TRAINING AND LOSS

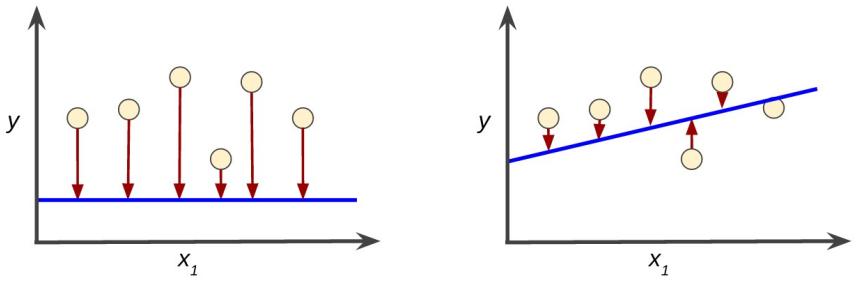
- **Training** a model simply means learning (determining) good values for all the weights and the bias from labeled examples
- In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called **empirical risk minimization**

TRAINING AND LOSS

- Loss is the penalty for a bad prediction. That is, **loss** is a number indicating how bad the model's prediction was on a single example
- If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater
- The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples

TRAINING AND LOSS

- For example, Figure shows a high loss model on the left and a low loss model on the right.



LOSS FUNCTION

- A mathematical function
- A Loss function
  - that would aggregate the individual losses in a meaningful fashion.

Squared loss: a popular loss function

The linear regression models we'll examine here use a loss function called **squared loss** (also known as **L_2 loss**). The squared loss for a single example is as follows:

```
= the square of the difference between the label and the prediction
= (observation - prediction(x))^2
= (y - y')^2
```

LOSS FUNCTION

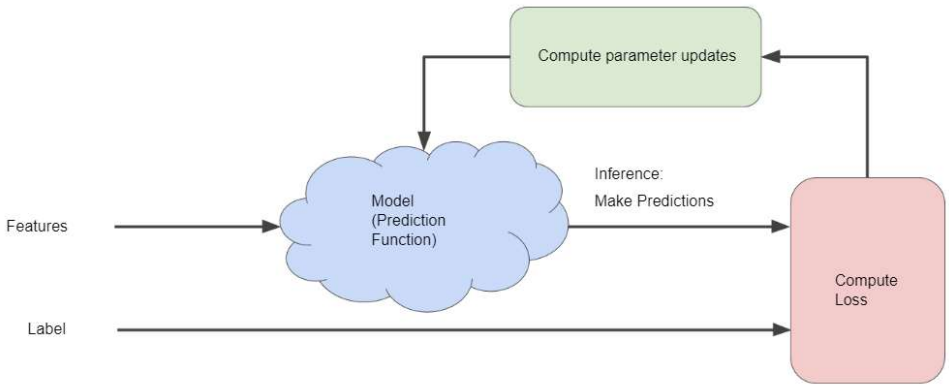
Mean square error (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

where:

- (x, y) is an example in which
  - x is the set of features (for example, chirps/minute, age, gender) that the model uses to make predictions.
  - y is the example's label (for example, temperature).
- $prediction(x)$ is a function of the weights and bias in combination with the set of features x.
- D is a data set containing many labeled examples, which are (x, y) pairs.
- N is the number of examples in D.

REDUCING LOSS



LOSS FUNCTION

Mean square error (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:

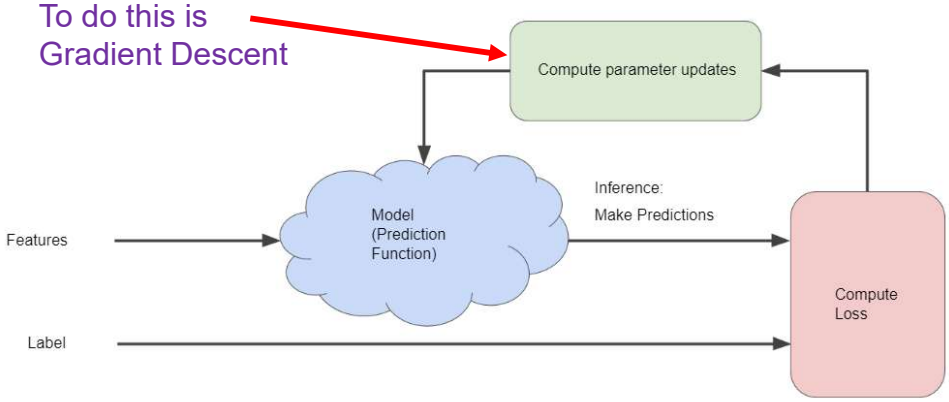
$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

where:

- (x, y) is an example in which
  - x is the set of features (for example, chirps/minute, age, gender) that the model uses to make predictions.
  - y is the example's label (for example, temperature).
- $prediction(x)$ is a function of the weights and bias in combination with the set of features x.
- D is a data set containing many labeled examples, which are (x, y) pairs.
- N is the number of examples in D.

REDUCING LOSS

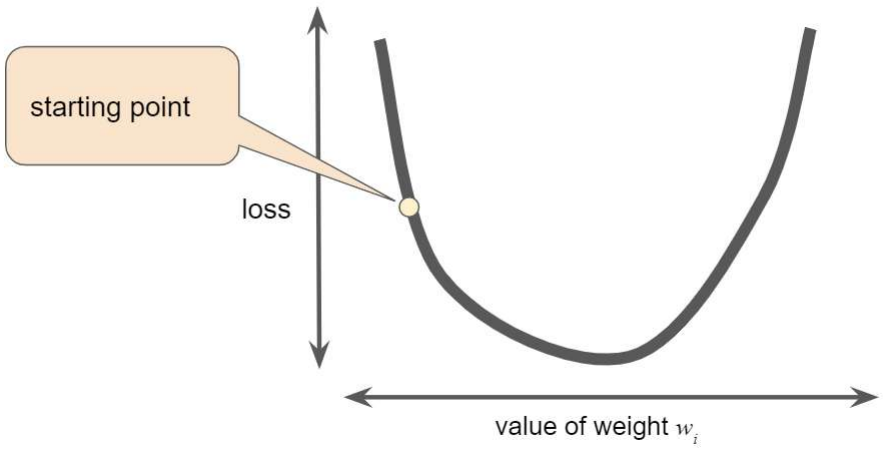
More substantial approach
To do this is
Gradient Descent



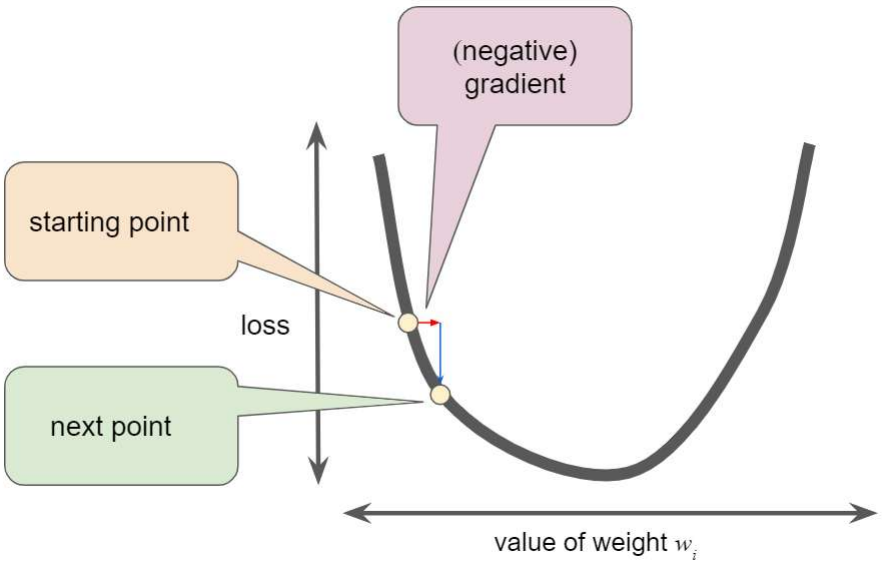
REDUCING LOSS

- The green hand-wavy box entitled "Compute parameter updates." the algorithmic fairy dust can be replaced with something more substantial
- Gradient Descent

REDUCING LOSS



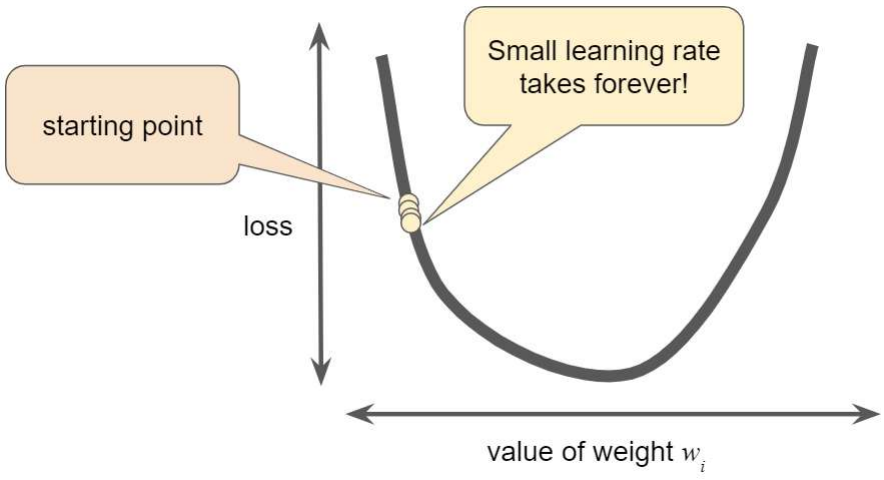
REDUCING LOSS



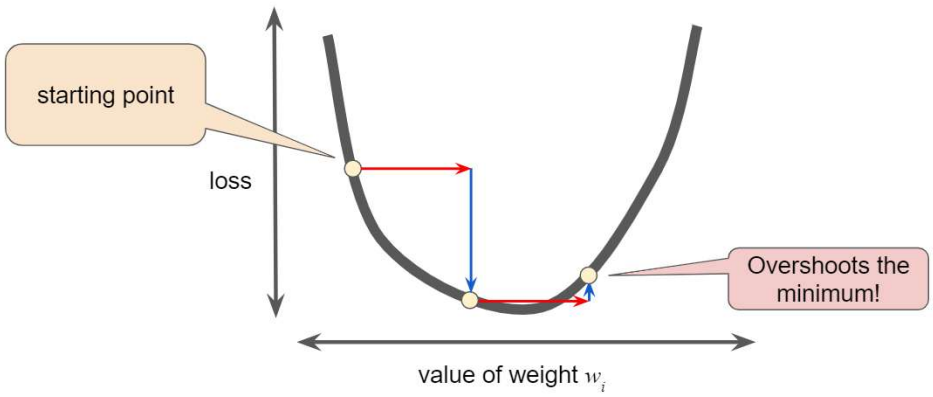
REDUCING LOSS-LEARNING RATE

- The gradient vector has both a direction and a magnitude
- Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point

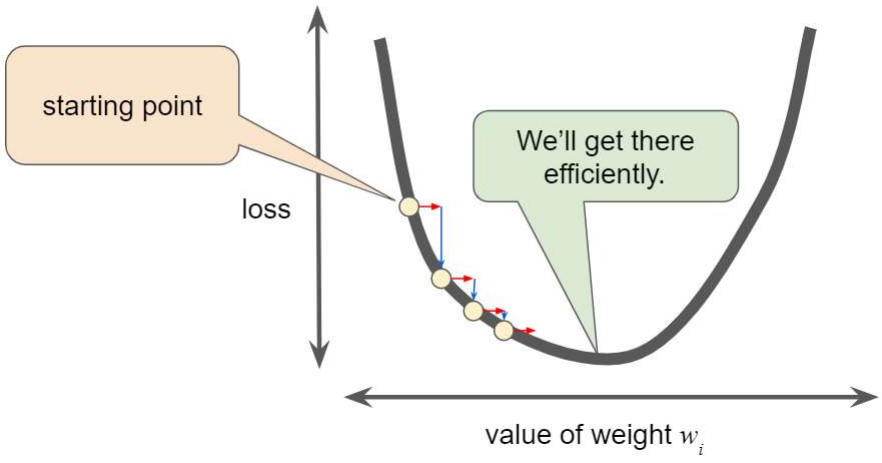
REDUCING LOSS-LEARNING RATE



REDUCING LOSS-LEARNING RATE

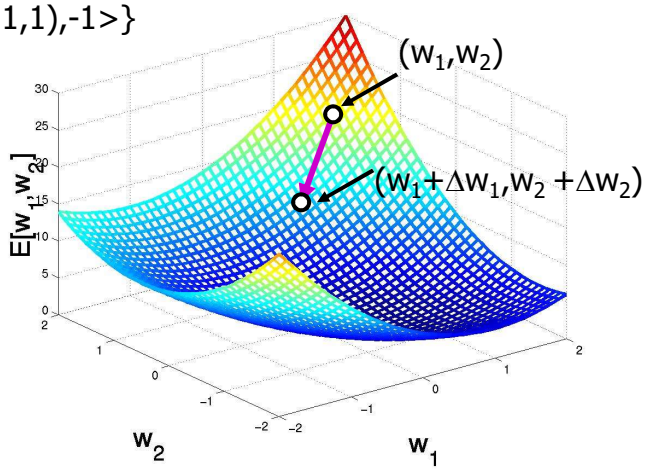


REDUCING LOSS-LEARNING RATE



Gradient Descent

D={<(1,1),1>,<(-1,-1),1>,
<(1,-1),-1>,<(-1,1),-1>}

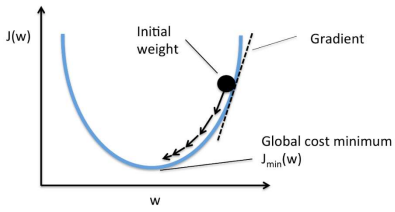


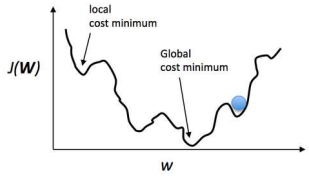
TRAINING PROCEDURE

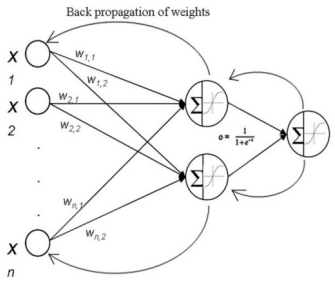
It involves three stages:

- Feed-forward of the input training pattern
- Calculation and Backpropagation of the associated error
- Adjustment of the weights

Gradient Descent- ANN Training Algorithm







Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}

Incremental Gradient Descent

- start from an arbitrary point in the weight space
- the direction in which the error E of an example (as a function of the weights) is decreasing most rapidly is the opposite of the gradient of E:

$$-(\text{gradient of } E(\mathrm{w(n)})) = -\left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}\right]$$

- take a small step (of size η) in that direction

$$\mathrm{w(n+1)} = \mathrm{w(n)} - \eta(\text{gradient of } \mathrm{E(w(n))})$$

Supervised Learning

- ***The aim is to minimize the operation error of the network.***

- Perceptron learning rule(applicable to a single neuron)

$$\mathbf{w_i^{t+1} = w_i^t + c \cdot (d - o) \cdot x_i^t}$$

- Back-propagation algorithm (applicable to feed-forward ANNs)

$$\mathbf{E = \sum_{i=1}^{m} (d_i - o_i)^2}$$

$$\mathbf{w_i^{t+1} = w_i^t - c \cdot \frac{\partial E}{\partial w_i} \cdot x_i}$$

Weights Update Rule

- Computation of Gradient(E):

$$\frac{\partial E(\mathrm{w})}{\partial \mathrm{w}} = \mathrm{e}\frac{\partial \mathrm{e}}{\partial \mathrm{w}}$$
$$= \mathrm{e}[-\mathrm{x}^{\mathrm{T}}]$$

- Delta rule (update rule) for weight update:

$$\mathrm{w}(\mathrm{n}+1) = \mathrm{w}(\mathrm{n}) + \eta \mathrm{e}(\mathrm{n})\mathrm{x}(\mathrm{n})$$

Gradient Descent

- Train w_i's such that they minimize squared error
  - $E[w_1,\ldots,w_m] = \frac{1}{2} \Sigma_{d\in D} (t_d-o_d)^2$

Gradient:

$\nabla E[w]=[\partial E/\partial w_0,\ldots \partial E/\partial w_m]$

$\Delta w=-\eta \nabla E[w]$

$\Delta w_i=-\eta \partial E/\partial w_i$

$= -\eta \partial/\partial w_i \; 1/2\Sigma_d(t_d-o_d)^2$

$= -\eta \partial/\partial w_i \; 1/2\Sigma_d(t_d-\Sigma_i w_i x_i)^2$

$= -\eta \Sigma_d(t_d- o_d)(-x_i)$

TYPES OF GRADIENT DESCENT

- Batch mode : gradient descent

  $w=w - \eta \nabla E_D[w]$ over the entire data D

  $E_D[w]=1/2\Sigma_d(t_d-o_d)^2$

- Incremental mode: gradient descent

  $w=w - \eta \nabla E_d[w]$ over individual training examples d

  $E_d[w]=1/2 (t_d-o_d)^2$

IGD approximate BGD arbitrarily closely if η is small

TYPES-TWO EXTREME CASES

- **Batch** is the total number of examples to calculate the gradient in a single iteration,
  - The batch has been the entire data set
- At Google scale, data sets contain billions of examples & huge # of features
  - Consequently, a batch is enormous causing even a single iteration to take a long time to compute

SGD AND MINI-BATCH SGD

- Redundancy becomes more likely as the batch size grows with lesser predictive value at the cost of smoothing out noisy gradients
- The right gradient can be estimated noisily *on average* for much less computation by choosing examples at random from data set

SGD AND MINI-BATCH SGD

- **SGD** takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration
- Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random

SGD AND MINI-BATCH SGD

- **Mini-batch SGD** is a compromise between full-batch iteration and SGD
- A mini-batch is typically between 10 and 1,000 examples, chosen at random
- Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch

LMS learning algorithm (gradient descent of error)

n=1;

initialize **w**(n) randomly;

while (E_tot unsatisfactory and n<max_iterations)

Select an example (**x**(n),*d(n)*)

$$e(n) = d(n) - w(n)^T x(n)$$
$$w(n+1) = w(n) + \eta e(n) x(n)$$

n = n+1;

end-while;

1. Gradient descent refers to the method to hunt for the minimum-cost solution
2. It can use any particular cost function
3. LMS is a specialization using MSE cost function

η = learning rate parameter (real number)

A modification uses $$w(n+1) = w(n) + \eta\, e(n) \frac{x(n)}{\| x(n) \|}$$

MATLAB/PYTHON VARIATIONS OF GRADIENT DESCENT ALGORITHM

- Traingd
  - Gradient descent backpropagation
  - net = feedforwardnet(3,'traingd')

The following code creates a training set of inputs `p` and targets `t`. For batch training, all the input vectors are placed in one matrix.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
```

Create the feedforward network.

```
net = feedforwardnet(3,'traingd');
```

In this simple example, turn off a feature that is introduced later.

```
net.divideFcn = '';
```

At this point, you might want to modify some of the default training parameters.

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

If you want to use the default training parameters, the preceding commands are not necessary.

Now you are ready to train the network.

```
[net,tr] = train(net,p,t);
```

```
dX = lr * dperf/dX
```

MATLAB/PYTHON VARIATIONS OF GRADIENT DESCENT ALGORITHM

- traingdm

Gradient Descent with Momentum

In addition to `traingd`, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by `traingdm`, allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12–9 of [HDB96] for a discussion of momentum.

Gradient descent with momentum depends on two training parameters. The parameter `lr` indicates the learning rate, similar to the simple gradient descent. The parameter `mc` is the momentum constant that defines the amount of momentum. `mc` is set between 0 (no momentum) and values close to 1 (lots of momentum). A momentum constant of 1 results in a network that is completely insensitive to the local gradient and, therefore, does not learn properly.

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingdm');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net = train(net,p,t);
y = net(p)
```

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

MATLAB/PYTHON VARIATIONS OF GRADIENT DESCENT ALGORITHM

- traingda
  - Gradient descent with adaptive learning rate backpropagation

```
dX = lr*dperf/dX
```

At each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor `lr_inc`. If performance increases by more than the factor `max_perf_inc`, the learning rate is adjusted by the factor `lr_dec` and the change that increased the performance is not made.

Gradient Descent with Adaptive Learning Rate Backpropagation

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

You can improve the performance of the steepest descent algorithm if you allow the learning rate to change during the training process. An adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by `traingd`. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, `max_perf_inc` (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by `lr_dec` = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by `lr_inc` = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it is decreased until stable learning resumes.

Backpropagation training with an adaptive learning rate is implemented with the function `traingda`, which is called just like `traingd`, except for the additiona training parameters `max_perf_inc`, `lr_dec`, and `lr_inc`. Here is how it is called to train the previous two-layer network:

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingda');
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net = train(net,p,t);
y = net(p)
```

MATLAB/PYTHON VARIATIONS OF GRADIENT DESCENT ALGORITHM

- traingdx
  - Gradient descent with momentum and adaptive learning rate backpropagation

```
dX = mc*dXprev + lr*mc*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

Algorithms

The function `traingdx` combines adaptive learning rate with momentum training. It is invoked in the same way as `traingda`, except that it has the momentum coefficient `mc` as an additional training parameter.

`traingdx` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*mc*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor `lr_inc`. If performance increases by more than the factor `max_perf_inc`, the learning rate is adjusted by the factor `lr_dec` and the change that increased the performance is not made.

MATLAB/PYTHON VARIATIONS OF GRADIENT DESCENT ALGORITHM

- trainlm
  - Levenberg-Marquardt backpropagation

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainlm');
net = train(net, x, t);
```

Levenberg-Marquardt Algorithm

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} \quad (1)$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e} \quad (2)$$

where **J** is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and **e** is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

$$\mathbf{J} = \frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_1} \cdots \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_u}\right] = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_u} \\ \vdots & & \vdots \\ \frac{\partial f_v(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_v(\mathbf{x})}{\partial x_u} \end{bmatrix}$$

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

That is, the entry of the ith row and the jth column is

$$(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i\,\partial x_j}.$$