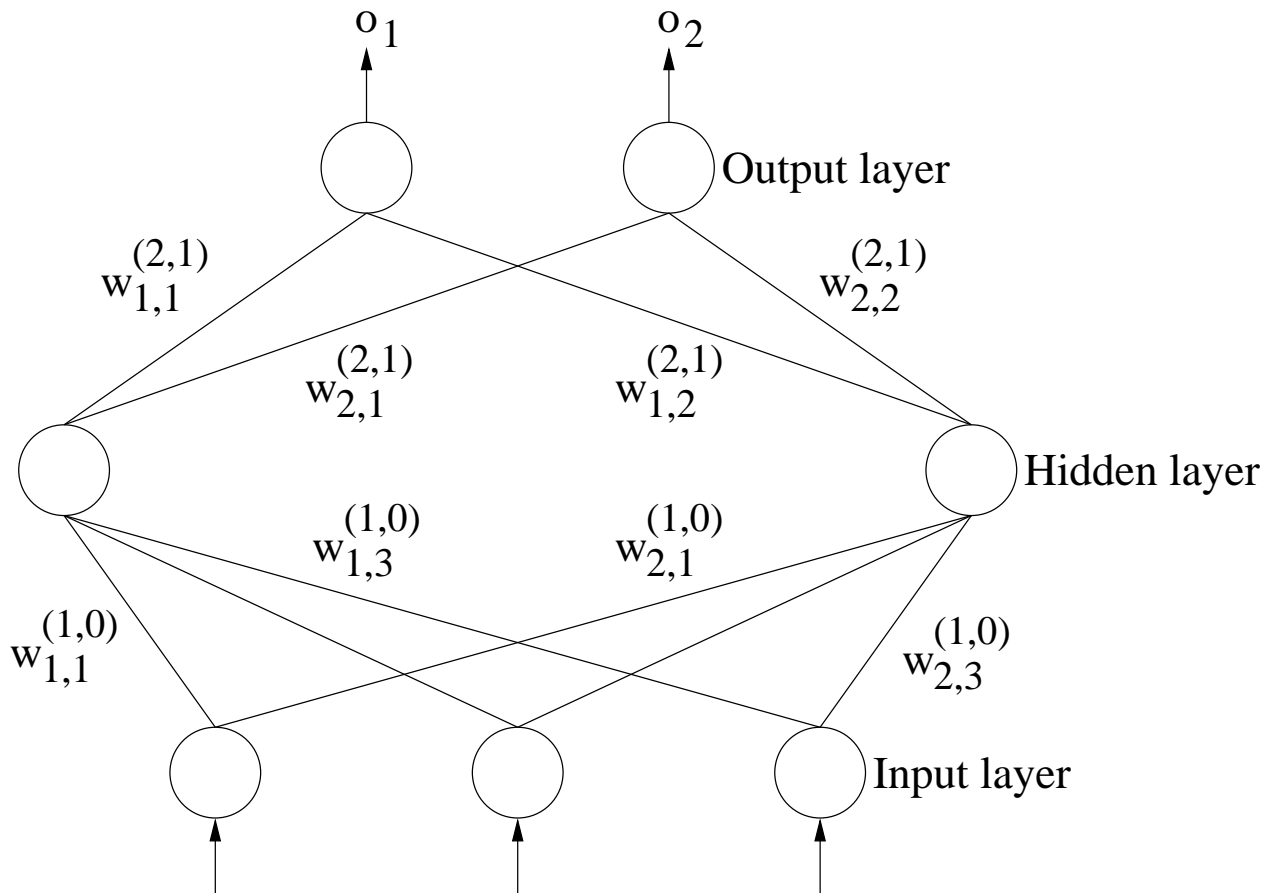


BACK-PROPAGATION NETWORKS

- Serious limitations of (single-layer) perceptrons:
 - Cannot learn non-linearly separable tasks
 - Cannot approximate (learn) non-linear functions
- Difficult (if not impossible) to design learning algorithms for multi-layer networks of perceptrons
- Solution:
 - Use multi-layer networks for non-linearly separable tasks
 - Use continuous differentiable non-linear activation functions
 - Solve the credit assignment problem

Multi-layer Perceptrons



- Activation function: sigmoid
- Error-correction learning
- Least mean square and gradient descent
- learning:
 - Forward pass
 - Backward pass

Preliminaries

- Training set: $\{(\vec{x}_p, \vec{d}_p), 1 \leq p \leq P\}$

P = number of input patterns

$$\vec{x}_p = (x_{p,0}, \dots, x_{p,N})$$

$$\vec{d}_p = (d_{p,1}, \dots, d_{p,K}) = \text{desired output for } \vec{x}_p$$

N = input space dimension

K = number of output neurons

$$\vec{o}_p = (o_{p,1}, \dots, o_{p,K}) = \text{actual output}$$

- Objectives: minimize cumulative error

$$Error = \sum_{p=1}^P Err(\vec{d}_p, \vec{o}_p)$$

- Err should be a metric (distance measure)

$$- Err(\vec{x}, \vec{y}) \geq 0$$

$$- Err(\vec{x}, \vec{y}) = 0 \text{ if } \vec{x} = \vec{y}$$

$$- Err(\vec{x}, \vec{y}) = Err(\vec{y}, \vec{x})$$

$$- Err(\vec{x}, \vec{y}) + Err(\vec{y}, \vec{z}) \geq Err(\vec{x}, \vec{z})$$

Preliminaries

(continuous)

- Popular choices based on norms of $\vec{d}_p - \vec{o}_p$:
 - $e_{p,j} = |d_{p,j} - o_{p,j}|$
 - $E_p = Err(\vec{d}_p, \vec{o}_p) = (e_{p,1}^u + \dots + e_{p,K}^u)^{\frac{1}{u}}, u > 0$
 - * $u = 1$: Manhattan distance
 - * $u = 2$: Euclidean distance
- Let $u = 2$, then cumulative error to minimize
 - Sum of Squared Error

$$SSE = \sum_{p=1}^P \sum_{j=1}^K e_{p,j}^2$$

- Mean Squared Error

$$MSE = \frac{1}{P} \frac{1}{K} \sum_{p=1}^P \sum_{j=1}^K e_{p,j}^2$$

Back-Propagation Algorithm

Scenario for One-Hidden-Layer Networks

1. $x_{p,i}$: value in i -th input node
2. $net_j^{(1)} = \sum_{i=0}^n w_{j,i}^{(1,0)} x_{p,i}$: net input of j -th node in hidden layer
3. $x_{p,j}^{(1)} = \sigma(\sum_{i=0}^n w_{j,i}^{(1,0)} x_{p,i})$: output of j -th in hidden layer
4. $net_k^{(2)} = \sum_j w_{k,j}^{(2,1)} x_{p,j}^{(1)}$: net input of k -th node in output layer
5. $o_{p,k} = \sigma(\sum_j w_{k,j}^{(2,1)} x_{p,j}^{(1)})$: output of k -th node in output layer
6. $e_{p,k}^2 = |d_{p,k} - o_{p,k}|^2$: squared error at k -th output node
7. $w_{k,j}^{(i+1,i)}$: weight from j -th node in i -th layer to k -th node $i + 1$ -th layer
8. $x_{p,j}^{(i)}$: output of j -th node of i -th layer for pattern \vec{x}_p

Back-Propagation Algorithm

(continued)

- $E_p = \sum_k e_{p,k}^2$: error for pattern \vec{x}_p
for simplicity write $E = \sum_{k=1}^K e_k^2$
(for now, we drop suffix p , for convenience)
- MSE is minimal when E is minimal for each pattern \vec{x}_p . Since o_k depends on the network weight \vec{w} , then E is also a function of \vec{w} . According to gradient descent, we the weight update

$$\Delta \vec{w} = -\frac{\partial E}{\partial \vec{w}}$$

That is

$$\Delta w_{k,j}^{(2,1)} = -\eta \frac{\partial E}{\partial w_{k,j}^{(2,1)}}$$

and

$$\Delta w_{j,i}^{(1,0)} = -\eta \frac{\partial E}{\partial w_{j,i}^{(1,0)}}$$

- How does E depends on the weights?

1. $E \leftarrow o_k \leftarrow net_k^{(2)} \leftarrow w_{k,j}^{(2,1)}$

2. $E \leftarrow o_k \leftarrow net_k^{(2)} \leftarrow x_j^{(1)} \leftarrow net_j^{(1)} \leftarrow w_{j,i}^{(1,0)}$

Back-Propagation Algorithm

(continued)

- By the chain rule, we have

$$1. \frac{\partial E}{\partial w_{k,j}^{(2,1)}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k^{(2)}} \cdot \frac{\partial net_k^{(2)}}{\partial w_{k,j}^{(2,1)}}$$

$$2. \frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^K \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k^{(2)}} \cdot \frac{\partial net_k^{(2)}}{\partial x_j^{(1)}} \cdot \frac{\partial x_j^{(1)}}{\partial net_j^{(1)}} \cdot \frac{\partial net_j^{(1)}}{\partial w_{j,i}^{(1,0)}}$$

After substitutions we obtain

$$\frac{\partial E}{\partial w_{k,j}^{(2,1)}} = -2(d_k - o_k) \cdot \sigma'(net_k^{(2)}) \cdot x_j^{(1)}$$

$$\frac{\partial E}{\partial w_{j,i}^{(1,0)}} = \sum_{k=1}^K -2(d_k - o_k) \cdot \sigma'(net_k^{(2)}) \cdot w_{k,j}^{(2,1)} \cdot \sigma'(net_j^{(1)}) \cdot x_i$$

- Applying gradient descent rule we have

$$1. \Delta w_{k,j}^{(2,1)} = \eta \cdot \delta_k \cdot x_j^{(1)}$$

$$2. \Delta w_{j,i}^{(1,0)} = \eta \cdot \mu_j \cdot x_i$$

$$\delta_k = (d_k - o_k) \cdot \sigma'(net_k^{(2)})$$
$$\mu_j = (\sum_k \delta_k w_{k,j}^{(2,1)}) \cdot \sigma'(net_j^{(1)})$$

Back-Propagation Algorithm

(continued)

- Changes in weights for the two layers are similar
 1. δ_k proportional to actual error $(d_k - o_k)$ multiplied by derivative of output node with respect to net input that node
 2. μ_j proportional to "weighted sum of errors" coming to the hidden node from node in upper layer
- We made no assumption about the activation function σ except it should be differentiable. For logistic sigmoid function we have

$$- \sigma(net) = \frac{1}{1 + e^{-\alpha net}}$$

$$- \sigma'(net) = \sigma(net) \cdot (1 - \sigma(net))$$

Therefore

$$1. \delta_k = (d_k - o_k) \cdot o_k(1 - o_k)$$

$$2. \mu_j = \sum_k \delta_k w_{k,j}^{(2,1)} \cdot x_j^{(1)}(1 - x_j^{(1)})$$

Back-Propagation Algorithm

(continued)

Start with initial random \vec{w} ;

Repeat

For each input pattern \vec{x}_p do

{Propagate \vec{x}_p (forward pass), that is:}

From first hidden layer to output layer do

Compute hidden node activations: $net_{p,j}^{(1)}$;

Compute hidden node outputs: $x_{p,j}^{(1)}$;

Compute output node activations: $net_{p,k}^{(2)}$;

Compute network outputs: $o_{p,k}$;

Compute network's error $e_{p,k} = d_{p,k} - o_{p,k}$;

{Back-propagate \vec{e}_p (backward pass), that is:}

From output layer to first hidden layer do

For the output layer do

Update output layer weights:

$$\delta_{p,k} = (d_{p,k} - o_{p,k}) \cdot o_{p,k} \cdot (1 - o_{p,k});$$

$$\Delta w_{k,j}^{(2,1)} = \eta \cdot \delta_{p,k} \cdot x_{p,j}^{(1)};$$

For a hidden layer do

Update hidden layer weights:

$$\mu_{p,j} = \sum_k \delta_{p,k} w_{k,j}^{(2,1)} \cdot x_{p,j}^{(1)} \cdot (1 - x_{p,j}^{(1)});$$

$$\Delta w_{j,i}^{(1,0)} = \eta \cdot \mu_{p,j} \cdot x_{p,i};$$

Until $MSE(\vec{w})$ is minimal;

General Multiclass Classification Problem

- K classes: C_1, \dots, C_K

n_k examples of class C_k

$$T_k = \{(\vec{x}_p^k, \vec{d}_p^k) : 1 \leq p \leq n_k, 1 \leq k \leq K\}$$

Training set: $T = T_1 \cup \dots \cup T_K$

- Output representation:

1. $\log_2 K$ output nodes: (bad)

– Training is difficult since many output nodes must be *high* simultaneously.

– *Cross-talk* phenomenon: different training examples require conflicting changes to be made to the same weight.

2. K output nodes: (better)

– One output node assigned to one class.

– Each output node focus only on learning one class rather than performing multiple duties.

3. Error-correcting output code: (best)

– Minimize cross-talk

General Multiclass Classification Problem

(continued)

- Desired output representation:
 1. High weight magnitudes for output 0 or 1
 2. $|\sigma'(net)| \rightarrow 0$ when $|net| \rightarrow +\infty$
 3. Desired output $\vec{d}^k \mapsto (\varepsilon, \dots, \varepsilon, 1 - \varepsilon, \varepsilon, \dots, \varepsilon)$, where $\varepsilon > 0$ (typical choices are 0.01 and 0.1)
- Perfect classification is possible even if the error $|d_{p,j} - o_{p,j}| \neq 0$ (in absolute value)
 1. $d_{p,j} = (1 - \varepsilon)$ and $o_{p,j} \geq d_{p,j}$ then $e_{p,j} = 0$
 2. $d_{p,j} = \varepsilon$ and $o_{p,j} \leq d_{p,j}$ then $e_{p,j} = 0$
 3. Otherwise $e_{p,j} = |d_{p,j} - o_{p,j}|$
- Class membership of \vec{x}_p after training
 1. Assign \vec{x}_p to that class k for which $\|\vec{d}^k - \vec{o}_p\| \leq \|\vec{d}^j - \vec{o}_p\|$ for $j \neq k$ where $1 \leq j \leq K$
 2. Assign \vec{x}_p to class k if $o_{k,p} > o_{j,p}$ for all $j \neq k$ where $1 \leq j \leq K$

Heuristic Modifications of Back-Propagation

- Frequency of weight updates

1. *Sequential learning*

- Weights are updated after each example presentation
- Slower but uses less memory
- Easy to implement
- Local minimization → less ability to escape local optimum

2. *Batch learning*: $\Delta w = \sum_{p=1}^P \Delta w_p$

- Weights are updated after each epoch
- Faster but uses more memory
- Can be parallelized
- Global minimization → better ability to escape local optimum

- It is good practice to *randomize the order of presentation of training examples* from epoch to the next.

Heuristic Modifications of Back-Propagation

(continued)

- *Maximizing information content*: Select examples that have the largest possible information content for the learning problem
 1. Use samples that result in largest training error
 2. Use radically different samples in training
 3. \Rightarrow Use random presentation
 4. \Rightarrow Use *emphasizing scheme*
- *Activation function*
 1. Should be *Antisymmetric*: $\sigma(-y) = -\sigma(y)$
Ex: *hyperbolic tangent* $\sigma(y) = a \tanh(by)$
 2. Bipolar representation $(-1, 1)$ vs. binary $(0, 1)$
 - (a) Weights are *always* updated
 - (b) Noise representation
 3. Faster learning and better generalizability

Heuristic Modifications of Back-Propagation (continued)

- *Normalizing the inputs*

1. Weights should be updated at approximately the *same speed*
2. Prevent network bias toward particular inputs

- *Initialization of weights*

1. Random values: $-1 \leq w_i \leq +1$
2. Normalized
3. Averaged:

$$w_{j,i}^{(1,0)} = \pm \frac{1}{P} \sum_{p=1}^P \frac{1}{|x_i|}$$

$$w_{k,j}^{(2,1)} = \pm \frac{1}{P} \sum_{p=1}^P \frac{1}{\sigma(\text{net}_j^{(1)})}$$

or

$$w_{j,i}(\text{new}) = \frac{w_{j,i}(\text{old})}{\|\bar{w}_j(\text{old})\|}$$

where $\bar{w}_j(\text{old})$ denotes the average weight, computed over all values of i .

Heuristic Modifications of Back-Propagation (continued)

- *Choice of learning rate:*
 1. Constant rate η
 2. A rate η_i for each w_i
 3. Start with large η (or η_i) and decrease steadily
 4. At each iteration where
 - (a) Performance improves: increase η (or η_i)
 - (b) Performance worsens: decrease η (or η_i)
 5. Double η (or η_i) until performance worsens

$$i := 0; \vec{w}_{new} := \vec{w} - E'(\vec{w})\epsilon;$$

While $E(\vec{w}_{new}) < E(\vec{w})$ do

$$j := j + 1; \vec{w} := \vec{w}_{new};$$

$$\vec{w}_{new} := \vec{w} - E'(\vec{w})2^j\epsilon;$$

End-While

$$\eta \text{ (or } \eta_i) := 2^{j-1}\epsilon;$$

Searches for a large enough rate ($2^j\epsilon$) at which network's error no longer decreases. (We assume $\epsilon > 0$ such that $E(\vec{w} - E'(\vec{w})\epsilon) < E(\vec{w})$)

6. Make use of the second derivative of MSE

Heuristic Modifications of Back-Propagation (continued)

- *Momentum*

1. $\Delta\vec{w}$ grows in magnitude when $\frac{\partial E}{\partial\vec{w}}$ has same sign on consecutive iterations: therefore *accelerate descent* for faster learning
2. $\Delta\vec{w}$ shrinks in magnitude when $\frac{\partial E}{\partial\vec{w}}$ has opposite sign on consecutive iterations: therefore *stabilize* to avoid oscillation
3. 1 and 2 can be achieved by adding a *momentum term* in the learning rule:

$$\Delta w_{k,j}(t+1) = \eta\delta_k x_j + \alpha\Delta w_{k,j}(t)$$

where momentum α : $0 < \alpha \leq 1$

4. Momentum term has an averaging effect: weights "*move*" in the "*general*" (or "*average*") direction of motion.

Heuristic Modifications of Back-Propagation

(continued)

- *Generalizability*

1. Neural networks are useless if they don't generalize
2. Design of Neural Network systems:
 - (a) Training and Testing
 - (b) Choice of training and testing set are very important
 - (c) NN should give good training and test results
3. When to stop training?
 - When testing error start to worsen
 - Otherwise: *Overfitting* occurs: NN only memorizes the input, it doesn't generalize
 - Small training error is acceptable when sampling is good
4. Small networks achieve better for generalizability