

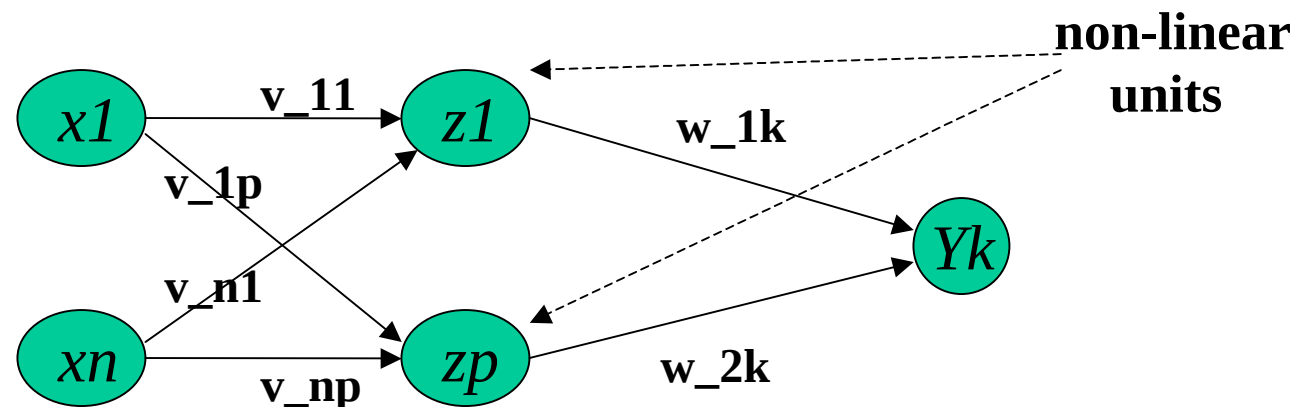
Chapter 6:

Backpropagation Nets

- **Architecture:** at least one layer of non-linear hidden units
- **Learning:** supervised, error driven, generalized delta rule
- Derivation of the weight update formula (with gradient descent approach)
- Practical considerations
- Variations of BP nets
- Applications

Architecture of BP Nets

- **Multi-layer, feed-forward network**
 - Must have at least one hidden layer
 - Hidden units must be non-linear units (usually with sigmoid activation functions)
 - Fully connected between units in two consecutive layers, but no connection between units within one layer.
 - For a net with only one hidden layer, each hidden unit z_j receives input from all input units x_i and sends output to all output units y_k



– Additional notations: (nets with one hidden layer)

$x = (x_1, \dots, x_n)$: input vector

$z = (z_1, \dots, z_p)$: hidden vector (after x applied on input layer)

$y = (y_1, \dots, y_m)$: output vector (computation result)

delta_k : error term on Y_k

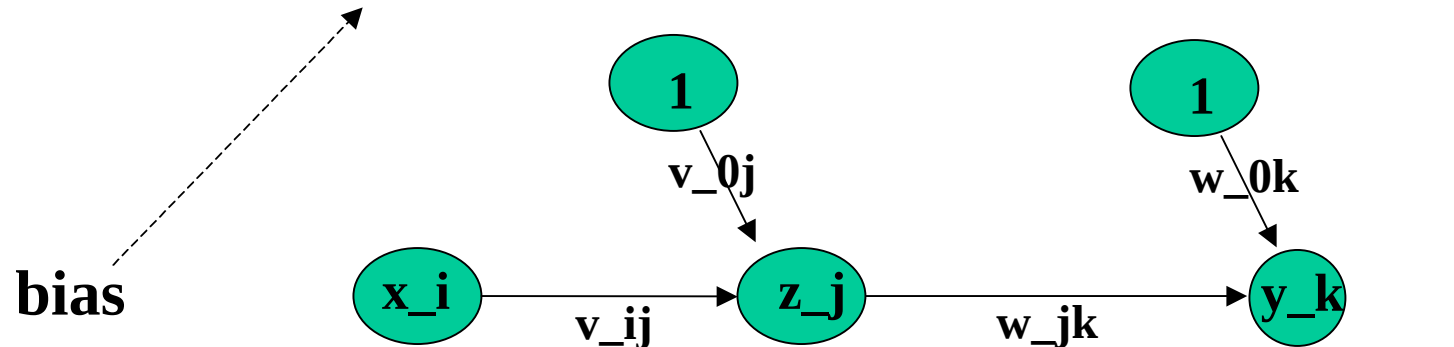
- Used to update weights w_{jk}
- Backpropagated to z_j

delta_j : error term on Z_j

- Used to update weights v_{ij}

$z_{\text{inj}} = v_{0j} + \text{Sum}(x_i * v_{ij})$: input to hidden unit Z_j

$y_{\text{inj}} = w_{0k} + \text{Sum}(z_j * w_{jk})$: input to output unit Y_k



- Forward computing:
 - Apply an input vector $\mathbf{x}$ to input units
 - Computing activation/output vector $\mathbf{z}$ on hidden layer

$$\mathbf{z}_j = f(\mathbf{z}_{0j} + \sum_i \mathbf{v}_{ij} \mathbf{x}_i)$$
 - Computing the output vector $\mathbf{y}$ on output layer

$$\mathbf{y}_k = f(\mathbf{y}_{0k} + \sum_j \mathbf{w}_{jk} \mathbf{z}_j)$$

$\mathbf{y}$ is the result of the computation.
 - The net is said to be a map from input $\mathbf{x}$ to output $\mathbf{y}$
- Theoretically nets of such architecture are able to approximate any L2 functions (all integral functions, including almost all commonly used math functions) to any given degree of accuracy, provided there are sufficient many hidden units
- Question: How to get these weights so that the mapping is what you want

Learning for BP Nets

- Update of weights in W (between output and hidden layers): delta rule as in a single layer net
- Delta rule is not applicable to updating weights in V (between input and hidden layers) because we don't know the target values for hidden units $z_1, \dots, z_p$
- **Solution:** Propagating errors at output units to hidden units, these computed errors on hidden units drives the update of weights in V (again by delta rule), thus called error **BACKPROPAGATION** learning
- How to compute errors on hidden units is the key
- Error backpropagation can be continued downward if the net has more than one hidden layer.

BP Learning Algorithm

step 0: initialize the weights (W and V), including biases, to small random numbers

step 1: while stop condition is false do steps 2 – 9

step 2: for each training sample $x:t$ do steps 3 – 8

/ Feed-forward phase (computing output vector y) */*

step 3: apply vector x to input layer

step 4: compute input and output for each hidden unit Z_j

$$z_{inj} := v_{0j} + \text{Sum}(x_i * v_{ij});$$

$$z_j := f(z_{inj});$$

step 5: compute input and output for each output unit Y_k

$$y_{ink} := w_{0k} + \text{Sum}(v_j * w_{jk});$$

$$y_k := f(y_{ink});$$

/* Error backpropagation phase */

step 6: for each output unit Y_k

$\text{delta}_k := (t_k - y_k) * f'(y_{in_k})$ */* error term */*

$\text{delta}_{w_{jk}} := \alpha * \text{delta}_k * z_j$ */* weight change */*

step 7: For each hidden unit Z_j

$\text{delta}_{inj} := \text{Sum}(\text{delta}_k * w_{jk})$ */* erro BP */*

$\text{delta}_j := \text{delta}_{inj} * f'(z_{inj})$ */*error term */*

$\text{delta}_{v_{ij}} := \alpha * \text{delta}_j * x_i$ */* weight change */*

step 8: Update weights (incl. biases)

$w_{jk} := w_{jk} + \text{delta}_{w_{jk}}$ for all j, k ;

$v_{ij} := v_{ij} + \text{delta}_{v_{ij}}$ for all i, j ;

step 9: test stop condition

Notes on BP learning:

- The error term for a hidden unit z_j is the weighted sum of error terms δ_k of all output units Y_k

$$\delta_{inj} := \text{Sum}(\delta_k * w_{jk})$$

times the derivative of its own output ($f'(z_{inj})$)

In other words, δ_{inj} plays the same role for hidden units v_j as $(t_k - y_k)$ for output units y_k

- Sigmoid function can be either binary or bipolar
- For multiple hidden layers: repeat step 7 (downward)
- Stop condition:
 - Total output error $E = \text{Sum}(t_k - y_k)^2$ falls into the given acceptable error range
 - E changes very little for quite awhile
 - Maximum time (or number of epochs) is reached.

Derivation of BP Learning Rule

- Objective of BP learning: **minimize the mean squared output error** over all training samples $E = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^m (t_k(p) - y_k(p))^2$

For clarity, the derivation is for error of one sample $\mathbf{x} : t$

$$E = 0.5 \sum_{k=1}^m (t_k - y_k)^2$$

- Approach: **gradient descent**. Gradient ∇f given the direction and magnitude of change of f w.r.t its arguments
- For a function of single argument $y = f(x)$, $\nabla y = \frac{dy}{dx} = f'(x)$
Gradient descent requires that $\mathbf{x}$ changes in the **opposite direction of the gradient**, i.e. $\Delta \mathbf{x} = -\nabla y = -f'(x)$.

Then since $\Delta y / \Delta x \approx dy / dx$ for small Δx

we have $\Delta y \approx f'(x) \Delta x = -f'^2(x) \leq 0$

y monotonically decreases

- For a multi-variable function (e.g., our error function E)

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right)$$

Gradient descent requires each argument w_i changes in the opposite direction of the corresponding $\frac{\partial E}{\partial w_i}$ (i.e., $\Delta w_i = -\frac{\partial E}{\partial w_i}$)

Then because

$$\frac{dE}{dt} = \left(\frac{\partial E}{\partial w_1} \frac{dw_1}{dt} + \dots + \frac{\partial E}{\partial w_n} \frac{dw_n}{dt} \right) = \nabla E \cdot \left(\frac{dw_1}{dt}, \dots, \frac{dw_n}{dt} \right)^T$$

we have

$$\Delta E \approx \nabla E \cdot (\Delta w_1, \dots, \Delta w_n)^T = \sum_{i=1}^n \frac{\partial E}{\partial w_i} \Delta w_i = -\sum_{i=1}^n \left(\frac{\partial E}{\partial w_i} \right)^2 \leq 0$$

- Gradient descent guarantees that E monotonically decreases, and $\Delta E = 0$ iff the partial derivatives $\partial E / \partial w_i = 0 \forall i$
- Chain rule of derivatives is used for deriving partial derivatives

$$\text{If } y = f(x) \text{ and } x = g(z), \text{ then } \frac{dy}{dz} = \frac{dy}{dx} \cdot \frac{dx}{dz}$$

Update W , the weights of the output layer

For a particular weight w_{JK} (from units Z_J to Y_K)

$$\begin{aligned}\frac{\partial E}{\partial w_{JK}} &= \frac{\partial}{\partial w_{JK}} \left(0.5 \sum_{k=1}^m (t_k - y_k)^2 \right) = \frac{\partial}{\partial w_{JK}} (0.5 (t_K - y_K)^2) \\ &= (t_K - y_K) \frac{\partial}{\partial w_{JK}} (-y_K) = -(t_K - y_K) \frac{\partial}{\partial w_{JK}} f(y_in_K) \\ &= -(t_K - y_K) f'(y_in_K) \frac{\partial}{\partial w_{JK}} (y_in_K) \quad (\text{by chain rule}) \\ &= -(t_K - y_K) f'(y_in_K) z_J\end{aligned}$$

The last equality comes from the fact that only one of the terms in $y_in_K = \sum_{j=1}^p w_{jK} z_j$, namely $w_{JK} z_J$ involves w_{JK}

Let $d_K = (t_K - y_K) f'(y_in_K)$. Then $\Delta w_{JK} = a \cdot \left(-\frac{\partial E}{\partial w_{JK}} \right) = a \cdot d_K \cdot z_J$

This is the update rule in Step 6 of the algorithm

Update V , the weights of the hidden layer

For a particular weight $\mathbf{v}_{IJ}$ (from unit X_I to Z_J)

$$\begin{aligned}
 \frac{\partial E}{\partial \mathbf{v}_{IJ}} &= \frac{\partial}{\partial \mathbf{v}_{IJ}} (0.5 \sum_{k=1}^m (\mathbf{t}_k - \mathbf{y}_k)^2) \\
 &= - \sum_{k=1}^m ((\mathbf{t}_k - \mathbf{y}_k) \frac{\partial}{\partial \mathbf{v}_{IJ}} \mathbf{y}_k) \quad (\text{every } \mathbf{y}_k \text{ involves } \mathbf{v}_{IJ} \text{ as it is connected to } \mathbf{z}_J) \\
 &= - \sum_{k=1}^m ((\mathbf{t}_k - \mathbf{y}_k) \frac{\partial}{\partial \mathbf{v}_{IJ}} f(\mathbf{y_in}_k)) \\
 &= - \sum_{k=1}^m ((\mathbf{t}_k - \mathbf{y}_k) f'(\mathbf{y_in}_k) \frac{\partial}{\partial \mathbf{v}_{IJ}} (\mathbf{y_in}_k)) \\
 &= - \sum_{k=1}^m (d_k \frac{\partial}{\partial \mathbf{v}_{IJ}} (\mathbf{y_in}_k)) \quad (\text{because } d_k = (\mathbf{t}_k - \mathbf{y}_k) f'(\mathbf{y_in}_k)) \\
 &= - \sum_{k=1}^m (d_k \mathbf{w}_{Jk} \frac{\partial}{\partial \mathbf{v}_{IJ}} \mathbf{z}_J)
 \end{aligned}$$

The last equality comes from the fact that only one of the terms in $\mathbf{y_in}_k = \sum_{j=1}^p \mathbf{w}_{jk} \mathbf{z}_j$, namely $\mathbf{w}_{Jk} \mathbf{z}_J$ involves $\mathbf{v}_{IJ}$ (via $\mathbf{z}_J$)

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{v}_{IJ}} &= -\sum_{k=1}^m (d_k \mathbf{w}_{Jk} \frac{\partial}{\partial \mathbf{v}_{IJ}} z_J) \\
&= -\sum_{k=1}^m (d_k \mathbf{w}_{Jk} f'(z - \mathbf{in}_J) \frac{\partial}{\partial \mathbf{v}_{IJ}} (z - \mathbf{in}_J)) \\
&= -\sum_{k=1}^m (d_k \mathbf{w}_{Jk} f'(z - \mathbf{in}_J) \cdot \mathbf{x}_I) \quad (\text{only } \mathbf{v}_{IJ} \mathbf{x}_I \text{ in } z - \mathbf{in}_J \text{ involves } \mathbf{v}_{IJ}) \\
&= -f'(z - \mathbf{in}_J) \cdot \mathbf{x}_I \cdot \sum_{k=1}^m d_k \mathbf{w}_{Jk} \quad (\text{because } \mathbf{x}_I \text{ and } z - \mathbf{in}_J \text{ are indep. of } k) \\
&= -f'(z - \mathbf{in}_J) \mathbf{x}_I \cdot d - \mathbf{in}_J
\end{aligned}$$

Let $d_J = d - \mathbf{in}_J f'(y - \mathbf{in}_J)$. Then $\Delta \mathbf{v}_{IJ} = a \left(-\frac{E}{\mathbf{v}_{IJ}} \right) = a d_J \mathbf{x}_I$

This is the update rule in Step 7 of the algorithm

Strengths of BP Nets

- **Great representation power**
 - Any L2 function can be represented by a BP net (multi-layer feed-forward net with non-linear hidden units)
 - Many such functions can be learned by BP learning (gradient descent approach)
- **Wide applicability of BP learning**
 - Only requires that a good set of training samples is available)
 - Does not require substantial prior knowledge or deep understanding of the domain itself (ill structured problems)
 - Tolerates noise and missing data in training samples (graceful degrading)
- **Easy to implement** the core of the learning algorithm
- **Good generalization power**
 - Accurate results for inputs outside the training set

Deficiencies of BP Nets

- Learning often takes a **long time** to converge
 - Complex functions often need hundreds or thousands of epochs
- The net is essentially a **black box**
 - It may provide a desired mapping between input and output vectors ($\mathbf{x}$, $\mathbf{y}$) but does not have the information of why a particular $\mathbf{x}$ is mapped to a particular $\mathbf{y}$.
 - It thus cannot provide an intuitive (e.g., causal) explanation for the computed result.
 - This is because the hidden units and the learned weights do not have a semantics. What can be learned are operational parameters, not general, abstract knowledge of a domain
- Gradient descent approach only guarantees to reduce the total error to a **local minimum**. (E may be reduced to zero)
 - Cannot escape from the local minimum error state
 - Not every function that is representable can be learned

- How bad: depends on the shape of the error surface. Too many valleys/wells will make it easy to be trapped in local minima
- Possible remedies:
 - Try nets with different # of hidden layers and hidden units (they may lead to different error surfaces, some might be better than others)
 - Try different initial weights (different starting points on the surface)
 - Forced escape from local minima by random perturbation (e.g., simulated annealing)
- **Generalization** is not guaranteed even if the error is reduced to zero
 - Over-fitting/over-training problem: trained net fits the training samples perfectly (E reduced to 0) but it does not give accurate outputs for inputs not in the training set
- Unlike many statistical methods, there is no theoretically well-founded way to **assess the quality** of BP learning
 - What is the confidence level one can have for a trained BP net, with the final E (which not or may not be close to zero)

- **Network paralysis** with sigmoid activation function

- Saturation regions: $|x| \gg 1$

$f(x) = 1/(1 + e^{-x})$, its derivative $f'(x) = f(x)(1 - f(x)) \rightarrow 0$ when $x \rightarrow \pm\infty$.

When x falls in a saturation region, $f(x)$ hardly changes its value regardless how fast the magnitude of x increases

- Input to an unit may fall into a saturation region when some of its incoming weights become very large during learning. Consequently, weights stop to change no matter how hard you try.

$$\Delta w_{jk} = a \cdot \left(-\frac{\partial E}{\partial w_{jk}} \right) = a \cdot (t_k - y_k) \cdot f'(y_{in_k}) \cdot z_j$$

- Possible remedies:

- Use non-saturating activation functions
- Periodically normalize all weights $w_{jk} := w_{jk} / \|w_{.k}\|_2$

- The learning (accuracy, speed, and generalization) is highly dependent of a set of learning **parameters**
 - Initial weights, learning rate, # of hidden layers and # of units...
 - Most of them can only be determined **empirically** (via experiments)

Practical Considerations

- A good BP net requires more than the core of the learning algorithms. Many parameters must be carefully selected to ensure a good performance.
- Although the deficiencies of BP nets cannot be completely cured, some of them can be eased by some practical means.
- **Initial weights (and biases)**
 - Random, $[-0.05, 0.05]$, $[-0.1, 0.1]$, $[-1, 1]$
 - Normalize weights for hidden layer (v_{ij}) (Nguyen-Widrow)
 - Random assign v_{ij} for all hidden units V_j
 - For each V_j , normalize its weight by $\mathbf{v}_{ij} = b \cdot \mathbf{v}_{ij} / \|\mathbf{v}_{.j}\|_2$
where $\|\mathbf{v}_{.j}\|_2$ is the normalization factor and $b = 0.7\sqrt[n]{p}$
where $p = \#$ of hidden nodes, $n = \#$ of input nodes
 - Avoid bias in weight initialization: $\|\mathbf{v}_{.j}\|_2 = b$ after normalization

- **Training samples:**

- Quality and quantity of training samples determines the quality of learning results
- Samples must be good representatives of the problem space
 - Random sampling
 - Proportional sampling (with prior knowledge of the problem space)
- # of training patterns needed:
 - There is no theoretically ideal number. Following is a rule of thumb
 - W : total # of weights to be trained (depends on net structure)
 - e : desired classification error rate
 - If we have $P = W/e$ training patterns, and we can train a net to correctly classify $(1 - e/2)P$ of them,
 - Then this net would (in a statistical sense) be able to correctly classify a fraction of $1 - e$ input patterns drawn from the same sample space
 - Example: $W = 80$, $e = 0.1$, $P = 800$. If we can successfully train the network to correctly classify $(1 - 0.1/2)*800 = 760$ of the samples, we would believe that the net will work correctly 90% of time with other input.

- **Data representation:**

- Binary vs bipolar

- Bipolar representation uses training samples more efficiently

$$\Delta \mathbf{w}_{jk} = a \cdot d_k \cdot \mathbf{z}_j \quad \Delta \mathbf{v}_{ij} = a \cdot d_j \cdot \mathbf{x}_i$$

no learning will occur when $\mathbf{x}_i = 0$ or $\mathbf{z}_j = 0$ with binary rep.

- # of patterns can be represented n input units:

binary: 2^n

bipolar: $2^{(n-1)}$ if no biases used, this is due to (anti)symmetry
(if the net outputs y for input x , it will output $-y$ for input $-x$)

- Real value data

- Input units: real value units (may subject to normalization)
 - Hidden units are sigmoid
 - Activation function for output units: often linear (even identity)

e.g., $y_k = y_{in_k} = \sum \mathbf{w}_{jk} \mathbf{z}_j$

- Training may be much slower than with binary/bipolar data (some use binary encoding of real values)

- **How many hidden layers and hidden units per layer:**
 - Theoretically, one hidden layer (possibly with many hidden units) is sufficient for any L2 functions
 - There is no theoretical results on minimum necessary # of hidden units (either problem dependent or independent)
 - Practical rule of thumb:
 - $n = \text{\# of input units}$; $p = \text{\# of hidden units}$
 - For binary/bipolar data: $p = 2n$
 - For real data: $p \gg 2n$
 - Multiple hidden layers with fewer units may be trained faster for similar quality in some applications

- **Over-training/over-fitting**
 - Trained net fits very well with the training samples (total error $E \approx 0$), but not with new input patterns
 - Over-training may become serious if
 - Training samples were not obtained properly
 - Training samples have noise
 - Control over-training for better generalization
 - **Cross-validation:** dividing the samples into two sets
 - 90% into training set: used to train the network
 - 10% into test set: used to validate training results
 - periodically test the trained net with test samples, stop training when test results start to deteriorating.
 - Stop training early (before $E \approx 0$)
 - Add noise to training samples: $\mathbf{x:t}$ becomes $\mathbf{x+noise:t}$ (for binary/bipolar: flip randomly selected input units)

Variations of BP nets

- **Adding momentum term** (to speedup learning)
 - Weights update at time $t+1$ contains the momentum of the previous updates, e.g.,
$$\Delta \mathbf{w}_{jk}(t+1) = a \cdot d_k \cdot \mathbf{z}_j + m \cdot \Delta \mathbf{w}_{jk}(t), \text{ where } 0 < m < a \ll 1$$
 - then
$$\Delta \mathbf{w}_{jk}(t+1) = \sum_{s=1}^t m^{t-s} a \cdot d_k(s) \cdot \mathbf{z}_j(s)$$
an exponentially weighted sum of all previous updates
 - Avoid sudden change of directions of weight update (smoothing the learning process)
 - Error is no longer monotonically decreasing
- **Batch mode** of weight updates
 - Weight update once per each epoch
 - Smoothing the training sample outliers
 - Learning independent of the order of sample presentations
 - Usually slower than in sequential mode

- **Variations on learning rate α**
 - Give known underrepresented samples higher rates
 - Find the maximum safe step size at each stage of learning (to avoid overshoot the minimum E when increasing α)
 - **Adaptive learning rate** (delta-bar-delta method)
 - Each weight w_{jk} has its own rate α_{jk}
 - If $\Delta \mathbf{w}_{jk}$ remains in the same direction, increase α_{jk} (E has a smooth curve in the vicinity of current $\mathbf{W}$)
 - If $\Delta \mathbf{w}_{jk}$ changes the direction, decrease α_{jk} (E has a rough curve in the vicinity of current $\mathbf{W}$)

$$\Delta_{jk} = \partial E / \partial w_{jk} = -d_k \mathbf{z}_j$$

$$\bar{\Delta}_{jk}(t) = (1 - b) \Delta_{jk}(t) + b \bar{\Delta}_{jk}(t-1)$$

$$a_{jk}(t+1) = \begin{cases} a_{jk}(t) + k & \text{if } \Delta_{jk}(t) \bar{\Delta}_{jk}(t-1) > 0 \\ (1 - g) a_{jk}(t) & \text{if } \Delta_{jk}(t) \bar{\Delta}_{jk}(t-1) < 0 \\ a_{jk}(t) & \text{if } \Delta_{jk}(t) \bar{\Delta}_{jk}(t-1) = 0 \end{cases}$$

- delta-bar-delta also involves momentum term (of α)
- Experimental comparison
 - Training for XOR problem (batch mode)
 - 25 simulations: success if E averaged over 50 consecutive epochs is less than 0.04
 - results

method	simulations	success	Mean epochs
BP	25	24	16,859.8
BP with momentum	25	25	2,056.3
BP with delta-bar-delta	25	22	447.3

- **Other activation functions**

- Change the range of the logistic function from (0,1) to (a, b)

Let $f(x) = 1/(1 + e^{-x})$, $r = b - a$, $h = -a$.

$g(x) = rf(x) - h$ is a sigmoid function with range (a, b)

$$\begin{aligned} g'(x) &= rf(x)(1 - f(x)) = (g(x) + h)(1 - g(x)/r - h/r) \\ &= \frac{1}{r}(g(x) + h)(r - g(x) - h) \end{aligned}$$

In particular, for bipolar sigmoid function, we have

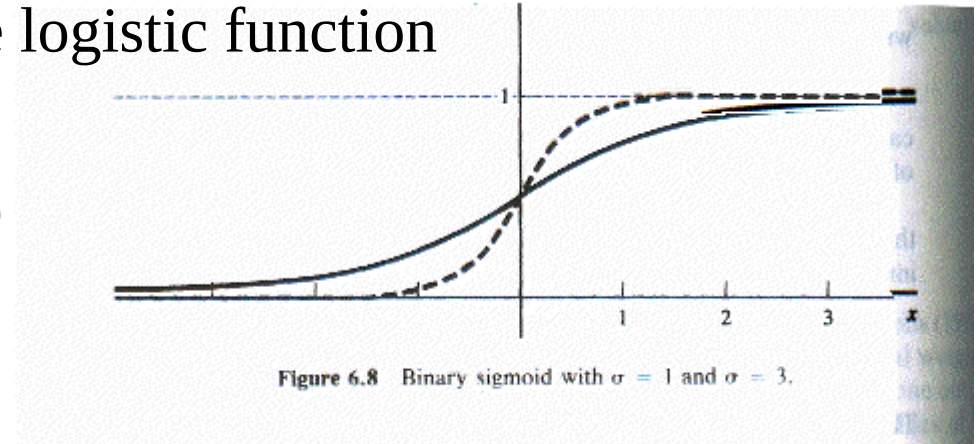
$a = -1$, $b = 1$, then $r = 2$, $h = 1$

$$g(x) = 2f(x) - 1, \text{ and } g'(x) = \frac{1}{2}(1 + g(x))(1 - g(x))$$

- Change the slope of the logistic function

$$f(x) = 1/(1 + e^{-sx}),$$

$$f'(x) = sf(x)(1 - f(x))$$



- Larger slope: quicker to move to saturation regions; faster convergence
 - Smaller slope: slow to move to saturation regions, allows refined weight adjustment
 - σ thus has a effect similar to the learning rate α (but more drastic)
- Adaptive slope (**each node has a learned slope**)

$y_k = f(s_k y_{in_k})$, $z_j = f(s_j z_{in_j})$. Then we have

$$\Delta w_{jk} = -a \partial E / \partial w_{jk} = a d_k s_k z_j, \text{ where } d_k = (t_k - y_k) f'(s_k y_{in_k})$$

$$\Delta v_{ij} = -a \partial E / \partial v_{ij} = a d_j s_j x_i, \text{ where } d_j = \sum d_k s_k w_{jk} f'(s_j z_{in_j})$$

$$\Delta s_k = -a \partial E / \partial s_k = a d_k y_{in_k}, \quad \Delta s_{ij} = -a \partial E / \partial s_{ij} = a d_j z_{in_j}$$

- Another sigmoid function with slower saturation speed

$$f(x) = \frac{2}{p} \arctan(x), \quad f'(x) = \frac{2}{p} \frac{1}{1+x^2}$$

For large $|x|$, $\frac{1}{1+x^2}$ is much smaller than $\frac{1}{(1+e^{-x})(1+e^x)}$

the derivative of logistic function

- A non-saturating function (also differentiable)

$$f(x) = \begin{cases} \log(1+x) & \text{if } x \geq 0 \\ -\log(1-x) & \text{if } x < 0 \end{cases}$$

$$f'(x) = \begin{cases} \frac{1}{1+x} & \text{if } x \geq 0 \\ \frac{1}{1-x} & \text{if } x < 0 \end{cases}, \text{ then, } f'(x) \rightarrow 0 \text{ when } |x| \rightarrow \infty$$

- Non-sigmoid activation function

Radial based function: it has a center $\mathbf{c}$.

$f(\mathbf{x}) > 0$ for all $\mathbf{x}$

$f(\mathbf{x})$ becomes smaller when $|\mathbf{x} - \mathbf{c}|$ becomes larger

$f(\mathbf{x}) \rightarrow 0$ when $|\mathbf{x} - \mathbf{c}| \rightarrow \infty$

e.g., Gaussian function :

$$f(\mathbf{x}) = e^{-x^2},$$

$$f'(\mathbf{x}) = -2\mathbf{x}f(\mathbf{x})$$

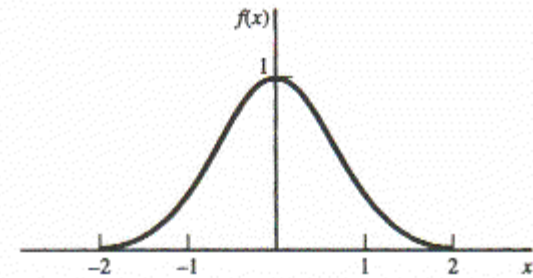


Figure 6.9 Gaussian activation function.

Applications of BP Nets

- **A simple example: Learning XOR**
 - Initial weights and other parameters
 - **weights**: random numbers in $[-0.5, 0.5]$
 - **hidden** units: single layer of 4 units (A 2-4-1 net)
 - **biases** used;
 - **learning rate**: 0.02
 - Variations tested
 - binary vs. bipolar representation
 - different stop criteria (targets with ± 1.0 and with ± 0.8)
 - normalizing initial weights (Nguyen-Widrow)
 - Bipolar is faster than binary
 - convergence: ~ 3000 epochs for binary, ~ 400 for bipolar
 - Why?

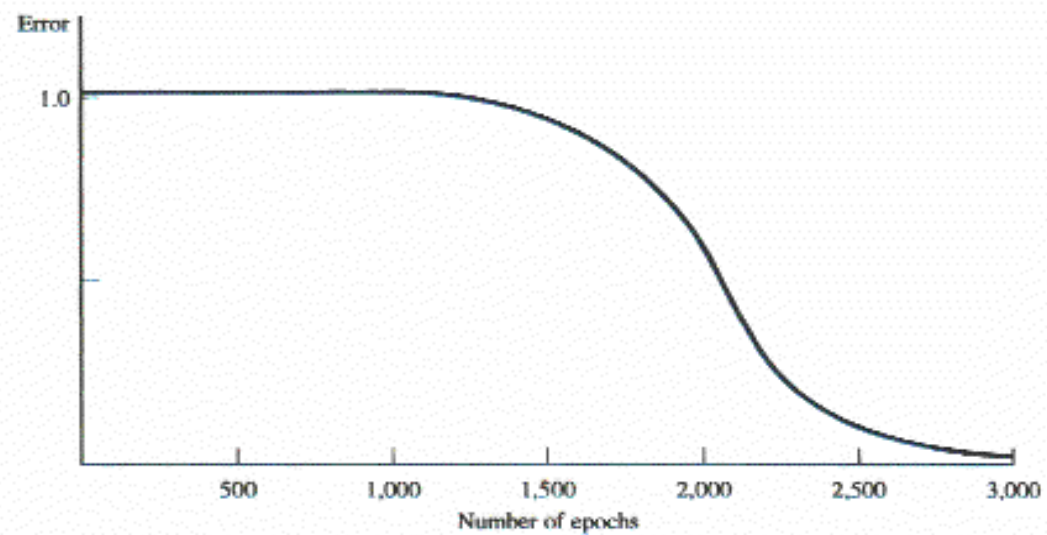


Figure 6.4 Total squared error for binary representation of XOR problem.

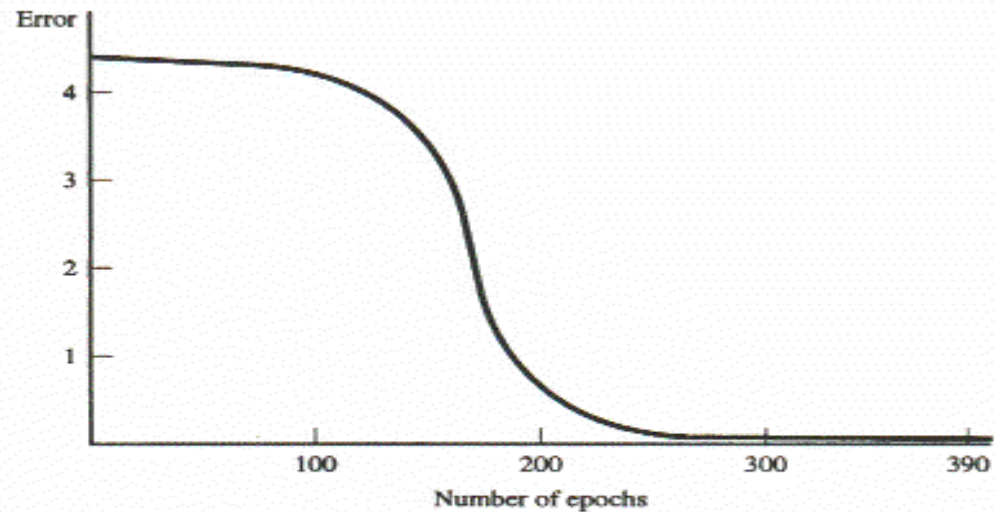


Figure 6.5 Total squared error for bipolar representation of XOR problem.

- Relaxing acceptable error range **may** speed up convergence
 - ± 1.0 is an asymptotic limits of sigmoid function,
 - When an output approaches ± 1.0 , it falls in a saturation region
 - Use $\pm \mathbf{a}$ where $0 < \mathbf{a} < 1.0$ (e.g., ± 0.8)
- Normalizing initial weights **may** also help

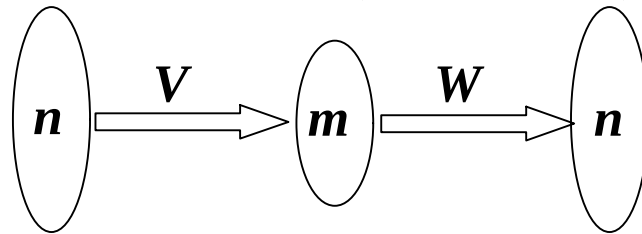
	Random	Nguyen-Widrow
Binary	2,891	1,935
Bipolar	387	224
Bipolar with targets = ± 0.8	264	127

- **Data compression**

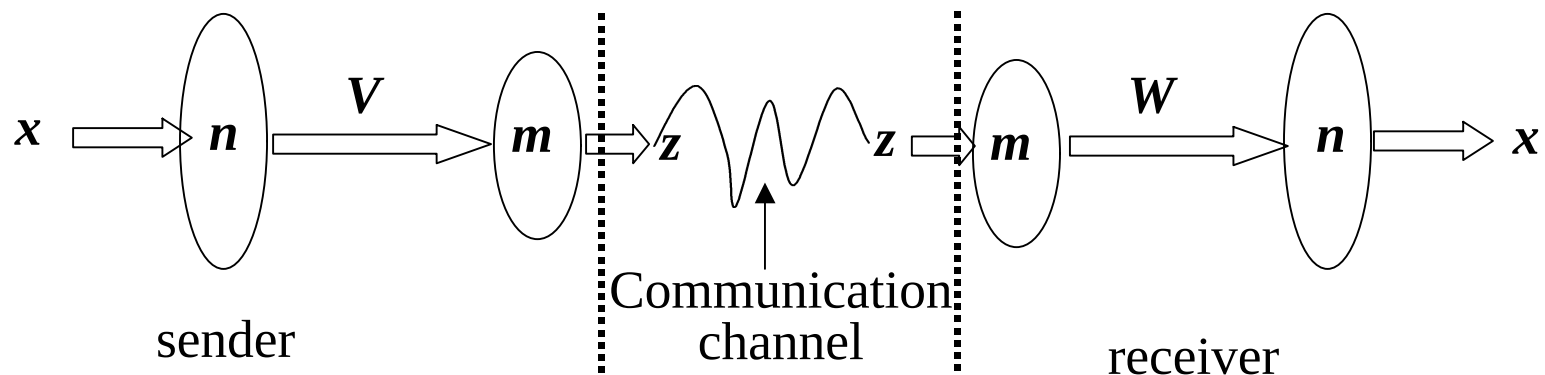
- **Autoassociation** of patterns (vectors) with themselves using a small number of hidden units:

- **training samples**:: $x:x$ (x has dimension n)

- hidden units**: $m < n$ (A n - m - n net)



- If training is successful, applying any vector x on input units will generate the same x on output units
 - Pattern z on hidden layer becomes a compressed representation of x (with smaller dimension $m < n$)
 - Application: reducing transmission cost



- **Example:** compressing character bitmaps.
 - Each character is represented by a 7 by 9 pixel bitmap, or a binary vector of dimension 63
 - 10 characters (A – J) are used in experiment
 - Error range:
 - tight: 0.1 (off: 0 – 0.1; on: 0.9 – 1.0)
 - loose: 0.2 (off: 0 – 0.2; on: 0.8 – 1.0)
 - Relationship between # hidden units, error range, and convergence rate (Fig. 6.7, p.304)
 - relaxing error range may speed up
 - increasing # hidden units (to a point) may speed up
- error range: 0.1 hidden units: 10 # epochs 400+
- error range: 0.2 hidden units: 10 # epochs 200+
- error range: 0.1 hidden units: 20 # epochs 180+
- error range: 0.2 hidden units: 20 # epochs 90+
- no noticeable speed up when # hidden units increases to beyond 22

- **Other applications.**

- Medical diagnosis

- Input: manifestation (symptoms, lab tests, etc.)

- Output: possible disease(s)

- Problems:

- no causal relations can be established

- hard to determine what should be included as inputs

- Currently focus on more restricted diagnostic tasks

- e.g., predict prostate cancer or hepatitis B based on standard blood test

- Process control

- Input: environmental parameters

- Output: control parameters

- Learn ill-structured control functions

- Stock market forecasting
 - Input: financial factors (CPI, interest rate, etc.) and stock quotes of previous days (weeks)
Output: forecast of stock prices or stock indices (e.g., S&P 500)
 - Training samples: stock market data of past few years
- Consumer credit evaluation
 - Input: personal financial information (income, debt, payment history, etc.)
 - Output: credit rating
- And many more
- Key for successful application
 - Careful design of input vector (including all **important** features): some domain knowledge
 - Obtain good training samples: time and other cost

Summary of BP Nets

- **Architecture**
 - Multi-layer, feed-forward (full connection between nodes in adjacent layers, no connection within a layer)
 - One or more hidden layers with non-linear activation function (most commonly used are sigmoid functions)
- **BP learning algorithm**
 - Supervised learning (samples $s:t$)
 - Approach: gradient descent to reduce the total error (why it is also called generalized delta rule)
 - Error terms at output units
error terms at hidden units (why it is called error BP)
 - Ways to speed up the learning process
 - Adding momentum terms
 - Adaptive learning rate (delta-bar-delta)
 - Generalization (cross-validation test)

- **Strengths of BP learning**
 - Great representation power
 - Wide practical applicability
 - Easy to implement
 - Good generalization power
- **Problems of BP learning**
 - Learning often takes a long time to converge
 - The net is essentially a black box
 - Gradient descent approach only guarantees a local minimum error
 - Not every function that is representable can be learned
 - Generalization is not guaranteed even if the error is reduced to zero
 - No well-founded way to assess the quality of BP learning
 - Network paralysis may occur (learning is stopped)
 - Selection of learning parameters can only be done by trial-and-error
 - BP learning is non-incremental (to include new training samples, the network must be re-trained with all old and new samples)