MultiLayer Neural Networks

Xiaogang Wang

xgwang@ee.cuhk.edu.hk

January 15, 2019

Outline

History of neural network

- Pioneering work on the mathematical model of neural networks
  - McCulloch and Pitts 1943
  - Include recurrent and non-recurrent (with “circles”) networks
  - Use thresholding function as nonlinear activation
  - No learning
- Early works on learning neural networks
  - Starting from Rosenblatt 1958
  - Using thresholding function as nonlinear activation prevented computing derivatives with the chain rule, and so errors could not be propagated back to guide the computation of gradients
- Backpropagation was developed in several steps since 1960
  - The key idea is to use the chain rule to calculate derivatives
  - It was reflected in multiple works, earliest from the field of control

History of neural network

- Standard backpropagation for neural networks
  - Rumelhart, Hinton, and Williams, *Nature* 1986. Clearly appreciated the power of backpropagation and demonstrated it on key tasks, and applied it to pattern recognition generally
  - In 1985, Yann LeCun independently developed a learning algorithm for three-layer networks in which target values were propagated, rather than derivatives. In 1986, he proved that it was equivalent to standard backpropagation
- Prove the universal expressive power of three-layer neural networks
  - Hecht-Nielsen 1989
- Convolutional neural network
  - Introduced by Kunihiko Fukushima in 1980
  - Improved by LeCun, Bottou, Bengio, and Haffner in 1998

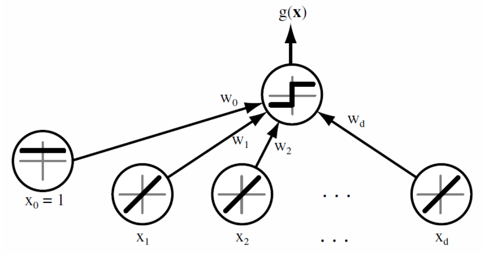
History of neural network

- Deep belief net (DBN)
  - Hinton, Osindero, and Tech 2006
- Auto encoder
  - Hinton and Salakhutdinov 2006 (*Science*)
- Deep learning
  - Hinton. Learning multiple layers of representations. *Trends in Cognitive Sciences*, 2007.
  - Unsupervised multilayer pre-training + supervised fine-tuning (BP)
- Large-scale deep learning in speech recognition
  - Geoff Hinton and Li Deng started this research at Microsoft Research Redmond in late 2009.
  - Generative DBN pre-training was not necessary
  - Success was achieved by large-scale training data + large deep neural network (DNN) with large, context-dependent output layers

History of neural network

- Unsupervised deep learning from large scale images
  - Andrew Ng et al. 2011
  - Unsupervised feature learning
  - 16000 CPUs
- Large-scale supervised deep learning in ImageNet image classification
  - Krizhevsky, Sutskever, and Hinton 2012
  - Supervised learning with convolutional neural network
  - No unsupervised pre-training

Two-layer neural networks model linear classifiers



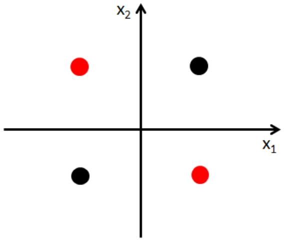
(Duda et al. Pattern Classification 2000)

$$g(\mathbf{x}) = f(\sum_{i=1}^{d} x_i w_i + w_0) = f(\mathbf{w}^t \mathbf{x})$$

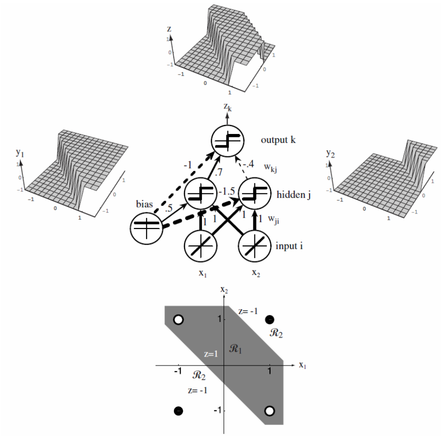
$$f(s) = \begin{cases} 1, & \text{if } s \geq 0 \\ -1, & \text{if } s < 0 \end{cases}.$$

Two-layer neural networks model linear classifiers

- A linear classifier cannot solve the simple exclusive-OR problem

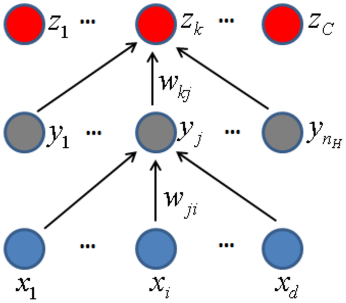


Add a hidden layer to model nonlinear classifiers



(Duda et al. Pattern Classification 2000)

Three-layer neural network



For C-class classification problem, the target vectors are represented as

$$(1 \dots 0 \dots 0)$$
$$\dots$$
$$(0 \dots 1 \dots 0)$$
$$\dots$$
$$(0 \dots 0 \dots 1)$$

Three-layer neural network

- Net activation: each hidden unit *j* computes the weighted sum of its inputs

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \mathbf{w}_j^t \mathbf{x}$$

- Activation function: each hidden unit emits an output that is a **nonlinear** function of its activation

$$y_j = f(net_j)$$

$$f(net) = Sgn(net) = \begin{cases} 1, & \text{if } net \geq 0 \\ -1, & \text{if } net < 0 \end{cases}.$$

There are multiple choices of the activation function as long as they are continuous and differentiable **almost everywhere**. Activation functions could be different for different nodes.

Three-layer neural network

- Net activation of an output unit k

$$net_k = \sum_{i=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^t \mathbf{y}$$

- Output unit emits

$$z_k = f(net_k)$$

- The output of the neural network is equivalent to a set of discriminant functions

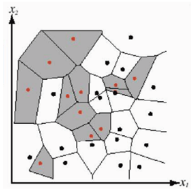
$$g_k(\mathbf{x}) = z_k = f\left(\sum_{j=1}^{n_H} w_{kj} f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right)$$

Expressive power of a three-layer neural network

- It can represent **any** discriminant function
- However, the number of hidden units required can be very large...
- Most widely used pattern recognition models (such as SVM, boosting, and KNN) can be approximated as neural networks with one or two hidden layers. They are called models with shallow architectures.
- Shallow models divide the feature space into regions and match templates in local regions. $O(N)$ parameters are needed to represent N regions.
- Deep architecture: the number of hidden nodes can be reduced exponentially with more layers for certain problems.

Expressive power of a three-layer neural network

KNN

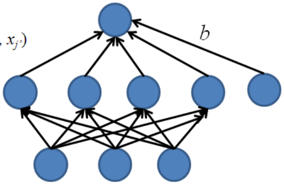


output $g(x) = label_j$

$j = \text{argmin}_{j'}\, d(x, x_{j'})$

hidden $d(x, x_j)$

input x

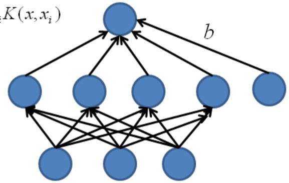


SVM

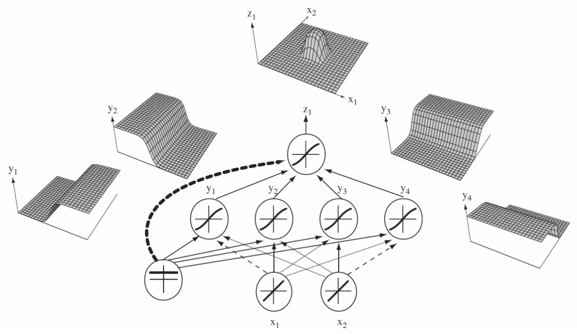
output $g(x) = b + \sum_i \alpha_i K(x, x_i)$

hidden $K(x, x_j)$

input x



Expressive power of a three-layer neural network



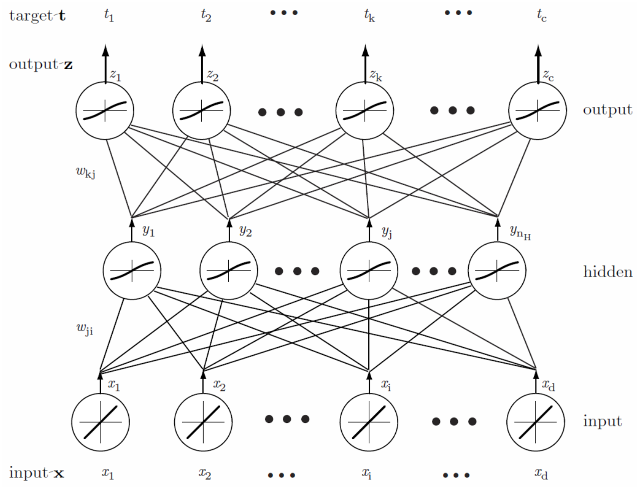
(Duda et al. Pattern Classification 2000)

With a tanh activation function $f(s) = (e^s - e^{-s})/(e^s + e^{-s})$, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. With four hidden units, a local mode (template) can be modeled. Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network.

Backpropagation

- The most general method for supervised training of multilayer neural network
- Present an input pattern and change the network parameters to bring the actual outputs closer to the target values
- Learn the input-to-hidden and hidden-to-output weights
- However, there is no explicit teacher to state what the hidden unit's output should be. Backpropagation calculates an effective error for each hidden unit, and thus derive a learning rule for the input-to-hidden weights.

A three-layer network for illustration



$$z_k = f(net_k)$$

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0}$$

$$y_j = f(net_j)$$

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0}$$

(Duda et al. Pattern Classification 2000)

Training error

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}||\mathbf{t} - \mathbf{z}||^2$$

- Differentiable
- There are other choices, such as cross entropy

$$J(\mathbf{w}) = -\sum_{k=1}^{c} t_k \log(z_k)$$

Both $\{z_k\}$ and $\{t_k\}$ are probability distributions.

Gradient descent

- Weights are initialized with random values, and then are changed in a direction reducing the error

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}},$$

  or in component form

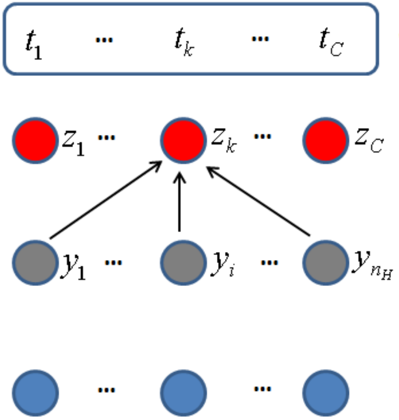
$$\Delta w_{pq} = -\eta \frac{\partial J}{\partial w_{pq}}$$

  where η is the learning rate.

- Iterative update

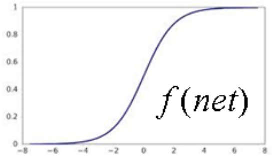
$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta \mathbf{w}(m)$$

Hidden-to-output weights w_{kj}


t_1 ... t_k ... t_C
Group truth
z_1 ... z_k ... z_C
y_1 ... y_i ... y_{n_H}
...
...

$$z_k = f(net_k)$$

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0}$$



Hidden-to-output weights w_{kj}

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

- Sensitivity of unit k

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k)$$

  Describe how the overall error changes with the unit's net activation.

- Weight update rule. Since $\partial net_k / \partial w_{kj} = y_j$,

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j.$$

Activation function

- Sign function is not a good choice for $f(\cdot)$. Why?
- Popular choice of $f(\cdot)$
  - Sigmoid function
    $$f(s) = \frac{1}{1+e^{-s}}$$
  - Tanh function (shift the center of Sigmoid to the origin)
    $$f(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$
  - Hard thanh
    $$f(s) = \max(-1, \min(1, x))$$
  - Rectified linear unit (ReLU)
    $$f(s) = \max(0, x)$$
  - Softplus: smooth version of ReLU
    $$f(s) = \log(1 + e^s)$$

Activation function

- Popular choice of $f(\cdot)$
  - Softmax: mostly used as output non-linearrity for predicting discrete probabilities

$$f(s_k) = \frac{e^{s_k}}{\sum_{k'=1}^{C} e^{s_{k'}}}$$

  - Maxout: it generalizes the rectifier assuming there are multiple net activations

$$f(s_1, \ldots, s_n) = \max_i(s_i)$$

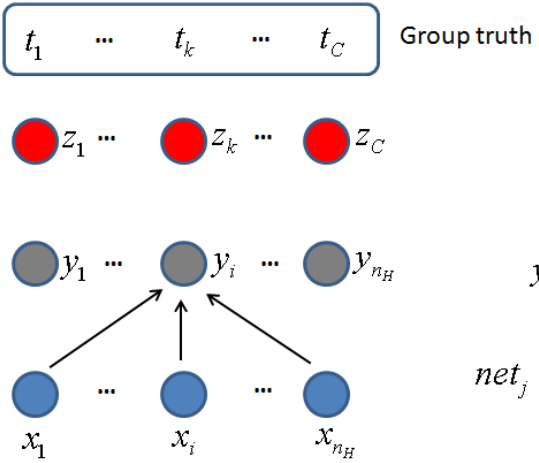
Example 1

- Choose squared error as training error measurement and sigmoid as activation function at the output layer
- When the output probabilities approach to 0 or 1 (i.e. saturate), $f'(net)$ gets close to zero and δ_k is small even if the error $(t_k - z_k)$ is large, which is bad.

Example 2

- Choose cross entropy as training error measurement and softmax as activation function at the output layer
- Sensitivity $\delta_k = -t_k(z_k - 1)$ (how to get it?)
- δ_k is large if error is large, even if z_k gets close to 0
- Softmax leads to sparser output

Input-to-hidden weights


t_1 ··· t_k ··· t_C
Group truth
z_1 ··· z_k ··· z_C
y_1 ··· y_i ··· y_{n_H}
x_1 ··· x_i ··· x_{n_H}

$$y_j = f(net_j)$$

$$net_j = \sum_{j=1}^{d} x_j w_{ji} + w_{j0}$$

Input-to-hidden weights

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

- How the hidden unit output y_j affects the error at each output unit

$$\begin{aligned}
\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\right] \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{kj} = \sum_{k=1}^{c}\delta_k w_{kj}
\end{aligned}$$

Input-to-hidden weights

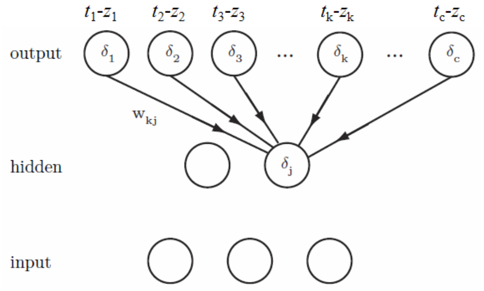
- Sensitivity for a hidden unit j

$$\delta_j = -\frac{\partial J}{\partial net_j} = -\frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j} = f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

- $\sum_{k=1}^{c} w_{kj}\delta_k$ is the effective error for hidden unit j
- Weight update rule. Since $\partial net_j / \partial w_{ji} = x_i$,

$$\Delta w_{ji} = \eta x_i \delta_j = \eta f'(net_j)\left[\sum_{k=1}^{c} w_{kj}\delta_k\right] x_i$$

Error backpropagation



(Duda et al. Pattern Classification 2000)

The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units.

Stochastic gradient descent

- Given n training samples, our target function can be expressed as

$$J(\mathbf{w}) = \sum_{p=1}^{n} J_p(\mathbf{w})$$

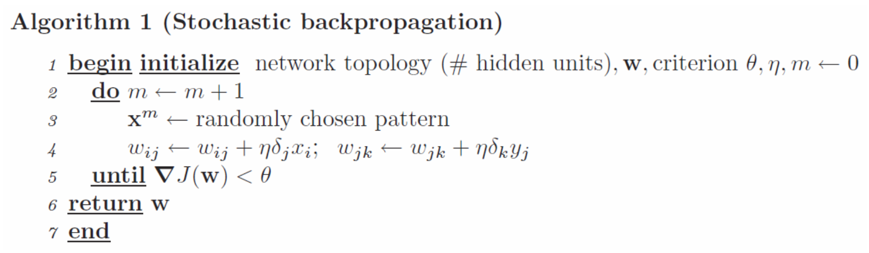
- Batch gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \sum_{p=1}^{n} \nabla J_p(\mathbf{w})$$

- In some cases, evaluating the sum-gradient may be computationally expensive. Stochastic gradient descent samples a subset of summand functions at every step. This is very effective in the case of large-scale machine learning problems. In stochastic gradient descent, the true gradient of $J(\mathbf{w})$ is approximated by a gradient at a single example (or a mini-batch of samples):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J_p(\mathbf{w})$$

Stochastic backpropagation



Algorithm 1 (Stochastic backpropagation)

```
begin initialize network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, m \leftarrow 0$
    do $m \leftarrow m + 1$
        $\mathbf{x}^m \leftarrow$ randomly chosen pattern
        $w_{ij} \leftarrow w_{ij} + \eta\delta_j x_i; \quad w_{jk} \leftarrow w_{jk} + \eta\delta_k y_j$
    until $\nabla J(\mathbf{w}) < \theta$
return $\mathbf{w}$
end
```

(Duda et al. Pattern Classification 2000)

- In stochastic training, a weight update may reduce the error on the single pattern being presented, yet increase the error on the full training set.

Mini-batch based stochastic gradient descent

- Divide the training set into mini-batches.
- In each epoch, randomly permute mini-batches and take a mini-batch sequentially to approximate the gradient
  - One epoch corresponds to a single presentations of all patterns in the training set
- The estimated gradient at each iteration is more reliable
- Start with a small batch size and increase the size as training proceeds

Batch backpropagation

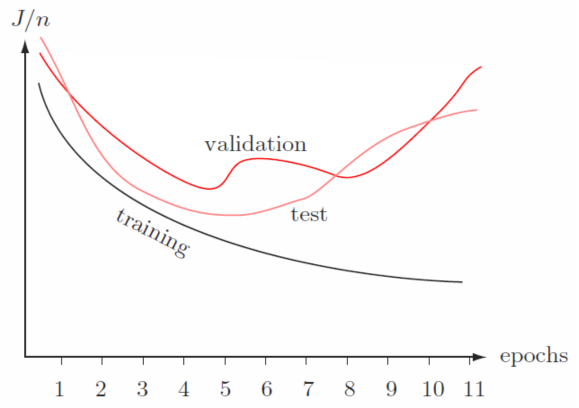
Algorithm 2 (Batch backpropagation)

```
begin initialize network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, r \leftarrow 0$
  do $r \leftarrow r + 1$ (increment epoch)
      $m \leftarrow 0;\ \Delta w_{ij} \leftarrow 0;\ \Delta w_{jk} \leftarrow 0$
      do $m \leftarrow m + 1$
          $\mathbf{x}^m \leftarrow$ select pattern
          $\Delta w_{ij} \leftarrow \Delta w_{ij} + \eta \delta_j x_i;\quad \Delta w_{jk} \leftarrow \Delta w_{jk} + \eta \delta_k y_j$
      until $m = n$
      $w_{ij} \leftarrow w_{ij} + \Delta w_{ij};\quad w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$
  until $\nabla J(\mathbf{w}) < \theta$
return $\mathbf{w}$
end
```

Summary

- Stochastic learning
  - Estimate of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
  - Faster than batch learning, especially when training data has redundance
  - Noise often results in better solutions
  - The weights fluctuate and it may not fully converge to a local minimum
- Batch learning
  - Conditions of convergence are well understood
  - Some acceleration techniques only operate in batch learning
  - Theoretical analysis of the weight dynamics and convergence rates are simpler

Plot learning curves on the training and validation sets



(Duda et al. Pattern Classification 2000)

Plot the average error per pattern (i.e. $1/n \sum_p J_p$) versus the number of epochs.

Learning curve on the training set

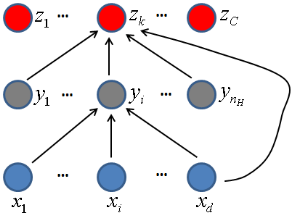
- The average training error typically decreases with the number of epochs and reaches an asymptotic value
- This asymptotic value could be high if **underfitting** happens. The reasons could be
  - The classification problem is difficult (Bayes error is high) and there are a large number of training samples
  - The expressive power of the network is not enough (the numbers of weights, layers and nodes in each layer)
  - Bad initialization and get stuck at local minimum (pre-training for better initialization)
- If the learning rate is low, the training error tends to decrease monotonically, but converges slowly. If the learning rate is high, the training error may oscillate.

Learning curve on the test and validation set

- The average error on the validation or test set is virtually always higher than on the training set. It could increase or oscillate when **overfitting** happen. The reasons could be
  - Training samples are not enough
  - The expressive power of the network is too high
  - Bad initialization and get stuck at local minimum (pre-training for better initialization)
- Stop training at a minimum of the error on the validation set

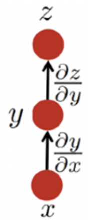
BP on general flow graphs

- BP be applied to a general flow graph, where each node u_i is the value obtained with a computation unit and partial orders are defined on nodes. $j < i$ means u_j is computed before u_i
- The final node is the objective function depending on all the other nodes
- Directed acyclic graphs
- Example: network with skipping layers (i.e. connecting non-adjacent layers)

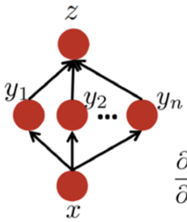


BP is an application of the chain rule

$$\frac{\partial C(g(\theta))}{\partial \theta} = \frac{\partial C(g(\theta))}{\partial g(\theta)} \frac{\partial g(\theta)}{\partial \theta}$$


z
$\frac{\partial z}{\partial y}$
y
$\frac{\partial y}{\partial x}$
x

$$\Delta z = \frac{\partial z}{\partial y} \Delta y$$
$$\Delta y = \frac{\partial y}{\partial x} \Delta x$$
$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

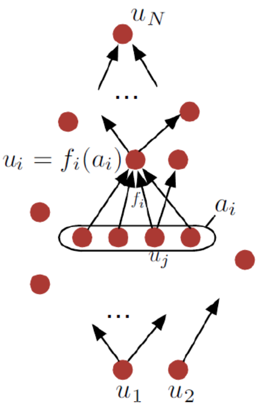


$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

(Bengio et al. Deep Learning 2014)

Flow graph forward computation

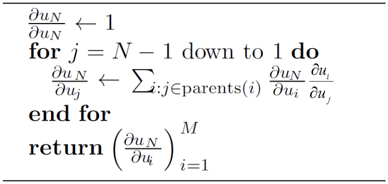
- $u_1, \ldots, u_N$ are the nodes with defined partial orders
- $a_i = (u_j)_{j \in \text{parents}(i)}$ is the set of parents of node u_i and $u_i = f_i(a_i)$



```
for $i = 1 \ldots M$ do
    $u_i \leftarrow x_i$
end for
for $i = M+1 \ldots N$ do
    $a_i \leftarrow (u_j)_{j \in \text{parents}(i)}$
    $u_i \leftarrow f_i(a_i)$
end for
return $u_N$
```

(Bengio et al. Deep Learning 2014)

BP on a flow graph



```
$\frac{\partial u_N}{\partial u_N} \leftarrow 1$
for $j = N-1$ down to 1 do
    $\frac{\partial u_N}{\partial u_j} \leftarrow \sum_{i:j \in \text{parents}(i)} \frac{\partial u_N}{\partial u_i} \frac{\partial u_i}{\partial u_j}$
end for
return $\left(\frac{\partial u_N}{\partial u_i}\right)_{i=1}^{M}$
```

(Bengio et al. Deep Learning 2014)

$$\frac{\partial u_N}{\partial w_{ji}} = \frac{\partial u_N}{\partial u_i} \frac{\partial u_i}{\partial net_i} \frac{\partial net_i}{\partial w_{ji}} = \frac{\partial u_N}{\partial u_i} f'(net_i) u_j$$

- BP has optimal computational complexity in the sense that there is no algorithm that can compute the gradient faster (in the $O(\cdot)$ sense)
- It is an application of the principles of dynamic programming

BP on a flow graph

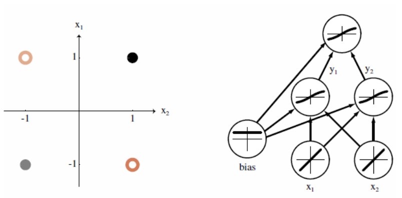
- The derivative of the output with respect to any node can be written in the following intractable form:

$$\frac{\partial u_N}{\partial u_i} = \sum_{\text{paths } u_{k_1} \ldots u_{k_n}:\ k_1=i, k_n=N} \prod_{j=2}^{n} \frac{\partial u_{k_j}}{\partial u_{k_{j-1}}}$$

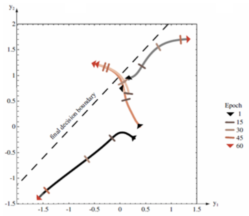
where the graphs $u_{k_1}, \ldots, u_{k_n}$ go from the node $k_1 = i$ to the final node $k_n = N$ in the flow graph. Computing the sum as above would be intractable because the number of possible paths can be exponential in the depth of the graph. BP is efficient because it employs a dynamic programming strategy to re-use rather than re-compute partial sums associated with the gradients on intermediate nodes.

Nonlinear feature mapping

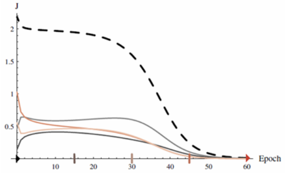
(a)



(b)



(c)



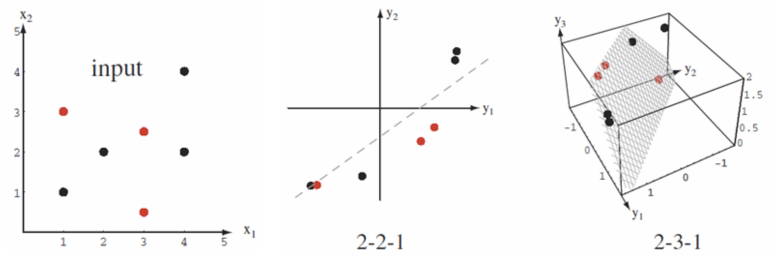
(a) 2-2-1 backpropagation network and the four patterns of the XOR problem;
(b) Outputs of the hidden units for each of the four patterns; these outputs move across the y_1y_2-space as the network learns;
(c) Learning curves of individual patterns and total error as a function of epoch

(Duda et al. Pattern Classification 2000)

Nonlinear feature mapping

- The multilayer neural networks provide nonlinear mapping of the input to the feature representation at the hidden units
- With small initial weights, the net activation of each hidden unit is small, and thus the linear portion of their activation function is used. Such a linear transformation from $\mathbf{x}$ to $\mathbf{y}$ leaves the patterns linearly inseparable in the XOR problem.
- As learning progresses and the input-to-hidden weights increase in magnitude, the nonlinearities of the hidden units warp and distort the mapping from input to the hidden unit space
- The linear decision boundary at the end of learning found by the hidden-to-output weights is shown by the straight dashed line; the nonlinearly separable problem at the inputs is transformed into a linearly separable at the hidden units

Nonlinear feature mapping

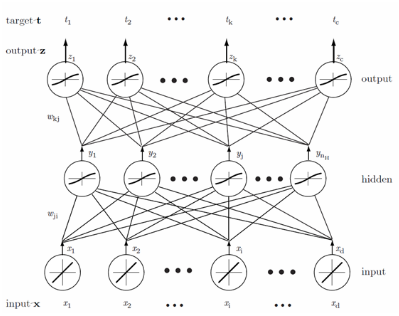


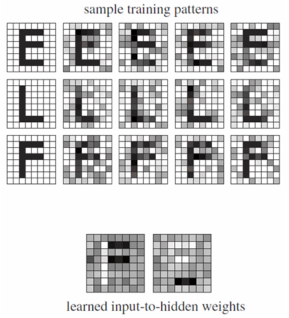
(Duda et al. Pattern Classification 2000)

- The expressive power of the 2-2-1 network is not high enough to separate all the seven patterns, even after the global minimum error is reached by training.
- These patterns can be separated by increasing one more hidden unit to enhance the expressive power.

Filter learning

- The input-to-hidden weights at a single hidden unit describe the input patterns that leads to maximum activation of that hidden unit, analogous to a “matched filter”
- Hidden units find feature groupings useful for the linear classifier implemented by the hidden-to-output layer weights





(Duda et al. Pattern Classification 2000)

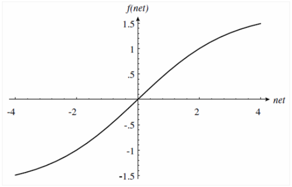
Filter learning

- The top images represent patterns from a large training set used to train a 64-2-3 neural network for classifying three characters
- The bottom figures show the input-to-hidden weights, represented as patterns, at the two hidden units after training
- One hidden unit is tuned to a pair of horizontal bars while the other is tuned to a single lower bar
- Both of these feature groups are useful building blocks for the pattern presented

A recommended sigmoid function for activation function

$$f(x) = 1.79159 \frac{e^{\frac{2}{3}x} - e^{-\frac{2}{3}x}}{e^{\frac{2}{3}x} + e^{-\frac{2}{3}x}}$$

$f(\pm 1) = \pm 1$, liner in the range of $-1 < x < 1$, $f''(x)$ has extrema near $x = \pm 1$.



Y. LeCun, Generalization and network design strategies. *Proc. Int'l Cibf, Cinectuibusn ub Oersoectuve*, 1988.

Desirable properties of activation functions

- Must be nonlinear: otherwise it is equivalent to a linear classifier
- Its output has maximum and minimum value: keep the weights and activations bounded and keep training time limited
    - Desirable property when the output is meant to represent a probability
    - Desirable property for models of biological neural networks, where the output represents a neural firing rate
    - May not be desirable in networks for regression, where a wide dynamic range may be required
- Continuous and differentiable **almost everywhere**
- Monotonicity: otherwise it introduces additional local extrema in the error surface
- Linearity for a small value of *net*, which will enable the system to implement a linear model if adequate for yielding low error

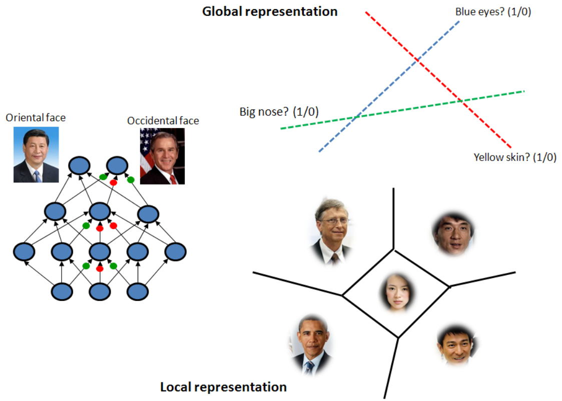
Desirable properties of activation functions

- The average of the outputs at a node is close to zero because these outputs are the inputs to the next layer
- The variance of the outputs at a node is also 1.

Global versus local representations

- Tanh function (shifting the center of *sigmoid* to 0) has all the properties above
  - It has large response for input in a large range. Any particular input **x** is likely to yield activity through several hidden units. This affords a **distributed** or **global** representation of the input.
  - If hidden units have activation functions that have significant response only for input within a small range, then an input **x** generally leads to fewer hidden units being active - a **local** representation.
  - Distributed representations are superior because more of the data influences the posteriors at any given input region.
  - The global representation can be better achieved with more layers

Global versus local representations



Choosing target values

- Avoid setting the target values as the sigmoid's asymptotes
    - Since the target values can only be achieved asymptotically, it drives the output and therefore the weights to be very large, which the sigmoid derivative is close to zero. So the weights may become stuck.
    - When the outputs saturate, the network gives no indication of confidence level. Large weights fore all outputs to the tails of the sigmoid instead of being close to decision boundary.
- Insure that the node is not restricted to only the linear part of the sigmoid
- Choose target values at the point of the maximum second derivative on the sigmoid so as to avoid saturating the output units

Initializing weights

- Randomly initialize weights in the linear region. But they should be large enough to make learning proceed.
    - The network learns the linear part of the mapping before the more difficult nonlinear parts
    - If weights are too small, gradients are small, which makes learning slow
- To obtain a standard deviation close to 1 at the output of the first hidden layer, we just need to use the recommended sigmoid and require that the input to the sigmoid also have a standard deviation $\sigma_y = 1$. Assuming the inputs to a unit are uncorrelated with variance 1, the standard deviation of σ_{y_i} is

$$\sigma_{y_i} = (\sum_{j=1}^{m} w_{ij}^2)^{1/2}$$

Initializing weights

- To ensure $\sigma_{y_i} = 1$, weights should be randomly drawn from a distribution (e.g. uniform) with mean zero and standard deviation

$$\sigma_w = m^{-1/2}$$

Reading materials

- R. O. Duda, P. E. Hart, and D. G. Stork, “Pattern Classification,” Chapter 6, 2000.
- Y. Bengio, I. J. GoodFellow, and A. Courville, “Feedforward Deep Networks,” Chapter 6 in “Deep Learning”, Book in preparation for MIT Press, 2014.

Reference

- W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115-133, 1943.
- F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386-408, 1958.
- D. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323:533-536,1986.
- Y. LeCun. A learning scheme for asymmetric threshold networks. In *Proceedings of Cognitiva* 85, pages 599-604, Paris, France, 1985.
- Y. LeCun. Learning processes in an asymmetric threshold network. In Elie Bienenstock, Francoise Fogelman-Soulie, and Gerard Weisbuch, editors, Disor- dered systems and biological organization, pages 233-240, Les Houches, France, 1986.Springer-Verlag.
- R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, Vol. 1, pages 593-605, 1989.

Reference

- K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4): 93-202, 1980.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11): 2278-2324, 1998.
- G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527-1554, 2006.
- G. E. Hinton, and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313:504-507, 2006.
- G. E. Hinton. Learning multiple layers of representation. *Trends in Cognitive Sciences*, Vol. 11, pp. 428-434, 2007.