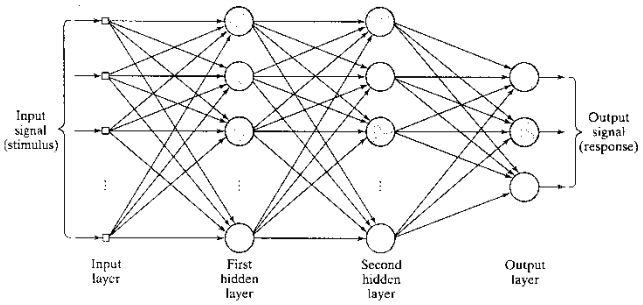
MULTILAYER PERCEPTRON

Source: Haykin, 2009

- Typically, the multilayer perceptron (MLP) network consists of a set of inputs that constitute the *input layer*, one or more *hidden layers* of computation nodes, and an *output layer* of computation nodes.
- MLP networks are feedforward networks, i.e., the signal propagates through the network in a forward direction, on a layer-by-layer basis.
- Typically, these networks are trained in a supervised fashion using *error back-propagation algorithm* based on the *error-correction learning rule*.
- Error back-propagation learning consists of two passes (*forward* and *backward*):
  - *Forward Pass* - In this pass, the effect of an activity pattern (input vector) propagates through the network layer by layer; The synaptic weights are all fixed during this pass.
  - *Backward Pass* - In this pass, the error signal is propagated backward through the network, against the direction of synaptic connection; The synaptic weights are adjusted to improve the performance of the network in a statistical sense.
- In designing an MLP network to be trained using the back-propagation algorithm, each hidden or output neuron is designed to perform two computations:
  1. Computation of the function signal.
  2. Computation of an instantaneous estimate of the gradient vector.
- Three distinctive characteristics (that offer advantages and pose challenges):
  1. Neuron model includes a *differentiable nonlinear activation function*.
  2. The network contains one or more number of *hidden layers*.
  3. The network exhibits a high degree of *connectivity*.
- The development of the back-propagation algorithm represents a landmark in that it provides a *computationally efficient* method for implementing the error-correction rule.

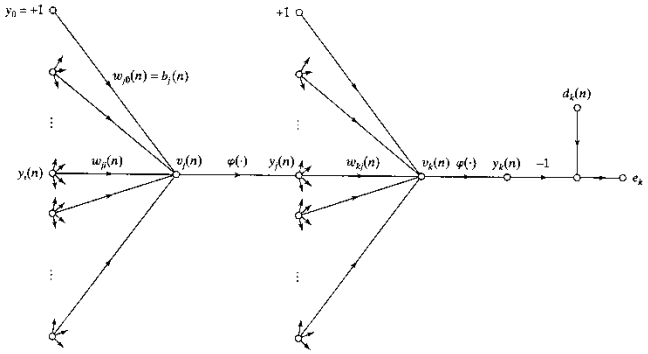


Architectural Graph of an MLP Network with Two Hidden Layers.

BACK-PROPAGATION ALGORITHM

Notation:

- The indices *i*, *j*, and *k* refer to different neurons in consecutive layers of the network.
- $\xi(n)$ denotes the instantaneous sum of error squares at iteration *n*.
- $y_j(n)$ and $d_j(n)$ denote the actual and desired responses, respectively, at output neuron *j* for iteration *n*.
- $e_j(n)$ denotes the error signal at output neuron *j* for iteration *n* {i.e., $e_j(n) = (d_j(n) - y_j(n))$ }.
- $w_{ji}(n)$ denotes the synaptic weight connecting the output of neuron *i* to the input of neuron *j* at iteration *n*.
- $\varphi_j(\cdot)$ denotes the activation function associated with neuron *j*.
- b_j denotes the bias applied to neuron *j*; its effect is represented by a synapse of weight $w_{j0} = b_j$ connected to a fixed input equal to ± 1.
- η denotes the learning rate parameter.
- m_l denotes the number of nodes in layer *l* of the network; $l = 0,1,...,L$, where *L* denotes the number of hidden and output layers in the network.
- *N* denotes the total number of patterns in the training set.



Signal-flow Graph Highlighting the Details of Neuron *k* Connected to Neuron *j*.

Algorithm:

- *Instantaneous* and *averaged* sum of squared errors for the network are written as:

$$\xi(n) = \frac{1}{2}\sum_j e_j(n)^2 \qquad \xi_{av} = \frac{1}{N}\sum_{n=1}^{N}\xi(n)$$

- The objective of the learning process is to adjust the free parameters of the network to minimize ξ_{av} (without compromising generalization capability).
- It is a common practice to consider a simple method of training in which the weights are adjusted on a pattern-by-pattern basis.
- The average of the individual weight changes over the training set is therefore an estimate of the true change that would result from directly minimizing ξ_{av} .
- The back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the instantaneous gradient $\partial\xi(n)/\partial w_{ji}(n)$.
- The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ as defined by the *delta rule* is:

$$\begin{aligned}\Delta w_{ji}(n) &= -\eta\frac{\partial\xi(n)}{\partial w_{ji}(n)} \\ &= \eta\left[-\frac{\partial\xi(n)}{\partial v_j(n)}\right]\frac{\partial v_j(n)}{\partial w_{ji}(n)} \qquad \text{(by chain rule)} \\ &= \eta\delta_j(n)y_i(n)\end{aligned}$$

where $\delta_j(n)$ denotes the *local gradient* of neuron *j*.

Case I: Neuron *j* is an Output Node

- According to the chain rule:

$$\frac{\partial \xi(n)}{\partial w_{ji}(n)} = \frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

However, from our notation, we know the following:

$$\frac{\partial \xi(n)}{\partial e_j(n)} = e_j(n) \qquad \frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \qquad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

- For the current case where neuron *j* is an output node,

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\varphi_j'(v_j(n))\end{aligned}$$

Case II: Neuron *j* is a Hidden Node

- If neuron *j* is a hidden node, there is no specified desired response for the neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of error signals of all the neurons to which that hidden neuron is directly connected.
- The local gradient $\delta_j(n)$ for hidden neuron *j* is:

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \xi(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \xi(n)}{\partial y_j(n)} \varphi_j'(v_j(n))\end{aligned}$$

Suppose *j* denotes a neuron in the last hidden layer, and *k* denotes a neuron in the output layer. Then, the instantaneous error is defined as

$$\xi(n) = \frac{1}{2}\sum_k e_k(n)^2$$

Now, the local gradient $\delta_j(n)$ for hidden neuron *j* can be rewritten as

$$\begin{aligned}\delta_j(n) &= -\varphi_j'(v_j(n)) \cdot \sum_k \left[e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} \right] \\ &= -\varphi_j'(v_j(n)) \cdot \sum_k \left[e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \right] \\ &= -\varphi_j'(v_j(n)) \cdot \sum_k \left[e_k(n) \frac{\partial e_k(n)}{\partial y_k(n)} \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \right] \\ &= -\varphi_j'(v_j(n)) \cdot \sum_k \left[e_k(n)(-1)\varphi_k'(v_k(n)) \frac{\partial v_k(n)}{\partial y_j(n)} \right] \\ &= \varphi_j'(v_j(n)) \cdot \sum_k \left[e_k(n)\varphi_k'(v_k(n)) \frac{\partial \sum_j w_{kj}(n) y_j(n)}{\partial y_j(n)} \right] \\ &= \varphi_j'(v_j(n)) \cdot \sum_k \left[\delta_k(n) w_{kj}(n) \right]\end{aligned}$$

- The above $\delta_j(n)$ can be shown to be valid for any neuron in any hidden layer.

Activation Function:

- The computation of the *local gradient* for any given neuron in an MLP network requires knowledge of the derivative of the action function (i.e., $\varphi'(\cdot)$).

Logistic Function:

$$\varphi_j(v_j(n)) = \frac{a}{1+\exp(-bv_j(n))} \quad (a,b) > 0 \text{ and } -\infty < v_j(n) < +\infty$$

$$\begin{aligned}\varphi_j'(v_j(n)) &= \frac{ab\exp(-bv_j(n))}{\left[1+\exp(-bv_j(n))\right]^2} \\ &= \frac{b}{a}\varphi_j(v_j(n))\left[a - \varphi_j(v_j(n))\right]\end{aligned}$$

Hyperbolic Tangent Function:

$$\begin{aligned}\varphi_j(v_j(n)) &= a\tanh(bv_j(n)) \quad (a,b) > 0 \text{ and } -\infty < v_j(n) < +\infty \\ &= a \cdot \frac{1-\exp(-bv_j(n))}{1+\exp(-bv_j(n))}\end{aligned}$$

$$\begin{aligned}\varphi_j'(v_j(n)) &= ab\ \text{sech}^2(bv_j(n)) \\ &= ab(1-\tanh^2(bv_j(n))) \\ &= \frac{b}{a}\left[a+\varphi_j(v_j(n))\right]\left[a-\varphi_j(v_j(n))\right]\end{aligned}$$

Softmax Function:

$$p_j = \text{softmax}(\boldsymbol{y} = [y_1, \ \ldots, \ y_K]) = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_k}}$$

$$\frac{\partial p_j(n)}{\partial y_i(n)} = \begin{cases} p_j(1-p_i) & \text{if } j = i \\ -p_j \cdot p_i & \text{if } j \neq i \end{cases}$$

See Full Derivation Here: Link

ReLu Functions:

Rectifer:

$$\varphi_j(v_j(n)) = \max(0, v_j(n))$$

$$\varphi'_j(v_j(n)) = \begin{cases} 1 & \text{if } v_j(n) > 0 \\ 0 & \text{if } v_j(n) \leq 0 \end{cases}$$

Softplus:

$$\varphi_j(v_j(n)) = \log(1 + e^{v_j(n)})$$

$$\varphi'_j(v_j(n)) = \frac{e^{v_j(n)}}{1+e^{v_j(n)}}$$

Rate of Learning – Need for "Momentum Term" and Variable η:

- The back-propagation algorithm provides an "approximation" to steepest descent.
- For small values of η, a smooth trajectory is attained at the cost of a slower rate of learning. On the other hand, for large values of η, the learning can be faster, however, there is a danger that the network may become unstable.
- A simple method of increasing the rate of learning yet avoiding the danger of instability is to include a *momentum term* in the weight update equation:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \quad \text{Convergent if } 0 \leq |\alpha| < 1$$

$$= \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

  - The current adjustment represents the sum of an exponentially weighted time series.
  - The inclusion of the momentum tends to *accelerate descent* in steady downhill directions and has a *stabilizing effect* in directions that oscillate in sign.
- The learning rate parameter can be adapted as well.

Sequential and Batch Modes of Training:

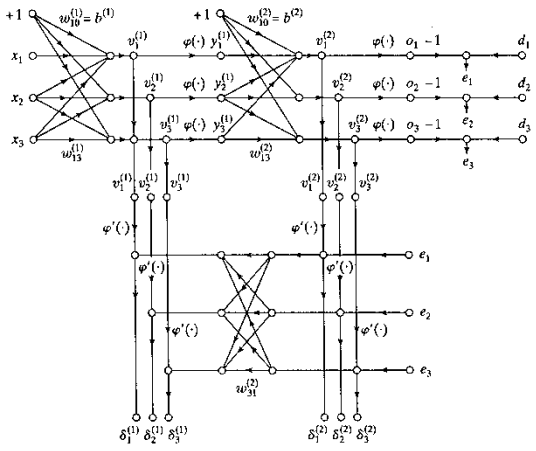
- Sequential mode of training involves adjusting the network parameters after the presentation of each training example.
  $\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$
  Offers the following properties and advantages:
  - Search in the weight space tends to be *stochastic* over the learning cycles, thus avoiding the possibility of limit cycles. This is however only possible if the order of presentation of training examples is randomized.
  - Simple to implement for it requires less local memory storage.
  - Provides more effective solutions to large and difficult problems.
- Batch mode of training involves adjusting the network parameters after the presentation of all the training examples. However, the local gradients themselves have to be calculated after the presentation of each training example.
  $$\Delta w_{ji} = \frac{\eta}{N} \sum_{n=1}^{N} \delta_j(n) y_i(n)$$
  Offers the following advantage:
  - Makes it relatively less difficult to establish theoretical conditions for convergence of the algorithm.

Note: The computed $\delta_j(n)$ and $y_i(n)$ values tend to differ for the two modes of training due to the fact that the weights are updated at different frequencies.

Termination/Stopping Criteria for Network Training:

- Actual criteria should depend on the application at hand. Here are some useful criteria:
  - Predetermined number of epochs.
  - Predetermined processor computation time.
  - Acceptable minimum learning error.
  - Acceptable minimum learning rate.
  - Acceptable generalization performance.
  - Apparent peak in generalization performance.

Signal-flow Summary of Back-propagation Algorithm:



Representation: Top Part → Forward Pass.
Bottom Part → Backward Pass.

Heuristics for Making Back-propagation Algorithm Perform Better:

- *Sequential versus batch update*. Sequential update leads to stochastic gradient search and can be valuable with difficult problems.
- *Nature of activation function*. Antisymmetric functions seem to help.
- *Target values*. Linear neurons are recommended for output neurons.
- *Normalizing of inputs and outputs*. Is crucial.
- *Weight initialization*. It is desirable for the uniform distribution from which the synaptic weights are selected to have a mean of zero and a variance equal to the reciprocal of the number of synaptic connections of a neuron.
- *Learning rate parameters*. All neurons should learn at the same rate. Since the last layers have usually larger local gradients, their learning rate parameters should be smaller.
- *A priori information*. Utilize this wisdom to achieve invariance properties, symmetries, etc.

Training MLP Networks using Gradient Search Methods–Some Issues:

- Back-propagation is a specific technique for implementing *gradient descent* in weight space for MLP networks, and it does so by computing *partial derivatives* efficiently.
- There is experimental evidence to suggest that many neural network training problems are intrinsically *ill-conditioned*, leading to a Jacobian **J** (i.e., the *N*-by-*W* matrix of partial derivatives where *N* denotes number of patterns and *W* the number of free synaptic parameters) that is rank deficient (Saarinen et al., 1991).
- Any rank deficiency in the Jacobian causes the back-propagation algorithm to obtain only partial information of the possible search directions, causing long training times.
- The *Hessian Matrix*, defined as the second derivative of $\xi_{av}(\mathbf{w})$ with respect to **w**, plays an important role in the study of neural networks and their training algorithms:
  - The eigenvalues* of the Hessian matrix have a profound influence on the dynamics of back-propagation learning.
  - The Hessian inverse provides a basis for pruning insignificant synaptic weights.
  - The Hessian matrix is basic to the formulation of second-order optimization methods as an alternative to back-propagation learning.
- Typically the Hessian matrix of an MLP network trained with the back-propagation algorithm has the following composition of eigenvalues (LeCun, 1993):
  - A small number of small eigenvalues
  - A large number of medium-sized eigenvalues
  - A small number of large eigenvalues
- Experimental results show that the learning time of the back-propagation algorithm is sensitive to variations in the condition number $\lambda_{max}/\lambda_{min}$ (where λ_{max} and λ_{min} represent the largest and smallest nonzero eigenvalues), the learning time increasing with higher condition numbers.
- For inputs with nonzero mean, the condition number is larger than its corresponding value for zero-mean inputs (the larger the mean the larger the condition number).
- For the learning time to be minimized, the use of nonzero-mean inputs should be avoided:
  - This can be achieved at the input layer by normalizing the inputs (for example, $x_{new} = (x - \mu_x)/\sigma_x$).
  - This can be partially achieved in the hidden layers by using antisymmetric activation functions.

Achieving Generalization in MLP Networks–Some Issues:

- A network is said to *generalize* well when the input-output mapping computed by the network is correct for test data never used in creating or training the network; The assumption here is that the test data are drawn from the same population used to generate the training data.
- If the learning process can be viewed as a "curve-fitting" problem, then, generalization is equivalent to achieving good nonlinear interpolation of the input data.
- *Overtraining* or *overfitting* can result if the network starts memorizing the training data. It may do so by finding a feature (due to noise, for example) that is present in the training data and not true of the underlying function that is to be modeled.
- Generalization is normally influenced by three factors:
  1. Size and efficiency of the training data set
  2. The architecture of the network
  3. Physical complexity of the problem at hand (normally out of our control)
- The issue of generalization can be viewed from two different perspectives:
  1. Architecture is fixed, what should be the size of the training data set to achieve good generalization?
     - The VC dimension provides the theoretical basis for a principled solution to this problem. In particular, there exist *distribution-free*, *worst-case* formulas for estimating the required size of the training data set.
     - In general, it is observed that there is a huge numerical gap between the size of the training sample actually needed and that predicted by these formulas.
     - In practice, for classification problems, it seems that all we really need for achieving good generalization is to satisfy the following condition (Baum and Haussler, 1989):

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

       $O(\cdot)$ denotes the order of quantity enclosed within,
       ε denotes the fraction of classification errors permitted on test data, and
       W denotes the total number of free parameters.
  2. The size of the training data set is fixed, what is the best architecture for the network to achieve good generalization? (More Common Approach)

Achieving Generalization through Cross-Validation:

- The network training problem can be looked upon as a problem of making the network learn enough about the past to generalize to the future.
  - From such a perspective, the learning process amounts to a choice of network parameterization for the existing data set.
  - More specifically, one may view the network selection problem as choosing, within a set of candidate model structures (parameterizations), the "best" one according to a certain criterion.
- In this context, a standard tool in statistics known as *cross-validation* provides an appealing guiding principle:
  - First, the available data set is partitioned into a *training set* (*N*) and a *test set* (*M*).
  - The training set is further partitioned into two disjoint subsets:
    Estimation subset, used to "select" the model ((1-*r*)/*N*).
    Validation subset, used to "validate" the model (*rN*).
  - There is a distinct probability that the "best" model so selected may end up overfitting the validation subset. To guard against this possibility, the generalization performance of the selected model is measured on a test set.
  - As the target function becomes more complex relative to the sample size *N*, the choice of optimum *r* has a more pronounced effect on cross-validation performance, and its own value decreases (Kearns, 1996).
- Early stopping method of training using cross-validation:
  - The procedure involves the following steps:
    Periodically stop the network training process and test the network (keeping the network parameters constant) on the validation subset.
    Permanently stop the network training process if an increase is noted in the validation error over successive evaluations.

---

* A square matrix **A** is said to have an *eigenvalue* λ, with corresponding *eigenvector* $\mathbf{x} \neq \mathbf{0}$, if $\mathbf{Ax} = \lambda\mathbf{x}$.

- The effectiveness of the cross-validation procedure decreases if $N \gg W$.

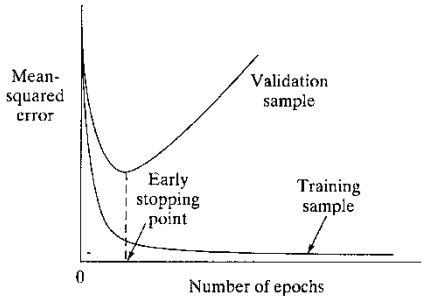


Illustration of the Early-stopping Rule Based on Cross-Validation.

Variants of Cross-Validation:

- For relatively small training data sets, two variants of cross-validation that offer more promise include:

  Multifold Method
  - Involves dividing the available training set of *N* examples into *K* subsets, *K* > 1.
  - The model is trained on all subsets except for one, and the validation error is measured by testing it on the subset left out.
  - The procedure is repeated for a total of *K* trials, each time using a different subset for validation.
  - The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment.
  - Normally requires excessive amount of computation.

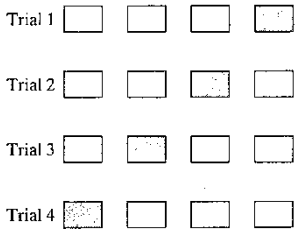


Illustration of the Hold-Out Method of Cross-Validation.

  Leave-one-out Method
  - When the data set is severely small, one can use the extreme form of multifold cross-validation where *K* equals *N*.

Complexity-Regularization and Network Pruning/Growing:

- A network with minimum/optimal size is less likely to learn the idiosyncrasies or noise in the training data, and might offer good generalization.
- One can achieve the above objective in one of two ways:
  - Network Growing
  - Network Pruning
- Complexity Regularization Methods:
  - Insofar as the network design is statistical in nature, we need an appropriate tradeoff between reliability of the training data and goodness of the model (i.e., a method for solving the bias-variance dilemma).
  - In the context of back-propagation learning, or any other supervised learning procedure for that matter, one can realize this tradeoff by minimizing the total risk expressed as:

    $R(\mathbf{w}) = \xi_S(\mathbf{W}) + \lambda \xi_C(\mathbf{w})$

    where:

    $\xi_C(\mathbf{w})$ denotes complexity penalty and depends on the network (model) alone,

    $\xi_S(\mathbf{W})$ denotes the standard performance measure (such as MSE) and depends on the network (model) and the input data, and

    λ denotes the complexity regularization parameter ($0 \le \lambda \le \infty$).
  - In a general setting, one choice of complexity-penalty term $\xi_C(\mathbf{w})$ is the *k*th order smoothing signal:

$$\xi_C(\mathbf{w},k) = \frac{1}{2}\int \left\| \frac{\partial^k}{\partial \mathbf{x}^k} F(\mathbf{x},\mathbf{w}) \right\|^2 \mu(\mathbf{x}) d\mathbf{x}$$

    where:

    $F(\mathbf{x},\mathbf{w})$ is the input-output mapping performed by the model and

    $\mu(\mathbf{x})$ is some weighting function that determines the region of input space over which the function $F(\mathbf{x},\mathbf{w})$ is required to be smooth.

    The larger we choose *k*, the smoother (i.e., less complex) the function $F(\mathbf{x},\mathbf{w})$.

  Weight Decay Method

$$\xi_C(\mathbf{w}) = \|\mathbf{w}\|^2$$

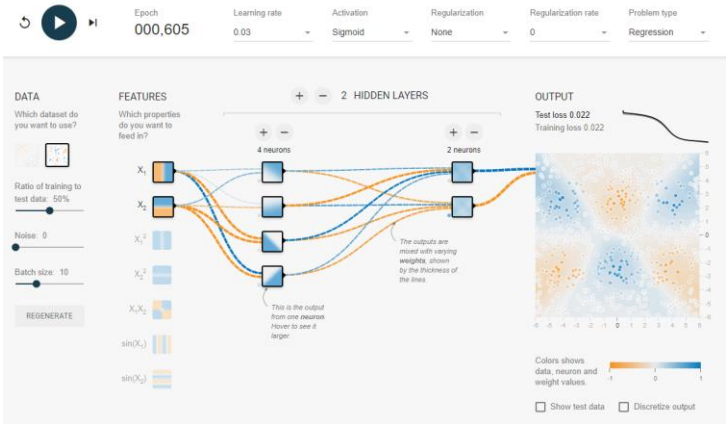
  Weight Elimination Method

$$\xi_C(\mathbf{w}) = \sum_{\forall i} \frac{(w_i / w_0)^2}{1 + (w_i / w_0)^2}$$

  where w_0 is a preassigned parameter.

Neural Network Playground: Tinker With a Neural Network in Your Browser:

- As part of a recent collaborative project called Tensor Flow, Daniel Smilkov and Shan Carter created a neural network playground, which aims to demystify the hidden layers by allowing users to interact and experiment with them.

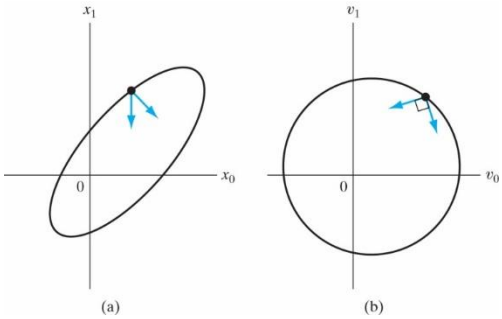


Virtues and Limitations of Back-Propagation Learning:

- Connectionism
- Feature Detection (hidden neurons)
- Function Approximation
- Computational Efficiency
- Sensitivity Analysis
- Robustness
- Convergence
- Local Minima
- Scaling

Supervised Learning Viewed as an Optimization Problem:

- Conjugate-Gradient Methods are very promising for accelerating learning



Initialization

Unless prior knowledge on the weight vector $\mathbf{w}$ is available, choose the initial value $\mathbf{w}(0)$ by using a procedure similar to that described for the back-propagation algorithm.

Computation

1. For $\mathbf{w}(0)$, use back propagation to compute the gradient vector $\mathbf{g}(0)$.
2. Set $\mathbf{s}(0) = \mathbf{r}(0) = -\mathbf{g}(0)$.
3. At time-step n, use a line search to find $\eta(n)$ that minimizes $\mathscr{E}_{av}(\eta)$ sufficiently, representing the cost function $\mathscr{E}_{av}$ expressed as a function of η for fixed values of $\mathbf{w}$ and $\mathbf{s}$.
4. Test to determine whether the Euclidean norm of the residual $\mathbf{r}(n)$ has fallen below a specified value, that is, a small fraction of the initial value $\|\mathbf{r}(0)\|$.
5. Update the weight vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n)$$

6. For $\mathbf{w}(n+1)$, use back propagation to compute the updated gradient vector $\mathbf{g}(n+1)$.
7. Set $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$.
8. Use the Polak–Ribière method to calculate:

$$\beta(n+1) = \max\left\{\frac{\mathbf{r}^T(n+1)(\mathbf{r}(n+1) - \mathbf{r}(n))}{\mathbf{r}^T(n)\mathbf{r}(n)}, 0\right\}$$

9. Update the direction vector:

$$\mathbf{s}(n+1) = \mathbf{r}(n+1) + \beta(n+1)\mathbf{s}(n)$$

10. Set $n = n + 1$, and go back to step 3.

Stopping criterion. Terminate the algorithm when the condition

$$\|\mathbf{r}(n)\| \leq \varepsilon\|\mathbf{r}(0)\|$$

is satisfied, where ε is a prescribed small number.