**newff**

**Purpose**

Create feed-forward backpropagation network

**Syntax**

\*

net = newff(P,T,[S1 S2...S(N-l)],{TF1 TF2...TFNl},

BTF,BLF,PF,IPF,OPF,DDF)

**Description**

newff(P,T,[S1 S2...S(N-l)],{TF1 TF2...TFNl}, BTF,BLF,PF,IPF,OPF,DDF) takes several arguments

P

R x Q1 matrix of Q1 sample R-element input vectors

T

SN x Q2 matrix of Q2 sample SN-element target vectors

Si

Size of ith layer, for N-1 layers, default = [ ].

(Output layer size SN is determined from T.)

TFi

Transfer function of ith layer. (Default = 'tansig' for

hidden layers and 'purelin' for output layer.)

BTF

Backpropagation network training function (default = 'trainlm')

BLF

Backpropagation weight/bias learning function (default = 'learngdm')

PF

Performance function. (Default = 'mse')

IPF

Row cell array of input processing functions. (Default = {'fixunknowns','removeconstantrows','mapminmax'})

OPF

Row cell array of output processing functions. (Default = {'removeconstantrows','mapminmax'})

DDF

Data divison function (default = 'dividerand')

,

and returns an N-layer feed-forward backpropagation network.

The transfer functions TFi can be any differentiable transfer function such as tansig, logsig, or purelin.

The training function BTF can be any of the backpropagation training functions such as trainlm, trainbfg, trainrp, traingd, etc.

Caution trainlm is the default training function because it is very fast, but it requires a lot of memory to run. If you get an out-of-memory error when training, try one of these:

\* Slow trainlm training but reduce memory requirements by setting net.trainParam.mem\_reduc to 2 or more. (See trainlm.)

\* Use trainbfg, which is slower but more memory efficient than trainlm.

\* Use trainrp, which is slower but more memory efficient than trainbfg.

The learning function BLF can be either of the backpropagation learning functions learngd or learngdm.

The performance function can be any of the differentiable performance functions such as mse or msereg.

**Examples**

Here is a problem consisting of inputs P and targets T to be solved with a network.

\*

P = [0 1 2 3 4 5 6 7 8 9 10];

T = [0 1 2 3 4 3 2 1 2 3 4];

Here a network is created with one hidden layer of five neurons.

\*

net = newff(P,T,5);

The network is simulated and its output plotted against the targets.

\*

Y = sim(net,P);

plot(P,T,P,Y,'o')

The network is trained for 50 epochs. Again the network's output is plotted.

\*

net.trainParam.epochs = 50;

net = train(net,P,T);

Y = sim(net,P);

plot(P,T,P,Y,'o')

Algorithm

Feed-forward networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer function.

The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

See Also

newcf, newelm, sim, init, adapt, train, trains

**initnw**

**Purpose**

Nguyen-Widrow layer initialization function

Syntax

\*

net = initnw(net,i)

**Description**

initnw is a layer initialization function that initializes a layer's weights and biases according to the Nguyen-Widrow initialization algorithm. This algorithm chooses values in order to distribute the active region of each neuron in the layer approximately evenly across the layer's input space. The values contain a degree of randomness, so they are not the same each time this function is called.

initnw requires that the layer it initializes have a transfer function with a finite active input range. This includes transfer functions such as tansig and satlin, but not purelin, whose active input range is the infinite interval [-inf, inf]. Transfer functions, such as tansig, will return their active input range as follows:

\*

activeInputRange = tansig('active')

activeInputRange =

-2 2

initnw(net,i) takes two arguments,

net Neural network

i Index of a layer

and returns the network with layer i's weights and biases updated.

There is a random element to Nguyen-Widrow initialization. Unless the default random generator is set to the same seed before each call to initnw, it will generate different weight and bias values each time.

**Network Use**

You can create a standard network that uses initnw by calling newff or newcf.

To prepare a custom network to be initialized with initnw,

1. Set net.initFcn to 'initlay'. This sets net.initParam to the empty matrix [], because initlay has no initialization parameters.

2. Set net.layers{i}.initFcn to 'initnw'.

To initialize the network, call init. See newff and newcf for training examples.

**Algorithm**

The Nguyen-Widrow method generates initial weight and bias values for a layer so that the active regions of the layer's neurons are distributed approximately evenly over the input space.

Advantages over purely random weights and biases are

\* Few neurons are wasted (because all the neurons are in the input space).

\* Training works faster (because each area of the input space has neurons). The Nguyen-Widrow method can only be applied to layers

o With a bias

o With weights whose weightFcn is dotprod

o With netInputFcn set to netsum

o With transferFcn whose active region is finite

If these conditions are not met, then initnw uses rands to initialize the layer's weights and biases.

See Also

initwb, initlay, init