Introduction to Artificial Intelligence

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Neural Networks in AI

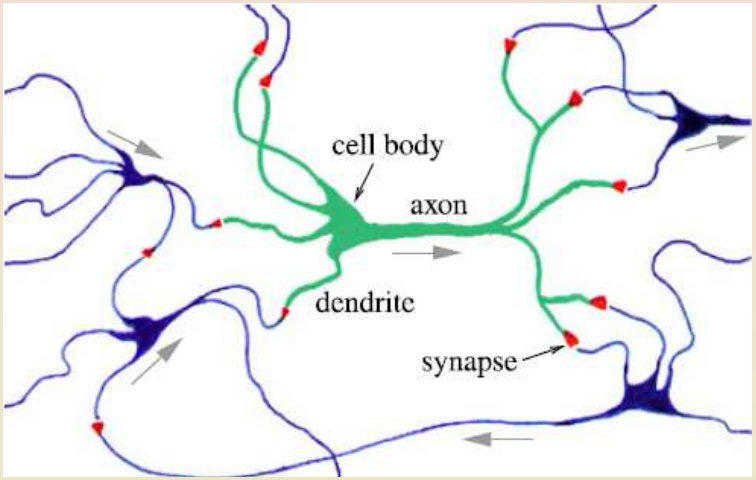
- Neural networks are networks of nerve cells in the brain.
- The human brain has about 100 billion nerve cells.
- Nerve cells and their connections, are responsible for awareness, associations, thoughts, consciousness, and the ability to learn.
- McCulloch and Pitts introduced neural networks in AI in 1943 by formulating a mathematical model of the neuron.
- The field of modeling and simulation of neural networks may be thought of as the bionics branch within AI.
- Nearly all areas of AI attempt to recreate cognitive processes, such as in logic or in probabilistic reasoning, but the tools used for modeling - mathematics, programming languages, and digital computers - have very little in common with the human brain.
- With artificial neural networks, one starts from knowledge about the function of natural neural networks and attempts to model, simulate, and even reconstruct them in hardware.

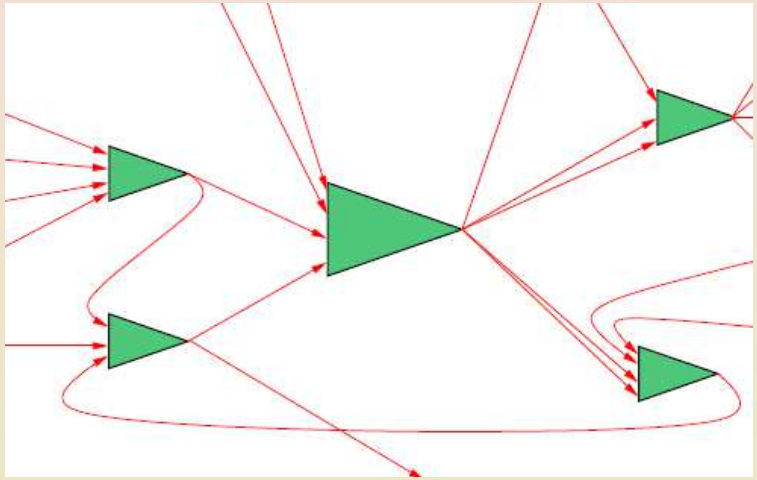
Subsection 1

From Biology to Simulation

Neuron Structure and Function

- Each neuron has the following structure and function:





  - Besides the cell body, the neuron has an axon, which can make local connections to other neurons over the dendrites.
  - The cell body can store small electrical charges. It is loaded by incoming electrical impulses from other neurons. If the voltage exceeds a certain threshold, the neuron will fire, i.e., unload its store by sending a spike over the axon and the synapses to many other neurons.
- Each of $\sim 10^{11}$ neurons is connected to $\sim$ 1000 to 10,000 other neurons, (10^{14} connections), so it is very likely that we will never be capable of understanding the diagram of the brain completely.

Adaptability: Neurotransmitters

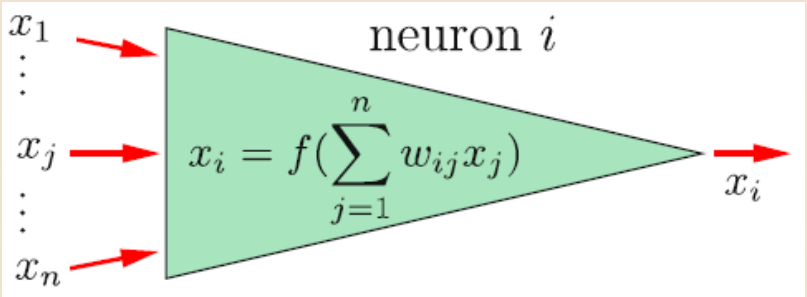
- The structure of the brain is adaptive.
- It continuously adapts according to the individual's activities and environmental influences.
- The central role here is played by the synapses, which create the connection between neurons.
- These connections are not perfectly conductively connected, rather there is a small gap, which the electrons cannot directly jump over.
- This gap is filled with chemical substances, neurotransmitters, which can be ionized by an applied voltage and then transport a charge over the gap.
- The function of the brain reacts very sensitively to changes of this synaptic connection, e.g., through the influence of alcohol or drugs.
- So it is not the actual neurons, which are adaptive, rather the connections between them, that is, the synapses.

Adaptability: Synapses

- A synapse is made stronger (higher conductivity) by however much more electrical current it must carry.
- All neurons in the brain work asynchronously and in parallel, but, compared to a computer, at very low speed (by a factor of 10^6).
- This disadvantage is more than compensated for in many complex cognitive tasks, such as image recognition, by the very high degree of parallel processing in the network of nerve cells.
- The connection to the outside world comes about through sensor neurons, e.g., on the retina in the eyes.
- It is still unclear how these principles make intelligent behavior possible.
- We use simulations via a simple mathematical model in attempting to explain how cognitive tasks (e.g., pattern recognition) become possible.

The Mathematical Model

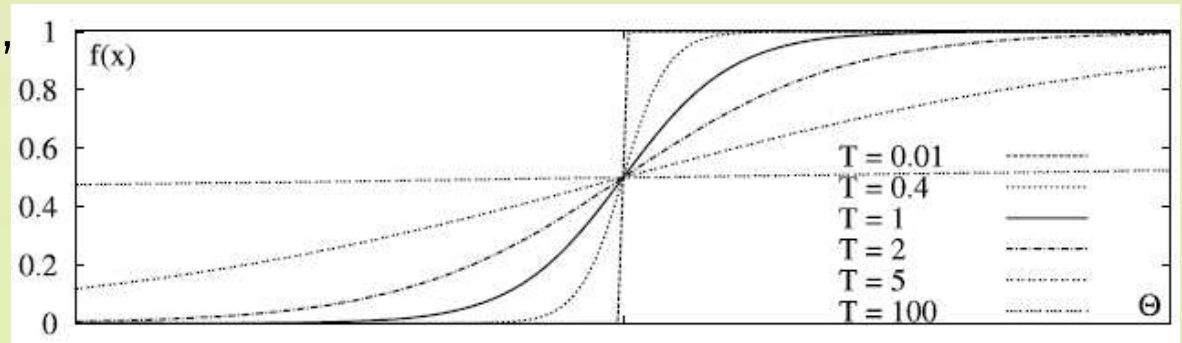
- We use a discrete time scale.
- The neuron i carries out the following calculation in a time step:
  - The "loading" of the activation potential is modeled by a weighted sum of incoming signals: $\sum_{j=1}^{n} w_{ij} x_j$.



  - Then an activation function f is applied $x_i = f(\sum_{j=1}^{n} w_{ij} x_j)$;
  - The result is passed on to the neighboring neurons as output over the synaptic weights.
- For the activation function there are a number of possibilities.
- The simplest choice is $f(x) = x$, i.e., the neuron calculates only the weighted sum of the input values and passes this on.
- This choice frequently leads to convergence problems because the function $f(x) = x$ is unbounded.

Step and Sigmoid Functions

- Often, the threshold function (Heaviside step function)
  $H_\Theta(x) = \begin{cases} 0, & \text{if } x < \Theta \\ 1, & \text{otherwise} \end{cases}$ is used.
- Then the neuron output is $x_i = \begin{cases} 0, & \text{if } \sum_{j=1}^{n} w_{ij} x_j < \Theta \\ 1, & \text{otherwise} \end{cases}$
- This formula is identical to that of a perceptron with the threshold Θ.
- The input neurons $1, \ldots, n$ have only the function of variables which pass on their externally set values $x_1, \ldots, x_n$ unchanged.
- The step function is quite sensible for binary neurons , but for continuous neurons the step function can be smoothed out by a sigmoid function, e.g., $f(x) = \frac{1}{1 + e^{-\frac{x-\Theta}{T}}}$:



The Hebb Rule

- Modeling learning is central to the theory of neural networks.
- One possibility of learning consists of strengthening a synapse according to how many electrical impulses it must transmit.
- This principle is known as the **Hebb rule**:

> If there is a connection w_{ij} between neuron j and neuron i and repeated signals are sent from neuron j to neuron i, which results in both neurons being simultaneously active, then the weight w_{ij} is reinforced.

- A possible formula for the weight change Δw_{ij} is

$$\Delta w_{ij} = \eta x_i x_j,$$

with the constant η (**learning rate**) determining the size of the individual learning steps.

- There are many modifications of this rule, which result in different types of networks or learning algorithms.

Subsection 2

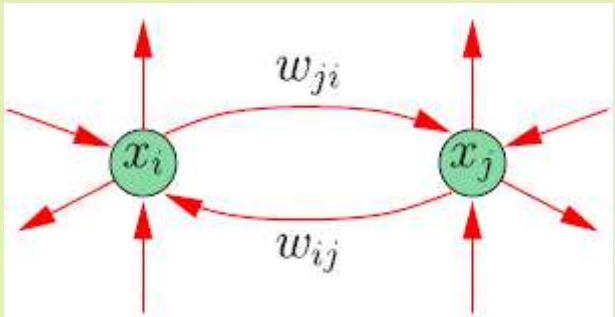
Hopfield Networks

Hopfield's Idea

- According to the Hebb rule, the weights of neurons with values between zero and one can only grow with time.
- It is not possible for a neuron to weaken or even die with this rule.
- Weakening can be modeled by a decay constant which weakens an unused weight by a constant factor per time step, such as 0.99.
- The model presented by Hopfield in 1982 uses binary neurons, but with the two values -1 for inactive and 1 for active.
- Using the Hebb rule a positive contribution to the weight is obtained whenever two neurons are simultaneously active, but Δw_{ij} is negative, if only one of the two is active.
- Hopfield networks are a beautiful and visualizable example of **autoassociative memory**, in which
  - patterns can be stored;
  - to call up a saved pattern, it is sufficient to provide a similar pattern.

Formalizing a Hopfield Network

- A classic application is handwriting recognition.
- In the learning phase of a Hopfield network, N binary coded patterns, saved in the vectors $\mathbf{q}^1, \ldots, \mathbf{q}^N$, are supposed to be learned.
- Each component $\mathbf{q}_i^j \in \{-1, 1\}$ of such a vector $\mathbf{q}^j$ represents a pixel of a pattern.
- For vectors consisting of n pixels, a neural network with n neurons is used, one for each pixel position.
- The neurons are fully connected with the restriction that the weight matrix is symmetric and all diagonal elements w_{ii} are zero.
- This means that there is no connection between a neuron and itself.
- The fully connected network includes complex feedback loops, which are called **recurrences**.



Dynamics of a Hopfield Network

- N patterns can be learned by simply calculating all weights: $w_{ij} = \frac{1}{n} \sum_{k=1}^{N} q_i^k q_j^k$.
- This formula points out an interesting relationship to the Hebb rule.
  - Each pattern in which the pixels i and j have the same value makes a positive contribution to the weight w_{ij}.
  - Each other pattern makes a negative contribution.
- Once all the patterns have been stored, the network can be used for pattern recognition.
- We give the network a new pattern $\mathbf{x}$ and update the activations of all neurons according to the rule $x_i = \begin{cases} -1, & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0 \\ 1, & \text{otherwise} \end{cases}$, until the network becomes stable, that is, until no more activations change.

The Hopfield Network Algorithm

HopfieldAssociator(q)

Initialize all neurons: $\mathbf{x} = \mathbf{q}$

Repeat

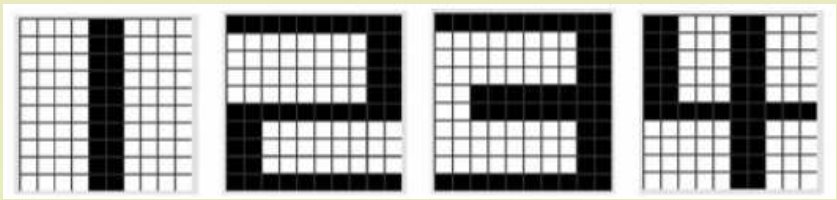
$i = \text{Random}(1, n)$

Update neuron i according to $x_i = \begin{cases} -1, & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0 \\ 1, & \text{otherwise} \end{cases}$

Until x converges

Return(x)

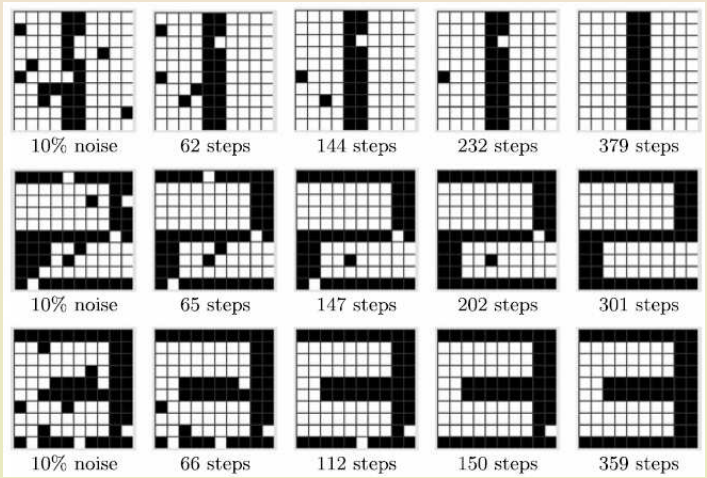
- Example: We apply the algorithm to a pattern recognition to recognize digits in a 10×10 pixel field.



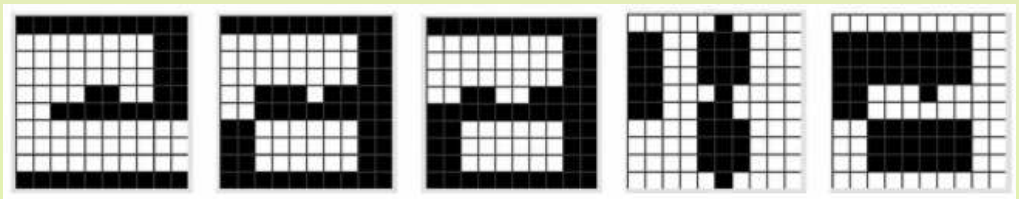
- The network has 100 neurons with a total of $\frac{100 \cdot 99}{2} = 4950$ weights.
- First the patterns of the digits 1, 2, 3, 4 above are trained, i.e., the weights are calculated using $w_{ij} = \frac{1}{n} \sum_{k=1}^{N} q_i^k q_j^k$.

The Digit Recognition Network

- Then we put in a pattern with added noise and let the Hopfield dynamics run until convergence.
- In the figure, five snapshots of the networks development are shown during recognition.

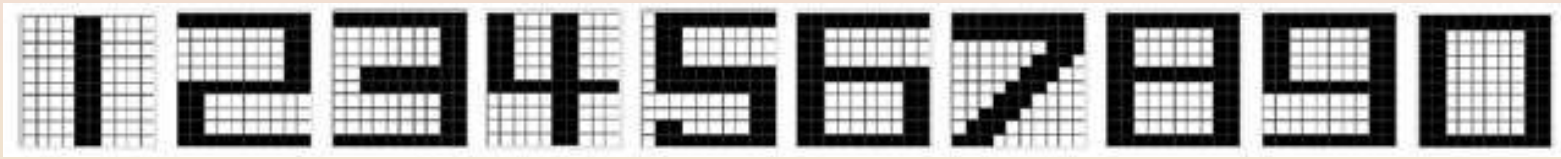


- At 10% noise all four learned patterns are very reliably recognized.
- Above about 20% noise the algorithm frequently converges to other learned patterns or even to patterns which were not learned.

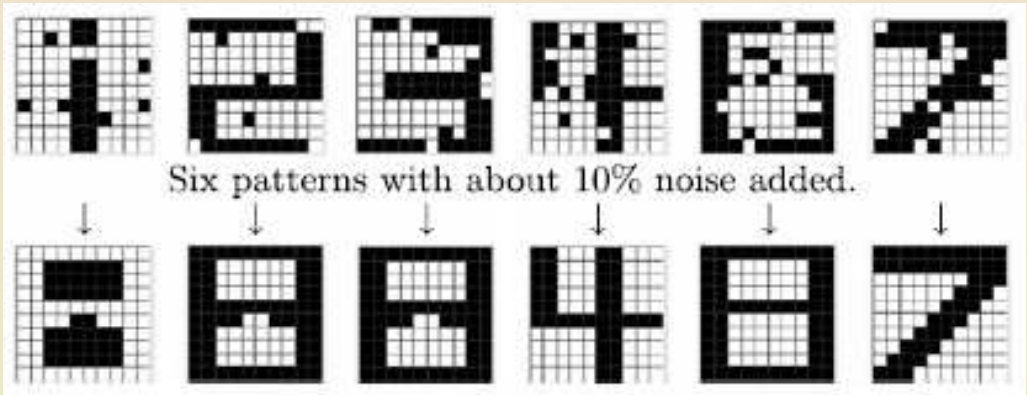


Expanding to 10 Digits

- Now we save the digits 0 to 9 in the same network:



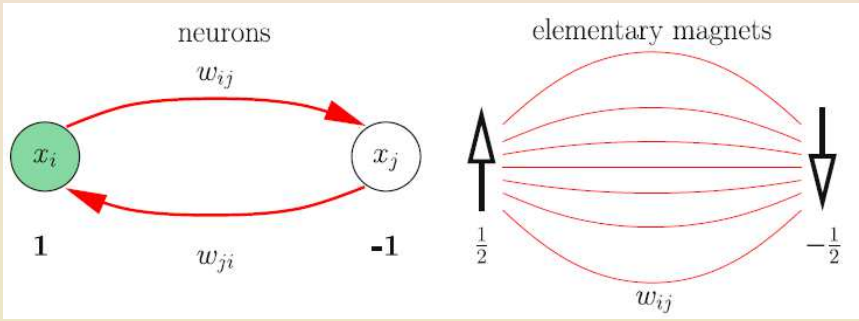
- We test the network again with patterns having 10% inverted pixels:



- The Hopfield iteration often does not converge to the most similar learned state even for only 10% noise.
- Evidently the network can securely save and recognize four patterns, but for ten patterns its memory capacity is exceeded.
- To understand this better, we turn to the theory of the network.

Hopfield Networks as Physical Systems I

- In 1982, John Hopfield showed that this model is formally equivalent to a physical model of magnetism.
- Small elementary magnets, called **spins**, mutually influence each other over their magnetic fields.



- If we observe two such spins i and j, they interact over a constant w_{ij} and the total energy of the system is $E = -\frac{1}{2}\sum_{i,j} w_{ij}x_i x_j$.
- It is true that
  - $w_{ii} = 0$ because particles have no self-interaction;
  - $w_{ij} = w_{ji}$ because physical interactions are symmetric.
- A physical system in equilibrium takes on a (stable) state of minimal energy and thus minimizes $E(\mathbf{x}, \mathbf{y})$.
- If such a system is brought into an arbitrary state, then it moves toward a state of minimal energy.

Hopfield Networks as Physical Systems II

- The Hopfield dynamics $x_i = \begin{cases} -1, & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0 \\ 1, & \text{otherwise} \end{cases}$ updates the state in each iteration such that, of the two states -1 and 1, the one with smaller total energy is taken on.
- Thus, they correspond exactly to this principle.
- The contribution of the neuron i to total energy is $-\frac{1}{2} x_i \sum_{j \neq i} w_{ij} x_j$.
- If $\sum_{j \neq i} w_{ij} x_j < 0$, then $x_i = -1$ results in a negative contribution to the total energy, and $x_i = 1$ in a positive contribution. For $x_i = -1$, the network takes on a state of lower energy than for $x_i = 1$.
- Analogously, we can assert that in the case of $\sum_{j \neq i} w_{ij} x_j \geq 0$, it must be true that $x_i = 1$.
- If each individual iteration results in a reduction of the energy, then, since there are only finitely many states, the network eventually reaches a state of minimal energy.
- What do these minima of the energy function mean?

Transition to a Chaotic Phase

- In the pattern recognition experiment, in the case of few learned patterns the system converges to one of the learned patterns.
- The learned patterns represent minima of the energy function.
- If too many patterns are learned, then the system converges to minima which do not correspond to learned patterns.
- A transition from an ordered dynamics into a chaotic one occurs.
- There is a phase transition at a critical number of learned patterns. If the number of learned patterns exceeds this value, then the system changes from the ordered phase into the chaotic.
- In magnetic physics there is such a transition from the ferromagnetic mode (all elementary magnets try to orient themselves parallel) to a so-called spin glass (the spins interact chaotically).
- A more visualizable example of such a physical phase transition is the melting of an ice crystal. The crystal is in a high state of order because the H_2O molecules are strictly ordered. In liquid water, by contrast, the structure is dissolved and positions are more random.

Summary

- Through its biological plausibility, the clear mathematical model, and the impressive simulations in pattern recognition, the Hopfield model contributed to the establishment of neural networks and to the rise of neuroinformatics as an important branch of AI.
- Subsequently many further network models were developed.
  - Networks without back-couplings were investigated because their dynamics is significantly easier to understand than recurrent networks.
  - Attempts were made to improve the storage capacity of the networks.
- Even when guaranteed to converge, it is not certain whether it will do so to a learned state or get stuck at a local minimum.
  - The Boltzmann machine, with continuous activation values and a probabilistic update rule for its network dynamics, was developed as an attempt to solve this problem.
  - In "simulated annealing" a "temperature" parameter is used to vary the amount of random state changes and thus attempt to escape local minima and find a stable global minimum. Annealing is a process of treating metals with heat to make them stronger and more "stable".

Subsection 3

Neural Associative Memory

List Memory and Associative Memory

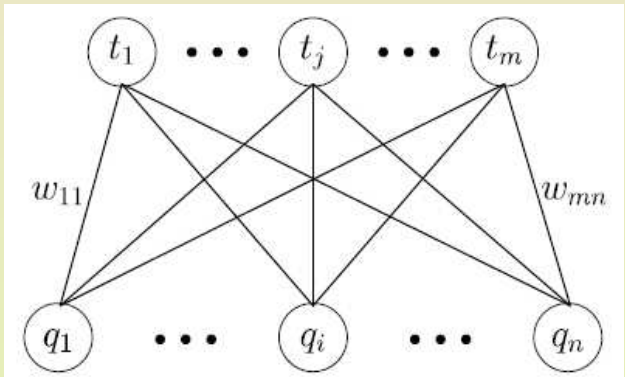
- A traditional list memory can in the simplest case be a text file in which strings of digits are saved line by line.
- If the file is sorted by line, then the search for an element can be done very quickly in logarithmic time, even for very large files.
- List memory can also be used to create mappings.
  - A telephone book is a mapping from the set of all entered names to the set of all telephone numbers.
  - Access control to a building using facial recognition is a similar task. However, the probability that the current photo matches the saved photo exactly is very small.
- In this case, what is needed is associative memory, which is capable of not only assigning the right name to the photo, but also to any of a potentially infinite set of "similar" photos.
- A function for finding similarity should be generated from a finite set of training data, namely the saved photos labeled with the names.
- A simple approach for this is the nearest neighbor method.

Implementing Associative Memory

- Learning consists simply of all of the photos being saved.
- To apply nearest neighbor, the photo most similar to the current one must be found in the database.
- This process, depending on the distance metric used, can require very long computation times and thus cannot be implemented in this form.
- Instead of a lazy algorithm, we prefer one which transfers the data into a function which then creates a very fast association.
- Finding a suitable distance metric presents a further problem.
  - We would like a person to be recognized even if the person's face appears in another place on the photo (translation), or if it is smaller, larger, or even rotated. The viewing angle and lighting might also vary.
  - Neural networks show their strengths, since without requiring the developer to think about a similarity metric, they deliver good results.
- The Hopfield model would be too difficult to use for two reasons:
  - It is only an auto-associative memory, that is, an approximately identical mapping which maps similar objects to the learned original.
  - The complex recurrent dynamics is often difficult to manage in practice.

The Kohonen Associative Memory Model

- The Kohonen associative memory model is based on linear algebra.
- It maps query vectors $\mathbf{x} \in \mathbb{R}^n$ to result vectors $\mathbf{y} \in \mathbb{R}^m$.
- We are looking for a matrix $\mathbf{W}$ which maps correctly a set of training data $T = \{(\mathbf{q}^1, \mathbf{t}^1), \ldots, (\mathbf{q}^N, \mathbf{t}^N)\}$.
- For $p = 1, \ldots, N$, $\mathbf{t}^p = \mathbf{W} \cdot \mathbf{q}^p$, or $t_i^p = \sum_{j=1}^{n} w_{ij} q_j^p$.
- To calculate the elements w_{ij}, we use the rule $w_{ij} = \sum_{p=1}^{N} q_j^p t_i^p$.
- These two linear equations can be viewed as a neural network with two layers:
  - $\mathbf{q}$ as the input layer;
  - $\mathbf{t}$ as the output layer.



- The neurons of the output layer have a linear activation function with learning rule $w_{ij} = \sum_{p=1}^{N} q_j^p t_i^p$, corresponding exactly to the Hebb rule.

Orthonormal Query Vectors in Training Data

Definition of Orthonormal Vectors

Two vectors $\mathbf{x}$ and $\mathbf{y}$ are called **orthonormal** if $\mathbf{x}^T \cdot \mathbf{y} = \begin{cases} 1, & \text{if } \mathbf{x} = \mathbf{y} \\ 0, & \text{otherwise} \end{cases}$.

Theorem

If all N query vectors $\mathbf{q}^p$ in the training data are orthonormal, then every vector $\mathbf{q}^p$ is mapped to the target vector $\mathbf{t}^p$ by multiplication with the matrix $w_{ij} = \sum_{p=1}^{N} q_j^p t_i^p$.

- We have:

$$\begin{aligned}
(\mathbf{W} \cdot \mathbf{q}^p)_i &= \sum_{j=1}^{n} w_{ij} q_j^p = \sum_{j=1}^{n} \sum_{r=1}^{N} q_j^r t_i^r q_j^p \\
&= \sum_{j=1}^{n} q_j^p q_j^p t_i^p + \sum_{r \neq p}^{N} \sum_{j=1}^{n} q_j^r q_j^p t_i^r \\
&= t_i^p \sum_{j=1}^{n} q_j^p q_j^p + \sum_{r \neq p}^{N} t_i^r \sum_{j=1}^{n} q_j^r q_j^p \\
&= t_i^p (\mathbf{q}^p)^T \mathbf{q}^p + t_i^r \sum_{r \neq p}^{N} (\mathbf{q}^r)^T \mathbf{q}^p = t_i^p.
\end{aligned}$$

- Orthonormality is too strong; it will be relaxed later.

Similarity and Adjustment of Responses

- Since linear mappings are continuous and injective, we know that the mapping from query vectors to target vectors preserves similarity.
- Similar queries are thus mapped to similar targets, but different queries are mapped to different targets.
  - If the network was trained to map faces to names and if the name Henry is assigned to a face, then we are sure that for the input of a similar face, an output similar to "Henry" will be produced, but "Henry" itself is guaranteed not to be calculated.
  - If the output was a string, then it could be "Genry" or "Gfnry".
- To arrive at the most similar learned case, Kohonen uses a binary coding for the output neuron.
- The calculated result of a query is rounded if its value is not 0 or 1.
- Even then we have no guarantee that we will hit the target vector.
- Alternatively, we could add a subsequent mapping of the calculated answer to the learned target vector with the smallest distance.

Weight Matrix and Invertibility

- We look at another approach to calculate the weight matrix $\mathbf{W}$.
- Think of all query vectors as columns of an $n \times N$ matrix $\mathbf{Q} = (\mathbf{q}^1, \ldots, \mathbf{q}^N)$.
- Analogously the target vectors as columns of an $m \times N$ matrix $\mathbf{T} = (\mathbf{t}^1, \ldots, \mathbf{t}^N)$.
- Now we get $\mathbf{T} = \mathbf{W} \cdot \mathbf{Q}$.
- We attempt to solve this equation for $\mathbf{W}$. Formally, we invert and obtain $\mathbf{W} = \mathbf{T} \cdot \mathbf{Q}^{-1}$.
- The requirement for this conversion is the invertibility of $\mathbf{Q}$.
- For this the matrix $\mathbf{Q}$:
  - must be square ($n = N$) and
  - must consist of linearly independent column vectors, i.e., the n query vectors $\mathbf{q}^1, \ldots, \mathbf{q}^n$ must all be linearly independent.
- This condition is inconvenient, but not as strict as orthonormality.

The Pseudoinverse

- A matrix $\mathbf{Q}$ is **invertible** if and only if there is a matrix $\mathbf{Q}^{-1}$ with the property $\mathbf{Q} \cdot \mathbf{Q}^{-1} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.
- If $\mathbf{Q}$ is not invertible (for example because $\mathbf{Q}$ is not square), then there is no matrix $\mathbf{Q}^{-1}$ with this property.
- There is, however, a matrix which comes close:

Definition of Pseudoinverse

Let $\mathbf{Q}$ be a real $n \times m$ matrix. An $m \times n$ matrix $\mathbf{Q}^+$ is a **pseudoinverse to $\mathbf{Q}$** if it minimizes $\|\mathbf{Q} \cdot \mathbf{Q}^+ - \mathbf{I}\|$, where $\|\mathbf{M}\|$ is the Euclidian norm.

- Using $\mathbf{W} = \mathbf{T} \cdot \mathbf{Q}^+$ we can calculate a weight matrix that minimizes crosstalk (association errors) $\mathbf{T} - \mathbf{W} \cdot \mathbf{Q}$.
- Least squares may be used for computing the pseudoinverse.

The Binary Hebb Rule

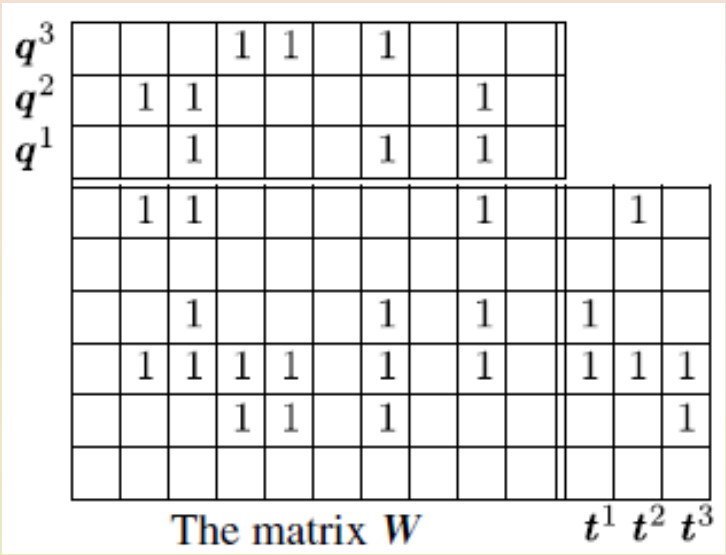
- In the context of associative memory, the so-called binary Hebb rule was suggested.
- It requires that the pattern is binary-encoded, i.e., all patterns $\mathbf{q}^p \in \{0,1\}^n$ and $\mathbf{t}^p \in \{0,1\}^m$.
- The summation is replaced by a simple logical OR and we obtain the **binary Hebb rule**

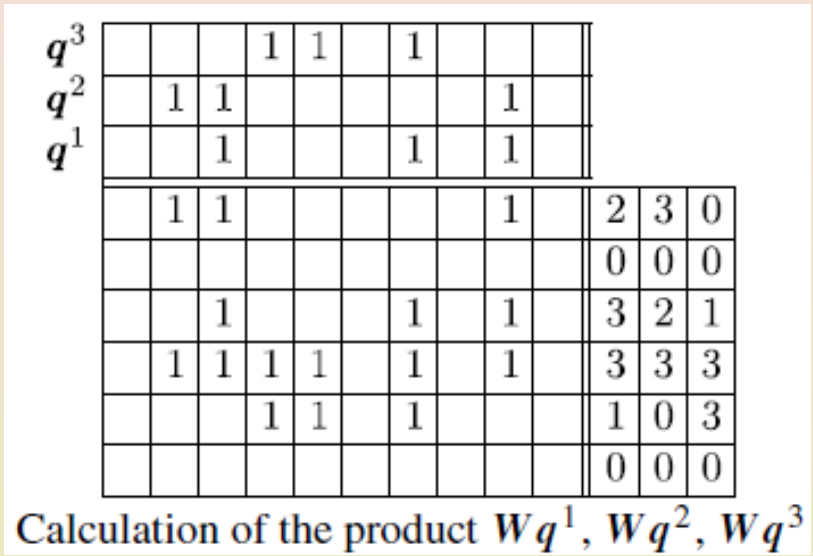
$$w_{ij} = \bigvee_{p=1}^{N} q_j^p t_i^p.$$

- The weight matrix is also binary, and a matrix element w_{ij} is equal to 1 if and only if at least one of the entries $q_j^1 t_i^1, \ldots, q_j^N t_i^N$ is not 0.
- We are tempted to believe that a lot of information is lost here during learning because, when a matrix element takes on the value 1 once, it cannot be changed by additional patterns.

An Example

- The matrix **W** for an example with $n = 10$, $m = 6$ after learning is





- To retrieve the saved patterns we look at **Wq**.
- In the target vector on the right side there is the value 3 in the place where the learned target vector had a one. The correct results would be obtained by setting a threshold value of 3.
- In the general case we choose the number of ones in the query vector as the threshold. Each output neuron thus works like a perceptron, albeit with a variable threshold.

Analysis of Performance

- If the weight matrix is sparse, this algorithm performs well.
- The matrix has a size of mn elements. A pair to be saved has $m + n$ bits. The number of memorizable patterns $N_{\max}$ is determined by the following condition:
  $$\alpha = \frac{\text{number of storable bits}}{\text{number of binary matrix elements}} = \frac{(m+n)N_{\max}}{mn} \leq \ln 2 \approx 0.69.$$
- The maximum memory efficiency:
  - List memory $\alpha = 1$;
  - Associative memory with the binary Hebb rule $\alpha = 0.69$;
  - Kohonen associative memory $\alpha = 0.72$;
  - Hopfield networks $\alpha = 0.292$.
- Note the surprisingly high memory capacity of the binary Hebb rule in comparison to the Kohonen model with continuous neurons.
- It is obvious that such memory becomes "full" less quickly when the query and target vectors are sparsely populated with ones.

Spelling Correction: The Setup

- Consider a program that corrects erroneous inputs and maps them to saved words from a dictionary.
- For the query vectors $\mathbf{q}$ we choose a pair encoding.
  - For 26 characters there are $26 \cdot 26 = 676$ ordered pairs.
  - With 676 bits, the query vector has one bit for each possible pair.
  - If a pair of letters occurs in the word, then we set the bit to 1.
  - For "henry", the slots for "he", "en", "nr", "ry" are 1's.
- For the target vector $\mathbf{t}$, 26 bits are reserved for each position in the word up to a maximum length (for example ten characters).
  - For the i-th letter in position j the bit number $(j-1) \cdot 26 + i$ is set.
  - For the word "henry", bits 8, 31, 66, 96 and 129 are set.
  - For a maximum of 10 letters, the target vector has length 260 bits.
- The weight matrix $\mathbf{W}$ has a size $676 \cdot 260$ bits $= 199420$ bits, which by can store at most $N_{\max} \leq 0.69 \frac{mn}{m+n} = 0.69 \frac{676 \cdot 260}{676+260} \approx 130$ words.
- With 72 first names, we save about half that many and test the system.

Spelling Correction: Training and Setting the Threshold

Stored words:

agathe, agnes, alexander, andreas, andree, anna, annemarie, astrid, august, bernhard, bjorn, cathrin, christian, christoph, corinna, corrado, dieter, elisabeth, elvira, erdmut, ernst, evelyn, fabrizio, frank, franz, geoffrey, georg, gerhard, hannelore, harry, herbert, ingilt, irmgard, jan, johannes, johnny, juergen, karin, klaus, ludwig, luise, manfred, maria, mark, markus, marleen, martin, matthias, norbert, otto, patricia, peter, phillip, quit, reinhold, renate, robert, robin, sabine, sebastian, stefan, stephan, sylvie, ulrich, ulrike, ute, uwe, werner, wolfgang, xavier

- The threshold is initialized to the number of bits in the encoded query, i.e., the number of letter pairs
- Then it is stepwise reduced to two.
- We could further automate the choice of the threshold by comparing with the dictionary for each attempted threshold and output the word found when the comparison succeeds.

Spelling Correction: Testing

Associations of the program:

```
input pattern: harry

threshold: 4, answer: harry
threshold: 3, answer: harry
threshold: 2, answer: horryrrde

------------------------------
input pattern: ute

threshold: 2, answer: ute

------------------------------
input pattern: gerhar

threshold: 5, answer: gerhard
threshold: 4, answer: gerrarn
threshold: 3, answer: jerrhrrd
threshold: 2, answer: jurtyrrde
```

```
input pattern: andrees

threshold: 6, answer: a
threshold: 5, answer: andree
threshold: 4, answer: andrees
threshold: 3, answer: mnnrens
threshold: 2, answer: morxsnssr

------------------------------
input pattern: johanne

threshold: 6, answer: johnnnes
threshold: 5, answer: johnnnes
threshold: 4, answer: jornnnrse
threshold: 3, answer: sorrnyrse
threshold: 2, answer: wtrrsyrse

------------------------------
```

```
------------------------------
input pattern: egrhard

threshold: 6, answer:
threshold: 5, answer:
threshold: 4, answer: gerhard
threshold: 3, answer: gernhrrd
threshold: 2, answer: irrryrrde

------------------------------
input pattern: andr

threshold: 3, answer: andrees
threshold: 2, answer: anexenser
```

```
input pattern: johnnnes

threshold: 6, answer: joh
threshold: 5, answer: johnnnes
threshold: 4, answer: johnnyes
threshold: 3, answer: jonnnyes
threshold: 2, answer: jornsyrse

------------------------------
input pattern: johnnyes

threshold: 7, answer:
threshold: 6, answer: joh
threshold: 5, answer: johnny
threshold: 4, answer: johnnyes
threshold: 3, answer: johnnyes
threshold: 2, answer: jonnnyes
```

- The reaction to the ambiguous inputs "andr" and "johanne" is interesting. In both cases, the network creates a mix of two saved words that fit.
- So neural networks are capable of making associations to similar objects without an explicit similarity metric, but there is no guarantee for a "correct" solution.

Subsection 4

Linear Networks with Minimal Errors

The Idea Behind Backpropagation

- The Hebb rule works very well when the query vectors are linearly independent.
- If this condition is not fulfilled, e.g., when too many training data are available, how do we find the optimal weight matrix?
- Optimal means that it minimizes the average error.
- The backpropagation algorithm uses an elegant solution known from function approximation to change the weights such that the error on the training data is minimized.
- Let N pairs of training vectors $T = \{(\mathbf{q}^1, \mathbf{t}^1), \ldots, (\mathbf{q}^N, \mathbf{t}^N)\}$ be given with $\mathbf{q}^p \in [0,1]^n, \mathbf{t}^p \in [0,1]^m$.
- We are looking for a function $f : [0,1]^n \to [0,1]^m$ which minimizes the squared error

$$\sum_{p=1}^{N} (f(\mathbf{q}^p) - \mathbf{t}^p)^2.$$

Existence of Fitting Functions

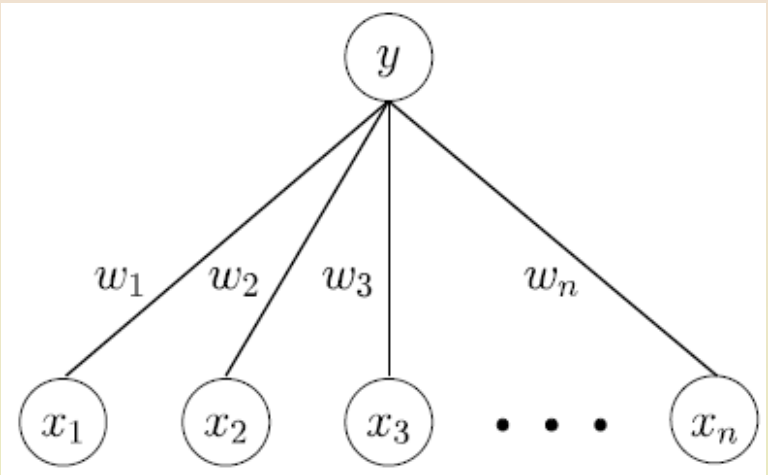
- If the data contains no contradictions, there exist infinitely many functions which make the error zero.
- We define the function

$$f(\mathbf{q}) = \begin{cases} \mathbf{0}, & \text{if } \mathbf{q} \notin \{\mathbf{q}^1, \ldots, \mathbf{q}^N\} \\ \mathbf{t}^p, & \text{if } \mathbf{q} = \mathbf{q}^p, \ p = 1, 2, \ldots, N \end{cases}$$

- This function makes the error zero, but does not represent an intelligent system because it does not generalize well from the training data to new, unknown data from the same representative data set.
- In fact this f overfits because it memorizes.
- Optimally, we would like a function that is smooth and "evens out" the space between the points.
- Continuity and the ability to take multiple derivatives would be sensible requirements.
- Even with these conditions there are infinitely zero-error functions, so we need to restrict the class of functions even further.

Square Error in Two-Layer Network

- The simplest choice is a linear mapping.
- We begin with a two-layer network



- Neuron y activates according to $y = f(\sum_{i=1}^{n} w_i x_i)$, where $f(x) = x$.
- We would like $\mathbf{w}$ to minimize the squared error $E(\mathbf{w}) = \sum_{p=1}^{N} (\mathbf{w}\mathbf{q}^p - \mathbf{t}^p)^2 = \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i q_i^p - t^p\right)^2$.
- As a necessary condition all partial derivatives must be zero.
- Therefore, for $j = 1, \ldots, n$:

$$\frac{\partial E}{\partial w_j} = 2 \sum_{p=1}^{N} (\sum_{i=1}^{n} w_i q_i^p - t^p) q_j^p = 0$$

Least Squares Method: Minimizing Square Error

- Continuing, we get

$$\frac{\partial E}{\partial w_j} = 2\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p - t^p\right)q_j^p = 0$$

$$\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p q_j^p - t^p q_j^p\right) = 0$$

$$\sum_{i=1}^{n} w_i \sum_{p=1}^{N} q_i^p q_j^p = \sum_{p=1}^{N} t^p q_j^p$$

$$\mathbf{Aw} = \mathbf{b}, \text{ if } A_{ij} = \sum_{p=1}^{N} q_i^p q_j^p \text{ and } b_j = \sum_{p=1}^{N} t^p q_j^p.$$

- These **normal equations** always have at least one solution.
- If **A** is invertible, they have exactly one solution.
- The matrix **A** is positive-definite, which implies that, in the unique solution case, the discovered solution is a global minimum.
- This algorithm is known as **least squares method**.
- The calculation time for setting up the matrix **A** is $\Theta(N \cdot n^2)$ and the time for solving the system is $O\left(n^3\right)$.

Incremental vs. Batch Learning

- Least squares (like the perceptron and decision tree learning) is a batch learning algorithm, as opposed to incremental learning.
- In **batch learning**, all training data must be learned in one run.
- If new training data is added, it cannot be learned in addition, rather the whole learning process must be repeated with the enlarged set.
- This problem is solved by **incremental learning algorithms**, which can adapt the learned model to each additional new example.
- In the following algorithms we will additively update the weights for each new training example by $w_j := w_j + \Delta w_j$.
- To derive an incremental variant of the least squares method, we reconsider the above calculated n partial derivatives of the error function

$$\frac{\partial E}{\partial w_j} = 2 \sum_{p=1}^{N} (\sum_{i=1}^{n} w_i q_i^p - t^p) q_j^p.$$

Using the Gradient of the Error for Incremental Learning

- The gradient $\nabla E = (\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n})$ as a vector of all partial derivatives of the error function points in the direction of the strongest rise of the error function in the n-dimensional space of the weights.
- While searching for a minimum, we will therefore follow the opposite direction:

$$\Delta w_j = -\frac{\eta}{2}\frac{\partial E}{\partial w_j} = -\eta \sum_{p=1}^{N}(\sum_{i=1}^{n} w_i q_i^p - t^p) q_j^p,$$

  where the learning rate η is a freely selectable positive constant.
- A larger η speeds up convergence but at the same time raises the risk of oscillation around minima or flat valleys. A large η, e.g., $\eta = 1$, is often used at the start and then slowly shrunk.

The Delta Learning Algorithm

- By replacing $y^p = \sum_{i=1}^{n} w_i q_i^p$ for training example $\mathbf{q}^p$, we obtain the delta rule $\Delta w_j = \eta \sum_{p=1}^{N} (t^p - y^p) q_j^p$.
- Thus, for every training example the difference between t^p and y^p is calculated for the given $\mathbf{q}^p$.
- The weights are changed proportionally to the sum over all patterns:

DeltaLearning(TrainingExamples, η)

Initialize all weights w_j randomly
Repeat
  $\Delta \mathbf{w} = \mathbf{0}$
  For all $(\mathbf{q}^p, t^p) \in$ TrainingExamples
    Calculate network output $y^p = \mathbf{w}^p \mathbf{q}^p$
    $\Delta \mathbf{w} = \Delta \mathbf{w} + \eta (t^p - y^p) \mathbf{q}^p$
  $\mathbf{w} = \mathbf{w} + \Delta \mathbf{w}$
Until w converges

The Incremental Delta Learning Algorithm

- The algorithm is still not really incremental because the weight changes only occur after all training examples have been applied once.
- We can correct this deficiency by directly changing the weights (incremental gradient descent) after every training example, which is no longer a correct implementation of the delta rule.

DeltaLearningIncremental(TrainingExamples, η)

Initialize all weights w_j randomly
Repeat
  For all $(\mathbf{q}^p, t^p) \in$ TrainingExamples
    Calculate network output $y^p = \mathbf{w}^p\mathbf{q}^p$
    $\Delta\mathbf{w} = \Delta\mathbf{w} + \eta(t^p - y^p)\mathbf{q}^p$
  $\mathbf{w} = \mathbf{w} + \eta(t^p - y^p)\mathbf{q}^p$
Until w converges

Comparing Perceptron, Least Squares and Delta Rule

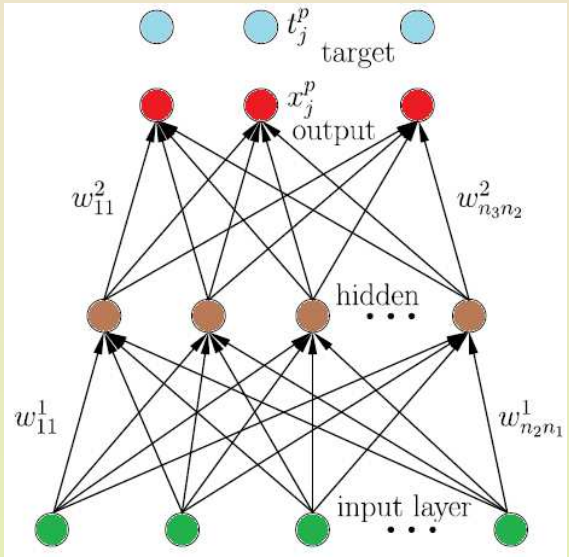
- Whereas for perceptrons a classifier for linearly separable classes is learned through the threshold decision, the other two methods generate a linear approximation to the data.
- A classifier can be generated from the linear mapping, if desired, by application of a threshold function.
- The perceptron and the delta rule are iterative algorithms for which the time until convergence depends heavily on the data.
  - In the case of linearly separable data, an upper limit on the number of iteration steps can be found for the perceptron.
  - For the delta rule, in contrast, there is only a guarantee of asymptotic convergence without limit.
  - For least squares, learning consists of setting up and solving a linear system of equations for the weight vector and, thus, there is a hard limit on the computation time.
- Because of this, the least squares method is always preferable when incremental learning is not needed.

Subsection 5

The Back-Propagation Algorithm

Back-Propagation Networks

- The back-propagation algorithm is the most-used neural model due to its versatility for arbitrary approximation tasks.
- The algorithm originates directly from the incremental delta rule, but
  - it applies a nonlinear sigmoid function on the weighted sum of the inputs as its activation function and
  - a back-propagation network can have more than two layers of neurons.
- In a typical backpropagation network there is an input layer, a hidden layer, and an output layer:
- The current output value x_j^p of the output layer is compared with the target t_j^p.



- Other than the input, all other neurons calculate their current value x_j by the rule $x_j = f(\sum_{i=1}^{n} w_{ji} x_i)$, where n is the number of neurons in the previous layer.

The Back-Propagation Update Rule

- We use the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$.
- Analogous to the incremental delta rule, the weights are changed proportional to the negative gradient of the quadratic error function summed over the output neurons for the training pattern p: $E_p(\mathbf{w}) = \frac{1}{2}\sum_{k\in\text{output}}(t_k^p - x_k^p)^2$
- So $\Delta_p w_{ji} = -\eta\frac{\partial E_p}{\partial w_{ji}}$.
- The above expression is substituted for E_p.
- Within the expression, x_k is replaced by $x_j = f(\sum_{i=1}^n w_{ji}x_i)$.
- The outputs x_i of the neurons of the next layer occur recursively, etc.
- Applying the chain rule multiple times, we get the learning rule

Back-Propagation Rule

$$\Delta_p w_{ji} = \eta\delta_j^p x_i^p \text{ with } \delta_j^p = \begin{cases} x_j^p(1-x_j^p)(t_j^p - x_j^p), & \text{if } j \text{ is output} \\ x_j^p(1-x_j^p)\sum_k \delta_k^p w_{kj}, & \text{if } j \text{ is hidden} \end{cases}.$$

Description of the Back-Propagation Algorithm

- For all neurons, the formula $w_{ji} = w_{ji} + \eta \delta_j^p x_i^p$ for changing the weight w_{ji} contains, like in the Hebb rule, a term $\eta x_i^p x_j^p$.
- The new factor $(1 - x_j^p)$ creates the symmetry, which is missing from the Hebb rule, between the activations 0 and 1 of neuron j.
- For the output neurons, the factor $(t_j^p - x_j^p)$ takes care of a weight change proportional to the error.
- For the hidden neurons, the value δ_j^p of neuron j is calculated recursively from all changes δ_k^p of the neurons of the next higher level.
- After calculating the output of the network (forward propagation) for a training example, the approximation error is calculated.
- This is then used during backward propagation to alter the weights backward from layer to layer.
- The process is applied to all training examples and repeated until the weights no longer change or a time limit is reached.

The Back-Propagation Algorithm

BackPropagation(TrainingExamples, η)

Initialize all weights w_j to random values

Repeat

For all $(\mathbf{q}^p, \mathbf{t}^p) \in$ TrainingExamples

1. **Apply the query vector** $\mathbf{q}^p$ to the input layer
2. **Forward propagation:**

For all layers from the first hidden layer upward

For each neuron of the layer

Calculate activation $x_j = f(\sum_{i=1}^{n} w_{ji} x_i)$

3. **Calculation of the square error** $E_p(\mathbf{w})$
4. **Backward propagation** :

For all levels of weights from the last downward

For each weight w_{ji}

$$w_{ji} = w_{ji} + \eta \delta_j^p x_i^p$$

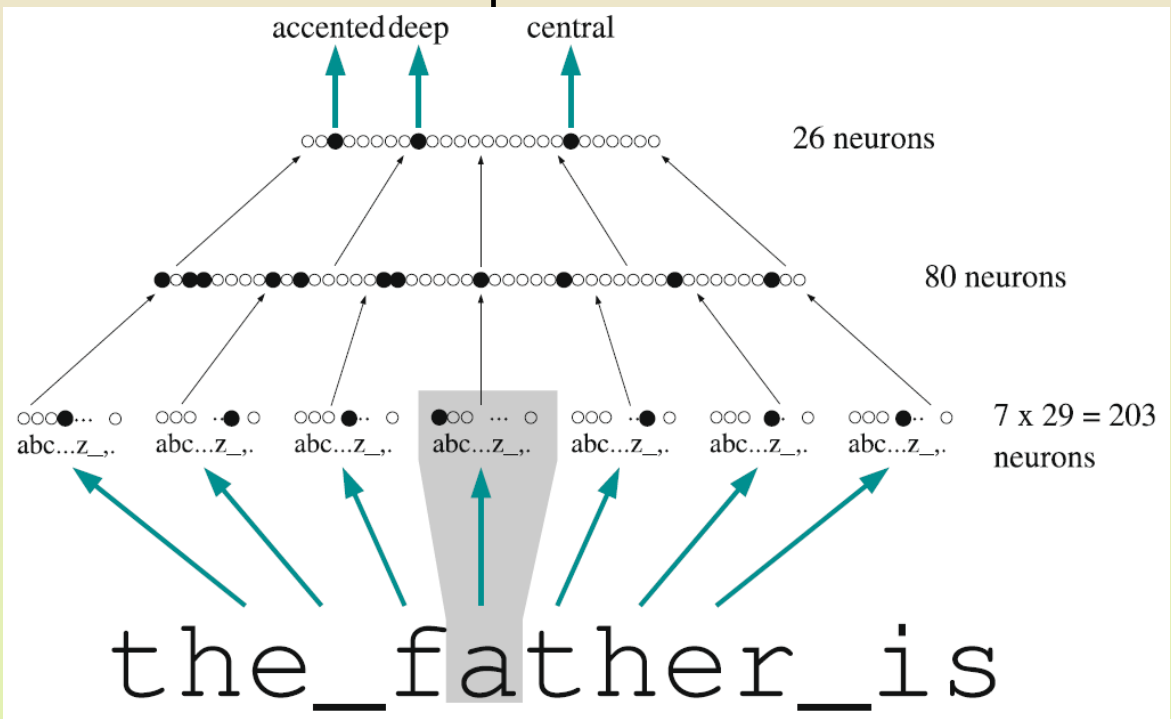
Until w converges or time limit is reached

Linear versus Non-Linear Behavior

- If we build a network with at least one hidden layer, nonlinear mappings can be learned.
- Without hidden layers, the output neurons are no more powerful than a linear neuron, despite the sigmoid function (because the sigmoid function is strictly monotonic).
- The same is true for multi-layer networks which only use a linear function as an activation function, for example the identity function, because chained executions of linear mappings is linear in aggregate.
- The class of the representable functions can be enlarged if we use a variable sigmoid function $f(x) = \frac{1}{1+e^{-(x-\Theta)}}$ with threshold Θ.
- This is implemented by a neuron, whose activation always has the value one and which is connected to neurons in the next highest level, that is inserted into the input layer and into each hidden layer.
- The weights of these connections are learned normally and represent the threshold Θ of the successor neurons.

NETtalk: A Network is Set up to Speak

- Sejnowski and Rosenberg demonstrated very impressively in 1986 what back-propagation is capable of performing.
- They built a system that is able to understandably read English text aloud from a text file.
- It has an input layer with $7 \cdot 29 = 203$ neurons in which the current, three previous and three subsequent letters are encoded.

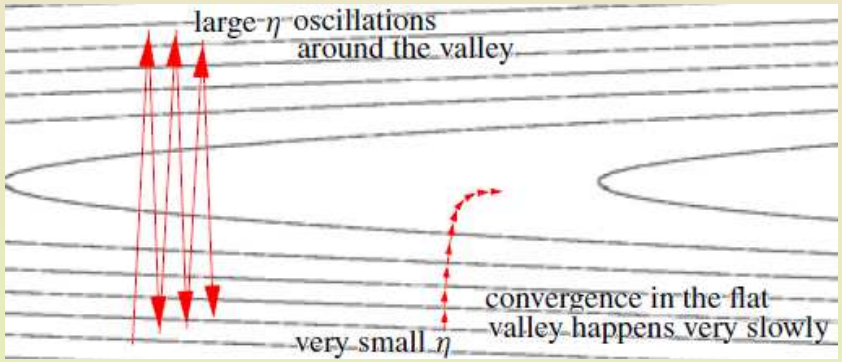


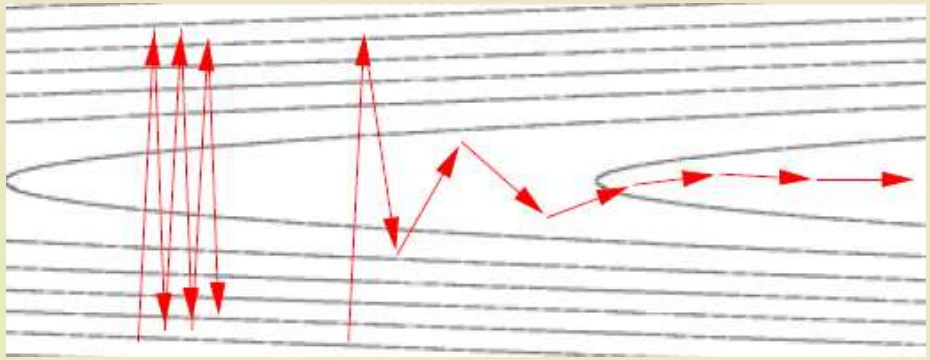
NETtalk: A Network Learns to Speak

- The input is mapped onto the 26 output neurons over 80 hidden neurons, each of which stands for a specific phoneme.
- It was trained with 1,000 words, applied randomly one after another letter by letter.
- For each letter, the target output was manually given.
- To translate the output attributes into actual sounds, part of the speech synthesis system DECtalk was used.
- The network has $203 \times 80 + 80 \times 26 = 18320$ weights.
- It was trained with about 50 cycles over all words. Thus, at about 5 characters per word on average, about $5 \cdot 50 \cdot 1000 = 250000$ iterations were needed. Training took roughly 69 hours.
- The developers observed many properties of the system which are quite similar to human learning. At first the system can only speak unclearly or use simple words. With time it continued to improve and finally reached 95% correctness of pronounced letters.

Summary

- Back-propagation has proved itself in various applications.
- When the network has many thousands of weights and there is a lot of training data to learn, two problems come up:
  - The network often converges to local minima of the error function.
  - Furthermore, back-propagation often converges very slowly.
- Many improvements have been suggested to alleviate these problems.
  - Oscillations can be avoided by slowly reducing the learning rate η





  - To reduce oscillations we may use a momentum term while updating the weights so that the direction of gradient descent does not change too dramatically: $\Delta_p w_{ji}(t) = \eta \delta_j^p x_i^p + \gamma \Delta_p w_{ji}(t-1)$, $0 < \gamma < 1$.
  - Another idea is to minimize the linear instead of the square error.

Subsection 6

Support Vector Machines

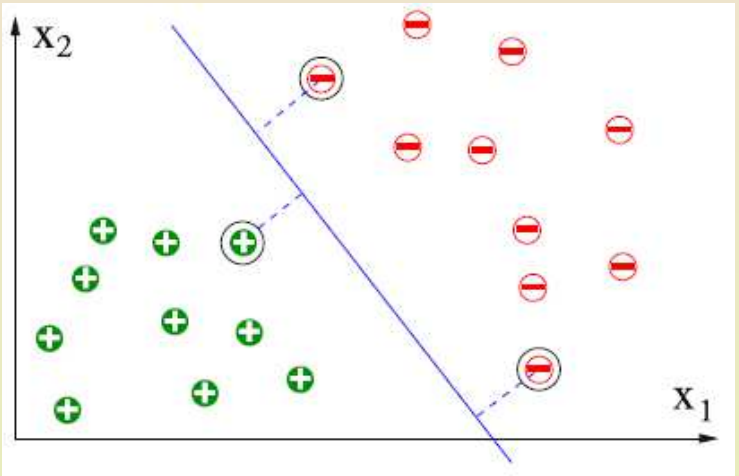
Taking Advantage of both Linearity and Non-Linearity

- Feed-forward neural networks with only one layer of weights are linear.
  - Linearity leads to simple networks and fast learning with guaranteed convergence.
  - Furthermore, the danger of overfitting is small for linear models.
- For many applications, however, the linear models are not strong enough, for example because the relevant classes are not linearly separable.
- Here multi-layered networks such as back-propagation come into use, with the consequence that local minima, convergence problems, and overfitting can occur.
- A promising approach, which brings together the advantages of linear and nonlinear models, follows the theory of **support vector machines** (**SVM**).

Support Vectors

- In the case of two linearly separable classes, it is easy to find a dividing hyperplane, for example with the perceptron learning rule, but there are usually infinitely many such planes:

- We are looking for a plane which has the largest minimum distance to both classes.
- This plane is usually uniquely defined by a few points in the border area, the **support vectors**, all at the same distance from the dividing line.



- To find the support vectors, there is an efficient optimizing algorithm.
- Since the optimal dividing hyperplane is determined by a few parameters, the support vectors, the danger of overfitting is small.

Support Vector Machines: The General Idea

- **Support vector machines** apply this algorithm to non linearly separable problems in a two-step process:
  - In the first step, a nonlinear transformation is applied to the data, with the property that the transformed data is linearly separable.
  - In the second step the support vectors are determined in the transformed space.
- It is always possible to make the classes linearly separable by transforming the vector space if the data contain no contradictions.
- Such a separation can be reached for example by introducing a new $(n+1)$-st dimension and the definition $x_{n+1} = \begin{cases} 1, & \text{if } x \in \text{class}_1 \\ 0, & \text{if } x \in \text{class}_0 \end{cases}$.
- This formula does not help much because it is not generalizable.
- We need a general transformation which is as independent as possible from the current data.

Support Vector Machines: Some Remarks

- There exist such generic transformations even for arbitrarily shaped class division boundaries in the original vector space and in the transformed space, the data are then linearly separable.
- The drawback is that the number of dimensions of the new vector space grows exponentially with the number of dimensions of the original vector space.
- However, the large number of new dimensions is not so problematic because, when using support vectors, the dividing plane is determined by only a few parameters.
- The central nonlinear transformation of the vector space is called the **kernel**, because of which support vector machines are also known as **kernel methods**.
- The original SVM theory developed for classification tasks has been extended and can now be used on regression problems also.
- The mathematics used are rather advanced, so we return to the topic later.

Subsection 7

Applications

Applications of Neural Networks

- There are countless applications for neural networks:
  - A very important area is pattern recognition (analysis of photographs, recognizing schools of fish from sonar images, recognition and classification of military vehicles from radar scans, etc.).
  - Another is recognition of spoken language and handwritten text.
  - They are also trained to control simple robots using sensor data, as well as for heuristically controlling search in backgammon, chess, etc.
- Additional areas include
  - Applications in combination with reinforcement learning techniques;
  - Supplementing statistical methods for forecasting stock markets and for evaluating the creditworthiness of banking customers.
- Neural networks have been more successful and popular than all other machine learning algorithms but their prevalence is being pushed back in favor of other methods because of the great commercial success of data mining and support vector machines.

Subsection 8

Summary

Associative Memory Models

- We saw the perceptron, the delta rule, and back-propagation and their relationship to naive Bayes and the least squares method.
- We also visited Hopfield networks, inspired by biological models, but difficult to manage in practice due to their complex dynamics.
- Associative memory models are important in practice.
- In all neural models, information is stored distributed over many weights, so the network is robust against small disruptions and has the ability to recognize patterns with errors.
- The disadvantage is that it is difficult to localize information and this is rectified in the learned decision tree, in which learned knowledge is easy to understand and, also, to represent as a logical formula.
- A problem with the networks introduced here comes up during incremental learning: In a trained back-propagation network, further training may result in forgetting old patterns.
- To solve this problem, Carpenter and Grossberg developed **adaptive resonance theory** (**ART**), a whole new line of neural models.