Slide04

Haykin Chapter 4 (both 2nd and 3rd ed.): Multi-Layer Perceptrons

CPSC 636-600
Instructor: Yoonsuck Choe
Spring 2012

Some materials from this lecture are from Mitchell (1997) *Machine Learning*, McGraw-Hill.

Introduction

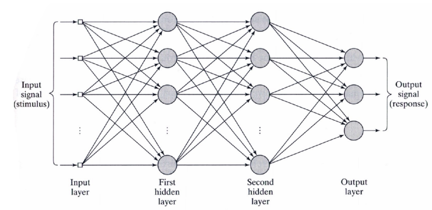


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

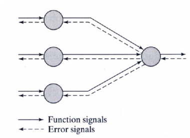


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

- Networks typically consisting of input, hidden, and output layers.
- Commonly referred to as *Multilayer perceptrons*.
- Popular learning algorithm is the *error backpropagation algorithm* (backpropagation, or backprop, for short), which is a generalization of the LMS rule.
  - Forward pass: activate the network, layer by layer
  - Backward pass: error signal backpropagates from output to hidden and hidden to input, based on which weights are updated.

Multilayer Perceptrons: Characteristics

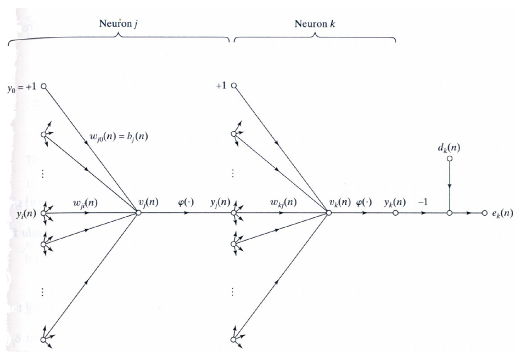


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j.

- Each model neuron has a *nonlinear activation function*, typically a *logistic function*: $y_j = \frac{1}{1+\exp(-v_j)}$
- Network contains one or more *hidden layers* (layers that are not either an input or an output layer).
- Network exhibits a high degree of *connectivity*.

Multilayer Networks

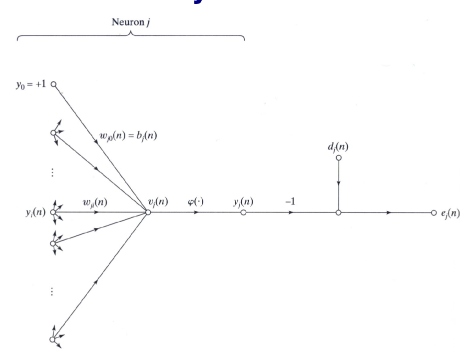
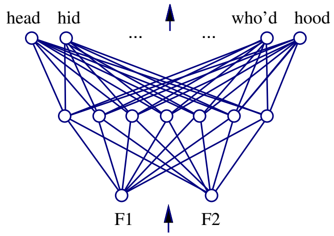
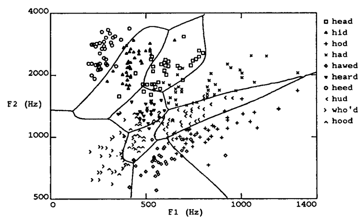


FIGURE 4.3 Signal-flow graph highlighting the details of output neuron j.

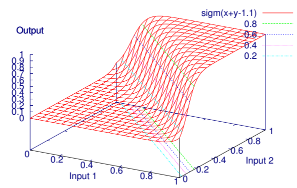
- Differentiable threshold unit: **sigmoid** $\phi(v) = \frac{1}{1+\exp(-v)}$. Interesting property: $\frac{d\phi(v)}{dv} = \phi(v)(1 - \phi(v))$.
- Output: $y = \phi(\mathbf{x}^T\mathbf{w})$
- Other functions: $\tanh(v) = \frac{1-\exp(-2v)}{1+\exp(-2v)}$

Multilayer Networks and Backpropagation

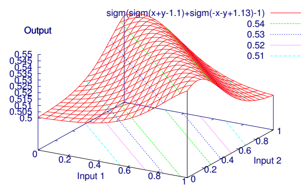




- Nonlinear decision surfaces.



(a) One output



(b) Two hidden, one output

- Another example: XOR

Error Gradient for a Single Sigmoid Unit

For n input-output pairs $\{(\mathbf{x}_k, d_k)\}_{k=1}^{n}$:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_k (d_k - y_k)^2 \\
&= \frac{1}{2} \sum_k \frac{\partial}{\partial w_i} (d_k - y_k)^2 \\
&= \frac{1}{2} \sum_k 2(d_k - y_k) \frac{\partial}{\partial w_i} (d_k - y_k) \\
&= \sum_k (d_k - y_k) \left(-\frac{\partial y_k}{\partial w_i} \right) \\
&= -\sum_k (d_k - y_k) \underbrace{\frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_i}}_{\text{Chain rule}}
\end{aligned}$$

Error Gradient for a Sigmoid Unit

From the previous page:

$$\frac{\partial E}{\partial w_i} = -\sum_k (d_k - y_k) \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_i}$$

But we know:

$$\frac{\partial y_k}{\partial v_k} = \frac{\partial \phi(v_k)}{\partial v_k} = y_k(1 - y_k)$$

$$\frac{\partial v_k}{\partial w_i} = \frac{\partial (\mathbf{x}_k^T \mathbf{w})}{\partial w_i} = x_{i,k}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_k (d_k - y_k) y_k (1 - y_k) x_{i,k}$$

Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied, Do

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit j
     $\delta_j \leftarrow y_j(1 - y_j)(d_j - y_j)$
  3. For each hidden unit h
     $\delta_h \leftarrow y_h(1 - y_h) \sum_{j \in outputs} w_{jh} \delta_j$
  4. Update each network weight $w_{i,j}$
     $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$ where
     $\Delta w_{ji} = \eta \delta_j x_i$.

Note: w_{ji} is the weight from i to j (i.e., $w_{j \leftarrow i}$).

The δ Term

- For output unit:

$$\delta_j \leftarrow \underbrace{y_j(1-y_j)}_{\phi'(v_j)}\underbrace{(d_j-y_j)}_{\text{Error}}$$

- For hidden unit:

$$\delta_h \leftarrow \underbrace{y_h(1-y_h)}_{\phi'(v_h)}\underbrace{\sum_{j\in outputs} w_{jh}\delta_j}_{\text{Backpropagated error}}$$

- In sum, δ is the derivative times the error.
- Derivation to be presented later.

Derivation of Δw

- Want to update weight as:

$$\Delta w_{ji} = -\eta\frac{\partial E}{\partial w_{ji}},$$

where error is defined as

$$E(\mathbf{w}) \equiv \frac{1}{2}\sum_{j\in outputs}(d_j-y_j)^2$$

- Given $v_j = \sum_j w_{ji}x_i$,

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}}$$

- Different formula for output and hidden.

Derivation of Δw: Output Unit Weights

From the previous page, $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}}$

- First, calculate $\frac{\partial E}{\partial v_j}$:

$$\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$

$$\begin{aligned}
\frac{\partial E}{\partial y_j} &= \frac{\partial}{\partial y_j}\frac{1}{2}\sum_{j\in outputs}(d_j-y_j)^2 \\
&= \frac{\partial}{\partial y_j}\frac{1}{2}(d_j-y_j)^2 \\
&= 2\frac{1}{2}(d_j-y_j)\frac{\partial(d_j-y_j)}{\partial y_j} \\
&= -(d_j-y_j)
\end{aligned}$$

Derivation of Δw: Output Unit Weights

From the previous page, $\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -(d_j-y_j)\frac{\partial y_j}{\partial v_j}$:

- Next, calculate $\frac{\partial y_j}{\partial v_j}$: Since $y_j = \phi(v_j)$, and $\phi'(v_j) = y_j(1-y_j)$,

$$\frac{\partial y_j}{\partial v_j} = y_j(1-y_j).$$

Putting everything together,

$$\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -(d_j-y_j)y_j(1-y_j).$$

Derivation of Δw: Output Unit Weights

From the previous page:

$$\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -(d_j - y_j)y_j(1 - y_j).$$

Since $\frac{\partial v_j}{\partial w_{ji}} = \frac{\partial \sum_{i'} w_{ji'}x_{i'}}{\partial w_{ji}} = x_i,$

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}} &= \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \\ &= -\underbrace{(d_j - y_j)y_j(1 - y_j)}_{\delta_j = error \times \phi'(net)}\ \underbrace{x_i}_{input}\end{aligned}$$

Derivation of Δw: Hidden Unit Weights

Start with $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}x_i$:

$$\begin{aligned}\frac{\partial E}{\partial v_j} &= \sum_{k \in Downstream(j)} \frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial v_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial v_k}{\partial v_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial v_k}{\partial y_j}\frac{\partial y_j}{\partial v_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial y_j}{\partial v_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj}\ \underbrace{y_j(1 - y_j)}_{\phi'(net)} \qquad (1)\end{aligned}$$

Derivation of Δw: Hidden Unit Weights

Finally, given

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}x_i,$$

and

$$\frac{\partial E}{\partial v_j} = \sum_{k \in Downstream(j)} -\delta_k w_{kj}\ \underbrace{y_j(1 - y_j)}_{\phi'(net)},$$

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = \eta\ \underbrace{\left[\underbrace{y_j(1 - y_j)}_{\phi'(net)}\ \underbrace{\sum_{k \in Downstream(j)} \delta_k w_k j}_{error}\right]}_{\delta_j} x_i$$

Summary

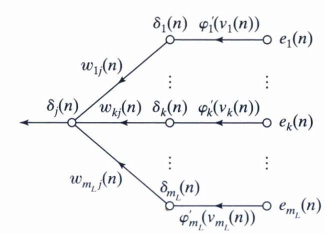
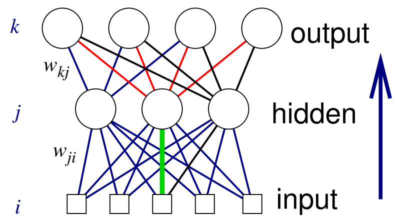


FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

$$\underbrace{\Delta w_{ji}(n)}_{\text{weight correction}} = \underbrace{\eta}_{\text{learning rate}} \cdot \underbrace{\delta_j(n)}_{\text{local gradient}} \cdot \underbrace{y_i(n)}_{\text{input signal}}$$

Extension to Different Network Topologies



- Arbitrary number of layers: for neurons in layer m:

$$\delta_r = y_r(1 - y_r) \sum_{s \in layer\ m+1} w_{sr}\delta s.$$

- Arbitrary acyclic graph:

$$\delta_r = y_r(1 - y_r) \sum_{s \in Downstream(r)} w_{sr}\delta s.$$

Backpropagation: Properties

- Gradient descent over entire *network* weight vector.
- Easily generalized to arbitrary directed graphs.
- Will find a local, not necessarily global error minimum:
  - In practice, often works well (can run multiple times with different initial weights).
- Minimizes error over *training* examples:
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using the network after training is very fast.

Learning Rate and Momentum

- Tradeoffs regarding learning rate:
  - Smaller learning rate: smoother trajectory but slower convergence
  - Larger learning rate: fast convergence, but can become unstable.
- Momentum can help overcome the issues above.

$$\Delta w_{ji}(n) = \eta\delta_j(n)y_i(n) + \alpha\Delta w_{ji}(n-1).$$

The update rule can be written as:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t}\delta_j(t)y_i(t) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}.$$

Momentum (cont'd)

$$\Delta w_{ji}(n) = \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

- The weight vector is the sum of an exponentially weighted time series.
- Behavior:
  - When successive $\frac{\partial E(t)}{\partial w_{ji}(t)}$ take the same sign: Weight update is accelerated (speed up downhill).
  - When successive $\frac{\partial E(t)}{\partial w_{ji}(t)}$ have different signs: Weight update is damped (stabilize oscillation).

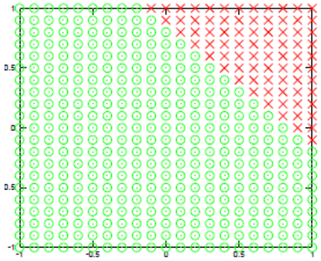
Sequential (online) vs. Batch Training

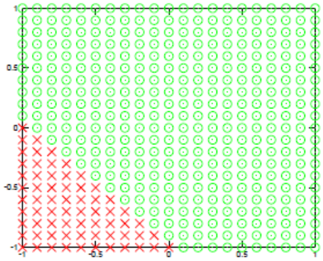
- Sequential mode:
  - Update rule applied after each input-target presentation.
  - Order of presentation should be randomized.
  - Benefits: less storage, stochastic search through weight space helps avoid local minima.
  - Disadvantages: hard to establish theoretical convergence conditions.
- Batch mode:
  - Update rule applied after all input-target pairs are seen.
  - Benefits: accurate estimate of the gradient, convergence to local minimum is guaranteed under simpler conditions.

Representational Power of Feedforward Networks

- Boolean functions: every boolean function representable with two layers (hidden unit size can grow exponentially in the worst case: one hidden unit per input example, and "OR" them).
- Continous functions: Every **bounded** continuous function can be approximated with an arbitrarily small error (output units are linear).
- Arbitrary functions: with three layers (output units are linear).

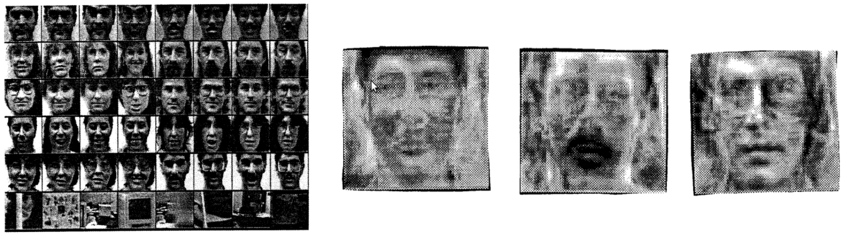
What the Hidden Layer Does





- A smooth ramped output, monotonically increasing.
- Ramp can be oriented in different angles.
- This kind of visualization is only possible with low-dimensional input.

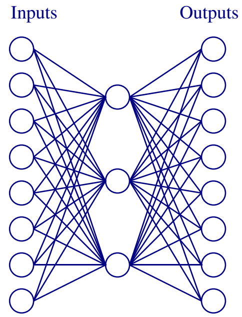
What the Hidden Layer Does (cont'd)



Fleming and Cottrell (1990)

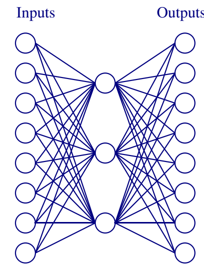
- We can also look at the hidden layer weight as a pattern or feature.
- Or, we can activate one hidden unit and see what output pattern it produces (example above).

Learning Hidden Layer Representations



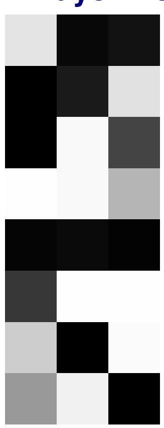
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Learned Hidden Layer Representations



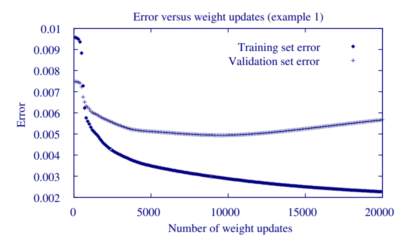
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

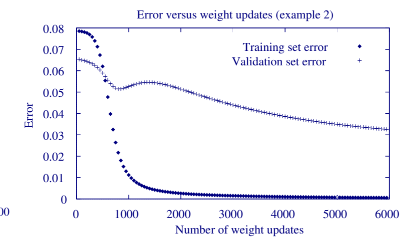
Learned Hidden Layer Representations



- Learned encoding is similar to standard 3-bit binary code.
- Automatic discovery of **useful hidden layer representations** is a key feature of ANN.
- Note: The hidden layer representation is **compressed**.

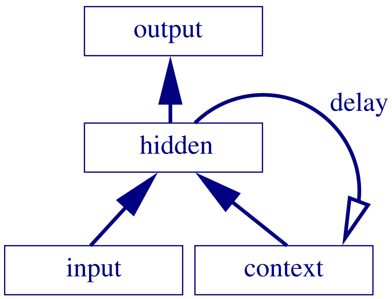
Overfitting





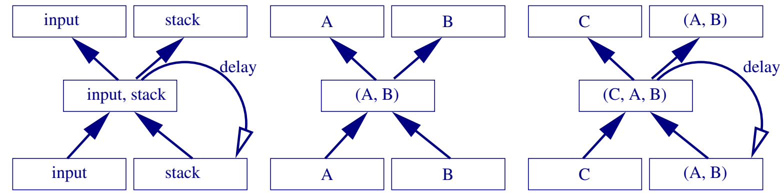
- Error in two different robot perception tasks.
- Training set and validation set error.
- Early stopping ensures good performance on unobserved samples, but must be careful.
- Weight decay, use of validation sets, use of k-fold cross-validation, etc. to overcome the problem.

Recurrent Networks



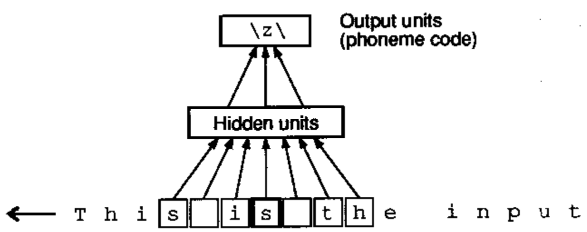
- Sequence recognition.
- Store tree structure (next slide).
- Can be trained with plain backpropagation.
- Generalization may not be perfect.

Recurrent Networks (Cont'd)



- Autoassociation (intput = output)
- Represent a stack using the hidden layer representation.
- Accuracy depends on numerical precision.

Some Applications: NETtalk



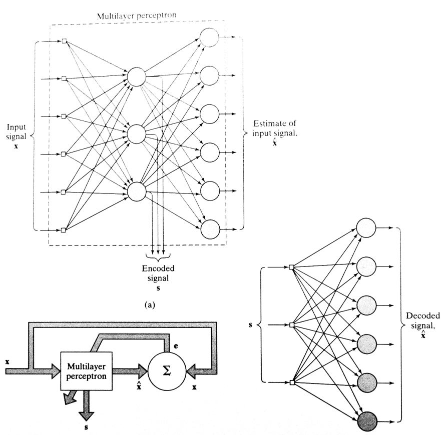
- NETtalk: Sejnowski and Rosenberg (1987).
- Learn to pronounce English text.
- Demo
- Data available in UCI ML repository

NETtalk data

```
aardvark a-rdvark 1<<<>2<<0
aback xb@k-0>1<<0
abacus @bxkxs 1<0>0<0
abaft xb@ft 0>1<<0
abalone @bxloni 2<0>1>0 0
abandon xb@ndxn 0>1<>0<0
abase xbes-0>1<<0
abash xb@S-0>1<<0
abate xbet-0>1<<0
abatis @bxti-1<0>2<2
...
```

- Word – Pronunciation – Stress/Syllable
- about 20,000 words

More Applications: Data Compression



- Construct an autoassociative memory where Input = Output.
- Train with small hidden layer.
- Encode using input-to-hidden weights.
- Send or store hidden layer activation.
- Decode received or stored hidden layer activation with the hidden-to-output weights.

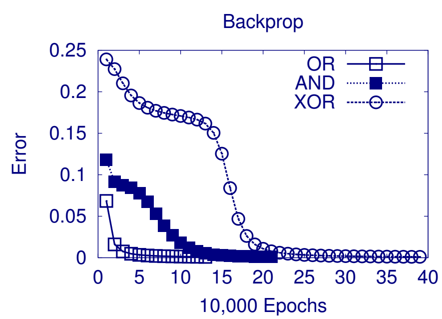
Backpropagation Exercise

- **URL:** http://www.cs.tamu.edu/faculty/choe/src/backprop-1.6.tar.gz
- Untar and read the `README` file:

```
gzip -dc backprop-1.6.tar.gz | tar
xvf -
```

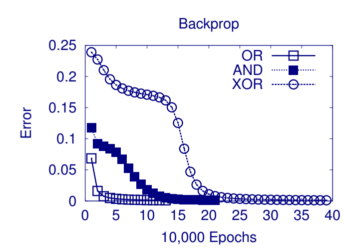
- Run `make` to build (on departmental unix machines).
- Run `./bp conf/xor.conf` etc.

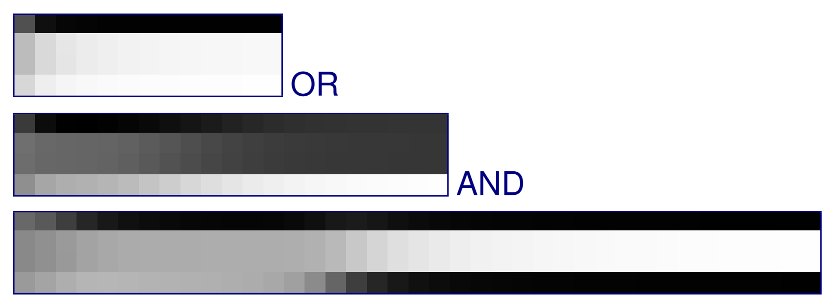
Backpropagation: Example Results



- Epoch: one full cycle of training through all training input patterns.
- `OR` was easiest, `AND` the next, and `XOR` was the most difficult to learn.
- Network had 2 input, 2 hidden and 1 output unit. Learning rate was 0.001.

Backpropagation: Example Results (cont'd)





OR

AND

XOR

Output to (0,0), (0,1), (1,0), and (1,1) form each row.

Backpropagation: Things to Try

- How does increasing the number of hidden layer units affect the (1) time and the (2) number of epochs of training?
- How does increasing or decreasing the learning rate affect the rate of convergence?
- How does changing the slope of the sigmoid affect the rate of convergence?
- Different problem domains: handwriting recognition, etc.

MLP as a General Function Approximator

- MLP can be seen as performing **nonlinear input-output mapping**.
- **Universal approximation theorem:** Let $\phi(\cdot)$ be a nonconstant, bounded, monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0, 1]^{m_0}$. The space of *continuous functions* on I_{m_0} is denoted by $C(I_{m_0})$. Then given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exists an integer m_1 and a set of real constants α_i, b_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$, such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \phi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

as an approximate realization of the function $f(\cdot)$; that is

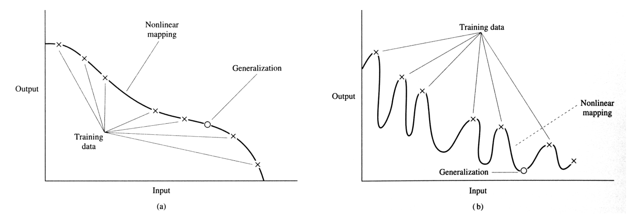
$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon$$

for all $x_1, \ldots, x_{m_0}$ that lie in the input space.

MLP as a General Function Approximator (cont'd)

- The *universal approximation theorem* is an *existence* theorem, and it merely generalizes approximations by finite Fourier series.
- The *universal approximation theorem* is directly applicable to neural networks (MLP), and it implies that one hidden layer is sufficient.
- The theorem does not say that a single hidden layer is optimum in terms of learning time, generalization, etc.

Generalization



- A network is said to *generalize* well when the input-output mapping computed by the network is correct (or nearly so) for test data never used during training.
- This view is apt when we take the *curve-fitting* view.
- Issues: **overfitting** or **overtraining**, due to memorization. *Smoothness* in the mapping is desired, and this is related to criteria like *Occam's razor*.

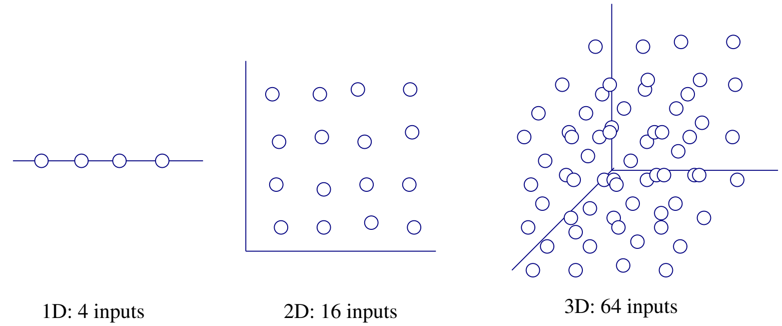
Generalization and Training Set Size

- Generalization is influenced by three factors:
  - Size of the training set, and how representative they are.
  - The architecture of the network.
  - Physical complexity of the problem.
- Sample complexity and VC dimension are related. In practice,

$$N = O\left(\frac{W}{\epsilon}\right),$$

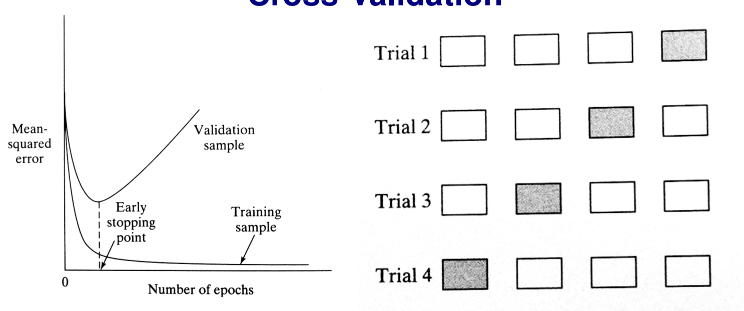
  where W is the total number of free parameters, and ϵ is the error tolerance.

Training Set Size and Curse of Dimensionality



- As the dimensionality of the input grows, exponentially more inputs are needed to maintain the same density in unit space.
- In other words, the **sampling density** of N inputs in m-dimensional space is proportional to $N^{1/m}$.
- One way to overcome this is to use *prior knowledge* about the function.

Cross-Validation



Use of **validation set** (not used during training, used for measuring generalizability).

- **Model selection**
- **Early stopping**
- **Hold-out method**: multiple cross-validation, leave-one-out method, etc.

Virtues and Limitations of Backprop

- **Connectionism**: biological metaphor, local computation, graceful degradation, paralellism. (Some limitations exist regarding the biological plausibility of backprop.)
- **Feature detection**: hidden neurons perform feature detection.
- **Function approximation**: a form of *nested sigmoid*.
- **Computational complexity**: computation is *polynomial* in the number of adjustable parameters, thus it can be said to be *efficient*.
- **Sensitivity analysis**: sensitivity $S_\omega^F = \frac{\partial F/F}{\partial \omega/\omega}$ can be estimated efficiently.
- **Robustness**: disturbances can only cause small estimation errors.
- **Convergence**: stochastic approximation, and it can be slow.
- **Local minima** and **scaling issues**

Heuristic for Accelerating Convergence

Learning rate adaptation

- Separate learning rate for each tunable weight.
- Each learning rate is allowed to adjust after each iteration.
- If the derivative of the cost function has the same sign for several iterations, increase the learning rate.
- If the derivative of the cost function alternates the sign over several iterations, decrease the learning rate.

Summary

- Backprop for MLP is **local** and **efficient** (in calculating the partial derivative).
- Backprop can handle **nonlinear** mappings.