Artificial Neural Networks

Computers vs. Neural Networks

"Standard" Computers	Neural Networks
one CPU	highly parallel processing
fast processing units	slow processing units
reliable units	unreliable units
static infrastructure	dynamic infrastructure

Why Artificial Neural Networks?

There are two basic reasons why we are interested in building artificial neural networks (ANNs):

- **Technical viewpoint:** Some problems such as character recognition or the prediction of future states of a system require massively parallel and adaptive processing.
- **Biological viewpoint:** ANNs can be used to replicate and simulate components of the human (or animal) brain, thereby giving us insight into natural information processing.

Why Artificial Neural Networks?

Why do we need another paradigm than symbolic AI for building "intelligent" machines?

- Symbolic AI is well-suited for representing **explicit** knowledge that can be appropriately formalized.
- However, learning in biological systems is mostly **implicit** – it is an adaptation process based on uncertain information and reasoning.
- ANNs are inherently parallel and work extremely **efficiently** if implemented in parallel hardware.

How do NNs and ANNs work?

- The "building blocks" of neural networks are the **neurons**.
- In technical systems, we also refer to them as **units** or **nodes**.
- Basically, each neuron
  - receives **input** from many other neurons,
  - changes its internal state (**activation**) based on the current input,
  - sends **one output signal** to many other neurons, possibly including its input neurons (recurrent network)

How do NNs and ANNs work?

- Information is transmitted as a series of electric impulses, so-called **spikes**.
- The **frequency** and **phase** of these spikes encodes the information.
- In biological systems, one neuron can be connected to as many as **10,000** other neurons.
- Usually, a neuron receives its information from other neurons in a confined area, its so-called **receptive field**.

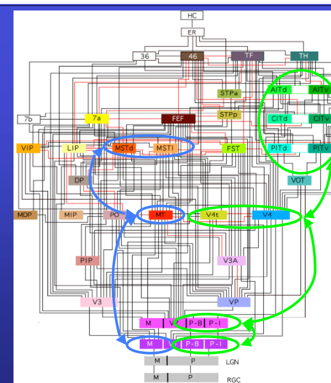
How do NNs and ANNs work?

- In biological systems, neurons of similar functionality are usually organized in separate **areas** (or **layers**).
- Often, there is a **hierarchy** of interconnected layers with the lowest layer receiving sensory input and neurons in higher layers computing more complex functions.
- For example, neurons in macaque visual cortex have been identified that are activated only when there is a **face** (monkey, human, or drawing) in the macaque's visual field.

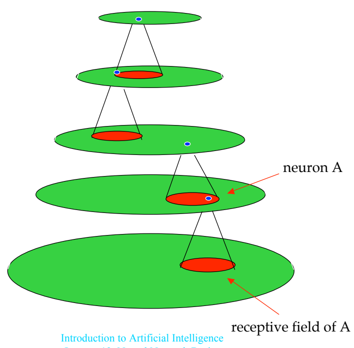
"Data Flow Diagram" of Visual Areas in Macaque Brain

Blue: motion perception pathway

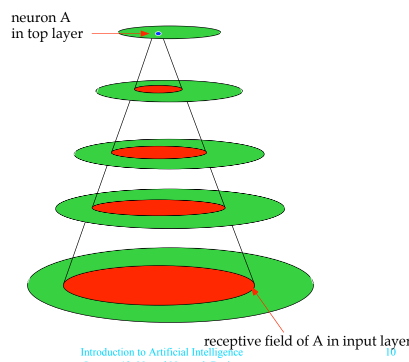
Green: object recognition pathway



Receptive Fields in Hierarchical Neural Networks



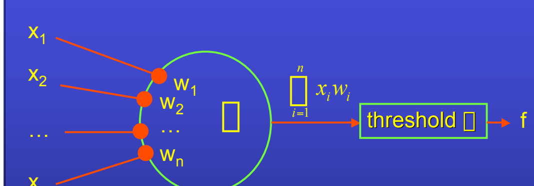
Receptive Fields in Hierarchical Neural Networks



How do NNs and ANNs work?

- NNs are able to **learn** by **adapting their connectivity patterns** so that the organism improves its behavior in terms of reaching certain (evolutionary) goals.
- The strength of a connection, or whether it is excitatory or inhibitory, depends on the state of a receiving neuron's **synapses**.
- The NN achieves **learning** by appropriately adapting the states of its synapses.

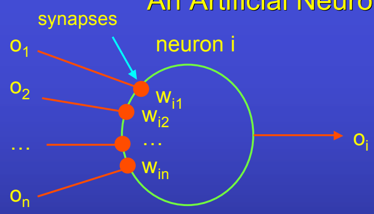
Do You Remember Threshold Logic Units?



$$f(x_1, x_2, \ldots, x_n) = 1, \quad \text{if } \sum_{i=1}^{n} x_i w_i \ge \theta$$

$$= 0, \quad \text{otherwise}$$

An Artificial Neuron



net input signal $\quad \text{net}_i(t) = \sum_{j=1}^{n} w_{ij}(t) o_j(t)$

activation $\quad a_i(t) = F_i(a_i(t-1), \text{net}_i(t))$

output $\quad o_i(t) = f_i(a_i(t))$

The Net Input Signal

The **net input signal** is the sum of all inputs after passing the synapses:

$$\text{net}_i(t) = \sum_{j=1}^{n} w_{ij}(t) o_j(t)$$

This can be viewed as computing the **inner product** of the vectors $\mathbf{w}_i$ and $\mathbf{o}$:

$$\text{net}_i(t) = \| w_i(t) \| \cdot \| o(t) \| \cdot \cos \alpha ,$$

where α is the **angle** between the two vectors.

The Net Input Signal

In most ANNs, the activation of a neuron is simply defined to **equal its net input signal**:

$$a_i(t) = \text{net}_i(i)$$

Then, the neuron's **activation function** (or output function) f_i is applied directly to $\text{net}_i(t)$:

$$o_i(t) = f_i(\text{net}_i(t))$$

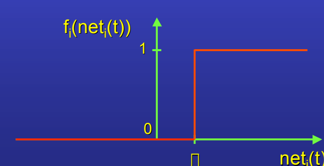
What do such functions f_i look like?

The Activation Function

One possible choice is a **threshold function**, just like the one we used in the Threshold Logic Units:

$$f_i(\text{net}_i(t)) = 1, \quad \text{if} \quad \text{net}_i(t) \geq \theta$$
$$= 0, \quad \text{otherwise}$$

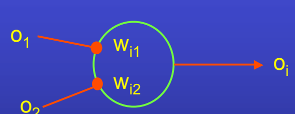
The graph of this function looks like this:



Capabilities of Threshold Neurons

What can threshold neurons do for us?

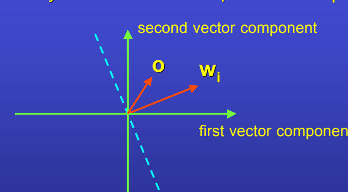
To keep things simple, let us consider such a neuron with two inputs:



The computation of this neuron can be described as the inner product of the **two-dimensional vectors o** and $\mathbf{w}_i$, followed by a threshold operation.

Capabilities of Threshold Neurons

Let us assume that the threshold $\theta = 0$ and illustrate the function computed by the neuron for sample vectors $\mathbf{w}_i$ and $\mathbf{o}$:



Since the inner product is positive for $-90° < \alpha < 90°$, in this example the neuron's output is 1 for any input vector **o** to the right of or on the dotted line, and 0 for any other input vector.

Capabilities of Threshold Neurons

By choosing appropriate weights w_i and threshold θ we can place the **line** dividing the input space into regions of output 0 and output 1in **any position and orientation**.

Therefore, our threshold neuron can realize any **linearly separable** function $\mathbf{R}^n \rightarrow \{0, 1\}$.

Although we only looked at two-dimensional input, our findings apply to **any dimensionality n**.

For example, for n = 3, our neuron can realize any function that divides the three-dimensional input space along a two-dimension plane.

Capabilities of Threshold Neurons

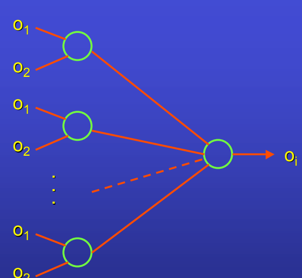
What do we do if we need a more complex function?

Just like Threshold Logic Units, we can also **combine** multiple artificial neurons to form networks with increased capabilities.

For example, we can build a two-layer network with any number of neurons in the first layer giving input to a single neuron in the second layer.

The neuron in the second layer could, for example, implement an AND function.

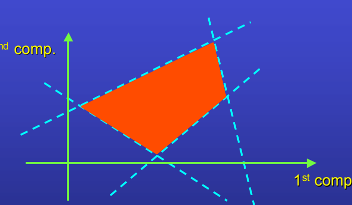
Capabilities of Threshold Neurons



What kind of function can such a network realize?

Capabilities of Threshold Neurons

Assume that the dotted lines in the diagram represent the input-dividing lines implemented by the neurons in the first layer:



Then, for example, the second-layer neuron could output 1 if the input is within a **polygon**, and 0 otherwise.

Capabilities of Threshold Neurons

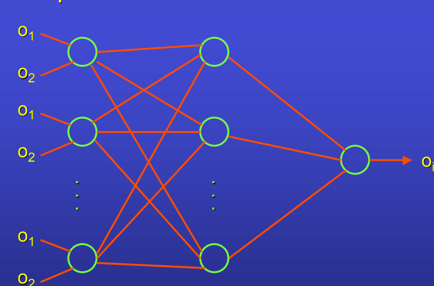
However, we still may want to implement functions that are more complex than that.

An obvious idea is to extend our network even further.

Let us build a network that has **three layers**, with arbitrary numbers of neurons in the first and second layers and one neuron in the third layer.

The first and second layers are **completely connected**, that is, each neuron in the first layer sends its output to every neuron in the second layer.

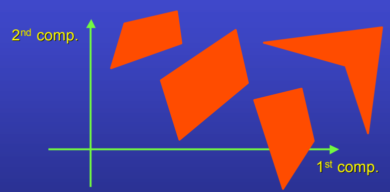
Capabilities of Threshold Neurons



What type of function can a three-layer network realize?

Capabilities of Threshold Neurons

Assume that the polygons in the diagram indicate the input regions for which each of the second-layer neurons yields output 1:



Then, for example, the third-layer neuron could output 1 if the input is within **any of the polygons**, and 0 otherwise.

Capabilities of Threshold Neurons

The more neurons there are in the first layer, the more vertices can the polygons have.

With a sufficient number of first-layer neurons, the polygons can approximate **any** given shape.

The more neurons there are in the second layer, the more of these polygons can be combined to form the output function of the network.

With a sufficient number of neurons and appropriate weight vectors $\mathbf{w}_i$, a three-layer network of threshold neurons can realize **any (!)** function $\mathbf{R}^n \to \{0, 1\}$.

Terminology

Usually, we draw neural networks in such a way that the input enters at the bottom and the output is generated at the top.

Arrows indicate the direction of data flow.

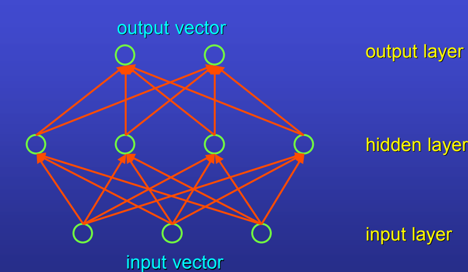
The first layer, termed **input layer**, just contains the input vector and does not perform any computations.

The second layer, termed **hidden layer**, receives input from the input layer and sends its output to the **output layer**.

After applying their activation function, the neurons in the output layer contain the output vector.

Terminology

Example: Network function f: $\mathbf{R}^3 \to \{0, 1\}^2$



Linear Neurons

Obviously, the fact that threshold units can only output the values 0 and 1 restricts their applicability to certain problems.

We can overcome this limitation by eliminating the threshold and simply turning f_i into the **identity function** so that we get:

$$o_i(t) = \text{net}_i(t)$$

With this kind of neuron, we can build networks with m input neurons and n output neurons that compute a function f: $R^m \to R^n$.

Linear Neurons

Linear neurons are quite popular and useful for applications such as interpolation.

However, they have a serious limitation: Each neuron computes a linear function, and therefore the overall network function f: $R^m \to R^n$ is also **linear**.

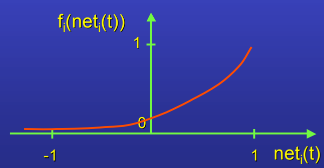
This means that if an input vector x results in an output vector y, then for any factor λ the input $\lambda \cdot x$ will result in the output $\lambda \cdot y$.

Obviously, many interesting functions cannot be realized by networks of linear neurons.

Gaussian Neurons

Another type of neurons overcomes this problem by using a **Gaussian** activation function:

$$f_i(\text{net}_i(t)) = e^{\frac{\text{net}_i(t)-1}{\sigma^2}}$$



Gaussian Neurons

Gaussian neurons are able to realize **non-linear** functions.

Therefore, networks of Gaussian units are in principle unrestricted with regard to the functions that they can realize.

The drawback of Gaussian neurons is that we have to make sure that their net input does not exceed 1.

This adds some difficulty to the learning in Gaussian networks.

Sigmoidal Neurons

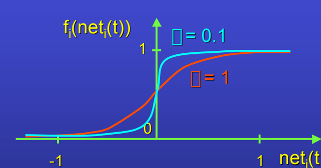
Sigmoidal neurons accept any vectors of real numbers as input, and they output a real number between 0 and 1.

Sigmoidal neurons are the most common type of artificial neuron, especially in learning networks.

A network of sigmoidal units with m input neurons and n output neurons realizes a network function $f: R^m \to (0,1)^n$

Sigmoidal Neurons

$$f_i(\text{net}_i(t)) = \frac{1}{1+e^{-(\text{net}_i(t)-\theta)/\tau}}$$



The parameter τ controls the slope of the sigmoid function, while the parameter θ controls the horizontal offset of the function in a way similar to the threshold neurons.

Supervised Learning in ANNs

In supervised learning, we train an ANN with a set of vector pairs, so-called **exemplars**.

Each pair (**x**, **y**) consists of an input vector **x** and a corresponding output vector **y**.

Whenever the network receives input **x**, we would like it to provide output **y**.

The exemplars thus describe the function that we want to "teach" our network.

Besides **learning** the exemplars, we would like our network to **generalize**, that is, give plausible output for inputs that the network had not been trained with.

Supervised Learning in ANNs

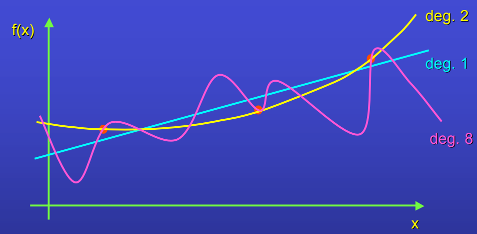
There is a tradeoff between a network's ability to precisely learn the given exemplars and its ability to generalize (i.e., inter- and extrapolate).

This problem is similar to **fitting a function** to a given set of data points.

Let us assume that you want to find a fitting function f:**R**→ **R** for a set of three data points.

You try to do this with polynomials of degree one (a straight line), two, and five.

Supervised Learning in ANNs



Obviously, the polynomial of degree 2 provides the most plausible fit.

Supervised Learning in ANNs

The same principle applies to ANNs:

- If an ANN has **too few** neurons, it may not have enough degrees of freedom to precisely approximate the desired function.
- If an ANN has **too many** neurons, it will learn the exemplars perfectly, but its additional degrees of freedom may cause it to show implausible behavior for untrained inputs; it then presents poor ability of generalization.

Unfortunately, there are **no known equations** that could tell you the optimal size of your network for a given application; you always have to experiment.

The Backpropagation Network

The backpropagation network (BPN) is the most popular type of ANN for applications such as classification or function approximation.

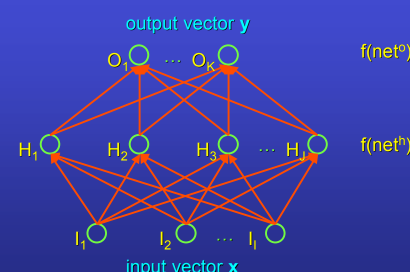
Like any other network using supervised learning, the BPN is not biologically plausible.

The structure of the network is identical to the one we discussed before:

- Three (sometimes more) layers of neurons,
- Only feedforward processing: input layer □ hidden layer □ output layer,
- Sigmoid activation functions

The Backpropagation Network

BPN units and activation functions:



Learning in the BPN

Before the learning process starts, all weights (synapses) in the network are **initialized** with pseudorandom numbers.

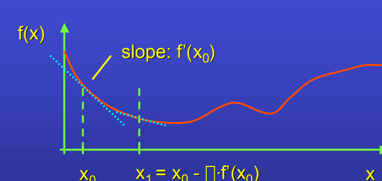
We also have to provide a set of **training patterns** (exemplars). They can be described as a set of ordered vector pairs $\{(x_1, y_1), (x_2, y_2), \ldots, (x_P, y_P)\}$.

Then we can start the backpropagation learning algorithm.

This algorithm iteratively minimizes the network's error by **finding the gradient** of the error surface in weight-space and **adjusting the weights** in the opposite direction (gradient-descent technique).

Learning in the BPN

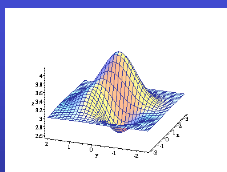
Gradient-descent example: Finding the absolute minimum of a one-dimensional error function f(x):

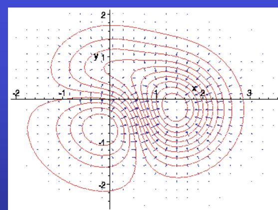


Repeat this iteratively until for some x_i, $f'(x_i)$ is sufficiently close to 0.

Learning in the BPN

Gradients of two-dimensional functions:





The two-dimensional function in the left diagram is represented by contour lines in the right diagram, where arrows indicate the gradient of the function at different locations. Obviously, the gradient is always pointing in the direction of the steepest increase of the function. In order to find the function's minimum, we should always move against the gradient.

Learning in the BPN

In the BPN, learning is performed as follows:

1. Randomly select a vector pair ($\mathbf{x}_p$, $\mathbf{y}_p$) from the training set and call it (**x**, **y**).
2. Use **x** as input to the BPN and successively compute the outputs of all neurons in the network (bottom-up) until you get the network output **o**.
3. Compute the error δ^o_{pk}, for the pattern p across all K output layer units by using the formula:

$$\delta^o_{pk} = (y_k - o_k) f'(net^o_k)$$

Learning in the BPN

4. Compute the error δ^h_{pj}, for all J hidden layer units by using the formula:

$$\delta^h_{pj} = f'(net^h_k) \sum_{k=1}^{K} \delta^o_{pk} w_{kj}$$

5. Update the connection-weight values to the hidden layer by using the following equation:

$$w_{ji}(t+1) = w_{ji}(t) + \eta \delta^h_{pj} x_i$$

Learning in the BPN

6. Update the connection-weight values to the output layer by using the following equation:

$$w_{kj}(t+1) = w_{kj}(t) + \eta \delta^o_{pk} f(net^h_j)$$

Repeat steps 1 to 6 for all vector pairs in the training set; this is called a training **epoch**.

Run as many epochs as required to reduce the network error E to fall below a **threshold** ε:

$$E = \sum_{p=1}^{P} \sum_{k=1}^{K} (\delta^o_{pk})^2$$

Learning in the BPN

The only thing that we need to know before we can start our network is the **derivative** of our sigmoid function, for example, $f'(net_k)$ for the output neurons:

$$f(\text{net}_k) = \frac{1}{1 + e^{-\text{net}_k}}$$

$$f'(\text{net}_k) = \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = o_k(1 - o_k)$$

Learning in the BPN

Now our BPN is ready to go!

If we choose the type and number of neurons in our network appropriately, after training the network should show the following behavior:

- If we input any of the training vectors, the network should yield the expected output vector (with some margin of error).
- If we input a vector that the network has never "seen" before, it should be able to generalize and yield a plausible output vector based on its knowledge about similar input vectors.

Backpropagation Network Variants

The standard BPN network is well-suited for learning **static functions**, that is, functions whose output depends only on the current input.

For many applications, however, we need functions whose output changes depending on **previous** inputs (for example, think of a deterministic finite automaton).

Obviously, pure feedforward networks are unable to achieve such a computation.

Only **recurrent neural networks** (RNNs) can overcome this problem.

A well-known recurrent version of the BPN is the **Elman** Network.

The Elman Network

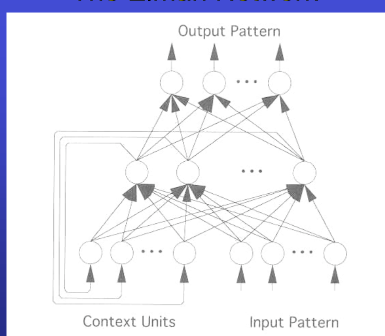
In comparison to the BPN, the Elman Network has an extra set of input units, so-called **context units**.

These neurons do not receive input from outside the network, but from the network's **hidden layer** in a one-to-one fashion.

Basically, the context units contain a copy of the network's **internal state** at the previous time step.

The context units feed into the hidden layer just like the other input units do, so the network is able to compute a function that not only depends on the current input, but also on the network's internal state (which is determined by **previous inputs**).

The Elman Network



The Counterpropagation Network

Another variant of the BPN is the **counterpropagation network** (CPN).

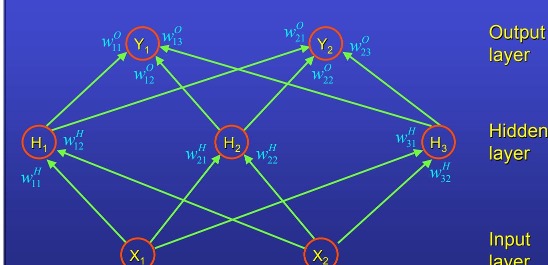
Although this network uses **linear** neurons, it can learn **nonlinear** functions by means of a hidden layer of **competitive units**.

Moreover, the network is able to learn a function and its **inverse** at the same time.

However, to simplify things, we will only consider the **feedforward** mechanism of the CPN.

The Counterpropagation Network

A simple CPN network with two input neurons, three hidden neurons, and two output neurons can be described as follows:



The Counterpropagation Network

The CPN learning process (general form for n input units and m output units):

1. Randomly select a vector pair (x, y) from the training set.
2. Normalize (shrink/expand to "length" 1) the input vector x by dividing every component of x by the magnitude ||x||, where

$$\| x \| = \sqrt{\sum_{j=1}^{n} x_j^2}$$

The Counterpropagation Network

3. Initialize the input neurons with the normalized vector and compute the activation of the **linear** hidden-layer units.
4. In the hidden (competitive) layer, determine the unit W with the largest activation (the winner).
5. Adjust the connection weights between W and all N input-layer units according to the formula:

$$w_{Wn}^{H}(t+1) = w_{Wn}^{H}(t) + \eta(x_n - w_{Wn}^{H}(t))$$

6. Repeat steps 1 to 5 until all training patterns have been processed once.

The Counterpropagation Network

7. Repeat step 6 until each input pattern is consistently associated with the same competitive unit.
8. Select the first vector pair in the training set (the current pattern).
9. Repeat steps 2 to 4 (normalization, competition) for the current pattern.
10. Adjust the connection weights between the winning hidden-layer unit and all M output layer units according to the equation:

$$w_{mW}^{O}(t+1) = w_{mW}^{O}(t) + \eta(y_m - w_{mW}^{O}(t))$$

The Counterpropagation Network

11. Repeat steps 9 and 10 for each vector pair in the training set.
12. Repeat steps 8 through 11 until the difference between the desired and the actual output falls below an acceptable threshold.

The Counterpropagation Network

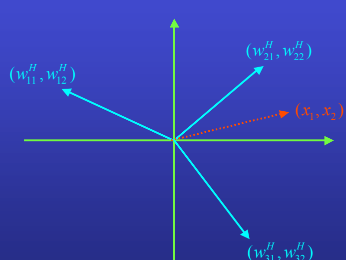
Because the input is two-dimensional, each unit in the hidden layer has two weights (one for each input connection).

Therefore, input to the network as well as weights of hidden-layer units can be represented and visualized by two-dimensional vectors.

For the current network, all weights in the hidden layer can be completely described by three 2D vectors.

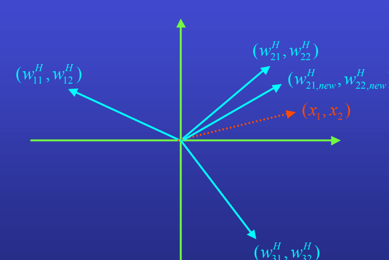
The Counterpropagation Network

This diagram shows a sample state of the hidden layer and a sample input to the network:



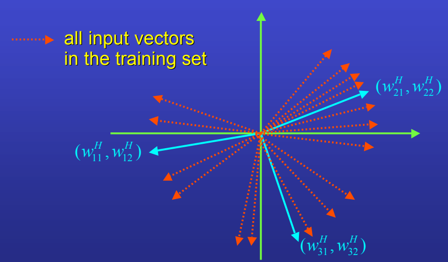
The Counterpropagation Network

In this example, hidden-layer neuron H_2 wins and, according to the learning rule, is moved closer towards the current input vector.



The Counterpropagation Network

After doing this through many epochs and slowly reducing the adaptation step size , each hidden-layer unit will win for a subset of inputs, and the angle of its weight vector will be in the center of gravity of the angles of these inputs.



The Counterpropagation Network

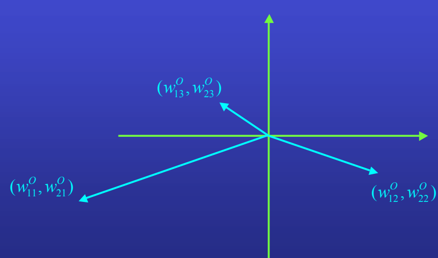
After the **first phase** of the training, each hidden-layer neuron is associated with a subset of input vectors.

The training process minimized the average angle difference between the weight vectors and their associated input vectors.

In the **second phase** of the training, we adjust the weights in the network's output layer in such a way that, for any winning hidden-layer unit, the network's output is as close as possible to the desired output for the winning unit's associated input vectors.

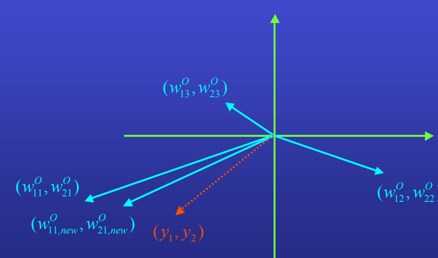
The Counterpropagation Network

Because there are two output neurons, the weights in the output layer that receive input from the same hidden-layer unit can also be described by 2D vectors:



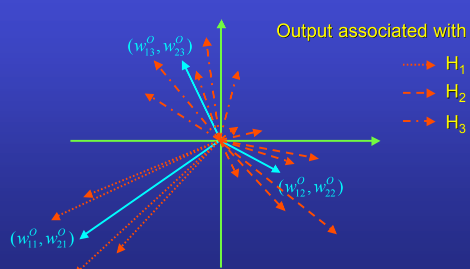
The Counterpropagation Network

For each input vector, the output-layer weights that are connected to the winning hidden-layer unit are made more similar to the desired output vector:



The Counterpropagation Network

The training proceeds with decreasing step size , and after its termination, the weight vectors are in the center of gravity of their associated output vectors:



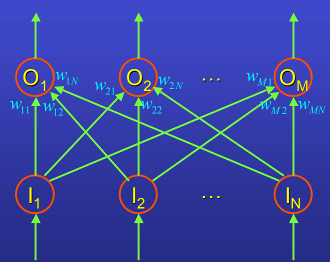
The Counterpropagation Network

Notice:

- In the first training phase, if a hidden-layer unit **does not win** for a long period of time, its weights should be set to **random** values to give that unit a chance to win subsequently.
- There is **no need for normalizing** the training output vectors.
- After the training has finished, the network maps the training vectors onto output vectors that are **close** to the desired ones.
- The **more** hidden units, the **better** the mapping.
- Thanks to the competitive neurons in the hidden layer, the linear neurons can realize **nonlinear** mappings.

Interpolative Associative Memory

If we just want to realize a linear function, a simple two-layer network suffices:



Interpolative Associative Memory

Sometimes it is possible to obtain a training set with **orthonormal** (that is, normalized and pairwise orthogonal) input vectors.

In that case, our two-layer network with linear neurons can solve its task perfectly and **does not even require training**.

We call such a network an **interpolative associative memory**.

You may ask: **How does it work?**

Interpolative Associative Memory

Well, if you look at the network's output function

$$o_m = \sum_{n=1}^{N} w_{mn} i_n \quad \text{for} \quad m = 1, \ldots, M$$

you will find that this is just like a matrix multiplication:

$$\begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_M \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{21} & w_{22} & & w_{2N} \\ \vdots & & \ddots & \vdots \\ w_{M1} & w_{M2} & \cdots & w_{MN} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_N \end{bmatrix} \quad \text{or} \quad \mathbf{o} = W\mathbf{i}$$

Interpolative Associative Memory

With an orthonormal set of exemplar input vectors (and any associated output vectors) we can simply **calculate a weight matrix** that realizes the desired function and does not need any training procedure.

For exemplars (x_1, y_1), (x_2, y_2), …, (x_P, y_P) we obtain the following weight matrix W:

$$W = \sum_{p=1}^{P} y_p x_p^t$$

Note that an N-dimensional vector space cannot have a set of more than N orthonormal vectors!

Interpolative Associative Memory

Example:

Assume that we want to build an interpolative memory with three input neurons and three output neurons.

We have the following three exemplars (desired input-output pairs):

$$\left\{ \left(\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -3 \\ -3 \\ -3 \end{bmatrix} \right), \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix} \right), \left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 5 \\ 2 \\ 8 \end{bmatrix} \right) \right\}$$

Interpolative Associative Memory

Then

$$W = \begin{bmatrix} -3 \\ -3 \\ -3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 5 \\ 2 \\ 8 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 & -3 & 0 \\ 0 & -3 & 0 \\ 0 & -3 & 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 \\ -2 & 0 & 0 \\ -3 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 0 & 0 & 2 \\ 0 & 0 & 8 \end{bmatrix} = \begin{bmatrix} -1 & -3 & 5 \\ -2 & -3 & 2 \\ -3 & -3 & 8 \end{bmatrix}$$

If you set the weights w_{mn} to these values, the network will realize the desired function.

Interpolative Associative Memory

So if you want to implement a **linear** function $R^N \to R^M$ and can provide exemplars with **orthonormal** input vectors, then an **interpolative associative memory** is the best solution.

It does **not** require any **training** procedure, realizes **perfect matching** of the exemplars, and performs **plausible interpolation** for new input vectors.

Of course, this interpolation is **linear**.