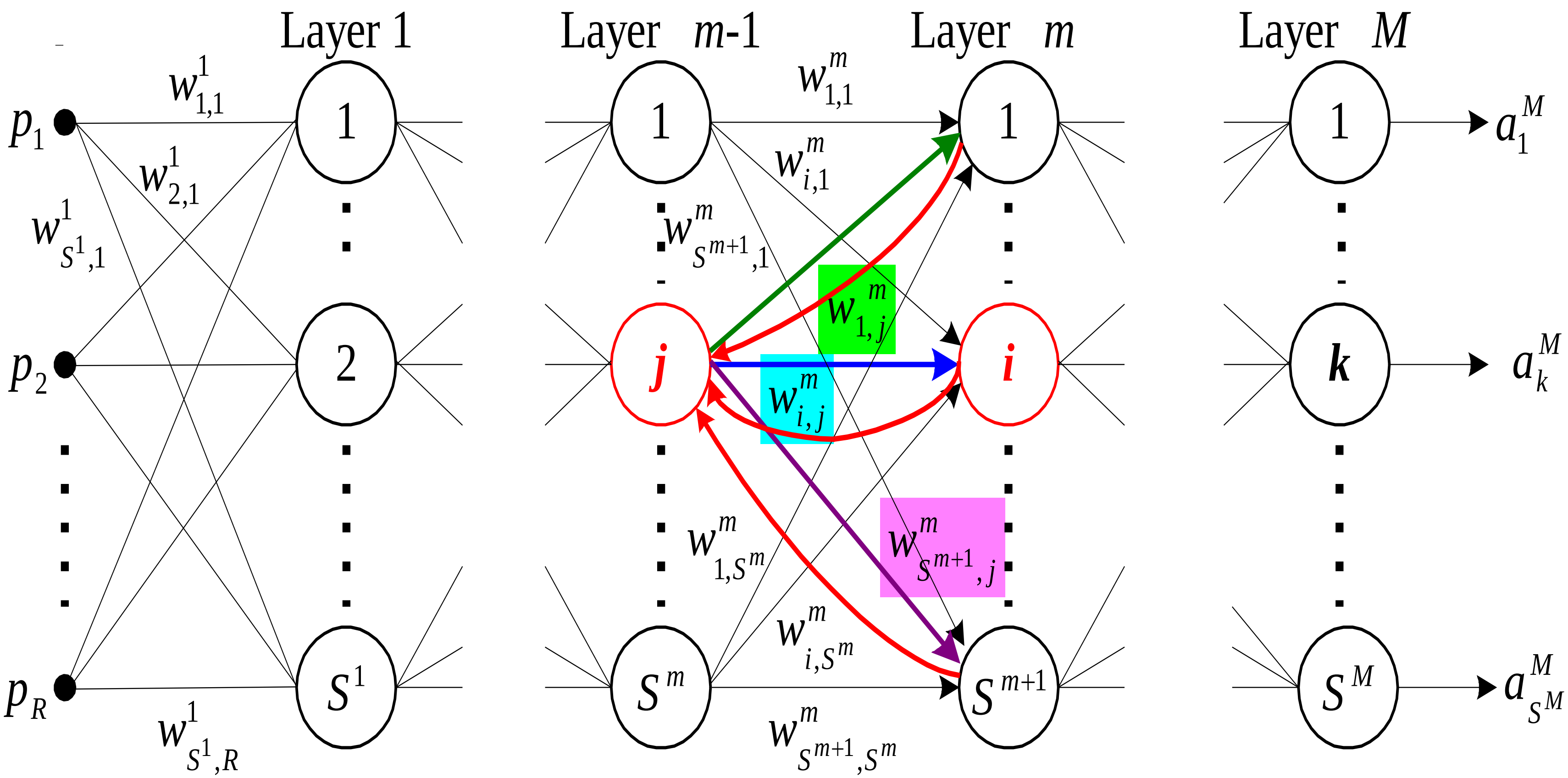


Figure from Answermath.com

BP Neural Network



How to Count Layers?

- **Some people count layers of *units***
  - **Some count the input layer and some don't**
- **Some people count layers of *weights***
  - **How they count skip-layer connections?**
- **We will count hidden-layers to refer to the latent features we are learning**

Backpropagation Application

- *Learning rate and local minima*
  - the selection of a learning rate is of critical importance in finding the true global minimum of the error distance
- *Backpropagation training with too small a learning rate will make agonizingly slow progress*
  - *Too large a learning rate will proceed much faster, but may simply produce oscillations between relatively poor solutions*
- *Both of these conditions are generally detectable through experimentation and sampling of results after a fixed number of training epochs*

Learning Rate

- Typical values for the learning rate parameter are numbers between 0 and 1:
  - $0.05 < \eta < 0.75$
- We would like to use the largest learning rate that still converges to the minimum solution

Momentum

- **The idea is to stabilize the weight change by using a combination of the gradient with a fraction of the previous weight change:**

$$\Delta w(t) = -\partial Ee / \partial w(t) + \alpha \, \Delta w(t-1)$$

where α is taken $0 \leq \alpha \leq 0.9$, and t is the index of the current weight change

Momentum

- This gives the system a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term
- The effects of momentum
  - smooths the weight changes and suppresses oscillations across an error valley
  - when all weight changes are all in the same direction it gives faster convergence
  - enables an escape from small local minima on the error surface

Momentum

- **Often, momentum will allow a larger learning rate and that this will speed convergence and avoid local minima**
- **On the other hand a learning rate of 1 with no momentum will be much faster when no problem with local minima or non-convergence is encountered**

Backpropagation Challenges

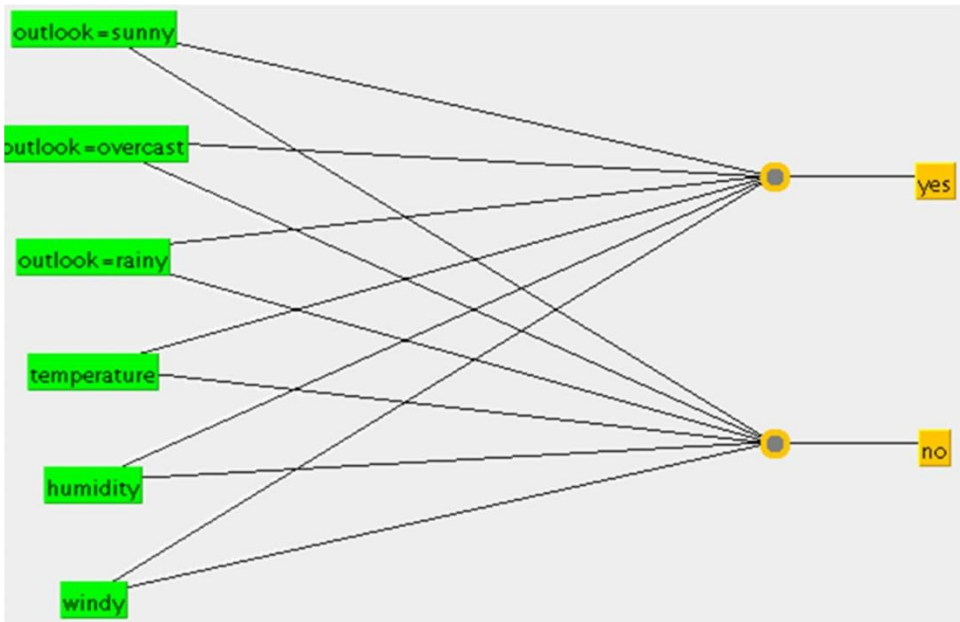
How many layers?

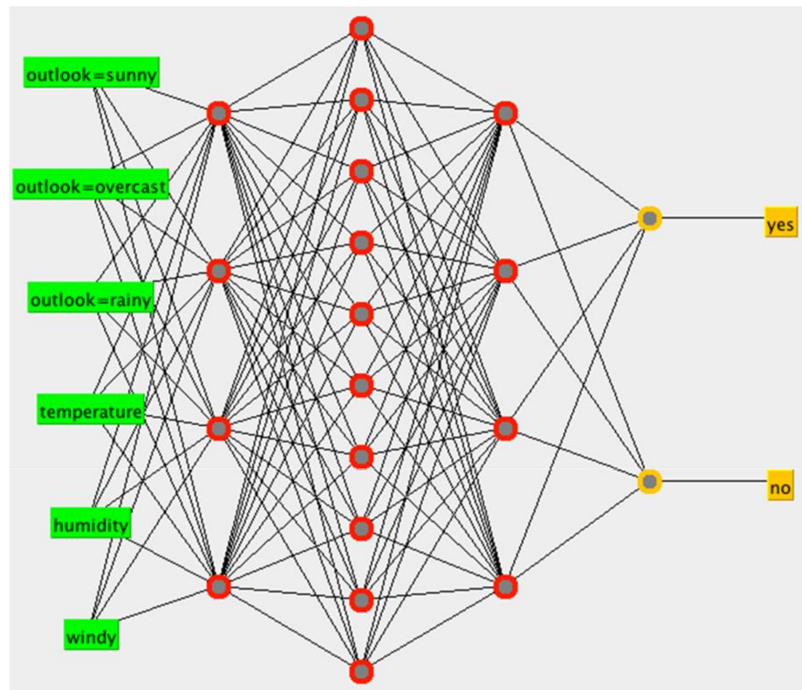
- Zero hidden layers:
  - standard Perceptron algorithm
  - suitable if data is linearly separable
- One-hidden-layer:
  - may be sufficiently accurate for many tasks encountered in practice
  - faster training
- >1 hidden-layer:
  - can represent non-linear features easier
  - error correction has to propagate back farther, each weight's derivative is less directly related to output

Backpropagation Challenges

- **Training may require thousands of backpropagations**
- **Backpropagation can get stuck or become unstable when varying the learning rate parameter**
  - **increasing too much of the learning parameter leads to unstable learning- errors decrease as well as increase during the training process**

Example: ANN in Weka





How to set up an ANN?

- **Input layer: one for each attribute (attributes are numeric, or binary)**
- **Output layer: one for each class (or just one for each variable if the target output is a numeric vector)**
- **How many and how large are hidden layers?**
  - Use prior expectations or start small
  - WEKA: (#input+ #output)/2

What Are The Weights?

- **They're learned from the training set**
- **Iteratively minimize the error using steepest descent**
- **Gradient is determined using the "backpropagation" algorithm**
- **Change in weight computed by multiplying the gradient by the "learning rate" and adding the previous change in weight multiplied by the "momentum"**
- **Often involves (much) experimentation**
  - *number and size of hidden layers*
  - *value of learning rate and momentum*

Parameters

- **hiddenLayers: set GUI to *true* and try 5, 10, 20 nodes**
- **learningRate, momentum**
- **makes multiple passes ("epochs") through the data**
- **training continues until**
  - *error on the validation set consistently increases*
  - *or training time is exceeded*

Thank you!