2.4.1 Operators and functions

An expression can contain the binary operators `+`, `-`, `*`, `/`, and `^` , the logical operators `&&` and `||`, the comparison operators >, <, >=, <=, ==, and `!=`, and many unary nonlinear functions such as `exp()` or `log()`.

By default, these operators and functions work according to an element-by-element rule. That is, the binary operators take two objects of same dimensionality and size and generate another object of the same dimensionality and size by performing the operation on every pair of corresponding elements in the original objects. Thus, the expression `vector1*vector2` evaluates to a vector whose *n*th element is the product between the *n*th element of `vector1` and the *n*th element of `vector2`.

Similarly, most unary functions take an object and apply the function to each of its element to generate another object of the same dimensionality and size. Thus, the expression `exp(matrix)` returns a matrix whose *ij*th element is the exponential of the *ij*th element of `matrix`.

Functions in this category include: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `sqrt()`, `log()`, `abs()`, `random()` (generates random values with gausian distribution with variance specified by the elements of the argument), `urandom()` (generates random values with uniform distribution in range [0,x] x being specified by the elements of the argument), and `randomstate()` (generates an object whose elements are turned on or off according to the probabilities given in its argument). Some of these will be used in concrete examples in Section 2.6.

Thus, in the following statement in the procedure `activateExpr` we saw in Section 2.3.2:

```
Hidden = logistic( Hidden:net  + Hidden:bias )
```

elements in the two vectors `Hidden:net` and `Hidden:bias` are added together element-by-element and then the function `logistic()` is applied to each of these results.

Functions that are not element-by-element

There are a few exceptions to this element-by-element rule. First, the return values of some functions do not have the same dimensionalities as their arguments. Some functions take a vector or a matrix and yield a scalar (a single value). An example is in the expression:

```
mean(($target-Output)^2)
```

which computes the mean squared error between the target vector and the Output activation vector. Here, the function `mean()` (this is equivalent to `average()`) takes a vector or a matrix and yields a scalar — the average of the values of its elements. Other functions in this category include: `sum()` which computes the sum of the elements of the argument; `min()` and `max()` which return the minimum and maximum, respectively, elements in the argument; `norm()` which is equivalent to `sqrt(sum(x^2))` and computes the norm of its argument; `indexmin()` and `indexmax()` which return the index for he minimum and maximum, respectively, elements in the argument; and `sizeof()` which returns the number of elements in its argument.

Some functions reduce the dimensionalities of their arguments by one. The function `sumrow()` takes a matrix as the argument and returns a column vector of which each element is the sum of each row of the matrix. Similarly, `sumcol()` takes a matrix and returns a row vector of which each element is the sum of each column of the matrix.

There are functions that return objects with the same dimensionalities as their arguments but do not work element-by-element. The function `transpose()` or equivalent `T()` takes a matrix as the argument and returns its transpose. The function `permute()` randomly permutes the elements of its argument.

Applying an operator to different size objects

The second exception to the element-by-element rule is that the binary operators +, -, *, /, and ^ can also be applied to two objects of different dimensionalities: for example, the expression `scalar*vector` or equivalently, `vector*scalar` does what you would expect - it multiplies each element of the vector by the scalar generating a vector of the same size. Similarly with `scalar*matrix` and matrix*scalar. This rule is more generally stated that the object with lower dimension is replicated as many times as necessary to make it compatible with the higher dimension object. This applies to cases where a vector and a matrix is combined with an operator. For example, in the expression vector+matrix, vector is added to every *row* in matrix to yield a matrix as the result. If vector is a column vector as in T(vector)+matrix, vector is added to every *column* in matrix. For example:

```
sqrt(sumrow((matrix-vector)^2))
```

generates a (column) vector of which each element is the Euclidean distance between vector and each row vector of matrix. We will see how this expression can be used to implement a competitive learning network in Section 2.6.2.

Matrix multiplication operator **

The last exception to the element-by-element rule is the operator ** which performs the matrix multiplication in the usual sense. For example the expression HiddenOutput**InputHidden would compute the product of the two weight matrices in our example network yielding another matrix with the same number of rows as HiddenOutput and the same number of columns as InputHidden. This operator can be used to compute vector-matrix multiplication as in InputHidden**T(Input). This computes the transpose of the activation vector Input multiplied by the weight matrix InputHidden.

Thus in the procedure activateExpr we saw in Section 2.3.2, the expression:

```
Hidden:net = InputHidden**T(Input)
```

is equivalent to the action forward InputHidden.

Using the ** operator and the function T() you can compute both the inner-product (or "dot" product) of two vectors vector1 and vector2:

```
T(vector1)**vector2
```

and the outer-product of them:

```
vector1**T(vector2) .
```

Logical and comparison operators

The logical operators && and || and the comparison operators $>$, $<$, $>=$, $<=$, $==$, and $!=$ which are all borrowed from the C language, syntactically behave in the same way as the binary operators. That is, they work element-by-element if used with two objects with a same size and dimensionality; and if used with two objects with different dimensionalities, the lower dimension object is duplicated to match the higher dimension object. The semantics of the logical operators are: x&&y is 1 if and only if both x and y are non-zero, otherwise it is 0; x||y is 1 if either one or both of x and y are non-zero otherwise it is 0. The semantics of the comparison operators are the same as in C: $>$ and $<$ mean "greater than" and "less than"; $>=$ and $<=$ mean "greater than or equal to" and "less than or equal to"; == and != mean "equal to" and "not equal to". The result of these operators are represented by 1 (true) or 0 (false).

The fact that these logical operators evaluate to real numbers (0 or 1) allows us to mix them in mathematical expression. For example:

```
sum(vector>0)
```

returns the number of positive elements in vector. The statement

```
vector = (vector>=threshold)*vector
```

would turn off (set to zero) those elements in vector whose values are less than threshold. threshold can be either a scalar or a vector (of the same size as vector).

Index operator [ ]

You can access a part of a vector or a matrix using the index operator [ ]. The expression `vector[n]` accesses the nth element (scalar) of `vector`: for example, `Hidden[n]` is the activation value of the nth *Hidden* unit; `Hidden:net[n]` is the net input to this unit. The expression `vector[n->m]` accesses the nth through mth elements of `vector`: for example, `Output[0->3]` is the activation values of the first four *Output* units.

The expression `matrix[n]` accesses the nth row of `matrix`, which is a vector. For example, `InputHidden[n]` is the weight vector going into the nth *Hidden* unit. The expression `matrix[n->m]` accesses the submatrix of `matrix` consisting of the nth through mth rows of `matrix`. The expression `matrix[][n]` accesses the nth column of `matrix`, which is a column vector. For example, `InputHidden[][n]` is the weight vector coming out from the nth *Input* unit. It is not possible to access multiple columns: `InputHidden[][n->m]` is not allowed. The `[][n]` form can be used to extract a column vector from an arbitrary expression. For example, the weights going from the kth *Input* unit to the ith through jth *Hidden* units can be accessed by the expression `(InputHidden[i->j])[][k]` (note that the parentheses separate the expression extracting the rows and the `[][k]` operator extracting the column.)

In any of these cases, the indeces (n, m, etc.) can be arbitrary expressions. For example, `vector[sizeof(vector)/2->sizeof(vector)-1]` returns the last half of `vector`.

2.4.2 In-expression assignment

Here is the example we saw in Section 2.3.2 for propagating the network forward. (Shorter layer names and connection names are used for readability.)

```
In = $input
Hid:net = InHid**T(In)
Hid = logistic( Hid:net  + Hid:bias )
Out:net = HidOut**T(Hid)
Out = logistic( Out:net + Out:bias )
```

It took three lines to compute a hidden activation vector, and a total of five lines to compute an output activation vector. It is possible to collapse these three or even five expressions into one expression using in-expression assignments. First `Hid` can be computed by:

```
Hid = logistic((Hid:net=InHid**T(In=$input)) + Hid:bias )
```

Here the expression `In=$input` is an in-expression assignment, which copies the input buffer `$input` to Input, and the result of the assignment is also the value of the expression. Any number of such in-expression assignments can be embedded in an expression, as shown in the following large expression which compute the output activation vector `Out`.

```
Out = logistic( (Out:net \
                 = HidOut**T(Hid \
                             = logistic( (Hid:net \
                                          = InHid**T(In=$input) \
                                          ) + Hid:bias \
                                        ) \
                            ) \
                ) + Out:bias )
```

Note that the escape character \ at the end of a line indicates that the line continues to the next line. I have tried to make the above expression more readable by indentation (a pair of parentheses on a same column match up), but it is not necessary to do so.

One interesting use of in-expression assignment is to use it in conjunction with the range operator `[]` to set all elements in a vector to a value and then set a subset of the elements to a differnt value. For example:

```
(Output=$min)[n->m] = $max
```

would first turn off all `Output` units and then turn on the nth through mth `Output` units. (`$min` and `$max` are parameters representing the minimum and maximum activation values which the user can set interactively as described in Section 3.1.3).

2.4.3 Other assignment operators

There are a number of assignment operators besides =. They are +=, -=, *= and /=, whose semantics are analogous to the same symbols in the C language. For example, the increment operator += increments the object on the left-hand side by the result of the expression on the right-hand side. It can be used to implement the training procedure `learn` we saw in Section 2.3.1 with expressions:

```
procedure learnExpr   # training procedure implemented with expressions
    call activateExpr
    Hidden:delta = Output:delta**HiddenOutput * Hidden*(1-Hidden)

    InputHidden += InputHidden:delta \
                 = $eta*T(Hidden:delta)**Input + $alpha*InputHidden:delta

    HiddenOutput += HiddenOutput:delta \
                  = $eta*T(Output:delta)**Hidden + $alpha*HiddenOutput:delta
end
```

After the call to `activateExpr`, it computes `Hidden:delta`, the deltas for the `Hidden` units by backward propagating `Output:delta`, the deltas for the `Output` units, through the connection `HiddenOutput` and multiplying the result with the derivative of the activation function `Hidden*(1-Hidden)` assuming the activation range is [0,1]. Then, the increment operator += is used to adjust the weights in the connection `InputHidden` by adding to them the changes stored in `HiddenOutput:delta`. The changes `HiddenOutput:delta` are computed as the sum of two terms: The first term is the outer product `T(Hidden:delta)**Input` between the deltas for the `Hidden` units and `Input` activation, which is scaled by the learning rate parameter `$eta`; The second term is the momentum term which is the previous changes to the weights `InputHidden:delta` scaled by the momentum parameter `$alpha`. The weights in the connection `HiddenOutput` are adjusted similarly by the last expression. I leave it as an exercise for the reader to implement adjusting the biases for `Hidden` and `Output` with similar expressions. (See Appendix B for an answer.)

2.5 Other Features of the Langauge

To facilitate more general procedure programming capability, the **PlaNet Language** provides various other features, some of which will be illustrated by examples in Section 2.6.

2.5.1 Variables — scalar, vector, matrix

To allow for more flexible use of expressions, you can use the commands `scalar`, `vector`, and `matrix` to declare a scalar, vector, and matrix variables, (all of which contain floating-point numbers), respectively, in a network file, and refer to them in expressions. The syntax for these commands are:

```
scalar   <name> X              # X is (optional) intial value
vector   <name> N              # N is number of elements
matrix   <name> Nrow Ncolumn   # numbers of rows and columns
```

Scalar variables can be set interactively, and therefore can act as parameters in the network.

Special variables

Also, any predefined **PlaNet** parameter that has a floating-point value [4] can be accessed in a procedure as a scalar variable by preceding it with a `$`. For example, `$eta` in a network file accesses the learning rate parameter *eta*. See Section 3.1.3 for setting parameters.

Other special variables starting with $ include:

$Error : (scalar) output error computed by the action `target`.

$globalError : (scalar) output error averaged across all patterns computed by the commands `cycle` and `present` by averaging the values of `$Error`.

$cumError : (scalar) cumulative output error summed across patterns presented so far in the commands `cycle` and `present`.

$mouseClick : (vector) this returns a vector when a mouse button is clicked, whose first, second, and third elements are the X and Y coordinates of the mouse cursor and the mouse button number that is clicked.

$windowClick : (scalar) this returns a scalar when a mouse button is clicked whose value is the window number that is clicked. -1 is returned if the user clicks at a location where there is no window.

Local Variables

Version 5.6 supports local variables in procedures. Scalar, vector, and matrix variables declared within a procedure definition body has a scope which is just inside the procedure. A variable can be declared anywhere in the procedure before it is first refered to. Note that, unlike local variables in many other languages and like static local variables in C, only a single memory space is allocated for each variable which retains its value from one procedure call to the next.[5]

2.5.2 Control statements

Control constructs such as the conditional `if` statements and `while` and `repeat` loops are also provided. Each of these takes an expression as its condition.

A repeat loop has the form:

```
repeat <expression>
    ...
    ...
endrepeat
```

The statements between `repeat` and `endrepeat` are repeatedly executed as many times as the value of the evaluated `<expression>` (only the first element of `<expression>` is used even if it has more than one value.)

A while loop has the form:

```
while <expression>
    ...
    ...
endrepeat
```

The statements between `while` and `endwhile` are repeatedly executed as long as the expression `<expression>` evaluates to true, that is all its elements have the value 1.

There are three kinds of `if` constructions. The first has the form:

[4] In the current version, **PlaNet** parameters with integer or binary (on/off) values cannot be accessed in a procedure.

[5] This makes it difficult, if not impossible, to write useful recursive procedures that manipulate structured objects. A room for future improvement.

```
if <expression>
  statement
```

If `<expression>` evaluates to true then the next statement is executed. Otherwise the next statement is skipped.

The second form of `if` statement is:

```
if <expression> keyword
```

where `keyword` is one of `end`, `stop`, `endrepeat`, `endwhile`, and `continue`. If `<expression>` evaluates to true (all its elements are 1), different action is taken depending on `keyword`. The following are the keywords and the corresponding actions.

end : terminate this procedure.

stop : terminate this and all parent procedures, if any.

endrepeat : terminate a repeat loop.

endwhile : terminate a while loop.

continue : continue a repeat/while loop.

The following example illustrates the difference between `end` and `stop`.

```
procedure p1
    call p2
    y = z
end

procedure p2
    if x==1 end
    if x==2 stop
    ...
end
```

In procedure `p2`, if `x==1` is true then `p2` is terminated but `p1` is not, thus executing the statement `y = z`. If `x==2` is true then both `p2` and `p1` are terminated and the statement `y = z` is not executed.

A third form of `if` statement is:

```
if <expression> then
    ....
    ....
endif
```

This means to execute the statements between `if` and `endif` only if `<expression>` evaluates to true. This form can optionally take one `else` in it:

```
if <expression> then
    ....
else
    ....
endif
```

in which case the statements between `else` and `endif` are executed when `<expression>` does not evaluate to true. Currently, there is no `else if`, but `if/else/then` can be nested to an arbitrary depth.

2.5.3 Other built-in actions

Some general-purpose vector operations are provided as built-in actions: `shift` shifts the contents of a vector; `push` and `pop` do stack-like operations that push and pop a vector to and from another vector. See Section III of the **PlaNet Reference** for a complete list of these and their usage.[6]

2.5.4 Procedure arguments

Up to Version 5.5, all variables were global, so a procedure could only deal with a fixed set of variables. Version 5.6 supports procedure arguments: You can pass layers, connections, and any expressions as arguments to a subprocedure call. This allows a same procedure to be used on different sets of variables. So for example, you can write a procedure for doing some common operations to be performed on each layer in the network, such as:

```
    procedure activateLayer layer L
        L = 1 / (1 + exp( -( L:net + L:bias )/Temp )
    end
```

which uses a special activation function to compute the activation vector in a layer. It implements a logistic function with a temperature parameter `Temp` that changes the slope of the curve. The two words `layer L` after the procedure name `activateLayer` specifies that this procedure has one argument of the type "layer" which is to be referred to by the name "L" in the procedure body. This procedure can be called for each layer in a network by giving the layer name as the argument, as in:

```
    call activateLayer(Hidden)
```

The general form to specify the types and the names of the arguments for a procedure is[7]:

```
    procedure <name> type1 name1 type2 name2 ...
... procedure body ..
    end
```

where `typeN` is one of `layer`, `connect`, `expression`, and `procedure` indicating the type of the argument. Each of these can be abbrebiated to its first one or more characters, so `expr` or just `e` is enough for `expression`. Arguments declared as each of these types can be used in the following ways:

`layer`: An argument declared as the type *layer* can be used in the procedure body as a layer. That is, it can be used as an argument to the actions `input`, `activation`, `delta`, `learnbias` and `target`, and also its component vectors can be accessed in expressions.

`connect`: Similarly, an argument declared as the type *connect* can be used as the argument for the actions `forward`, `backward`, and `learn` and its weight matrix can be accesses in expressions.

`procedure`: An argument declared as the type *procedure* can be called as a subprocedure within the procedure. That is, it can be used as an argument to the action `call`.

`expression`: An argument declared as the type *expression* can be used wherever an expression can be used, for example, as the right or left hand side of an = assignment, or inside an expression in the right or left hand side of an =. An argument appearing as the left hand side of an = is assumed to be an object to which values can be assigned (so-called an *lvalue* in the C language).

[6] In version 5.6, many actions that previous versions had have been implemented as equivalent expressions. For example, the action `randomstate x y` is now automatically converted into the expression `y=randomstate(x)`.

[7] This format may be changed in future versions to something more reasonable

When a procedure defined with arguments is called, it should be passed the appropriate number of arguments of the correct types. The general calling syntax is simply:

```
call <procedure-name>(arg1,arg2,..)
```

where `argN` is the Nth argument. We will see many examples in Section 2.6.

2.5.5 Library procedures

Since procedures can have arguments, one procedure can be used in many different contexts by giving it different sets of arguments. It is, then, convenient to have libraries of often-used procedures which can be accessed whenever they are needed without redefining them in the network file. There is a default directory with a number of standard library files. One library file called "libstat" contains some procedures for statistical analyses. You can access them by putting a line:

```
library libstat
```

in the network file. When a procedure is called in a procedure definition but not defined in the network file, this library file is looked upon for the definition of that procedure. If, for example, you need to compute a linear regression on two vectors `X` and `Y`, you can call a procedure `regress()` defined in this library with:

```
call regress(X,Y)
```

You can access the result in the variables `regress.intercept` and `regress.slope` which represent the intercept and the slope, respectively, of the regression line.

There is another library file called "libnet" which contains various utilities procedures for running networks. For example, a procedure `repeatInteractive()` in this library can be used (after `library libnet`) as in `call repeatInteractive(procName,nrepeat)` which will call a procedure `procName` repeatedly `nrepeat` times in an interactive manner, that is, stopping at each step to wait for a mouse click by the user.

You can also create a library file of your own and access the procedures in it by including a line `library <library-filename>` in the network file.

2.5.6 Changing the size of a data

In some cases it is desirable to change the size of some data structure (such as the number of units in a layer in the network) from what is initially declared (such as by the command `layer`.) In fact, some learning algorithms add (or sometimes remove) units to (from) a network as the network learns. The size of a layer can be changed in a procedure by an action `resize`. For example the following procedure can be used to change the size of any layer to any value:

```
procedure resizeLayer layer Layer expr nunit
    resize Layer nunit
end
```

A procedure call `call resizeLayer(Hidden,nhidden+=1)` will add one unit to the layer `Hidden` everytime it is called (provided that the variable `nhidden` is set to the correct size initially). Similarly, `call resizeLayer(Hidden,sizeof(Layer)+nunit)` will add `nunit` units to the layer instead of one.

Usually, it is also necessary to make new weights (or remove weights) coming out of and/or going into the units just created (or removed). If this is desired, a third argument `connect` can be given to the `resize` action. Thus in the following network with two input layers:

```
layer Input1 n1; layer Input2 n2; layer Hidden n3; layer Output n4
connect InOut1 Input1 to Hidden
connect InOut2 Input2 to Hidden
connect HidOut1 Hidden to Output
```

```
connect HidOut2 Hidden[0-4] to Output

procedure addUnit layer Layer
    resize Layer sizeof(Layer)+1 connect
    ...
```

a procedure call `call addUnit(Hidden)` will add one unit to `Hidden` and then create three sets of new weights: weights from the two input layers to this new unit; and weights from this new units to the output layer. It re-connects only the connections with *full connectivities*, namely, `InOut1`, `InOut2` and `HidOut1`. A connection that connects only a part of the layer, such as `HidOut2` in this case, is not re-connected. [8]

Note that if the connection has a sparse connectivity whose weights are specified by the network command `weight` or the action `rewire` as explained in the next section, the connectivity pattern is not changed even when the connection is resized. It is necessary to use `weight` or `rewire` to change the connectivity pattern.

The action `resize` can be used to change the size of a vector or a matrix. The syntax is:

```
resize <vector-name> <no-of-element>
resize <matrix-name> <no-of-rows> <no-of-columns>
```

where the second and third arguments specifying the size of the vector or matrix can be any expressions.

The resizing feature can be used just to *pretend* that a layer, a vector or a matrix is smaller than it actually is. For example, instead of:

```
Output[0->nunits-1] = logistic(Output:net[0->nunits-1]+Output:bias[0->nunits-1])
```

you can write:

```
resize Output nunits; activation Output; resize Output nOutput
```

assuming that `nOutput` is the correct size of `Output`. This is convenient (and more efficient) when the lengths of input and/or output vectors vary in time, for example.

2.5.7 Sparse connectivity

Two layers can be connected with a sparse connectivity in which not every unit in one layer is connected to every unit in the other layer. There are two basic ways to achieve a sparse connectivity. One is using the command `weight` described in Section 2.1.2, which can also specify which of the weights are modifiable. In this section we will look at a different way that uses the action `rewire`. With this action, you can change the connectivity pattern in a connection by giving a matrix containing the desired pattern of 0's 1's and -1's, where 0 indicates no weight, 1 (or any positive number) indicates a modifiable weight, and -1 (or any negative number) indicates a non-modifiable weight. For example a random connectivity can be achieved by:

```
connect InHid Input to Hidden
matrix connectivity sizeof(Hidden) sizeof(Input)

procedure prune ex probOn
    rewire InHid connectivity=randomstate(connectivity=probOn)
    ...
```

The matrix `connectivity` has the same dimensions as the weight matrix in `InHid`. When this procedure is called with, say, `call prune(0.3)` each weight in `InHid` will be removed with the probability 0.7 (thus 0.3 connectivity). In the expression given as the second argument to `rewire`,

[8] This can be confusing when a layer is shrunk.

all elements in the matrix `connectivity` are first set to the probability `probOn` and then passed to the function `randomstate()` which generates a pattern of on/off according to the probabilities specified by the elements of its argument.

If you want to remove weights that are smaller than a threshold, you can do:

```
procedure prune ex threshold
    rewire InHid InHid>threshold
    ...
```

If you want to further remove a specific subset of the weights, for example, all the weights going into the nth hidden unit, you can do:

```
connectivity[n] = 0
rewire InHid connectivity
```

By setting `connectivity[n]` to 1's instead of 0's you can connect all the weights going into the nth hidden unit. The value of a weight is kept unchanged even when the weight is removed and the weight will resume its value if it is connected again. Thus, removing and adding of weights with `rewire` have effects only with the actions `forward`, `backward` and `learn`. If you use the weight matrix in an expression, it will behave as if all the weights are there, even the ones that have been removed. Expressions like `InHid*connectivity` should be used as the effective weight matrix. [9]

2.6 Examples

In this section, three simple examples are shown that illustrates how the various features mentioned above can be used. First one is a use of the backpropagation algorithm in a temporal processing and uses, as in the previous example, the high-level actions for most of the network processing but uses `while` loops and subprocedure call with arguments for the control of the temporal flow. The next two are networks that do not use the backpropagation algorithm and therefore rely more on low-level expressions.

2.6.1 Sequential network

The first example is Mike Jordan's sequential network (Jordan, 1986) which generates a sequence of output vectors from a single input vector. In addition to the feedforward three-layer architecture that we have been using, a Jordan network has a layer of units called the *State* units with recurrent self-connections that acts as a memory of a temporal context of past output vectors stored as an exponentially decaying trace of past output vectors: The sequence of vectors $X = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_T)$, where $\vec{x}_t$ is the output vector at time t, is stored as $\sum_{t=1}^{T} \alpha^{T-t}\vec{x}_t$, where α is the decay factor $(0 < \alpha < 1)$. At each time step, the next output vector is determined both by its input vector (which does not change during the sequence), and by the *State* vector (which changes at each time step).

We first need to define the architecture of this network.

```
layer Plan Nplan
layer Hidden Nhid
layer Output Nout
layer State Nout

connect PlanHidden  Plan to Hidden
connect StateHidden  State to Hidden
connect HiddenOutput  Hidden to Output
```

[9] Doing this with a very sparse weight matrix is clearly wasting lots of computation. By contrast, computation time for the actions `forward`, `backward` and `learn` is roughly proportional to the number of *existing* weights.

```
input Nplan
target Tseq*Nout

matrix targetSeq Tseq Nout
vector errorSeq Tseq
vector prevTarg Nout
```

The Hidden layer receives two connections: one from the `Plan` layer and one from the `State` layer. Assuming we want to train the network to generate some fixed sequence, we need a target vector for each time step in the sequence. Thus, the size of the target buffer is the number of units in the `Output` layer (`Nout`) times the number of time steps in each sequence (`Tseq`). The matrix `targetSeq` is used to store the target vectors temporalily, and the vector `errorSeq` stores the sequence of output errors. The vector `prevTarg` is used to for remembering the target vector at previous time step (for feedback).

Now we can define a procedure to train this network.

```
procedure learn
    scalar t                      # local var. for time step in sequence

    targetSeq=target             # save target sequence
    Output=0; State=0            # clear network states
    prevTarg = 0
    input Plan                   # present input to Plan
    t = 0
    while t < Tseq               # for each time step in sequence
        call activateStep(targetSeq[t],errorSeq[t]) # activate one step
        call learnStep           # learn one step
        t +=1
    endwhile
    $Error = mean(errorSeq)      # average error for the sequence
end
```

We assume the target pattern contains the sequence of target vectors concatinated with each other. `targetSeq = target` copies this sequence of target vectors into `targetSeq`, storing each target vector into each row of the matrix `targetSeq`. As in this statement, even when the left and right hand sides of a `=` are of different types (matrix and vector in this case) the values of the right hand side object are assigned to the left hand side object as far as there are space. Values are assigned to a matrix in row-first order. As a special case, if the right hand side is a scalar (has a single value) its value is assigned to all the elements of the left hand side object. This can be seen in the next two statements `Output=0; State=0` where the activation values of all the units in these two layers are set to zero.

Then we let the network do processing for `Tseq` time steps, and compare the network's output at each time step with each of the `Tseq` target vectors. The `while` loop does this as the time step represented by the local variable `t` is varied from `0` to `Tseq-1`. For each step, two subprocedures `activateStep` and `learnStep` are called which are similar to `activate` and `learn` for the feedforward network we saw in Section 2.3.1 performing forward and backward propagations respectively. First, `activateStep` is called with two arguments: The first argument `targetSeq[t]` is the `t`th target vector to be used as the target for time step `t`; The second argument `errorSeq[t]` is the variable to be assigned the error for time step `t`. The errors stored in this vector is averaged after the while loop to compute the overall error for the sequence. Now we look at the two subprocedures `activateStep` and `learnStep`.

```
scalar feedbackTarget 1
scalar decay 0.6
```

```
procedure activateStep expr targVec expr error
    clear Output; clear Hidden          # clear net inputs and deltas
    if feedbackTarget==1 then
        State = State * decay + prevTarg
    else
        State = State * decay + Output
    endif
#          alternative implementation of the above
#   State = State * decay
#            + prevTarg*(feedbackTarget==1) + Output*(feedbackTarget!=1)
    target = targVec
    forward PlanHidden; forward StateHidden
    activation Hidden
    forward HiddenOutput; activation Output
    target Output
    prevTarg = target
    error = $Error
end
```

Basically, this procedure first updates the `State` vector by feeding back an appropriate vector, and then propagates activations forward in the network to generate a `Hidden` activation vector and then an `Output` activation vector. It is necessary to set the net inputs (and the deltas) in the `Hidden` and `Output` layers to zero because the action `forward` adds the result of the vector matrix product onto the net input vector that is already there. [10] This is what the `clear` action does.

> [*Comments: When does a layer need a* `clear`*?*] The `clear` action was not necessary in the feedforward network we saw in the very first example because the net inputs and the deltas are cleared automatically before the main procedure `activate` is started. And this was enough because the activation of each layer was computed just once within this procedure. However, if we update the activation in a layer more than once within the main procedure, as is the case in this recurrent network, then we have to `clear` the layer before each time we compute the layer's activation with `forward` and `activation`.

The `if-then-else` statements chooses a vector to be fed back to the `State` layer depending on the value of the flag `feedbackTarg`. If this flag is `true` (has the value 1) the previous target vector is fed back; otherwise, the previous output vector is fed back. [11] An alternative implementation of these `if-then-else` statements are also shown (and commented out with #s) which produces the same effect with one expression. [12]

At each time step, the `State` activation vector has the information about the sequence of target or output vectors up to this point. This information is stored as an exponentially decaying trace of the vectors superimposed with each other. This is achieved by multiplying the `State` vector with a decaying factor `decay` before adding a new vector at each time step. This decay factor is another parameter you can set (usually to a value between 0 and 1.) [13]

Finally, there is nothing really new in the `learnStep` procedure which is shown below. It is very similar to the `learn` procedure for the feedforward network we saw in Section 2.3.1 except that we have two connections to the `Hidden` layer which both get modified with the action `learn`.

---

[10] this is so in order that the inputs from more than one connection going into a single layer can be added together as is the case with the two `forward` statements before `activation Hidden` statement in this example.

[11] Feeding back the previous target keeps the sequence of State vectors consistent during training whereas feeding back the previous output vector would make this sequence change over the course of training. You can choose between these options (for whatever theoretical or practical reason) by setting the flag to 1 or 0.

[12] This implementation of if-then-else with an expression can be less efficient if the vectors involved are large because all the elements in both of the two alternative vectors are evaluated. If the vectors are relatively small it can be more efficient due to some overhead associated with the evaluation of an if-then-else construct.

[13] The decay factor `decay` is a single variable in this implementation, but it can be a vector of decay values in order to implement different decay factors for the individual units in `State`.

```
procedure learnStep
     backward     HiddenOutput; delta       Hidden
     learn        PlanHidden
     learn        StateHidden;  learnbias   Hidden
     learn        HiddenOutput; learnbias   Output
end
```

2.6.2 Competitive learning

Here is our first example of a non-backpropagation network: a competitive learning network, which is described in detail in Chapter 5 of the PDP book (mentioned in Section 1.2). The network we look at here is a simple version that has only two layers, an input layer and an output layer, with the input layer directly connected to the output layer.

```
layer Input nIn
layer Output nOut

connect InputOutput Input to Output
```

We will implement two different versions in a same network file and choose between the two depending on whether the input vector is assumed to be binary (0 or 1) or continuous. To distinguish between these two we first define a string `Binary` to be either 1 for binary input or 0 for continuous input.

```
define Binary 1     # set this to 0 for continuous input vector
```

We first write a procedure `activate` for selecting a winning output unit. This procedure should be defined differently for the two cases. We use `IF/ENDIF` to distinguish between the two cases. We first look at the binary input case.

```
scalar winner        # winner output unit

IF Binary==1        # for binary input vector
procedure activate
    Output = $min                       # turn off output units
    Input = $input/sum($input)          # normalize and present input
    forward InputOutput                 # compute Output:net
    winner = indexmax(Output:net)       # find the winner
    Output[winner] = $max               # turn on the winner
end
ENDIF
```

In order to simplify the learning rule as we will see later, the input vector is first normalized so that it always sums to 1. In the binary case, the winning output unit is the one that receives the largest net input. The function `indexmax()` returns the index for the largest element in its argument. The winning output unit is then turned on.

In the continuous input case which we will define next, the winning output unit is the one whose weight vector is closest to the input vector.

```
IF Binary==0        # for continuous input vector
procedure activate
    Output = $min                       # turn off output units
    Input = $input/sum($input)          # normalize & present input
    Output:net = sumrow((InputOutput-Input)^2) # compute Output:net
    winner = indexmin(Output:net)       # find the winner
```

```
    Output[winner] = $max                # turn on the winner
end
ENDIF
```

The expression `(InputOutput-Input)` subtracts the input vector `Input` from each row of the matrix `InputOutput` (see Section 2.4.1.) Each row of `InputOutput` contains the weight vector for each output unit. The result is a matrix whose rows contain the differences between the weight vectors and the input vector. The function `sumrow()` takes a matrix as the argument and returns a vector whose elements are sums of the rows of the matrix. Thus, the expression `sumrow((InputOutput-Input)^2)` computes the distances from the weight vectors of the output units to the input vector. The function `indexmin()` returns the index for the smallest of these, which is equal to the index for the winning output unit.

Actually, we can collapse these two `activate` procedures into one using `IF/ELSE/ENDIF` within the procedure to deal with the two cases. [14] We also collapse the statements by using in-expression assignments and eliminating the intermediate variables `Output:net`. Also, this time we are cautious about cases when `sum(input)` is zero: such an input vector cannot be normalized, so the procedure is terminated.

```
procedure activate
    scalar totalinput
    if ((totalinput=sum(input)) == 0) stop
IF Binary==1
    winner=indexmax(InputOutput**T(Input=input/totalinput))
ELSE
    winner=indexmin(sumrow((InputOutput-(Input=input/totalinput))^2))
ENDIF
    (Output=$min)[winner]=$max
end
```

Once we select a winner, we train the weights according to the competitive learning algorithm.

```
procedure learn
    call activate
                                        # adjust the winner's weights
    InputOutput[winner] += $eta * (Input - InputOutput[winner])
end
```

The weight vector for the winner selected in the procedure `activate` is adjusted by the second statement. One constraint assumed here is that the weight vector for each output unit is normalized to sum to 1. This weight adjustment rule maintains this constraint, because the input is normalized so that it sums to 1, and thus the weight adjustment `$eta*(Input - InputOutput[winner])` should sum to 0. We can interpret this rule as moving the winner's weight vector closer to the input vector it has just won so that it becomes more likely to win it again.

Finally, the next procedure `reset` sets each weight vector so that it sums to 1.

```
procedure reset                         # reset the weights
    InputOutput = urandom(InputOutput=1) # initialize weights
    scalar n;    n = 0
    repeat nOut                         # sum of each weight vector should be 1.
        InputOutput[n] /= sum(InputOutput[n])
        n += 1
    endrepeat
    $epoch = 0
end
exec reset
```

[14] The `IF/ELSE/ENDIF` construct is evaluated only at compile-time, and thus is more efficient than `if-then-else` which is evaluated at run-time.

This procedure first fills all the weight vectors with random numbers between 0 and 1, and then normalize each weight vector so that it sums to 1. Note the use of the function `urandom()`: `InputOutput = urandom(1)` would simply set all the weights to a same random value, so we need to first set them to 1 (`InputOutput=1`) and then call `urandom()` on all of them. Each weight vector `InputOutput[n]` is normalized simply by dividing it with the sum of its elements, `sum(InputOutput[n])`. `$epoch = 0` sets the learning epoch parameter `epoch`, which is incremented automatically when the network is actually trained. We put a **PlaNet** command `exec reset` to make sure the weights are initialized properly when the network file is read.

2.6.3 Boltzmann machine

The last example is an implementation of a fully connected Boltzmann machine. As usual, we start with an architecture definition, which is simply one layer of units called `units` fully connected with each other by a weight matrix called `weights`. Then we define a procedure `anneal` for performing simulated annealing.

```
layer units nUnit                # all units are in one layer.
connect weights units to units   # full connectivity

define nAnneal 32
vector schedule nAnneal
value schedule < scheduleFile

procedure anneal expr updateList         # procedure for annealing
    scalar t; t = 0
    repeat nAnneal
        if schedule[t] <= 0 endrepeat      # end of schedule
                                           # compute new activation pattern
        call update(schedule[t],permute(updateList))
        t = t+1
    endrepeat
end
```

The vector `schedule` contains a temperature schedule to be used for the annealing process, and its values are read in from the file `scheduleFile` with the network command `value`. The procedure `anneal` follows this schedule by taking one element at a time from this vector and passing it to another procedure `update` which updates the network at the temperature specified by (first) the argument. The end of the schedule is marked by a non-positive temperature value, which terminates the repeat loop at the line `if schedule[t] <= 0 endrepeat`. The vector `updateList`, which itself is an argument passed to this `anneal` procedure, specifies which units should be updated. The second argument to the `update` procedure specifies the units to be updated and the order in which to update them. We permute this vector with `permute(updateList)` so that the units are updated in random order. Now we look at how the procedure `update` updates the network.

```
procedure update expr temperature expr updateList
    scalar unit; scalar n
    clear units                            # clear net inputs
    n = 0
    repeat sizeof(updateList)
        unit = updateList[n]               # for each unit in the list
        units:net[unit] = sum(units*weights[unit])  # compute net input
                                           # compute activation level
        units[unit] = randomstate( logistic( (units:net[unit] +\
                                             units:bias[unit])/temperature ) )
        n = n+1
```

```
    endrepeat
end
```

Note that the action `clear` is used to clear the net input of the units first because this procedure is called many times in the procedure `anneal` and thus the net inputs will not get cleared automatically. (See the comment in Section 2.6.1.) This procedure goes through the vector `updateList`, which should contain the indeces for the units to be updated. For each of these units, its net input is first computed by the expression `sum(units*weights[unit])`, where `weight[unit]` is the vector of weights going into this unit. (We cannot use the action `forward` as we did for computing net inputs in previous examples, because we are doing *asynchronous update* here, updating one unit at a time.) The activation level (on or off) of this unit is then computed by the stochastic activation function `randomstate(logistic((net+bias)/temperature))` in which `temperature` determines the amount of randomness.

Next, we write a procedure for testing this network: We present an input vector to a subset of the units and then use the annealing process we have defined to let the network settle into a stable state. Since this is a stochastic network, we take a statistics of the state of the network after it settles.

```
define Input 0->nIn-1                 # index for input units
define Output nUnit-nOut->nUnit-1     # index for output units

input nIn                             # input buffer
target nOut                           # target buffer

vector updateHidOut nUnit-nIn

scalar nTest 10                       # no. of runs for testing.
define testTemp 0.5                   # temperature for testing
vector testOutput nOut                # test output vector

procedure activate                    # procedure for testing network
    units[Input] = $input             # present input
    call anneal(updateHidOut)         # let the net settle
    testOutput = 0                    # initialize data
    repeat nTest
        call update(testTemp,updateHidOut)         # update net
        testOutput += units[Output]                # take output statistics
    endrepeat
    testOutput /= nTest                            # average data
    $Error = mean((target-testOutput)^2)           # compute output error
end
```

Here, we assume that the units are divided into three groups: the input units, the hidden units, and the output units (such an assumption affects the annealing process only with respect to which units should be updated). Since the units are all in one layer, the two define statements define the strings `Input` and `Output` to be the indeces for the input units and the output units, respectively. We can use these to access the input units by `units[Input]`, for example, as in the first line in this procedure, where the current pattern in the input buffer is copied to the input units. Since we need to update only the hidden and the output units, the vector `updateHidOut` which is passed to the procedure `anneal` should contain the indeces for these units. After the annealing process, we update the network several times at a low temperature (the `testTemp` parameter) and collect a statistics of the output activation into the vector `testOutput` (of course other statistics can be taken as well), which is compared with the target pattern to compute the mean squared error by the last statement.

Finally, we implement the Boltzmann machine learning algorithm. It involves running the network in two different conditions, a clamped phase and a free-running phase, and adjusting the weights based on the difference in the stable states the network settles in in the two conditions. Since the only difference between these two phases is which units get updated, we can implement them in a single procedure which takes as an argument the list of units to be updated.

```
procedure phase expr updateList expr correl expr actv
    call anneal(updateList)                    # let the net settle
    correl = 0;  actv = 0                      # initialize data
    scalar n; n = 0
    repeat nTest
        call update(testTemp,updateList)       # update net
        correl += T(units)**units              # cross-correlation statistics
        actv += units                          # activation statistics
        n += 1
    endrepeat
    correl /= nTest; actv /= nTest             # average data
end
```

The second and the third arguments to this procedure `correl`, and `actv` are data structures for collecting some statistics of the state of the network after it settles. `correl` is a cross-correlation matrix whose *ij*th element represents an estimated probability that the *i*th unit and the *j*th unit are both on. These are computed after the network settles with the `anneal` procedure by computing a time average of `T(units)**units` which is the outer product of the activation vector `units` with itself. Similarly, `actv` is a vector whose *i*th element represents an estimated probability of the *i*th unit being on. This is computed as a time average of the activation vector `units` itself.

In the actual training procedure `learn`, we call this `phase` procedure twice, once for the clamped phase and once for the free-running phase. We first set up the data structures for taking statistics of network states in the two conditions in the training procedure.

```
matrix CorClamp nUnit nUnit       # cross-correlation matrix - clamped phase
matrix CorFree nUnit nUnit        # cross-correlation matrix - free running phase
vector actvClamp nUnit            # activation vector - clamped phase
vector actvFree nUnit             # activation vector - free running phase

procedure learn                   # training procedure
    units[Input] = $input; units[Output] = $target # present input/output
    call phase(updateHid,CorClamp,actvClamp)       # clamped phase
    call phase(updateHidOut,CorFree,actvFree)      # free-running phase

    weights += $eta*(CorClamp - CorFree)           # adjust weights
    units:bias += $eta * (actvClamp - actvFree)    # adjust biases

    $Error = mean((target-actvFree[Output])^2)     # output error
end
```

For the clamped phase, both the input and output units are clamped so we update only the hidden units: the first argument to `phase`, `updateHid` contains the indeces for the hidden units. For the free-running phase, only the input units are clamped, so the hidden units and the output units are updated: `updateHidOut` contains the indeces for these units. The weights adjustments are proportional to the difference between the two cross-correlation matrices computed in the two phases. Similarly, the biases of the units are adjusted in proportion to the difference between the average activation vectors in the two phases.

Finally, we define two procedures that make necessary setups.

```
procedure makeList expr list expr start expr nunits
    scalar n; scalar unit
```