Artificial Neural Networks

MultiLayer Perceptrons

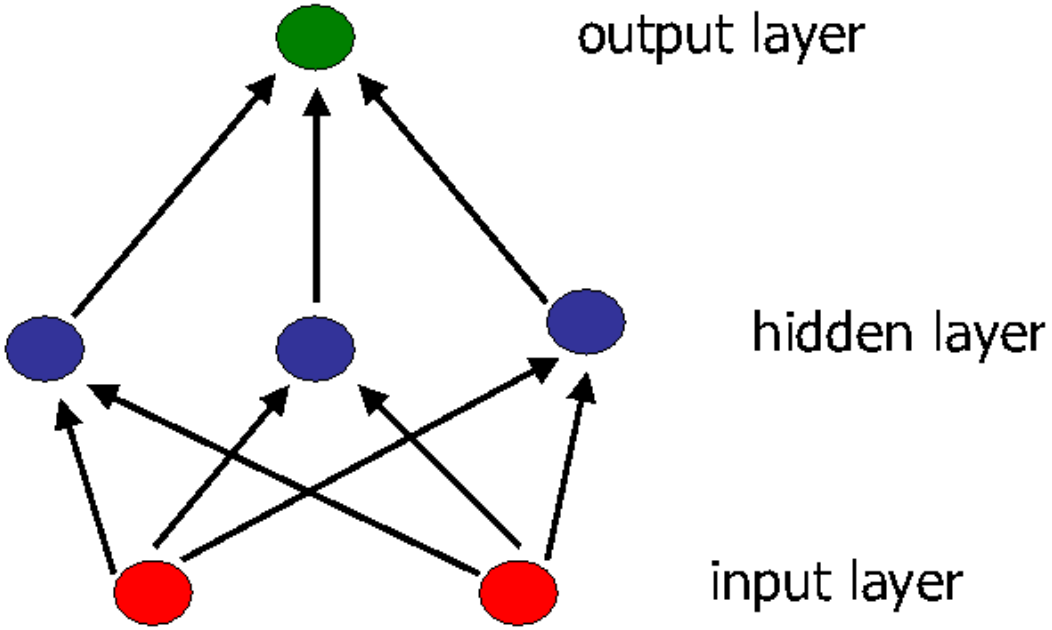
Backpropagation

Part 3/3

Berrin Yanikoglu

Capabilities of Multilayer Perceptrons

Multilayer Perceptron



In Multilayer perceptrons, there may be one or more hidden layer(s) which are called hidden since they are not observed from the outside.

Multilayer Perceptron

- Each layer may have different number of nodes and different activation functions:
  - Commonly, same activation function within one layer
  - Typically,
    - sigmoid/tanh activation function is used in the hidden units, and
    - sigmoid/tanh or linear activation functions are used in the output units depending on the problem (classification or function approximation)
- In feedforward networks, activations are passed only from one layer to the next.

Backpropagation

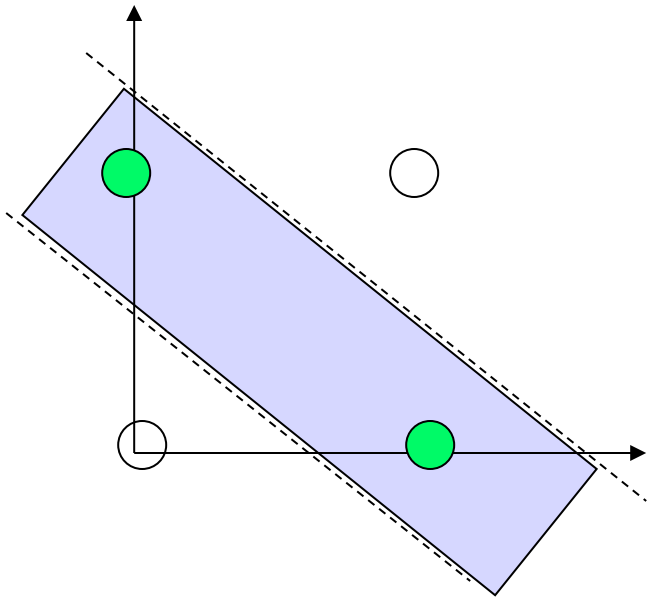
Capabilities of multilayer NNs were known, but ...

- a learning algorithm was introduced by Werbos (1974);
- made famous by Rumelhart and McClelland (mid 1980s - the PDP book)
  - Started massive research in the area

XOR problem

- Learning Boolean functions: 1/0 output can be seen as a 2-class classification problem

- Xor can be solved by a 1-hidden layer network



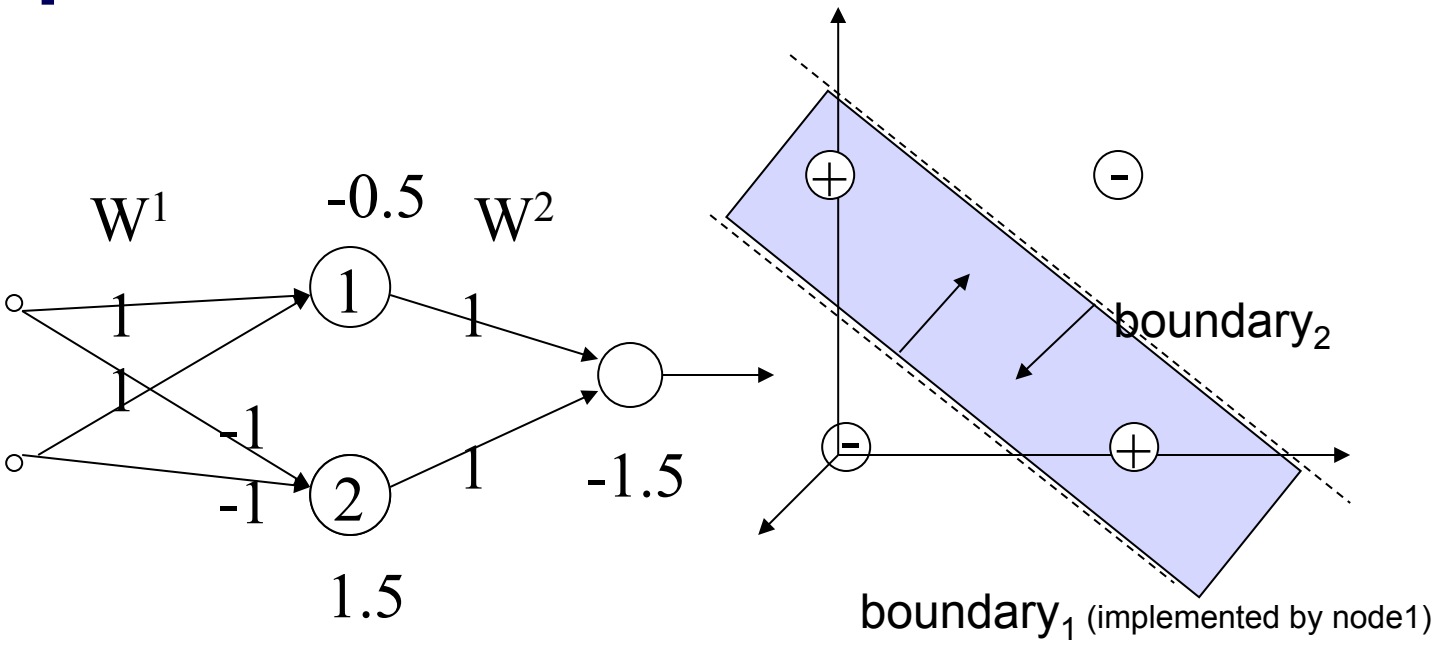
XOR problem

$W^1_1 = [\ 1\ \ 1\ -0.5]$

$W^1_2 = [-1\ -1\ \ 1.5]$

$W^2 = [\ 1\ \ 1\ -1.5]$

Notice how each node implements a decision boundary and the output node combines (AND) their result.



Capabilities (Hardlimiting nodes)

Single layer

- Hyperplane boundaries

1-hidden layer

- Can form any, possibly unbounded convex region

2-hidden layers

- Arbitrarily complex decision regions

Capabilities: Decision Regions

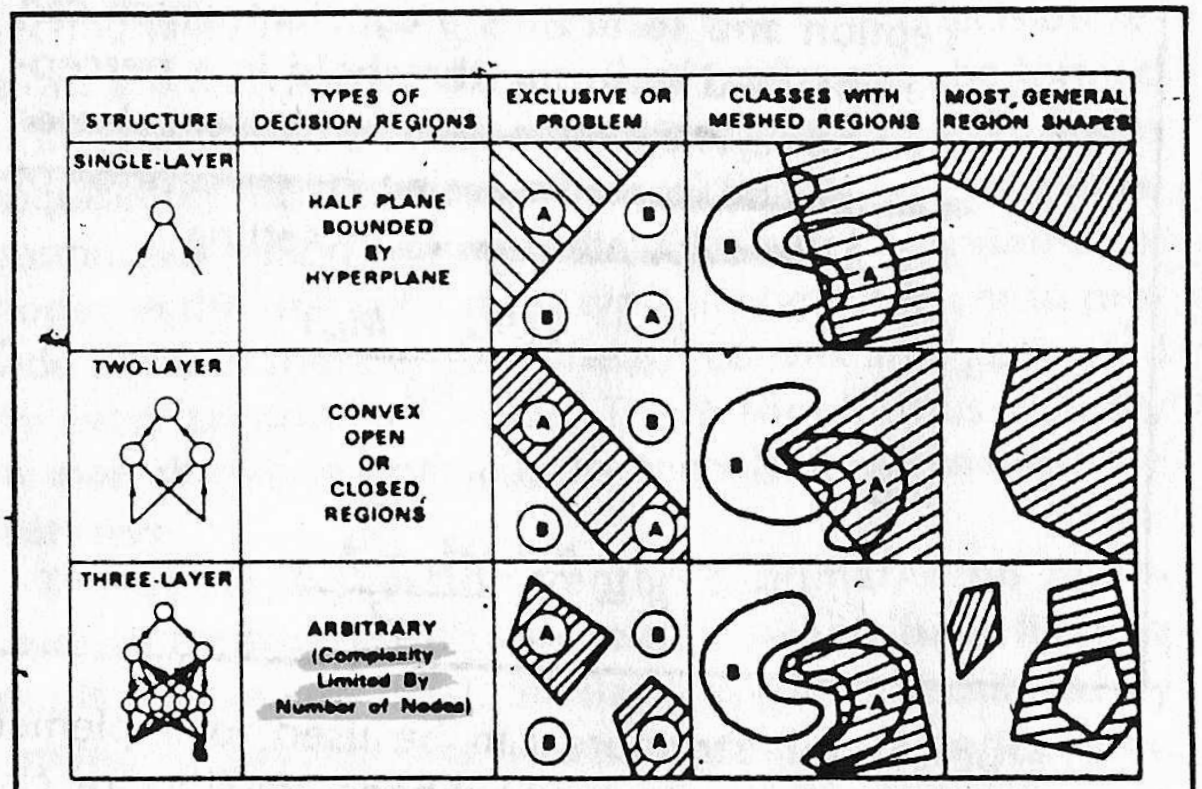


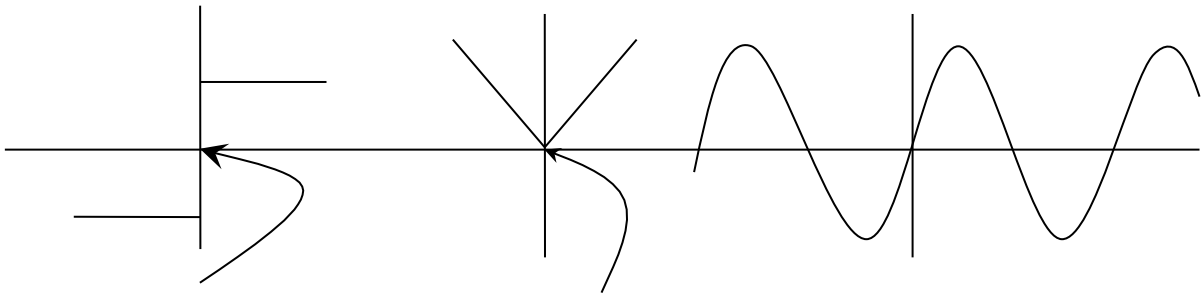
Figure 14. Types of decision regions that can be formed by single- and multi-layer perceptrons with one and two layers of hidden units and two inputs. Shading denotes decision regions for class A. Smooth closed contours bound input distributions for classes A and B. Nodes in all nets use hard limiting nonlinearities.

Capabilities

Thm.(Cybenko 1989, Hornik et al. 1989): Every bounded continuous function can be approximated arbitrarily accurately by 2 layers of weights (1-hidden layer) and sigmoidal units. All other functions can be learned by 2-hidden layer networks.

- Discontinuities can be theoretically tolerated for most real life problems. Also, functions without compact support can be learned under some conditions.

- *Proff is based on Kolmogorov's Thm.*



Discontinuity Continuous with discontinuous first derivative

Performance Learning

Performance Learning

A learning paradigm where the network adjusts its parameters (weights and biases) so as to optimize its "performance"

- Need to define a performance index
  - e.g. mean square error= $1/N \sum_{i=1} (t^i-o^i)^2$

    where we consider the difference between the target and the output for a pattern i.

- Search the parameter space to minimize the performance index with respect to the parameters

Performance Optimization

Iterative minimization techniques is a form of performance learning:

- Define E(.) as the performance index
- Starting with an initial guess w(0), find w(n+1) at each iteration such that

$$E(w(n+1)) < E(w(n))$$

- In particular, **we will see that Gradient Descent** is an iterative (error) minimization technique

Basic Optimization Algorithm

Start with initial guess $\mathbf{w}_0$ and update the guess in each stage moving along the search direction:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$$

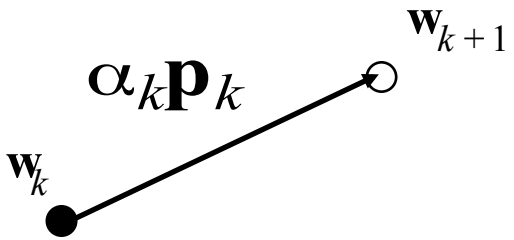
or

$$\Delta\mathbf{w}_k = (\mathbf{w}_{k+1} - \mathbf{w}_k) = \alpha_k \mathbf{p}_k$$

A new state in the search involves deciding on a search direction and the size of the step to take in that direction:

$\mathbf{p}_k$ - Search Direction

α_k - Learning Rate



Gradient Descent

Also known as

Steepest Descent

…

Performance Optimization

Iterative minimization techniques: Gradient Descent

- Successive adjustments to w are in the direction of the steepest descent (direction opposite to the gradient vector)

$$w(n+1) = w(n) - \eta \nabla E(w(n))$$

gradient

Performance Optimization: Iterative Techniques Summary - ADVANCED

Choose the next step so that the function decreases:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$$

For small changes in $\mathbf{x}$ we can approximate $F(\mathbf{x})$ using the Taylor Series Expansion:

$$F(\mathbf{x}_{k+1}) = F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{g}_k^T \Delta\mathbf{x}_k$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

If we want the function to decrease, we must choose p_k such that:

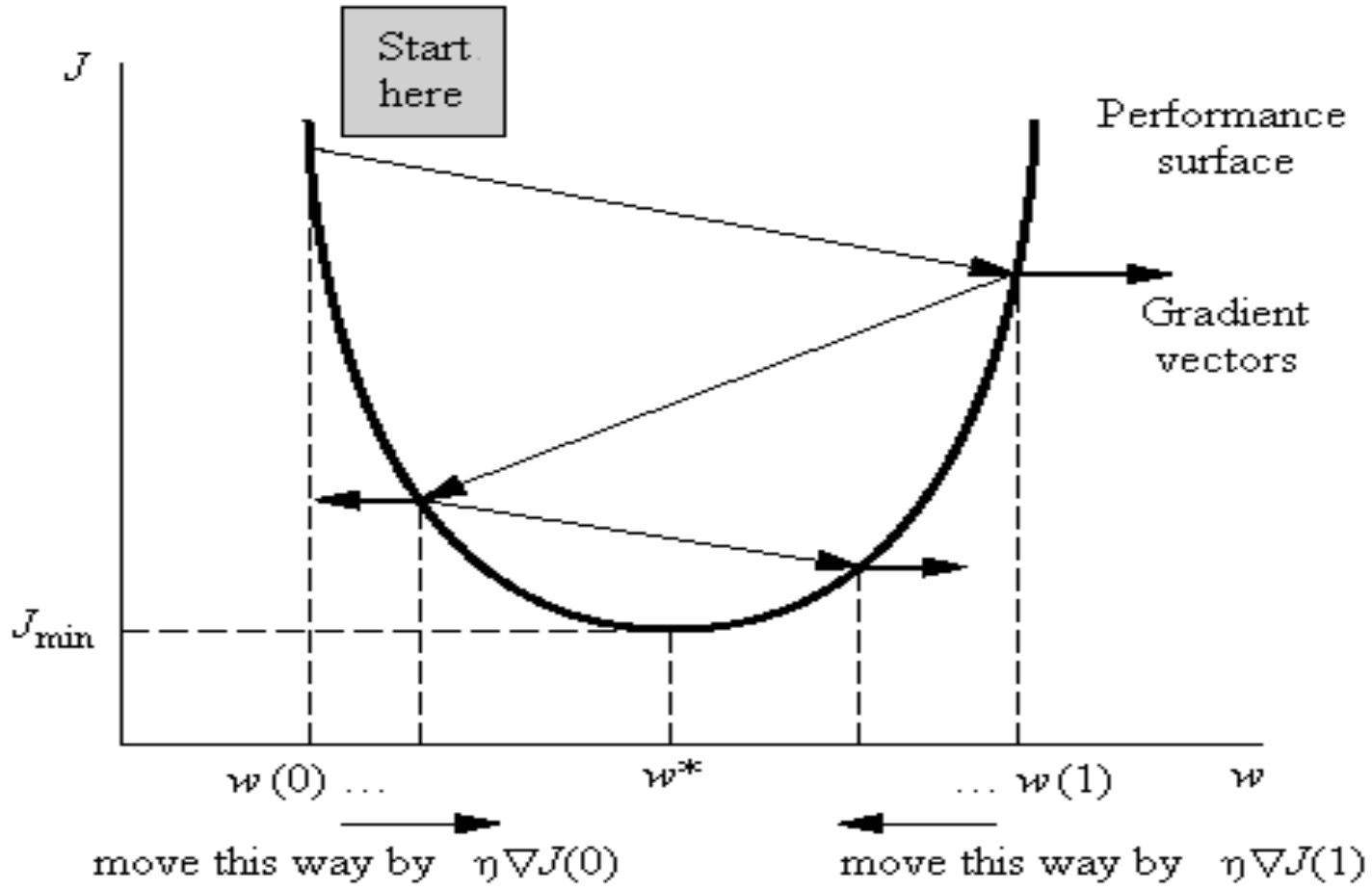
$$\mathbf{g}_k^T \Delta\mathbf{x}_k = \alpha_k \mathbf{g}_k^T \mathbf{p}_k < 0$$

We can maximize the decrease by choosing:

$$\mathbf{p}_k = -\mathbf{g}_k$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

Steepest Descent



Example

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

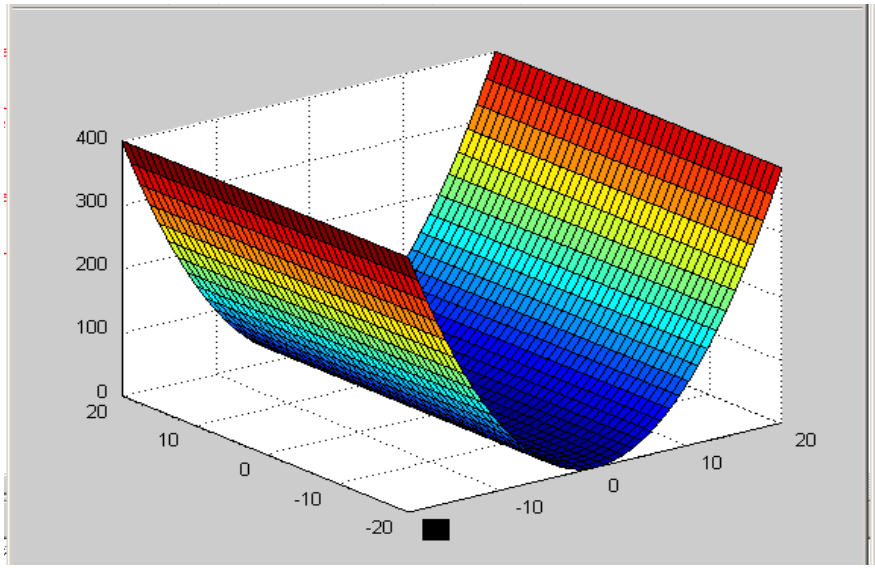
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x} = \mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

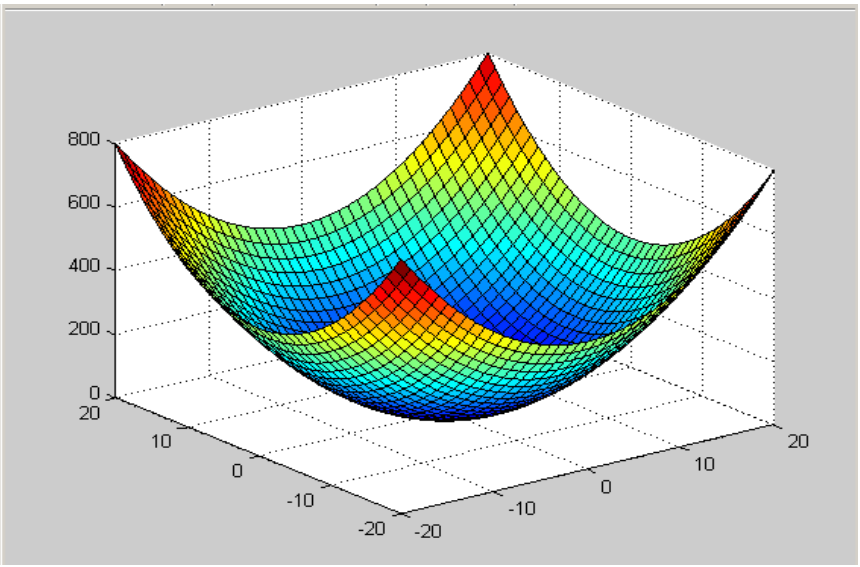
$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

Two simple error surfaces (for 2 weights)



a)



b)

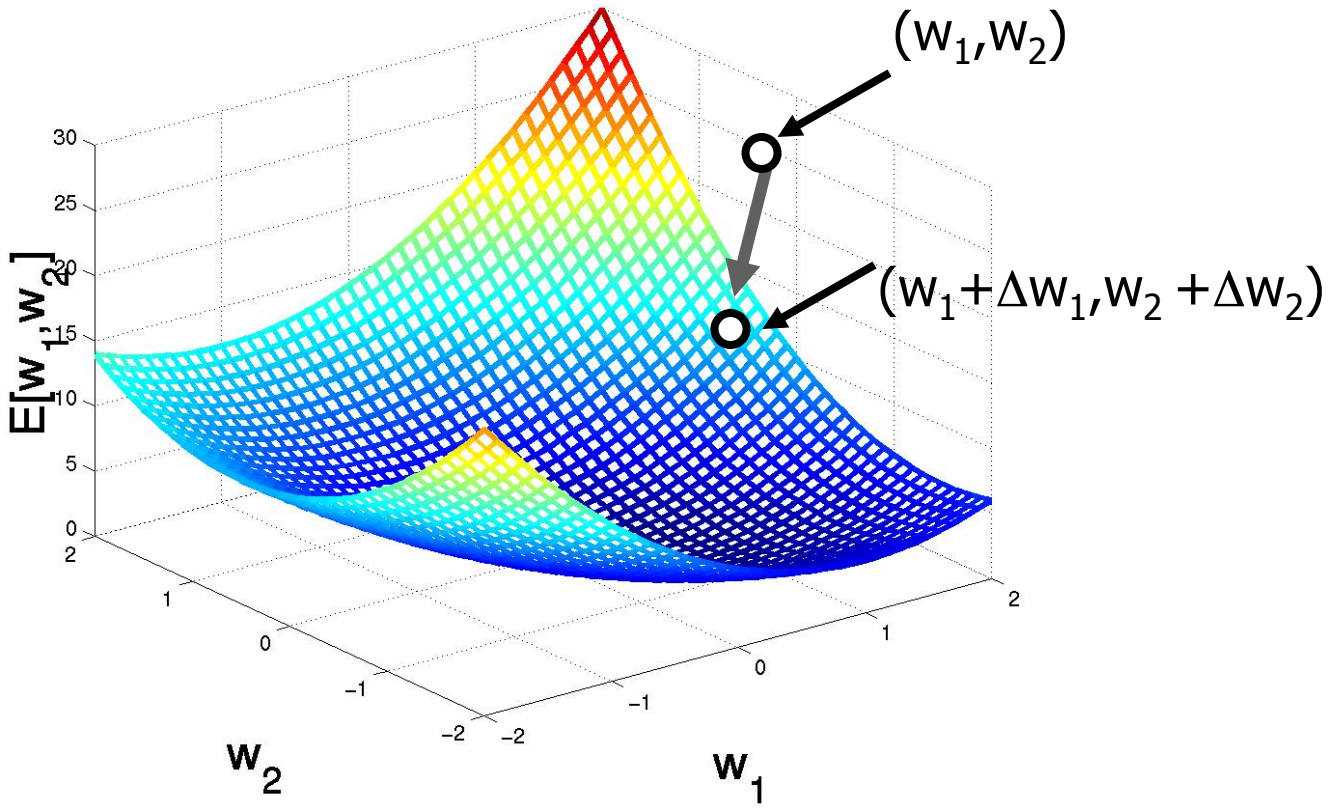
In a) as you move parallel to one of the axis, there is no change in error

In b) moving diagonally, rather than parallel to the axes, brings the biggest change.

Gradient Descent: illustration

Gradient:
$\nabla E[w]=[\partial E/\partial w_0,\ldots \partial E/\partial w_n]$

$\Delta w=-\eta \nabla E[w]$

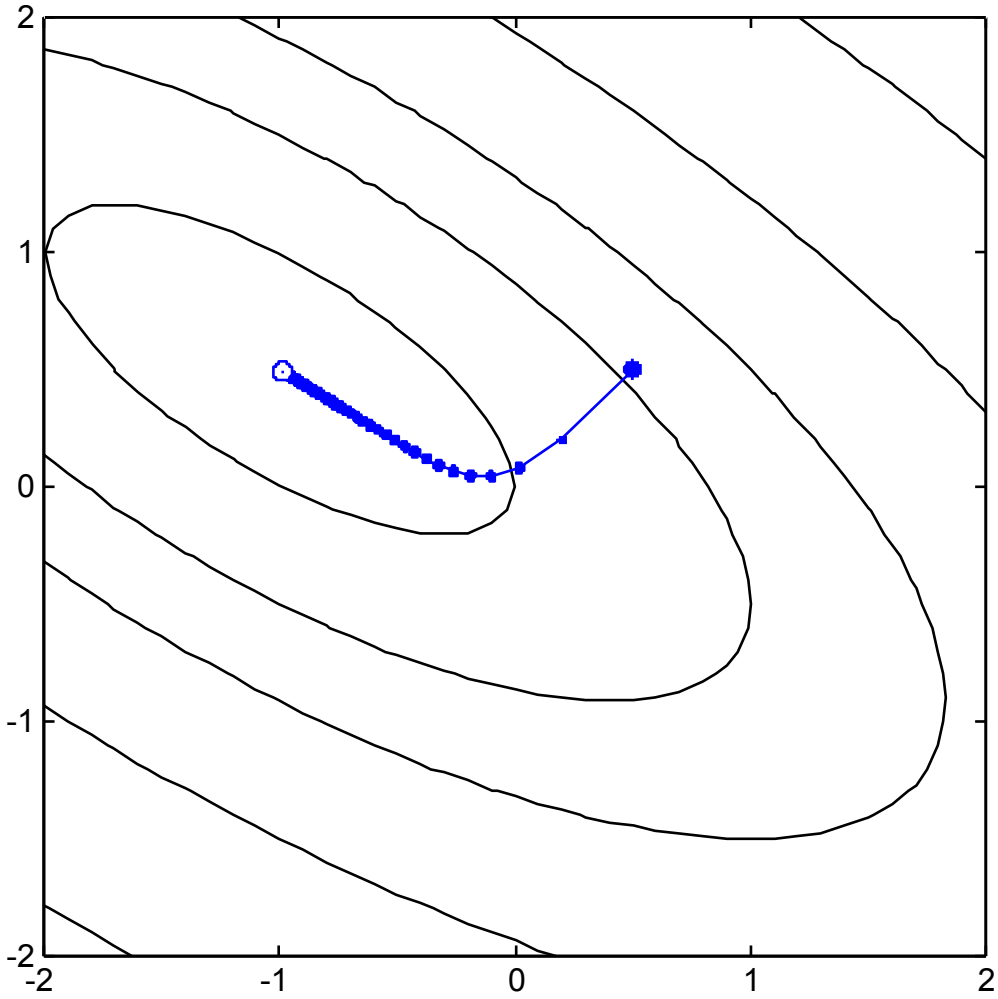


Gradient Vector

Each dimension of the gradient vector is the partial derivative of the function with respect to one of the dimensions.

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

Plot



Gradient Descent
for

Delta Rule for Adaline (Linear Activation)

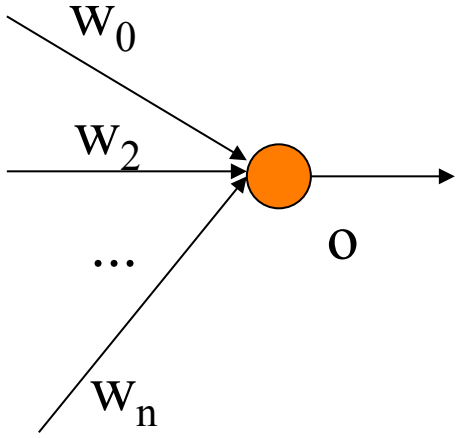
To understand, consider simpler *linear unit*, where

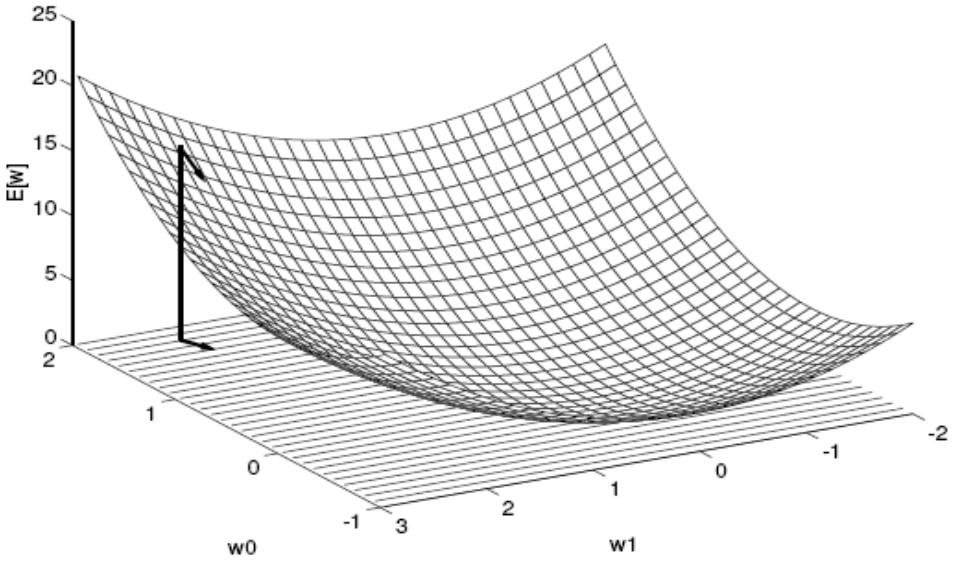
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's 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

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

Training rule:

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

i.e.,

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

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

Stochastic Gradient Descent

Approximate Gradient Descent
(Stochastic Backpropagation)

Normally, in gradient descent, we would need to compute how the error over all input samples (true gradient) changes with respect to a small change in a given weight.

But the common form of the gradient descent algorithm takes one input pattern, **compute the error of the network on that pattern only**, and updates the weights using only that information.

- Notice that the new weight may not be good/better for all patterns, but we expect that if we take a small step, we will average and approximate the true gradient.

Stochastic Approximation to Steepest Descent

Instead of updating every weight until all examples have been observed, **we update on every example**:

$$\nabla w_i \cong \eta (t-o) x_i$$

Remarks:

- Speeds up learning significantly when data sets are large
  - Standard gradient descent can be used with a larger step size.
- When there are multiple local minima, stochastic approximation to gradient descent may avoid the problem of getting stuck on a local minimum.

Gradient Descent Backpropagation Algorithm

Derivation for General Activation Functions

Transfer Function Derivatives

Sigmoid:

$$f'(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^2} = \left(1-\frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = \boxed{(1-a)(a)}$$

Linear:

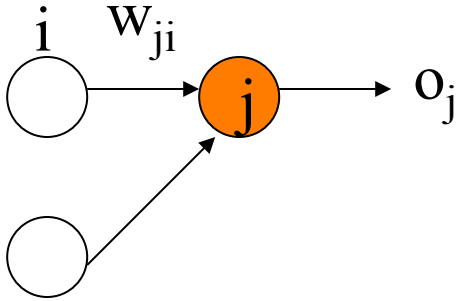
$$f'(n) = \frac{d}{dn}(n) = 1$$

Note: saturation of sigmoid nodes and efficiency of computing the derivative

Stochastic Backpropagation

To calculate the partial derivative of E_p (error on pattern p) w.r.t a given weight w_{ji}, we have to consider whether this is the weight of an output or hidden node:

If w_{ji} is an **output** node weight, the situation is simpler:



$$\frac{dE_p}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$\frac{dE_p}{dw_{ji}} = -(t_j - o_j) \times f'(net_j) \times o_i$$

Note that o_i is the input to node j.

$$E_p = (t_p - o_p)^2$$

$$o_j = f(net_j)$$

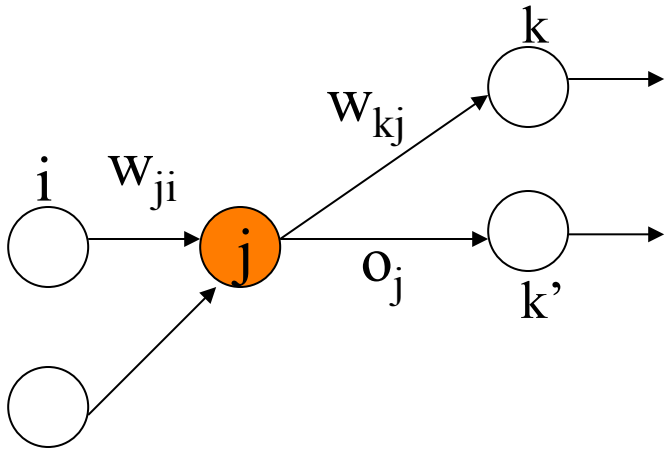
$$net_j = \sum_i o_i w_{ji}$$

The situation is more complex with a hidden node (you don't need to follow), because basically:

- while we know what the output of an output node should be,
- we don't know what the output of a hidden node should be.

Backpropagation – Hidden nodes

If w_{ji} is a **hidden** node weight:



$$\frac{dE_p}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$= \frac{dE}{do_j} \times f'(net_j) \times o_i$$

$$E_p = (t_p - o_p)^2$$

$$net_j = \sum_i o_i w_{ji}$$

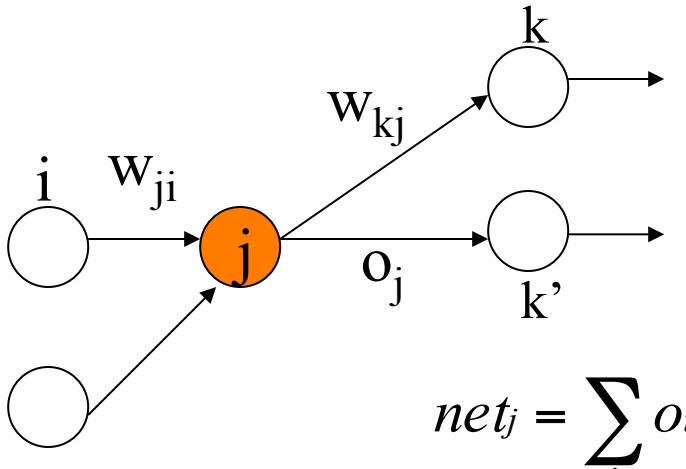
$$o_j = f(net_j)$$

Backpropagation – Hidden nodes

If w_{ji} is a **hidden** node weight:

$$\frac{dE_p}{dw_{ji}} = \frac{dE}{do_j} \times \frac{do_j}{dnet_j} \times \frac{dnet_j}{dw_{ji}}$$

$$= \frac{dE}{do_j} \times f'(net_j) \times o_i$$



$$net_j = \sum_i o_i w_{ji}$$

$$o_j = f(net_j)$$

Note that as j is a hidden node, **we do not know its target.**
Hence, dE/do_j can only be calculated through j's **contribution to the derivative of E w.r.t net_k at the output nodes:**

$$\frac{dE}{do_j} = \sum_k w_{kj} \times \frac{dE}{dnet_k}$$

Vanishing Gradient Problem

The use of chain rule in computing the gradient (for sigmoid or tanh) activations causes the gradient to decrease exponentially, as we move from the output towards the first layers.

Layers will learn in much slower speed.

We will cover the rest briefly.

1) Showing what a single neuron (or in the example 3 of them) can do, in a function approximation situation.
2) Showing networks in different complexity, solving the same problem (approximating a sinusoidal function)

Function Approximation & Network Capabilities

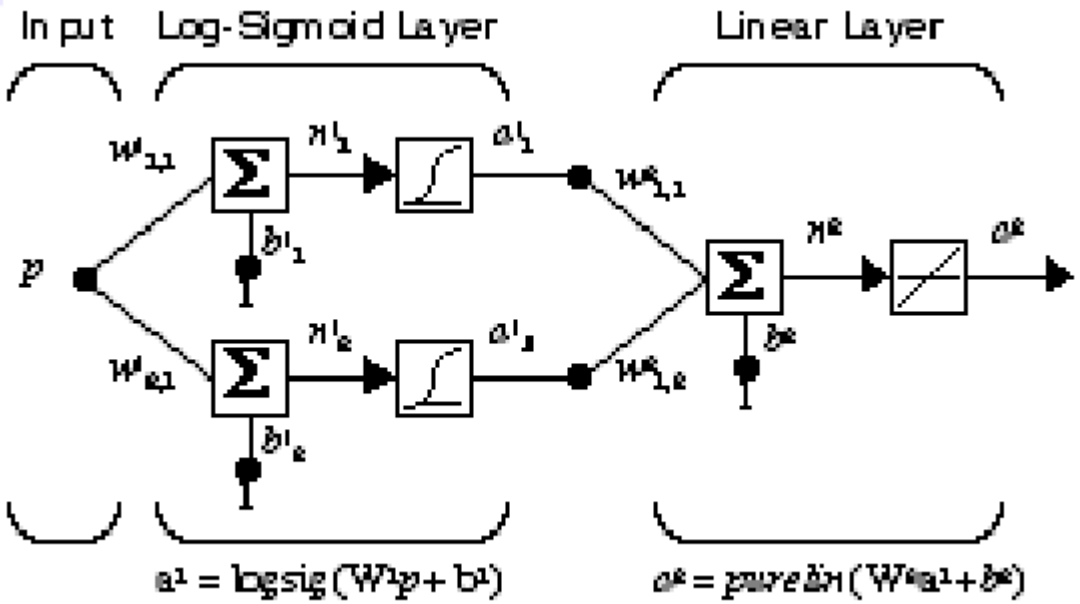
Function Approximation

Neural Networks are intrinsically function approximators:

- we can train a NN to map real valued vectors to real-valued vectors.

Function approximation capabilities of a simple network, in response to its parameters (weights and biases) are illustrated in the next slides.

Function Approximation: Example



$$f^1(n) = \frac{1}{1+e^{-n}}$$

$$f^2(n) = n$$

Superscripts are layer numbers

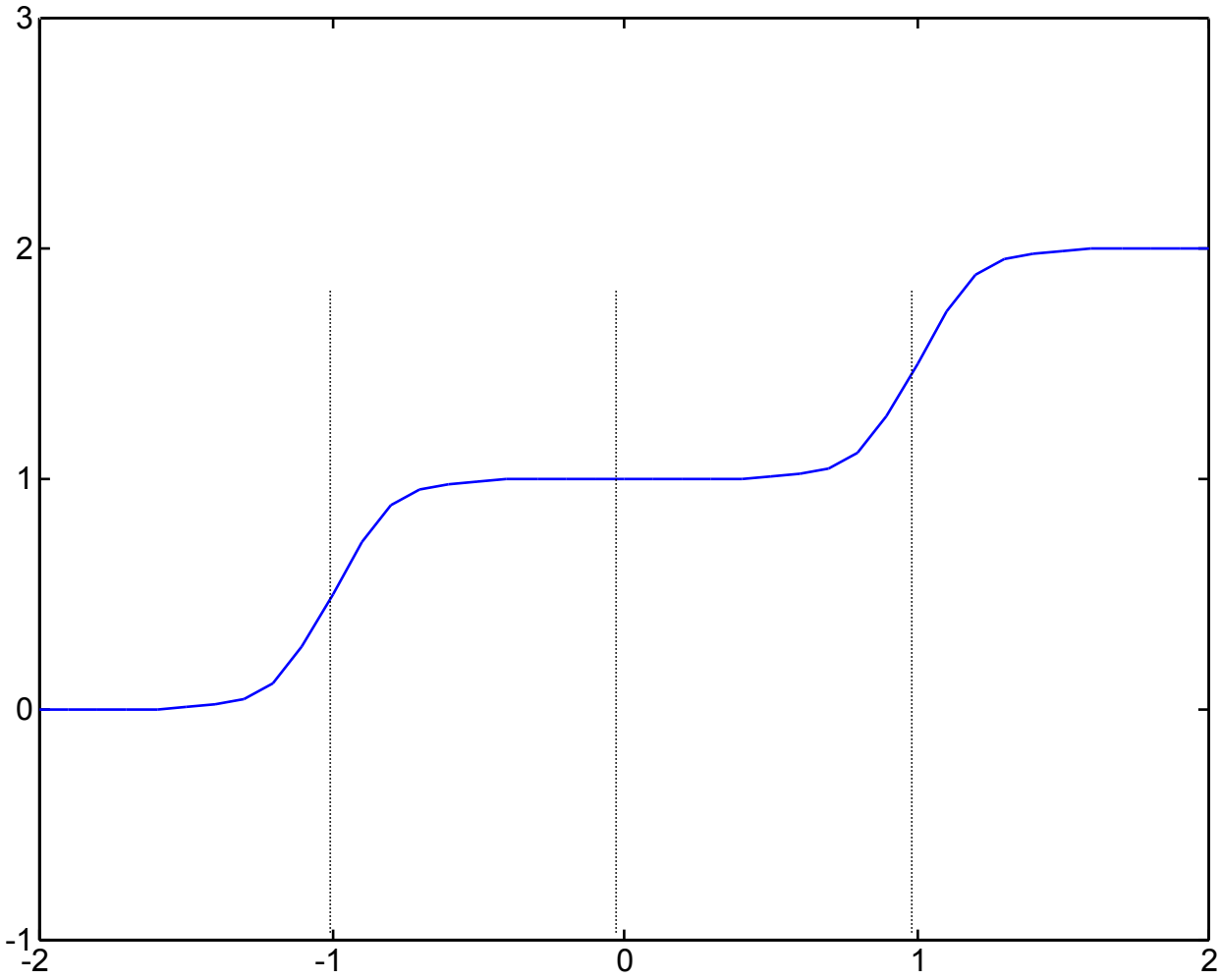
Nominal Parameter Values

$$w^1_{1,1} = 10 \qquad b^1_1 = -10 \qquad w^2_{1,1} = 1$$

$$w^1_{2,1} = 10 \qquad b^1_2 = 10 \qquad w^2_{1,2} = 1$$

$$b^2 = 0$$

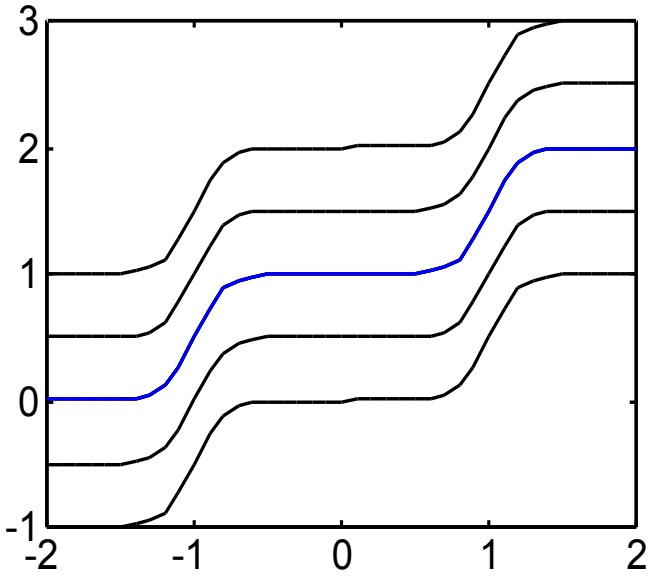
Nominal Response



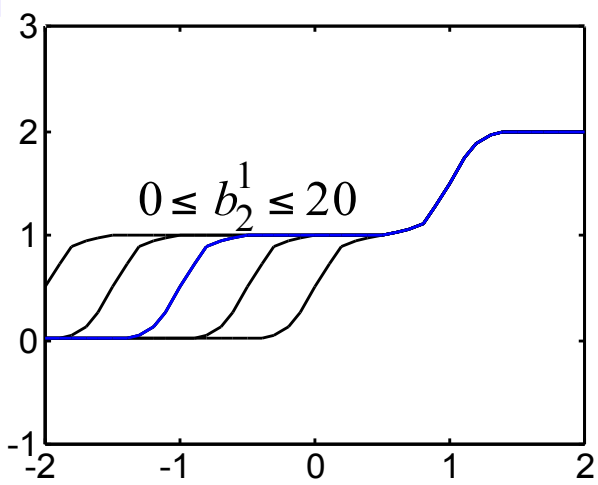
Parameter Variations

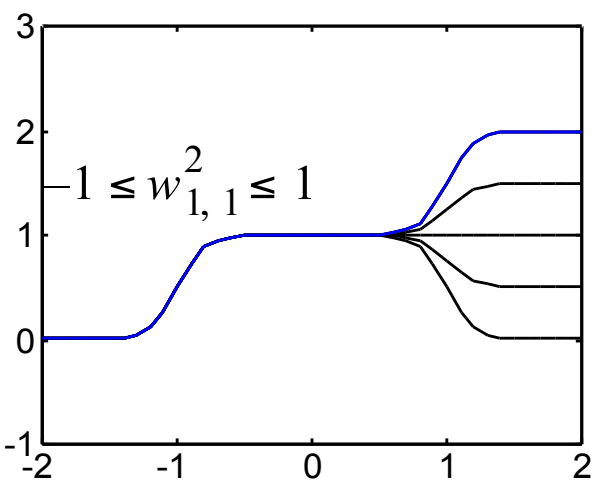
What would be the effect of varying the bias of the output neuron?

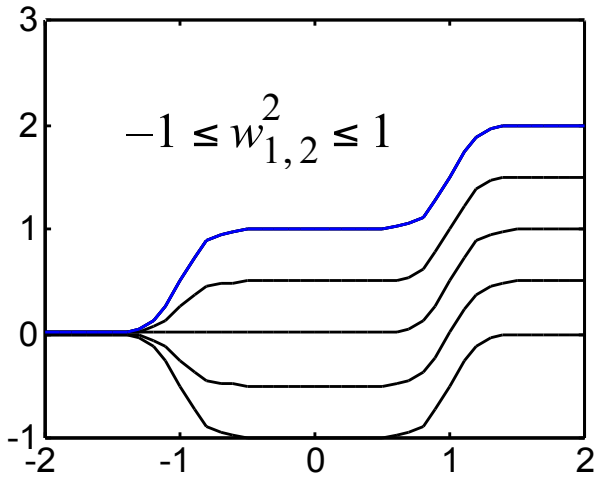
$$-1 \le b^2 \le 1$$

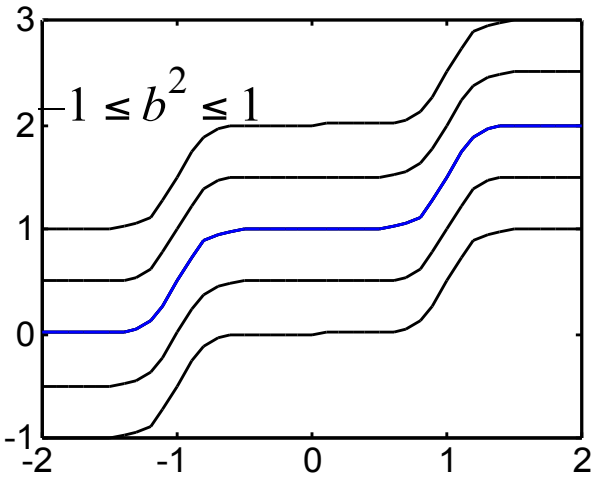


Parameter Variations





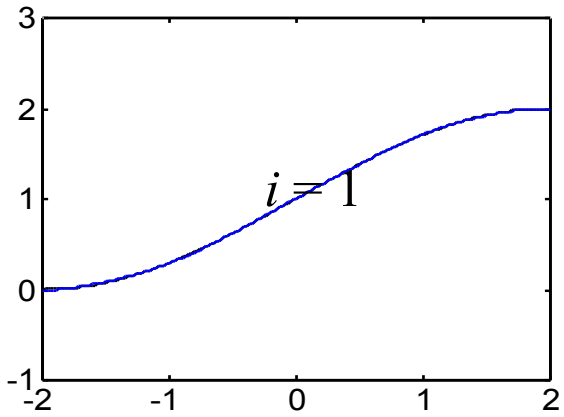


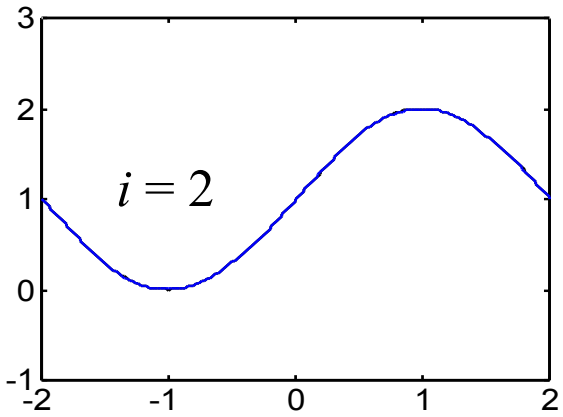


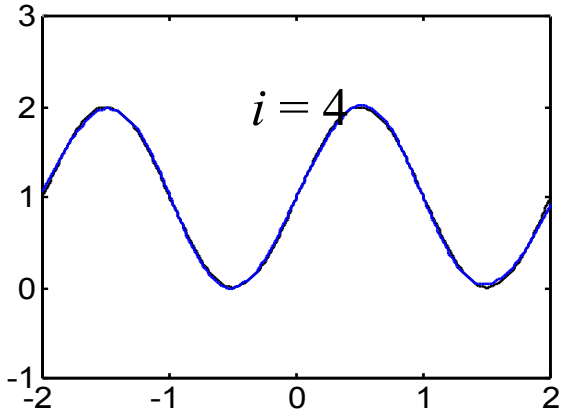
Network Complexity

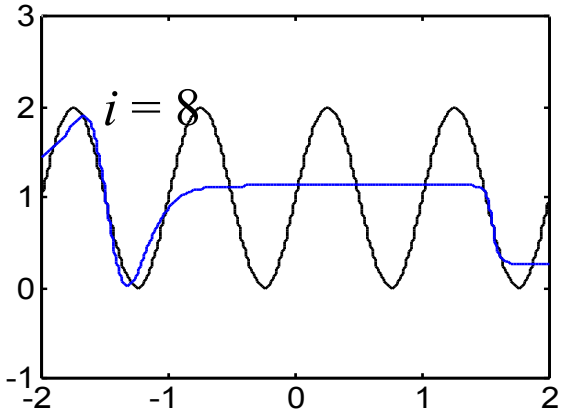
Choice of Architecture

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right)$$ 1-3-1 Network (1 input, 3 hidden, 1 output nodes)



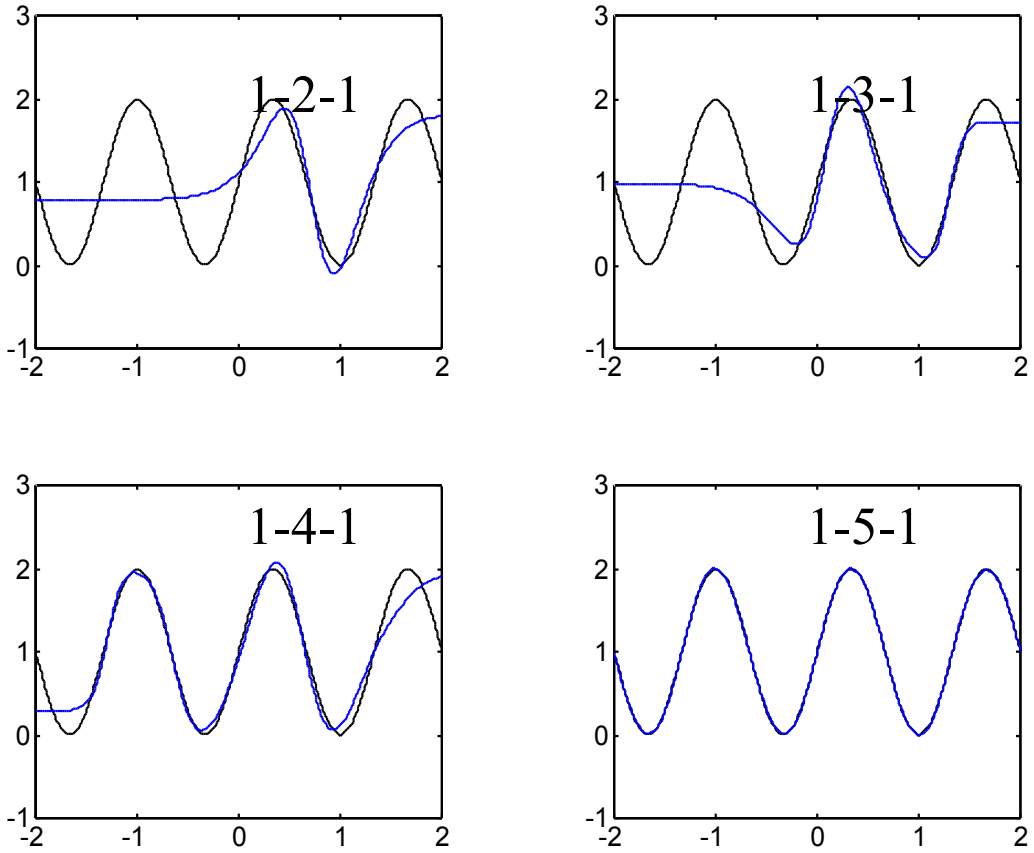






Choice of Network Architecture

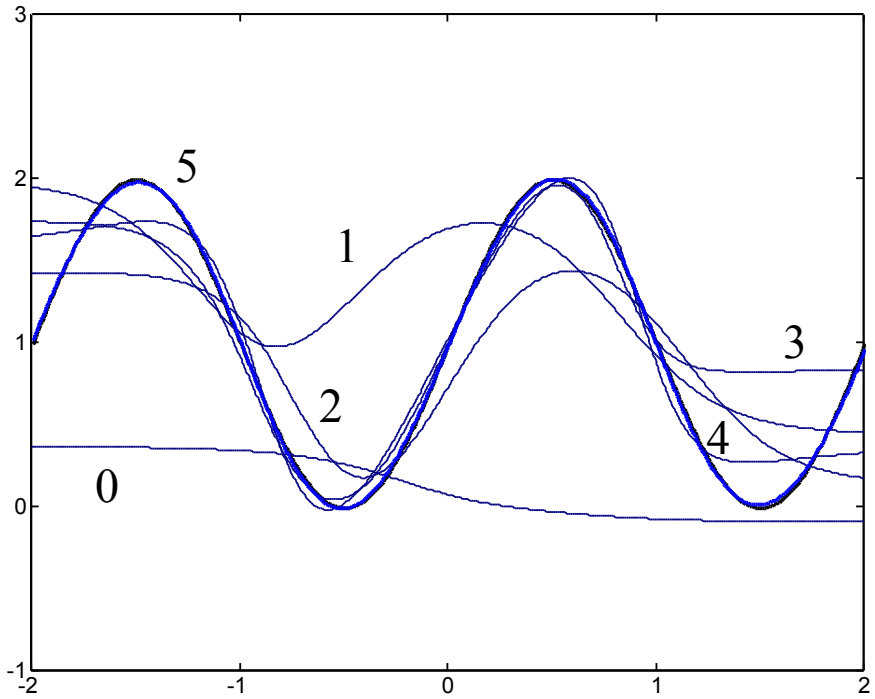
$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right)$$

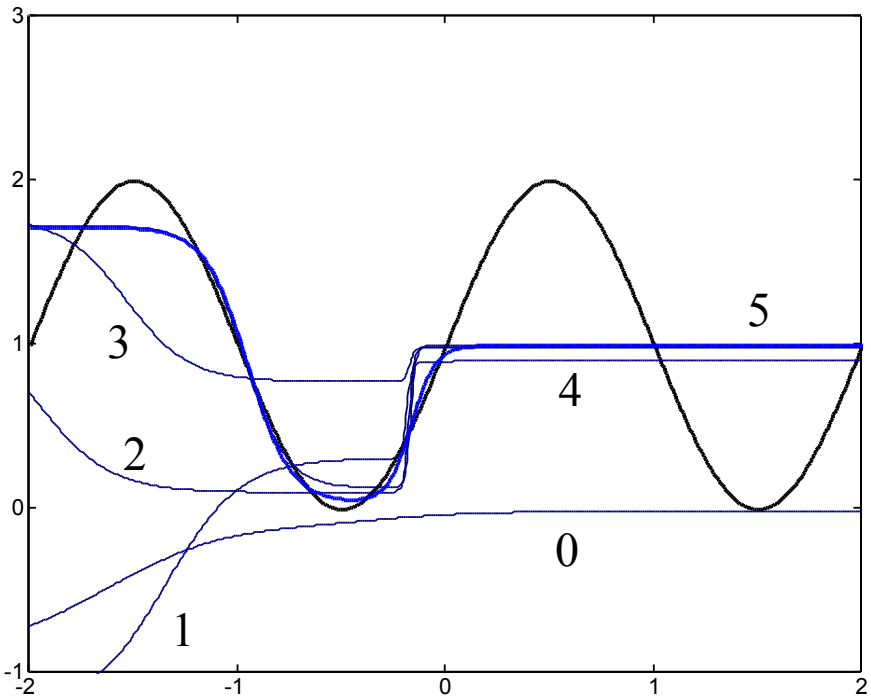


Residual error decreases with O(1/M) where M is the number of hidden units

Convergence in Time

$$g(p) = 1 + \sin(\pi p)$$

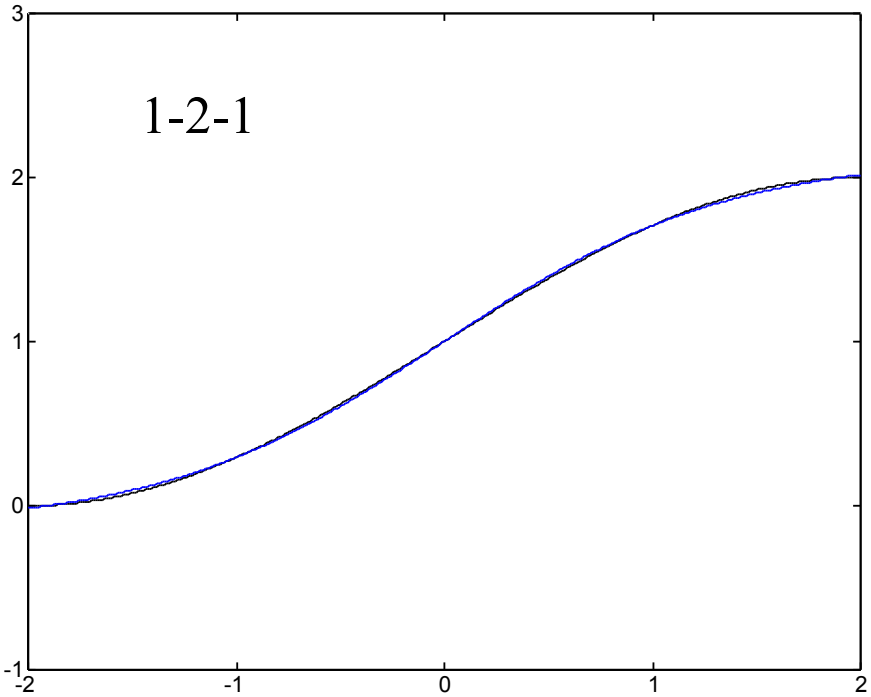


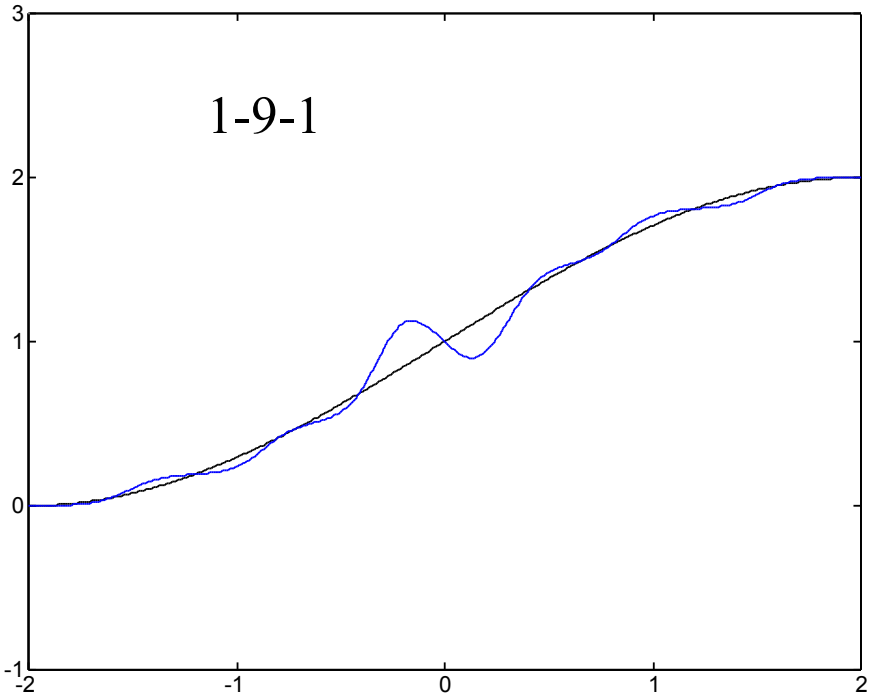


Generalization

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \qquad p = -2, -1.6, -1.2, \ldots, 1.6, 2$$





We will see later how to use complex models (e.g. a higher order polynomial or a MLP with large number of nodes) together with regularization to control model complexity.

- E.g. to keep the weight small, so that even if we use complex models, the weight are kept in check so that the overall model is not prone to overfit.

Next: Issues and Variations on Backpropagation