IV. Neural Network Learning

A. Artificial Neural Network Learning
Supervised Learning

- Produce desired outputs for training inputs
- Generalize reasonably & appropriately to other inputs
- Good example: pattern recognition
- Feedforward multilayer networks

Feedforward Network
Typical Artificial Neuron

- Connections weights
- Inputs: $s_1, s_j, ..., s_n$
- Connection weights: $w_{i1}, w_{ij}, ..., w_{in}$
- Threshold: $\theta$
- Output: $s_i$
- Linear combination: $h_i$
- Activation function: $\sigma$
- Net input (local field): $s_i$
Equations

Net input:

\[ h_i = \left( \sum_{j=1}^{n} w_{ij} s_j \right) - \theta \]

\[ \mathbf{h} = \mathbf{W} \mathbf{s} - \theta \]

Neuron output:

\[ s'_i = \sigma(h_i) \]

\[ s' = \sigma(\mathbf{h}) \]

Single-Layer Perceptron
Variables

Single Layer Perceptron Equations

Binary threshold activation function:

\[ \sigma(h) = \Theta(h) = \begin{cases} 
1, & \text{if } h > 0 \\
0, & \text{if } h \leq 0
\end{cases} \]

Hence, \[ y = \begin{cases} 
1, & \text{if } \sum_j w_j x_j > \theta \\
0, & \text{otherwise}
\end{cases} = \begin{cases} 
1, & \text{if } w \cdot x > \theta \\
0, & \text{if } w \cdot x \leq \theta
\end{cases} \]
2D Weight Vector

\[ w \cdot x = \|w\| \|x\| \cos \phi \]

\[ \cos \phi = \frac{v}{\|x\|} \]

\[ w \cdot x = \|w\| v \]

\[ w \cdot x > \theta \]
\[ \Leftrightarrow \|w\|v > \theta \]
\[ \Leftrightarrow v > \theta / \|w\| \]

N-Dimensional Weight Vector

separating hyperplane

normal vector
Goal of Perceptron Learning

- Suppose we have training patterns $x^1, x^2, \ldots, x^P$ with corresponding desired outputs $y^1, y^2, \ldots, y^P$
- where $x^p \in \{0, 1\}^n$, $y^p \in \{0, 1\}$
- We want to find $w, \theta$ such that $y^p = \Theta(w \cdot x^p - \theta)$ for $p = 1, \ldots, P$

Treating Threshold as Weight

$$h = \left( \sum_{j=1}^{n} w_j x_j \right) - \theta$$

$$= -\theta + \sum_{j=1}^{n} w_j x_j$$
Treating Threshold as Weight

\[ h = \left( \sum_{j=1}^{n} w_j x_j \right) - \theta \]

Let \( x_0 = -1 \) and \( w_0 = \theta \)

\[ h = w_0 x_0 + \sum_{j=1}^{n} w_j x_j = \sum_{j=0}^{n} w_j x_j = \tilde{w} \cdot \tilde{x} \]

Augmented Vectors

\[ \tilde{w} = \begin{pmatrix} \theta \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \quad \tilde{x}^p = \begin{pmatrix} -1 \\ x_1^p \\ \vdots \\ x_n^p \end{pmatrix} \]

We want \( y^p = \Theta(\tilde{w} \cdot \tilde{x}^p), \; p = 1, \ldots, P \)
Reformulation as Positive Examples

We have positive ($y^p = 1$) and negative ($y^p = 0$) examples

Want $\tilde{w} \cdot \tilde{x}^p > 0$ for positive, $\tilde{w} \cdot \tilde{x}^p \leq 0$ for negative

Let $z^p = \tilde{x}^p$ for positive, $z^p = -\tilde{x}^p$ for negative

Want $\tilde{w} \cdot z^p \geq 0$, for $p = 1, \ldots, P$

Hyperplane through origin with all $z^p$ on one side

Adjustment of Weight Vector

2/23/12
Outline of Perceptron Learning Algorithm

1. initialize weight vector randomly
2. until all patterns classified correctly, do:
   a) for $p = 1, \ldots, P$ do:
      1) if $z^p$ classified correctly, do nothing
      2) else adjust weight vector to be closer to correct classification

Weight Adjustment
Improvement in Performance

If $\tilde{w} \cdot z^p < 0$,

$$\tilde{w}' \cdot z^p = (\tilde{w} + \eta z^p) \cdot z^p$$

$$= \tilde{w} \cdot z^p + \eta z^p \cdot z^p$$

$$= \tilde{w} \cdot z^p + \eta \|z^p\|^2$$

$$> \tilde{w} \cdot z^p$$

---

Perceptron Learning Theorem

- If there is a set of weights that will solve the problem,
- then the PLA will eventually find it
- (for a sufficiently small learning rate)
- Note: only applies if positive & negative examples are linearly separable
NetLogo Simulation of Perceptron Learning

Run Perceptron-Geometry.nlogo

Classification Power of Multilayer Perceptrons

- Perceptrons can function as logic gates
- Therefore MLP can form intersections, unions, differences of linearly-separable regions
- Classes can be arbitrary hyperpolyhedra
- Minsky & Papert criticism of perceptrons
- No one succeeded in developing a MLP learning algorithm
Hyperpolyhedral Classes

Credit Assignment Problem

How do we adjust the weights of the hidden layers?

input layer
hidden layers
output layer

Desired output
NetLogo Demonstration of Back-Propagation Learning

Run Artificial Neural Net.nlogo

Adaptive System

System

Evaluation Function
(Fitness, Figure of Merit)

\[ S \]

\[ F \]

Control Parameters

Control Algorithm

\[ C \]
Gradient

\[ \frac{\partial F}{\partial P_k} \] measures how \( F \) is altered by variation of \( P_k \)

\[ \nabla F = \begin{pmatrix} \frac{\partial F}{\partial P_1} \\ \vdots \\ \frac{\partial F}{\partial P_k} \\ \vdots \\ \frac{\partial F}{\partial P_m} \end{pmatrix} \]

\( \nabla F \) points in direction of maximum local increase in \( F \)
Gradient Ascent by Discrete Steps

Gradient Ascent is Local But Not Shortest
Gradient Ascent Process

\[ \dot{P} = \eta \nabla F(P) \]

Change in fitness:

\[ \dot{F} = \frac{dF}{dt} = \sum_{k=1}^{m} \frac{\partial F}{\partial P_k} \frac{dP_k}{dt} = \sum_{k=1}^{m} (\nabla F)_k \dot{P}_k \]

\[ \dot{F} = \nabla F \cdot \dot{P} \]

\[ \dot{F} = \nabla F \cdot \eta \nabla F = \eta \| \nabla F \|^2 \geq 0 \]

Therefore gradient ascent increases fitness (until reaches 0 gradient)

General Ascent in Fitness

Note that any adaptive process \( P(t) \) will increase fitness provided:

\[ 0 < \dot{F} = \nabla F \cdot \dot{P} = \| \nabla F \| \| \dot{P} \| \cos \varphi \]

where \( \varphi \) is angle between \( \nabla F \) and \( \dot{P} \)

Hence we need \( \cos \varphi > 0 \)

or \( |\varphi| < 90^\circ \)
General Ascent on Fitness Surface

Fitness as Minimum Error

Suppose for \( Q \) different inputs we have target outputs \( t^1, \ldots, t^Q \)

Suppose for parameters \( P \) the corresponding actual outputs
are \( y^1, \ldots, y^Q \)

Suppose \( D(t,y) \in [0,\infty) \) measures difference between
target & actual outputs

Let \( E^q = D(t^q,y^q) \) be error on \( q \)th sample

Let \( F(P) = - \sum_{q=1}^Q E^q(P) = - \sum_{q=1}^Q D(t^q,y^q(P)) \)
Gradient of Fitness

\[ \nabla F = \nabla \left( - \sum_q E^q \right) = - \sum_q \nabla E^q \]

\[ \frac{\partial E^q}{\partial P_k} = \frac{\partial}{\partial P_k} D(t^q, y^q) = \sum_j \frac{\partial D(t^q, y^q)}{\partial y^q_j} \frac{\partial y^q_j}{\partial P_k} \]

\[ = \frac{dD(t^q, y^q)}{dy^q} \cdot \frac{\partial y^q}{\partial P_k} \]

\[ = \nabla_{y^q} D(t^q, y^q) \cdot \frac{\partial y^q}{\partial P_k} \]

Jacobian Matrix

Define Jacobian matrix \( J^q = \begin{pmatrix} \frac{\partial y^q_1}{\partial P_1} & \cdots & \frac{\partial y^q_1}{\partial P_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y^q_n}{\partial P_1} & \cdots & \frac{\partial y^q_n}{\partial P_m} \end{pmatrix} \)

Note \( J^q \in \mathbb{R}^{n \times m} \) and \( \nabla D(t^q, y^q) \in \mathbb{R}^{n \times 1} \)

Since \( \left( \nabla E^q \right)_k = \frac{\partial E^q}{\partial P_k} = \sum_j \frac{\partial y^q_j}{\partial P_k} \frac{\partial D(t^q, y^q)}{\partial y^q_j} \),

\[ \therefore \nabla E^q = (J^q)^T \nabla D(t^q, y^q) \]
Derivative of Squared Euclidean Distance

Suppose \( D(t, y) = \|t - y\|^2 = \sum_i (t_i - y_i)^2 \)

\[
\frac{\partial D(t - y)}{\partial y_j} = \frac{\partial}{\partial y_j} \sum_i (t_i - y_i)^2 = \sum_i \frac{\partial (t_i - y_i)^2}{\partial y_j} = \frac{d(t_j - y_j)^2}{dy_j} = -2(t_j - y_j)
\]

\[
\therefore \frac{dD(t,y)}{dy} = 2(y - t)
\]

Gradient of Error on \( q^{th} \) Input

\[
\frac{\partial E^q}{\partial P_k} = \frac{dD(t^q, y^q)}{dy^q} \cdot \frac{\partial y^q}{\partial P_k}
\]

\[
= 2(y^q - t^q) \cdot \frac{\partial y^q}{\partial P_k}
\]

\[
= 2 \sum_j (y^q_j - t^q_j) \frac{\partial y^q_j}{\partial P_k}
\]

\[
\nabla E^q = 2(J^q)^T (y^q - t^q)
\]
Recap

\[ \dot{P} = \eta \sum_q \left( J^q \right)^T (t^q - y^q) \]

To know how to decrease the differences between actual & desired outputs, we need to know elements of Jacobian, \( \frac{\partial y^q}{\partial P_k} \), which says how jth output varies with kth parameter (given the qth input)

The Jacobian depends on the specific form of the system, in this case, a feedforward neural network

Multilayer Notation
Notation

- $L$ layers of neurons labeled $1, \ldots, L$
- $N_l$ neurons in layer $l$
- $s^l$ = vector of outputs from neurons in layer $l$
- input layer $s^1 = x^q$ (the input pattern)
- output layer $s^L = y^q$ (the actual output)
- $W^l$ = weights between layers $l$ and $l+1$
- Problem: find out how outputs $y_i^q$ vary with weights $W_{jk}^l (l = 1, \ldots, L-1)$

Typical Neuron
Error Back-Propagation

We will compute $\frac{\partial E^q}{\partial W_{ij}^l}$ starting with last layer ($l = L - 1$) and working back to earlier layers ($l = L - 2, \ldots, 1$)

Delta Values

Convenient to break derivatives by chain rule:

$$\frac{\partial E^q}{\partial W_{ij}^{l-1}} = \frac{\partial E^q}{\partial h_i^l} \frac{\partial h_i^l}{\partial W_{ij}^{l-1}}$$

Let $\delta_i^l = \frac{\partial E^q}{\partial h_i^l}$

So

$$\frac{\partial E^q}{\partial W_{ij}^{l-1}} = \delta_i^l \frac{\partial h_i^l}{\partial W_{ij}^{l-1}}$$
Output-Layer Neuron

\[ s_i^L = \sigma_{h_i^L} \]

Output-Layer Derivatives (1)

\[ \delta_i^L = \frac{\partial E^q}{\partial h_i^L} = \frac{\partial}{\partial h_i^L} \sum_k (s_k^L - t_k^q)^2 \]

\[ = \frac{\partial (s_i^L - t_i^q)^2}{\partial h_i^L} = 2(s_i^L - t_i^q) \frac{\partial s_i^L}{\partial h_i^L} \]

\[ = 2(s_i^L - t_i^q) \sigma'(h_i^L) \]
Output-Layer Derivatives (2)

\[
\frac{\partial h_i^L}{\partial W_{ij}^{L-1}} = \frac{\partial}{\partial W_{ij}^{L-1}} \sum_k W_{ik}^{L-1} s_k^{L-1} = s_j^{L-1}
\]

\[\therefore \frac{\partial E_q}{\partial W_{ij}^{L-1}} = \delta_i^L s_j^{L-1}\]

where \(\delta_i^L = 2(s_i^L - t_i^q)\sigma'(h_i^L)\)

Hidden-Layer Neuron

\[
\begin{align*}
&s_1^{L-1} \\
&s_j^{L-1} \\
&s_N^{L-1}
\end{align*}
\]

\[
\begin{align*}
&\sum \\
&\omega
\end{align*}
\]

\[
\begin{align*}
&s_1^L \\
&s_j^L \\
&s_N^L
\end{align*}
\]

\[
\begin{align*}
&\sum \\
&\omega
\end{align*}
\]

\[
\begin{align*}
&s_1^{L+1} \\
&s_j^{L+1} \\
&s_N^{L+1}
\end{align*}
\]

\[
\begin{align*}
&\sum \\
&\omega
\end{align*}
\]

\[E^l\]
Hidden-Layer Derivatives (1)

Recall
\[
\frac{\partial E^q}{\partial W_{ij}^{l-1}} = \delta_i^l \frac{\partial h_i^l}{\partial W_{ij}^{l-1}}
\]

\[
\delta_i^l = \frac{\partial E^q}{\partial h_i^l} = \sum_k \frac{\partial E^q}{\partial h_{k}^{l+1}} \frac{\partial h_{k}^{l+1}}{\partial h_i^l} = \sum_k \delta_{k}^{l+1} \frac{\partial h_{k}^{l+1}}{\partial h_i^l}
\]

\[
\frac{\partial h_{k}^{l+1}}{\partial h_i^l} = \sum_m \frac{\partial W_{km}^l}{\partial h_i^l} s_m^l = \frac{\partial W_{ki}^l}{\partial h_i^l} s_k^l = \frac{d\sigma(h_i^l)}{dh_i^l} = W_{ki}^l \sigma'(h_i^l)
\]

\[
\therefore \delta_i^l = \sum_k \delta_k^{l+1} W_{ki}^l \sigma'(h_i^l) = \sigma'(h_i^l) \sum_k \delta_k^{l+1} W_{ki}^l
\]

Hidden-Layer Derivatives (2)

\[
\frac{\partial h_i^l}{\partial W_{ij}^{l-1}} = \frac{\partial}{\partial W_{ij}^{l-1}} \sum_k W_{ik}^{l-1} s_k^{l-1} = \frac{dW_{ij}^{l-1}}{dW_{ij}^{l-1}} s_j^{l-1} = s_j^{l-1}
\]

\[
\therefore \frac{\partial E^q}{\partial W_{ij}^{l-1}} = \delta_i^l s_j^{l-1}
\]

where \(\delta_i^l = \sigma'(h_i^l) \sum_k \delta_k^{l+1} W_{ki}^l\)
Derivative of Sigmoid

Suppose \( s = \sigma(h) = \frac{1}{1 + \exp(-ah)} \) (logistic sigmoid)

\[
D_h s = D_h \left[ \frac{1}{1 + \exp(-ah)} \right] = \frac{-1}{1 + \exp(-ah)} D_h \left( 1 + e^{-ah} \right)
\]

\[
= -\left( 1 + e^{-ah} \right)^{-2} (-ae^{-ah}) = \alpha \frac{e^{-ah}}{(1 + e^{-ah})^2}
\]

\[
= \alpha \frac{1}{1 + e^{-ah}} \frac{e^{-ah}}{1 + e^{-ah}} = \alpha s \left( \frac{1 + e^{-ah}}{1 + e^{-ah}} - \frac{1}{1 + e^{-ah}} \right)
\]

\[
= \alpha s (1 - s)
\]

Summary of Back-Propagation Algorithm

Output layer: \( \delta^L_i = 2\alpha s^L_i (1 - s^L_i) \left( s^L_i - t^L_i \right) \)

\[
\frac{\partial E^q}{\partial W^L_{ij}} = \delta^L_i s^L_j
\]

Hidden layers: \( \delta^l_i = \alpha s^l_i (1 - s^l_i) \sum_k \delta^{l+1}_k W^l_{ki} \)

\[
\frac{\partial E^q}{\partial W^l_{ij}} = \delta^l_j s^{l-1}_j
\]
Output-Layer Computation

\[ \Delta W_{ij}^{L-1} = \eta \delta_i^L s_j^{L-1} \]

\[ \delta_i^L = 2\alpha s_i^L (1 - s_i^L) (t_i^q - s_i^L) \]

Hidden-Layer Computation

\[ \Delta W_{ij}^{l-1} = \eta \delta_i^l s_j^{l-1} \]

\[ \delta_i^l = \alpha s_i^l (1 - s_i^l) \sum_k \delta_k^{l+1} W_{ki}^l \]
Training Procedures

- **Batch Learning**
  - on each *epoch* (pass through all the training pairs),
  - weight changes for all patterns accumulated
  - weight matrices updated at end of epoch
  - accurate computation of gradient

- **Online Learning**
  - weight are updated after back-prop of each training pair
  - usually randomize order for each epoch
  - approximation of gradient

- Doesn’t make much difference

---

Summation of Error Surfaces

![Error Surfaces Diagram]
Gradient Computation in Batch Learning

Gradient Computation in Online Learning
Testing Generalization

Problem of Rote Learning
A Few Random Tips

- Too few neurons and the ANN may not be able to decrease the error enough
- Too many neurons can lead to rote learning
- Preprocess data to:
  - standardize
  - eliminate irrelevant information
  - capture invariances
  - keep relevant information
- If stuck in local min., restart with different random weights
Run Example BP Learning

Beyond Back-Propagation

- Adaptive Learning Rate
- Adaptive Architecture
  - Add/delete hidden neurons
  - Add/delete hidden layers
- Radial Basis Function Networks
- Recurrent BP
- Etc., etc., etc…. 
What is the Power of Artificial Neural Networks?

- With respect to Turing machines?
- As function approximators?

Can ANNs Exceed the “Turing Limit”?

- There are many results, which depend sensitively on assumptions; for example:
- Finite NNs with real-valued weights have super-Turing power (Siegelmann & Sontag ‘94)
- Recurrent nets with Gaussian noise have sub-Turing power (Maass & Sontag ‘99)
- Finite recurrent nets with real weights can recognize all languages, and thus are super-Turing (Siegelmann ‘99)
- Stochastic nets with rational weights have super-Turing power (but only P/POLY, BPP/log”) (Siegelmann ‘99)
- But computing classes of functions is not a very relevant way to evaluate the capabilities of neural computation
A Universal Approximation Theorem

Suppose $f$ is a continuous function on $[0,1]^n$.
Suppose $\sigma$ is a nonconstant, bounded, monotone increasing real function on $\mathbb{R}$.
For any $\varepsilon > 0$, there is an $m$ such that

$$\exists a \in \mathbb{R}^m, \ b \in \mathbb{R}^n, \ W \in \mathbb{R}^{m \times n} \text{ such that if }$$

$$F(x_1, \ldots, x_n) = \sum_{i=1}^{m} a_i \sigma \left( \sum_{j=1}^{n} W_{ij} x_j + b_j \right)$$

[i.e., $F(x) = a \cdot \sigma(Wx + b)$]

then $|F(x) - f(x)| < \varepsilon$ for all $x \in [0,1]^n$

(see, e.g., Haykin, N.Nets 2/e, 208–9)

One Hidden Layer is Sufficient

- **Conclusion**: One hidden layer is sufficient to approximate any continuous function arbitrarily closely

![Diagram of a neural network with one hidden layer](image)
The Golden Rule of Neural Nets

Neural Networks are the *second-best way* to do *everything*!