January 24th 2005

Introduction to Neural Networks and Data Mining – lecture 2
BARGIELA Andrzej *(ISM – NTU, Hirota Lab. – TITech)*

7. Multi-layer networks

Consider again the "best linear fit" we found for the car data. Notice that the data points are not evenly distributed around the line: for low weights, we see more miles per gallon than our model predicts. In fact, it looks as if a simple **curve** might fit these data better than the straight line. We can enable our neural network to do such curve fitting by giving it an additional node, which has a suitably curved (**nonlinear**) activation function. A useful function for this purpose is the S-shaped **hyperbolic tangent** (tanh) function (Fig. 1).

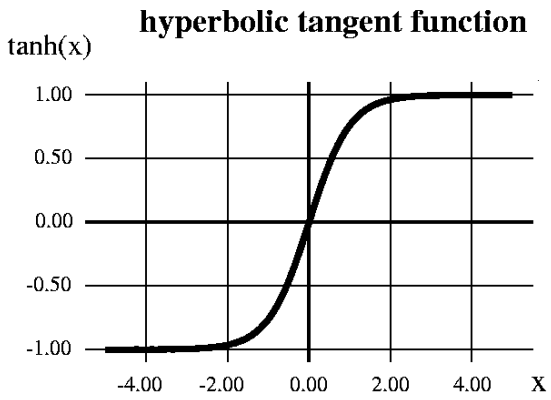


Figure 1.

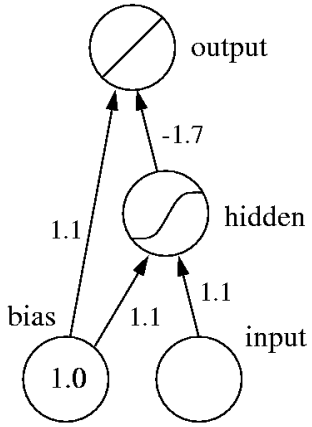


Figure 2.

FIg. 2 shows our new network: an extra node (unit 2) with tanh activation function has been inserted between input and output. Since such a node is "hidden" inside the network, it is commonly called a **hidden unit**. Note that the hidden unit also has a weight from the bias unit. In general, all non-input neural network units have such a bias weight. For simplicity, the bias unit and weights are usually omitted from neural network diagrams - unless it's explicitly stated otherwise, you should always assume that they are there.

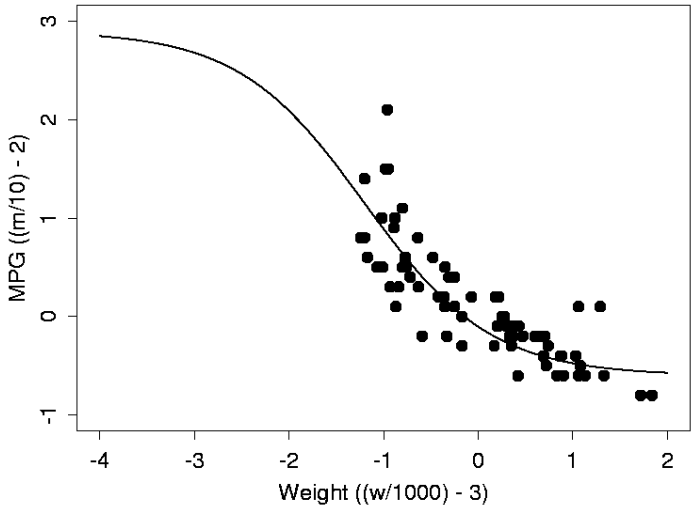


Figure 3.

When this network is trained by gradient descent on the car data, it learns to fit the tanh function to the data (Fig. 3). Each of the four weights in the network plays a particular role in this process: the two bias weights shift the tanh function in the x- and y-direction, respectively, while the other two weights scale it along those two directions. Fig. 2 gives the weight values that produced the solution shown in Fig. 3.

7.1. Hidden layers

One can argue that in the example above we have cheated by picking a hidden unit activation function that could fit the data well. What would we do if the data looks like this (Fig. 4)?

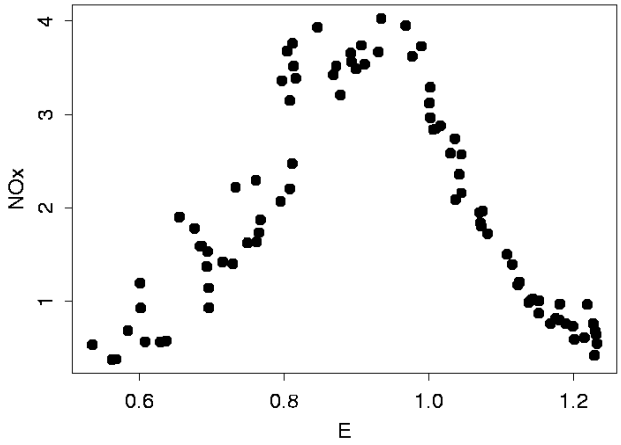


Figure 4. (Relative concentration of NO and NO_2 in exhaust fumes as a function of the richness of the ethanol/air mixture burned in a car engine.)

Obviously the tanh function can't fit this data at all. We could cook up a special activation function for each data set we encounter, but that would defeat our purpose of *learning* to model the data. We would like to have a general, non-linear function approximation method, which would allow us to fit *any* given data set, no matter how it looks like.

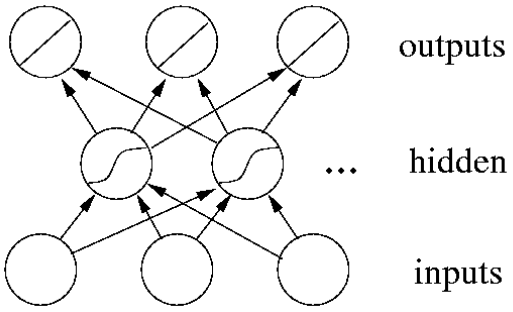


Figure 5.

Fortunately there is a very simple solution: add more hidden units! In fact, a network with just two hidden units using the tanh function (Fig. 5) can fit the data in Fig. 4 quite well - can you see how? The fit can be further improved by adding yet more units to the **hidden layer**. Note, however, that having too large a hidden layer - or too many hidden layers - can degrade the network's performance (more on this later). In general, one shouldn't use more hidden units than necessary to solve a given problem. (One way to ensure this is to start training with a very small network. If gradient descent fails to find a satisfactory solution, grow the network by adding a hidden unit, and repeat.)

Theoretical results indicate that given enough hidden units, a network like the one in Fig. 5 can approximate *any* reasonable function to any required degree of accuracy. In other words, any function can be expressed as a linear combination of tanh functions: tanh is a **universal basis function**. Many functions form a universal basis; the two classes of activation functions commonly used in neural networks are the **sigmoidal** (S-shaped) basis functions (to which tanh belongs), and the **radial** basis functions.

8. Error Backpropagation

We have already seen how to train linear networks by gradient descent. In trying to do the same for multi-layer networks we encounter a difficulty: we don't have any target values for the hidden units. This seems to be an insurmountable problem - how could we tell the hidden units just what to do? This unsolved question was in fact the reason why neural networks fell out of favour after an initial period of high popularity in the 1950s. It took 30 years before the **error backpropagation** (or in short: **backprop**) algorithm popularised a way to train hidden units, leading to a new wave of neural network research and applications.

In principle, backprop provides a way to train networks with any number of hidden units arranged in any number of layers. (There are clear practical limits, which we will discuss later.) In fact, the network does not have to be organized in layers - any pattern of connectivity that permits a **partial ordering** of the nodes from input to output is allowed. In other words, there must be a way to order the units such that all connections go from "earlier" (closer to the input) to "later" ones (closer to the output). This is equivalent to stating that their connection pattern must not contain any cycles. Networks that respect this constraint are called **feedforward** networks; their connection pattern forms a **directed acyclic graph** or **dag**.

8.1. The Algorithm

We want to train a multi-layer feedforward network by gradient descent to approximate an unknown function, based on some training data consisting of pairs (**x,t**). The vector **x** represents a pattern of input to the network, and the vector **t** the corresponding **target** (desired output). As we have seen before, the overall gradient with respect to the entire training set is just the sum of the gradients for each pattern; in what follows we will therefore describe how to compute the gradient for just a single training pattern. As before, we will number the units, and denote the weight from unit j to unit i by w_{ij}.

1. **Definitions:**
   - the **error** signal for unit j: $\delta_j = -\partial E / \partial net_j$
   - the (negative) **gradient** for weight w_{ij}: $\Delta w_{ij} = -\partial E / \partial w_{ij}$
   - the set of nodes **anterior** to unit i: $A_i = \{j : \exists w_{ij}\}$
   - the set of nodes **posterior** to unit j: $P_j = \{i : \exists w_{ij}\}$

2. **The gradient.** As we did for linear networks before, we expand the gradient into two factors by use of the chain rule:

$$\Delta w_{ij} = -\frac{\partial E}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}}$$

The first factor is the error of unit i. The second is

$$\frac{\partial net_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_{k \in A_i} w_{ik} y_k = y_j$$

Putting the two together, we get

$$\Delta w_{ij} = \delta_i y_j$$

To compute this gradient, we thus need to know the activity and the error for all relevant nodes in the network.

3. **Forward activaction.** The activity of the input units is determined by the network's external input **x**. For all other units, the activity is propagated forward:

$$y_i = f_i(\sum_{j \in A_i} w_{ij} y_j)$$

Note that before the activity of unit i can be calculated, the activity of all its anterior nodes (forming the set A_i) must be known. Since feedforward networks do not contain cycles, there is an ordering of nodes from input to output that respects this condition.

4. **Calculating output error.** Assuming that we are using the sum-squared loss

$$E = \frac{1}{2}\sum_o (t_o - y_o)^2$$

the error for output unit o is simply

$$\delta_o = t_o - y_o$$

5. **Error backpropagation.** For hidden units, we must propagate the error back from the output nodes (hence the name of the algorithm). Again using the chain rule, we can expand the error of a hidden unit in terms of its posterior nodes:

$$\delta_j = -\sum_{i \in P_j} \frac{\partial E}{\partial net_i} \frac{\partial net_i}{\partial y_j} \frac{\partial y_j}{\partial net_j}$$

Of the three factors inside the sum, the first is just the error of node i. The second is

$$\frac{\partial net_i}{\partial y_j} = \frac{\partial}{\partial y_j} \sum_{k \in A_i} w_{ik} y_k = w_{ij}$$

while the third is the derivative of node j's activation function:

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial f_j(net_j)}{\partial net_j} = f_j'(net_j)$$

For hidden units h that use the tanh activation function, we can make use of the special identity $\tanh(u)' = 1 - \tanh(u)^2$, giving us

$$f_h'(net_h) = 1 - y_h^2$$

Putting all the pieces together we get

$$\delta_j = f_j'(net_j) \sum_{i \in P_j} \delta_i w_{ij}$$

Note that in order to calculate the error for unit j, we must first know the error of all its posterior nodes (forming the set P_j). Again, as long as there are no cycles in the network, there is an ordering of nodes from the output back to the input that respects this condition. For example, we can simply use the reverse of the order in which activity was propagated forward.

8.2 Matrix form of the algorithm

For layered feedforward networks that are **fully connected** - that is, each node in a given layer connects to *every* node in the next layer - it is often more convenient to write the backprop algorithm in matrix notation rather than using more general graph form given above. In this notation, the biases weights, net inputs, activations, and error signals for all units in a layer are combined into vectors, while all the non-bias weights from one layer to the next form a matrix W. Layers are numbered from 0 (the input layer) to L (the output layer). The backprop algorithm then looks as follows:

1. Initialise the input layer:

   $$\vec{y}_0 = \vec{x}$$

2. Propagate activity forward: for l = 1, 2, ..., L,

   $$\vec{y}_l = f_l(W_l \vec{y}_{l-1} + \vec{b}_l)$$

   where b_l is the vector of bias weights.

3. Calculate the error in the output layer:

   $$\vec{\delta}_L = \vec{t} - \vec{y}_L$$

4. Backpropagate the error: for l = L-1, L-2, ..., 1,

   $$\vec{\delta}_l = (W_{l+1}^T \vec{\delta}_{l+1}) \cdot f_l'(\vec{net}_l)$$

where T is the matrix transposition operator.

5. Update the weights and biases:

$$\Delta W_l = \vec{\delta}_l \vec{y}_{l-1}^T \qquad \Delta \vec{b}_l = \vec{\delta}_l$$

You can see that this notation is significantly more compact than the graph form, even though it describes exactly the same sequence of operations.

9. Overfitting

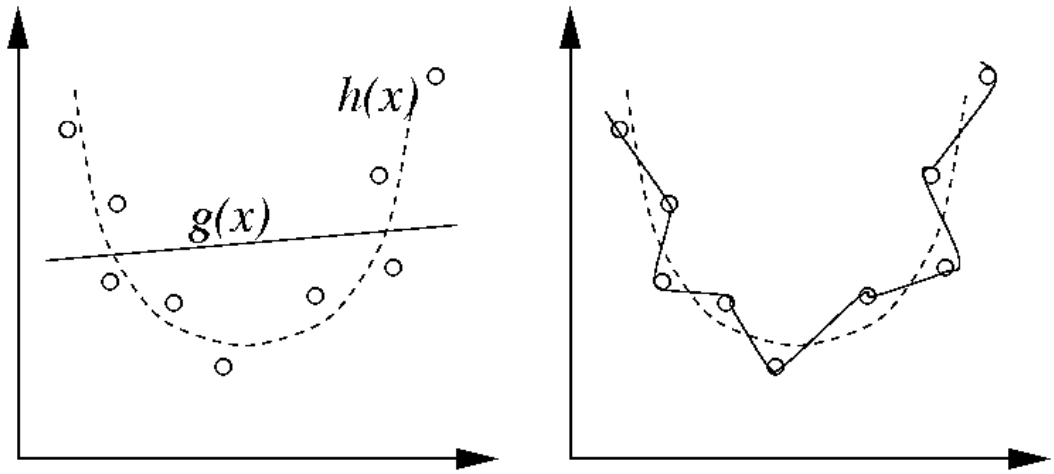
In the previous example we used a network with two hidden units. Just by looking at the data, it was possible to guess that two tanh functions would do a pretty good job of fitting the data. In general, however, we may not know how many hidden units, or equivalently, how many weights, we will need to produce a reasonable approximation to the data. Furthermore, we usually seek a model of the data which will give us, on average, the best possible predictions for novel data. This goal can conflict with the simpler task of modelling a specific training set well. In this section we will look at some techniques for preventing our model becoming too powerful (overfitting). In the next, we address the related question of selecting an appropriate architecture with just the right amount of trainable parameters.

9.1. Bias-variance tradeoff

Consider the two fitted functions below. The data points (circles) have all been generated from a smooth function, *h(x)*, with some added noise. Obviously, we want to end up with a model which approximates *h(x)*, given a specific set of data *y(x)* generated as:

$$y(x) = h(x) + \epsilon \qquad (1)$$

In the left hand panel we try to fit the points using a function *g(x)* which has too few parameters: a straight line. The model has the virtue of being simple; there are only two free parameters. However, it does not do a good job of fitting the data, and would not do well in predicting new data points. We say that the simpler model has a high **bias**.



The right hand panel shows a model which has been fitted using too many free parameters. It does an excellent job of fitting the data points, as the error at the data points is close to zero. However it would not do a good job of predicting *h(x)* for new values of *x*. We say that the model has a high **variance**. The model does not reflect the structure which we expect to be present in *any* data set generated by equation (1) above.

Clearly what we want is something in between: a model which is powerful enough to represent the underlying structure of the data (*h(x)*), but not so powerful that it faithfully models the noise associated with this particular data sample.

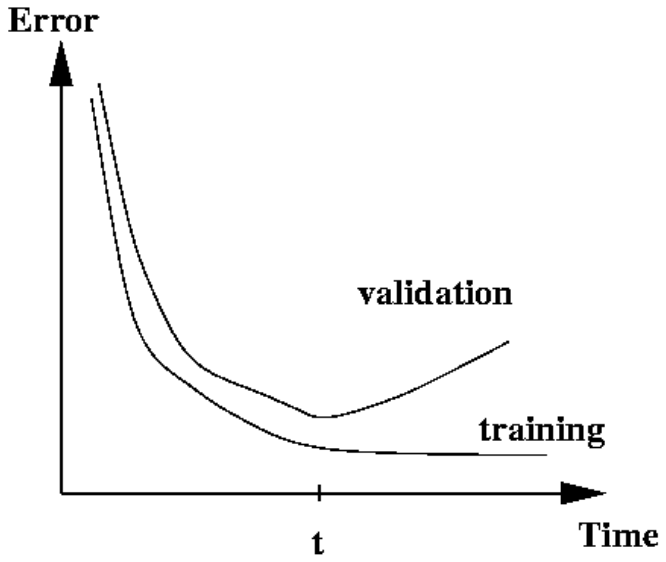
The bias-variance trade-off is most likely to become a problem if we have relatively few data points. In the opposite case, where we have essentially an infinite number of data points (as in continuous online learning), we are not usually in danger of overfitting the data, as the noise associated with any single data point plays a vanishingly small role in our overall fit. The following techniques therefore apply to situations in which we have a finite data set, and, typically, where we wish to train in batch mode.

9.2. Preventing overfitting

Early stopping

One of the simplest and most widely used means of avoiding overfitting is to divide the data into two sets: a training set and a **validation** set. We train using only the training data. Every now and then, however, we stop training, and test network performance on the independent validation set. No weight updates are made during this test! As the validation data is independent of the training data, network performance is a good measure of generalization, and as long as the network is learning the underlying structure of the data (*h(x)* above), performance on the validation set will improve with training. Once the network stops learning things which are expected to be true of any data sample and learns things which are true only of this sample (epsilon in Eqn 1 above), performance on the validation set will stop improving, and will typically get

worse. Schematic learning curves showing error on the training and validation sets are shown below. To avoid overfitting, we simply stop training at time *t*, where performance on the validation set is optimal.



One detail of note when using early stopping: if we wish to test the trained network on a set of independent data to measure its ability to generalize, we need a third, independent, test set. This is because we used the validation set to decide when to stop training, and thus our trained network is no longer entirely independent of the validation set. The requirements of independent training, validation and test sets means that early stopping can only be used in a data-rich situation.

Weight decay

The over-fitted function above shows a high degree of curvature, while the linear function is maximally smooth. **Regularization** refers to a set of techniques which help to ensure that the function computed by the network is no more curved than necessary. This is achieved by adding a penalty to the error function, giving:

$$\tilde{E} = E + \nu\Omega \qquad (2)$$

One possible form of the regularizer comes from the informal observation that an over-fitted mapping with regions of large curvature requires large weights. We thus penalize large weights by choosing

$$\Omega = \frac{1}{2}\sum_i w_i^2 \qquad (3)$$

Using this modified error function, the weights are now updated as

$$\Delta w_{ij} = -\mu \frac{\partial \tilde{E}}{\partial w_{ij}} = -\mu \frac{\partial E}{\partial w_{ij}} - \mu \nu w_{ij} \quad (4)$$

where the right hand term causes the weight to decrease as a function of its own size. In the absence of any input, all weights will tend to decrease exponentially, hence the term "weight decay".

Training with noise

A final method which can often help to reduce the importance of the specific noise characteristics associated with a particular data sample is to add an extra small amount of noise (a small random value with mean value of zero) to each input. Each time a specific input pattern x is presented, we add a different random number, and use $x + \varepsilon$ instead.

At first, this may seem a rather odd thing to do: to deliberately corrupt ones own data. However, perhaps you can see that it will now be difficult for the network to approximate any specific data point too closely. In practice, training with added noise has indeed been shown to reduce overfitting and thus improve generalization in some situations.

If we have a finite training set, another way of introducing noise into the training process is to use online training, that is, updating weights after every pattern presentation, and to randomly reorder the patterns at the end of each training epoch. In this manner, each weight update is based on a noisy estimate of the true gradient.

10. Growing and pruning networks

The neural network modeller is faced with a huge array of models and training regimes from which to select. In this presentation we only introduce the most common and general models. However, even after deciding, for example, to train a simple feed forward network, using some specific form of gradient descent, with tanh nodes in a single hidden layer, an important question to be addressed is remains: how big a network should we choose? How many hidden units and consequently how many weights?

By way of an example, the nonlinear data which formed our first example can be fitted very well using 40 tanh functions. Learning with 40 hidden units is considerably harder than learning with 2, and takes significantly longer. The resulting fit is no better (as measured by the sum squared error) than the 2-unit model.

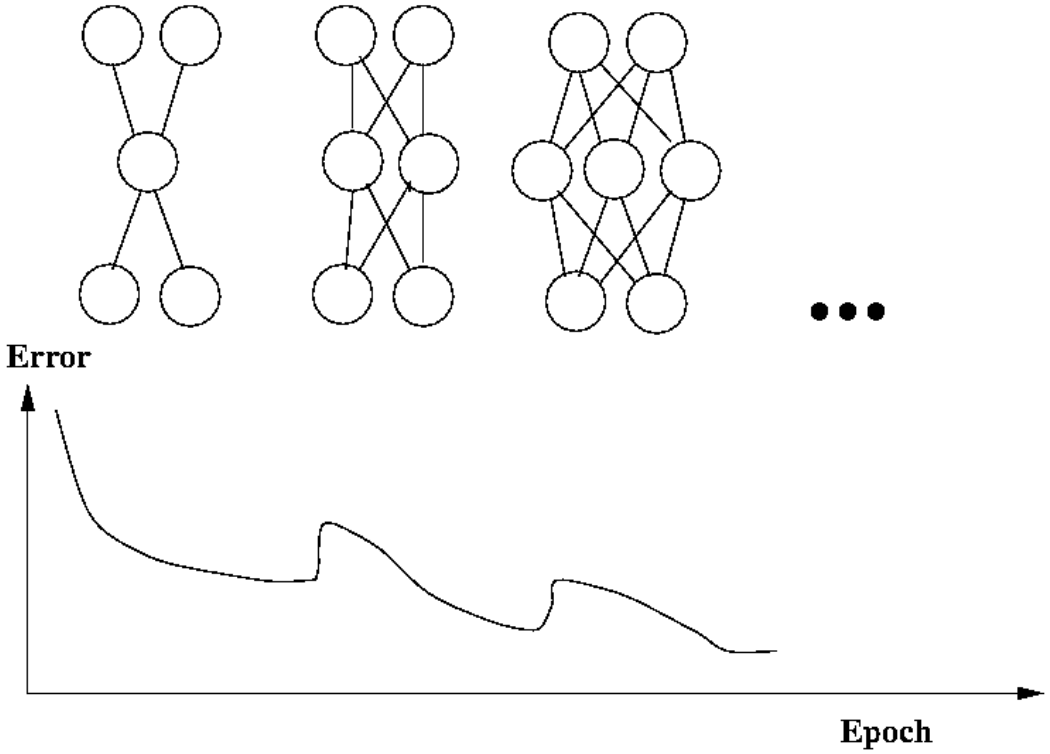
The most usual answer is not necessarily the best: we guess an appropriate number (as we did above).

Another common solution is to try out several network sizes, and select the most promising. Neither of these methods is very principled.

Two more rigorous classes of methods are available, however. We can either start with a network which we know to be too small, and iteratively add units and weights, or we can train an oversized network and remove units/weights from the final network. We will look briefly at each of these approaches.

10.1. Growing networks

The simplest form of network growing algorithm starts with a small network, say one with only a single hidden unit. The network is trained until the improvement in the error over one epoch falls below some threshold. We then add an additional hidden unit, with weights from inputs and to outputs. We initialize the new weights randomly and resume training. The process continues until no significant gain is achieved by adding an extra unit. The process is illustrated below.



10.2. Cascade correlation

Beyond simply having too many parameters (danger of overfitting), there is a problem with large networks which has been called the *herd effect*. Imagine we have a task which is essentially decomposable into two sub-tasks *A* and *B*. We have a number of hidden units and randomly weighted connections. If task *A* is responsible for most of the error signal arriving at the hidden units, there will be a tendency for all units to simultaneously try to solve *A*. Once the error attributable to *A* has been reduced, error

from subtask *B* will predominate, and all units will now try to solve that, leading to an increase again in the error from *A*. Eventually, due mainly to the randomness in the weight initialization, the herd will split and different units will address different sub-problems, but this may take considerable time.

To get around this problem, Fahlman (1991) proposed an algorithm called **cascade correlation** which begins with a minimal network having just input and output units. Training a single layer requires no back-propagation of error and can be done very efficiently. At some point further training will not produce much improvement. If network performance is satisfactory, training can be stopped. If not, there must be some remaining error which we wish to reduce some more. This is done by adding a new hidden unit to the network, as described in the next paragraph. The new unit is added, its input weights are frozen (i.e. they will no longer be changed) and all output weights are once again trained. This is repeated until the error is small enough (or until we give up).

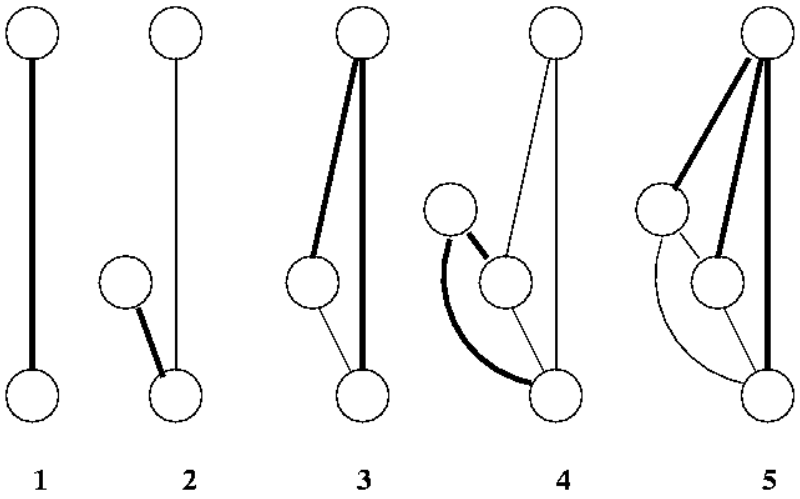
To add a hidden unit,we begin with a candidate unit and provide it with incoming connections from the input units and from all existing hidden units. We do not yet give it any outgoing connections. The new unit's input weights are trained by a process similar to gradient descent. Specifically, we seek to *maximize* the covariance between *v*, the new unit's value, and E_o, the output error at output unit *o*.

We define *S* as:

$$S = \sum_o \left| \sum_p (v^p - \bar{v})(E_o^p - \bar{E}_o) \right| \qquad (1)$$

where *o* ranges over the output units and *p* ranges over the input patterns. The terms $\bar{v}$, $\bar{E}_o$ are the mean values of *v* and E_o over all patterns. Performing gradient *ascent* on the partial derivative $\frac{\partial S}{\partial w_i}$ (we will skip the explicit formula here) ensures that we end up with a unit whose activation is maximally correlated (positively or negatively) with the remaining error. Once we have maximized *S*, we freeze the input weights, and install the unit in the network as described above. The whole process is illustrated below.

In (1) we train the weights from input to output. In (2), we add a candidate unit and train its weights to maximize the correlation with the error. In (3) we retrain the output layer, (4) we train the input weights for another hidden unit, (5) retrain the output layer, etc. Because we train only one layer at a time, training is very quick. What is more, because the weights feeding into each hidden unit do not change once the unit has been added, it is possible to record and store the activations of the hidden units for each pattern, and reuse these values without recomputation in later epochs.



10.3. Pruning networks

An alternative approach to growing networks is to start with a relatively large network and then remove weights so as to arrive at an optimal network architecture. The usual procedure is as follows:

1. Train a large, densely connected, network with a standard training algorithm
2. Examine the trained network to assess the relative importance of the weights
3. Remove the least important weight(s)
4. retrain the pruned network
5. Repeat steps 2-4 until satisfied

Deciding which are the least important weights is a difficult issue for which several heuristic approaches are possible. We can estimate the amount by which the error function E changes for a small change in each weight. The computational form for this estimate is outside the scope of this presentation. Various forms of this technique have been called **optimal brain damage**, and **optimal brain surgeon**.

11. Preconditioning the network

11.1. Ill-Conditioning

In the preceding section on overfitting, we have seen what can happen when the network learns a given set of data *too* well. Unfortunately a far more frequent problem encountered by backpropagation users is just the opposite: that the network does not learn well at all! This is usually due to **ill-conditioning** of the network.

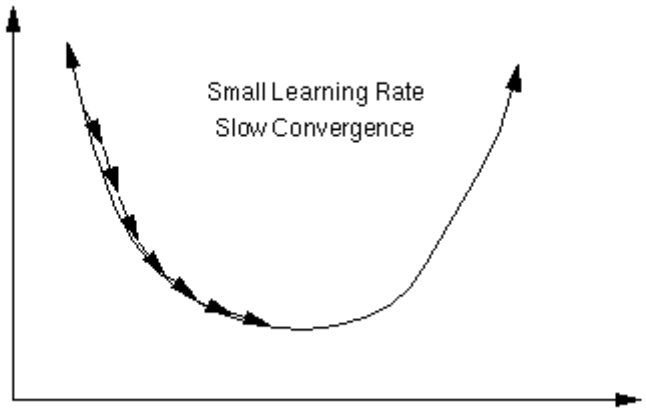


Figure 1a.

Recall that gradient descent requires a reasonable learning rate to work well: if it is too low (Fig. 1a), convergence will be very slow; set it too high, and the network will diverge (Fig. 1b).

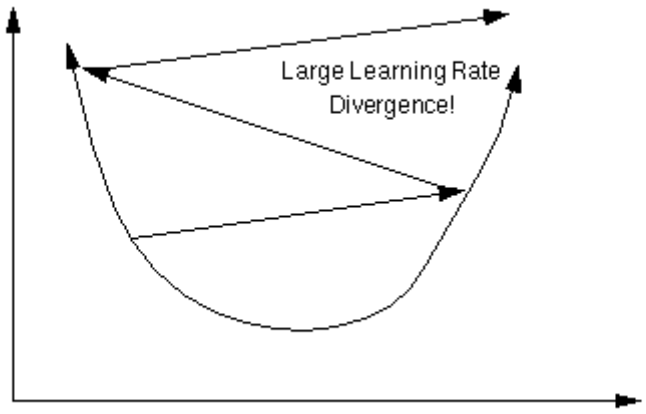


Figure 1b.

Unfortunately the best learning rate is typically *different* for each weight in the network! Sometimes these differences are small enough for a single, global compromise learning rate to work well - other times not. We call a network ill-conditioned if it requires learning rates for its weights that differ by so much that there is no global rate at which the network learns reasonably well. The error function for such a network is characterized by long, narrow valleys:

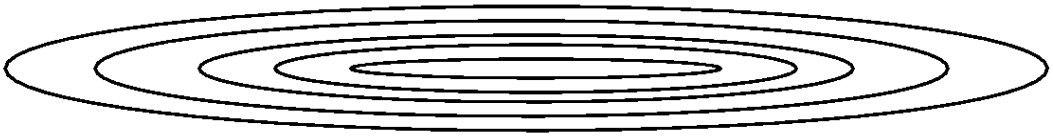


Figure 2.

(Mathematically, ill-conditioning is characterized by a high **condition number**. The condition number is the ratio between the largest and the smallest eigenvalue of the network's **Hessian**. The Hessian is the matrix of second derivatives of the loss function with respect to the weights. Although it is possible to calculate the Hessian for a multi-layer network and determine its condition number explicitly, it is a rather complicated procedure, and rarely done.)

Ill-conditioning in neural networks can be caused by the training data, the network's architecture, and/or its initial weights. Typical problems are: having large inputs or target valuess, having both large and small layers in the network, having more than one hidden layer, and having initial weights that are too large or too small. This should make it clear that ill-conditioning is a *very* common problem indeed! In what follows, we look at each possible source of ill-conditioning, and describe a simple method to remove the problem. Since these methods are all used *before* training of the network begins, we refer to them as **preconditioning** techniques.

11.2. Normalizing inputs and targets

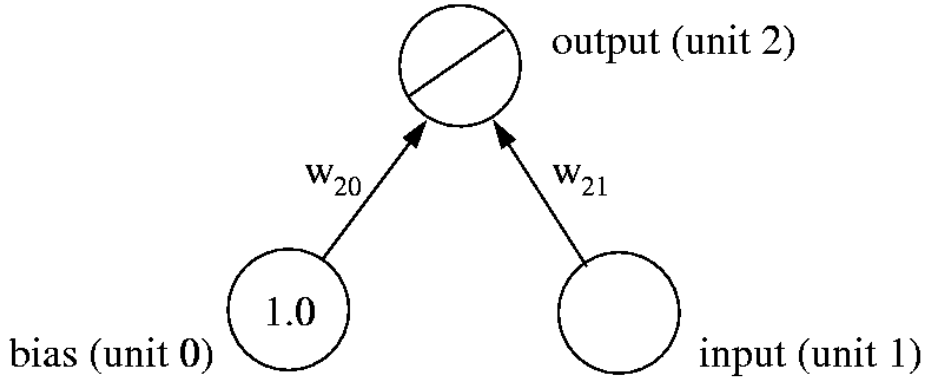


Figure 3.

Recall the simple linear network (Fig. 3) we first used to learn the car data set. When we presented the best linear fit, we had rescaled both the x (input) and y (target) axes. Why did we do this? Consider what would happen if we used the original data directly instead: the input (weight of the car) would be quite large - over 3000 (pounds) on average. To map such large inputs onto the far smaller targets, the weight from input to output must become quite small - about -0.01. Now assume that we are 10% (0.001) away from the optimal value. This would cause an error of (typically) 3000*0.001 = 3 at the output. At learning rate μ, the weight change resulting from this error would be

$\mu*3*3000 = 9000\ \mu$. For stable convergence, this should be smaller than the distances to the weight's optimal value: $9000\ \mu < 0.001$, giving us $\mu < 10^{-7}$, a *very* small learning rate. (And this is for online learning - for batch learning, where the weight changes for several patterns are added up, the learning rate would have to be even smaller!)

Why should such a small learning rate be a problem? Consider that the bias unit has a constant output of 1. A bias weight that is, say, 0.1 away from its optimal value would therefore have a gradient of 0.1. At a learning rate of 10^{-7}, however, it would take 10 million steps to move the bias weight by this distance! This is a clear case of ill-conditioning caused by the vastly different scale of input and bias values. The solution is simple: **normalize** the input, so that it has an average of zero and a standard deviation of one. Normalization is a two-step process:

To normalize a variable, first

1. (**centering**) subtract its average, then
2. (**scaling**) divide by its standard deviation.

Note that for our purposes it is not really necessary to calculate the mean and standard deviation of each input exactly - approximate values are perfectly sufficient. (In the case of the car data, the "mean" of 3000 and "standard deviation" of 1000 were simply guessed after looking at the data plot.) This means that in situations where the training data is not known in advance, estimates based on either prior knowledge or a small sample of the data are usually good enough. If the data is a time series x(t), you may also want to consider using the first differences x(t) - x(t-1) as network inputs instead; they have zero mean as long as x(t) is stationary. Whichever way you do it, remember that you should always

- normalize the inputs, and
- normalize the targets.

To see why the target values should also be normalized, consider the network we've used to fit a sigmoid to the car data (Fig. 4). If the target values were those found in the original data, the weight from hidden to output unit would have to be 10 times larger. The error signal propagated back to the hidden unit would thus be multiplied by 17 along the way. In order to compensate for this, the global learning rate would have to be lowered correspondingly, slowing down the weights that go directly to the output unit. Thus while large inputs cause ill-conditioning by leading to very small weights, large targets do so by leading to very large weights.

Finally, notice that the argument for normalizing the inputs can also be applied to the hidden units (which after all look like inputs to their posterior nodes). Ideally, we would like hidden unit activations as well to have a mean of zero and a standard deviation of one. Since the weights into hidden units keep changing during training, however, it would be rather hard to predict their mean and standard deviation accurately!

Fortunately we can rely on our tanh activation function to keep things reasonably well-conditioned: its range from -1 to +1 implies that the standard deviation cannot exceed 1, while its symmetry about zero means that the mean will typically be relatively small. Furthermore, its maximum derivative is also 1, so that backpropagated errors will be neither magnified nor attenuated more than necessary.

Note: For historic reasons, many people use the logistic sigmoid $f(u) = 1/(1 + e^{-u})$ as activation function for hidden units. This function is closely related to tanh (in fact, $f(u) = \tanh(u/2)/2 + 0.5$) but has a smaller, asymmetric range (from 0 to 1), and a maximum derivative of 0.25. We will later encounter a legitimate use for this function, but as activation function for hidden units it tends to worsen the network's conditioning. Thus

- do not use the logistic sigmoid $f(u) = 1/(1 + e^{-u})$ as activation function for hidden units.

Use tanh instead: your network will be better conditioned.

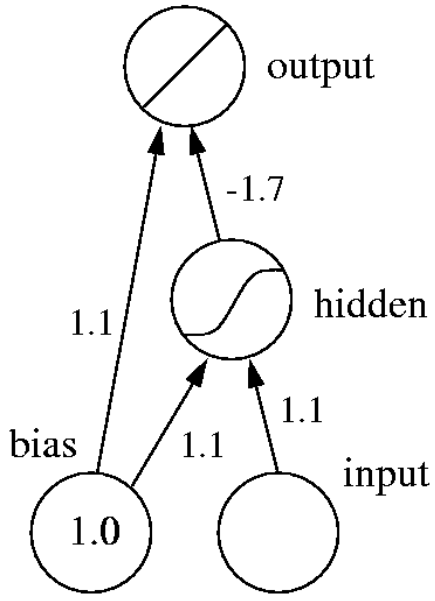


Figure 4.

11.3 Initialising the weights

Before training, the network weights are initialized to small random values. The random values are usually drawn from a uniform distribution over the range [-r,r]. What should r be? If the initial weights are too small, both activation and error signals will die out along their way through the network. Conversely, if they are too large, the tanh function of the hidden units will **saturate** - be very close to its asymptotic value of +/-1. This means that its derivative will be close to zero, blocking any backpropagated error signals from passing through the node; this is sometimes called **paralysis** of the node.

To avoid either extreme, we would initally like the hidden units' net input to be approximately normalized. We do not know the inputs to the node, but we do know that they're approximately normalized - that's what we ensured in the previous section. It seems reasonable then to model the expected inputs as independent, normalized random variables. This means that their variances add, so we can write

$$Var(net_i) \approx \sum_{j \in A_i} Var(w_{ij} y_j) = \sum_{j \in A_i} w_{ij}^2 Var(y_j) \approx \sum_{j \in A_i} w_{ij}^2 \leq |A_i| r^2$$

since the initial weights are in the range [-r,r]. To ensure that Var(net_i) is at most 1, we can thus set r to the inverse of the square root of the **fan-in** $|A_i|$ of the node - the number of weights coming into it:

- initialize weight w_{ij} to a uniformly random value in the range $[-r_i, r_i]$, where $r_i = \frac{1}{\sqrt{|A_i|}}$

11.4. Setting local learning rates

Above we have seen that the **architecture** of the network - specifically: the fan-in of its nodes - determines the range within which its weights should be initialized. The architecture also affects how the error signal scales up or down as it is backpropagated through the network. Modelling the error signals as independent random variables, we have

$$Var(\delta_j) \approx \sum_{i \in P_j} Var(\delta_i w_{ij}) = \sum_{i \in P_j} Var(\delta_i) w_{ij}^2 \leq \sum_{i \in P_j} \frac{Var(\delta_i)}{|A_i|}$$

Let us define a new variable v for each hidden or output node, proportional to the (estimated) variance of its error signal divided by its fan-in. We can calculate all the v by a backpropagation procedure:

- for all output nodes o, set $v_o = \frac{1}{|A_o|}$
- backpropagate: for all hidden nodes j, calculate $v_j = \frac{1}{|A_j|} \sum_{i \in P_j} v_i$

Since the activations in the network are already normalized, we can expect the gradient for weight w_{ij} to scale with the square root of the corresponding error signal's variance, $v_i|A_i|$. The resulting weight change, however, should be commensurate with the characteristic size of the weight, which is given by r_i. To achieve this,

- set the learning rate μ_i (used for all weights w_{ij} into node i) to $\mu_i = \frac{1}{|A_i|\sqrt{v_i}}$

If you follow all the points we have made in this section before the start of training, you should have a reasonably well-conditioned network that can be trained effectively. It remains to determine a good global learning rate μ. This must be done by trial and error; a good first guess (on the high size) would be the inverse of the square root of the batch size (by a similar argument as we have made above), or 1 for online learning. If this leads to divergence, reduce μ and try again.

12. Momentum and learning rate adaptation

12.1. Local minima

In gradient descent we start at some point on the error function defined over the weights, and attempt to move to the **global minimum** of the function. In the simplified function of Fig 1a the situation is simple. Any step in a downward direction will take us closer to the global minimum. For real problems, however, error surfaces are typically complex, and may more resemble the situation shown in Fig 1b. Here there are numerous **local minima**, and the ball is shown trapped in one such minimum. Progress here is only possible by climbing higher before descending to the global minimum.

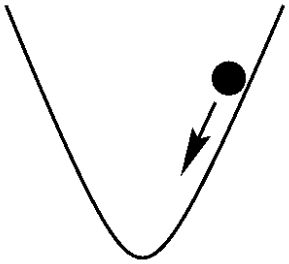


Figure 1a.

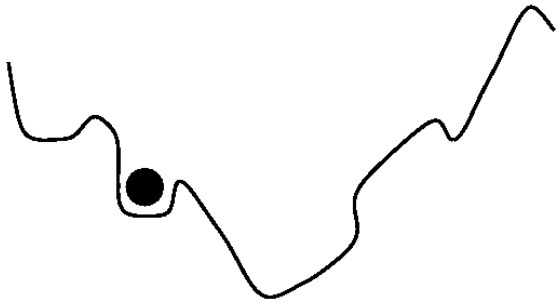


Figure 1b

We have already mentioned one way to escape a local minimum: use online learning. The noise in the stochastic error surface is likely to bounce the network out of local minima as long as they are not too severe.

12.2. Momentum

Another technique that can help the network out of local minima is the use of a **momentum** term. This is probably the most popular extension of the backprop algorithm; it is hard to find cases where this is not used. With momentum m, the weight update at a given time t becomes

$$\Delta w_{ij}(t) = \mu_i \delta_i y_j + m \Delta w_{ij}(t-1) \qquad (1)$$

where $0 < m < 1$ is a new global parameter which must be determined by trial and error. Momentum simply adds a fraction m of the previous weight update to the current one. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum. It is otherefore often necessary to reduce the global learning rate μ when using a lot of momentum (m close to 1). If you combine a high learning rate with a lot of momentum, you will rush past the minimum with huge steps!

When the gradient keeps changing direction, momentum will smooth out the variations. This is particularly useful when the network is not well-conditioned. In such cases the error surface has substantially different curvature along different directions, leading to the formation of long narrow valleys. For most points on the surface, the gradient does not point towards the minimum, and successive steps of gradient descent can oscillate from one side to the other, progressing only very slowly to the minimum (Fig. 2a). Fig. 2b shows how the addition of momentum helps to speed up convergence to the minimum by damping these oscillations.

Figure 2a.

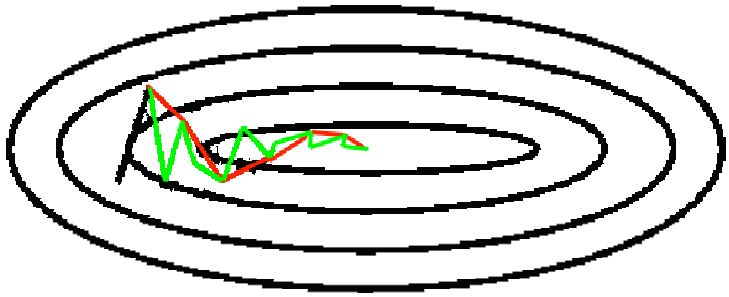


Figure 2b

12.3. Learning rate adaptation

In the section on preconditions, we have employed simple heuristics to arrive at reasonable guesses for the global and local learning rates. It is possible to refine these values significantly once training has commenced, and the network's response to the data can be observed. We will now introduce a few methods that can do so *automatically* by adapting the learning rates during training.

Bold Driver

A useful batch method for adapting the global learning rate μ is the **bold driver** algorithm. Its operation is simple: after each epoch, compare the network's loss E(t) to its previous value, E(t-1). If the error has decreased, increase μ by a small proportion (typically 1%-5%). If the error has increased by more than a tiny proportion (say, 10^{-10}), however, undo the last weight change, and decrease μ sharply - typically by 50%. Thus bold driver will keep growing μ slowly until it finds itself taking a step that has clearly gone too far up onto the opposite slope of the error function. Since this means that the network has arrived in a tricky area of the error surface, it makes sense to reduce the step size quite drastically at this point.

Annealing

Unfortunately bold driver cannot be used in this form for online learning: the stochastic fluctuations in E(t) would hopelessly confuse the algorithm. If we keep μ fixed, however, these same fluctuations prevent the network from ever properly converging to the minimum - instead we end up randomly dancing around it. In order to actually reach the minimum, and stay there, we must **anneal** (gradually lower) the global learning rate. A simple, non-adaptive annealing schedule for this purpose is the **search-then-converge** schedule

$$\mu(t) = \mu(0)/(1 + t/T) \qquad (2)$$

Its name derives from the fact that it keeps μ nearly constant for the first T training patterns, allowing the network to find the general location of the minimum, before annealing it at a (very slow) pace that is known from theory to guarantee convergence to the minimum. The characteristic time T of this schedule is a new free parameter that must be determined by trial and error.

Local Rate Adaptation

If we are willing to be a little more sophisticated, we go a lot further than the above global methods. First let us define an online weight update that uses a local, time-varying learning rate for each weight:

$$w_{ij}(t+1) = w_{ij}(t) + \mu_{ij}(t)\,\Delta w_{ij}(t) \qquad (3)$$

The idea is to adapt these local learning rates by gradient descent, while simultaneously adapting the weights. At time t, we would like to change the learning rate (before changing the weight) such that the loss E(t+1) at the next time step is reduced. The gradient we need is

$$\frac{\partial E(t+1)}{\partial \mu_{ij}(t)} = \frac{\partial E(t+1)}{\partial w_{ij}(t+1)} \frac{\partial w_{ij}(t+1)}{\partial \mu_{ij}(t)} = -\Delta w_{ij}(t)\,\Delta w_{ij}(t-1) \qquad (4)$$

Ordinary gradient descent in μ_{ij}, using the meta-learning rate q (a new global parameter), would give

$$\mu_{ij}(t) = \mu_{ij}(t-1) + q\,\Delta w_{ij}(t)\,\Delta w_{ij}(t-1) \tag{5}$$

We can already see that this would work in a similar fashion to momentum: increase the learning rate as long as the gradient keeps pointing in the same direction, but decrease it when you land on the opposite slope of the loss function.

Problem: μ_{ij} might become negative! Also, the step size should be proportional to μ_{ij} so that it can be adapted over several orders of magnitude. This can be achieved by performing the gradient descent on $\log(\mu_{ij})$ instead:

$$\log(\mu_{ij}(t)) = \log(\mu_{ij}(t-1)) + q\,\Delta w_{ij}(t)\,\Delta w_{ij}(t-1) \tag{6}$$

Exponentiating this gives

$$\begin{aligned} \mu_{ij}(t) &= \mu_{ij}(t-1)\, e^{q\,\Delta w_{ij}(t)\,\Delta w_{ij}(t-1)} \\ &\approx \mu_{ij}(t-1)\, \max(0.5,\ 1 + q\,\Delta w_{ij}(t)\,\Delta w_{ij}(t-1)) \end{aligned} \tag{7}$$

where the approximation serves to avoid an expensive exp function call. The multiplier is limited below by 0.5 to guard against very small (or even negative) factors.

Problem: the gradient is noisy; the product of two of them will be even noisier - the learning rate will bounce around a lot. A popular way to reduce the stochasticity is to replace the gradient at the previous time step (t-1) by an **exponential average** of past gradients. The exponential average of a time series u(t) is defined as

$$\bar{u}(t) = m\,\bar{u}(t-1) + (1-m)\,u(t) \tag{8}$$

where $0 < m < 1$ is a new global parameter.

Problem: if the gradient is ill-conditioned, the product of two gradients will be even worse - the condition number is squared. We will need to normalize the step sizes in some way. A radical solution is to throw away the magnitude of the step, and just keep the sign, giving

$$\mu_{ij}(t) = \begin{cases} r\,\mu_{ij}(t-1) & \text{if } \Delta w_{ij}(t)\,\overline{\Delta w_{ij}}(t-1) > 0 \\ (1/r)\,\mu_{ij}(t-1) & \text{otherwise} \end{cases} \tag{9}$$

where $r = e^q$. This works fine for batch learning, but...

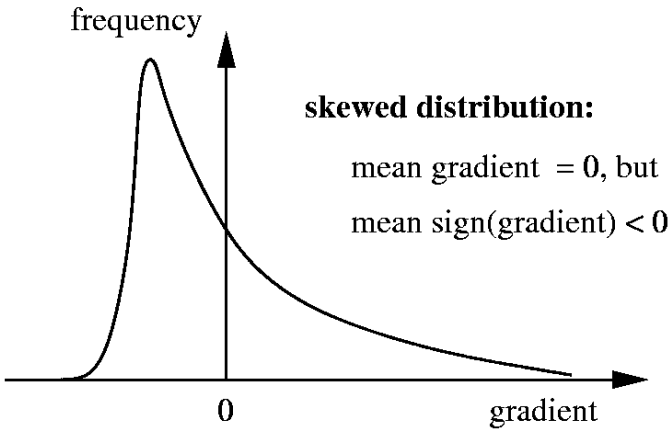


Figure 3.

Problem: Nonlinear normalizers such as the sign function lead to systematic errors in stochastic gradient descent (Fig. 3): a skewed but zero-mean gradient distribution (typical for stochastic equilibrium) is mapped to a normalized distribution with non-zero mean. To avoid the problems this is causing, we need a linear normalizer for online learning. A good method is to divide the step by $\overline{\Delta w_{ij}^2}$, an exponential average of the squared gradient. This gives

$$\mu_{ij}(t) \;=\; \mu_{ij}(t-1)\,\max\left(0.5,\; 1 + q\,\Delta w_{ij}(t)\,\frac{\overline{\Delta w_{ij}}(t-1)}{\overline{\Delta w_{ij}^2}(t)}\right) \quad (10)$$

Problem: successive training patterns may be correlated, causing the product of stochastic gradients to behave strangely. The exponential averaging does help to get rid of short-term correlations, but it cannot deal with input that exhibits correlations across long periods of time. If you are iterating over a fixed training set, make sure you **permute** (shuffle) it before each iteration to destroy any correlations. This may not be possible in a true online learning situation, where training data is received one pattern at a time.