Researcher's guide to Bayes Blocks library

Markus Harva, Antti Honkela, Alexander Ilin,
Tapani Raiko, Harri Valpola and Tomas Östman

December 4, 2006

Contents

1 Introduction

This package is supposed to be a flexible tool for building a wide variety of latent variable models by combining simple building blocks. After the connections have been defined, the computations are automatic. The theory is explained in [1].

At the moment, Gaussian, rectified Gaussian, and mixture-of-Gaussian variables, summation, addition, two types of nonlinearity and discrete variables have been implemented.

For computational efficiency there are vectorised nodes in addition to scalar nodes. There is a delay to support connections of different time instances of vectorised nodes. There is also support for on-line learning.

2 Restrictions on the structure

In general, the network has to be a directed acyclic graph (DAG). The delay nodes are an exception because the past values of any node can be the parents of any other nodes. The violation is not real in the sense that if the vectors were split into scalar nodes, the resulting network would be a DAG.

For computational efficiency each node assumes its inputs to be independent. This means that a latent variable cannot propagate its value to another variable (observed or latent) through more than one path. If there were multiple paths, the values through these paths would be dependent. This restriction can be overcome by adding intermediary variable nodes.

Self-delays for vectorised latent variable nodes are forbidden for computational efficiency. There are special nodes with built-in self-delays for overcoming this restriction.

3 Description of the nodes

Currently all nodes except Discrete and DiscreteDirichlet have continuous values.

The following nodes are in use:

- Constant, ConstantV,
- Evidence, EvidenceV,
- Prod, ProdV, Sum2, Sum2V, SumN, SumNV, Rectification, RectificationV, DelayV, Relay(computational nodes)
- Gaussian, GaussianV, DelayGaussV, RectifiedGaussian, RectifiedGaussianV, MoG, MoGV, GaussNonlin, GaussNonlinV, GaussRect, GaussRectV, Discrete and DiscreteV, DiscreteDirichlet, DiscreteDirichletV, Dirichlet(variable nodes)
- Memory, OLDelayS, OLDelayD (for on-line learning)

- Proxy (building temporal structures that would seemingly violate the DAG property)

3.1 Constant nodes

Constant has no parents. The output value is a scalar. Posterior variance is zero. The value is a double given for the constructor. ConstantV is similar but the value is a vector.

3.2 Evidence nodes

Evidence has one parent and no children. It is meant for providing fading clamps for initialising the parameters of the model. The node must be clamped with two values, one for mean and one for "variance" of the observation. The clamp is decayed by increasing the variance parameter after subsequent iterations. EvidenceV is similar but the value is a vector.

3.3 Computational nodes

3.3.1 Prod

Prod has two scalar parents which are given for the constructor. The output value is scalar and is the product of the parents.

3.3.2 Sum2

Sum2 has two scalar parents which are given for the constructor. The output value is scalar and is the sum of the parents.

3.3.3 SumN

SumN is like Sum2 but has N scalar parents which are given one by one by calling the `AddParent` method of the node.

3.3.4 Rectification

This node is used in conjunction with GaussRect, which is its only parent given in the constructor. See section 3.4.5.

3.3.5 DelayV

DelayV delays a vector. The first value is a special case and is given separately as the first parent of the node. It is a scalar value. The other parent is the vector to be delayed. To give it as a parameter to the constructor, you will probably want to use a `Proxy` node to create a structure that would seemingly create a cycle in the network.

3.3.6 Relay

Relay passes the value as it is. The purpose of this node is to make the role of a parent identifiable when a node is used for more than one purpose.

Example: node X has mean Y and variance Y. Node Y is now used for two purposes and computation of gradient does not work because the role of Y cannot be identified. Y calls X twice and X should first assume the mean is calling and second assume the variance is calling. Instead X assumes mean is calling both times. This can be fixed by adding a relay node Z = Y and then telling X its mean is Y and variance Z. This could be automated later on: a relay is needed if a node has identical parents for different roles.

Currently Relay can be used for scalar nodes only, but extension to vectors would be straight-forward. To be added as soon as needed.

3.3.7 ProdV, Sum2V, SumNV, RectificationV

These nodes are like their counterparts without V but the output value is a vector. Parents can be either scalars or vectors.

3.4 Variable nodes

Any variable node can be latent (hidden, unknown). Some variable nodes can be observed in which case the observations can be set by clamping.[1] When direct clamping is not supported (it doesn't make sense in some cases) similar effect can be achieved using evidence nodes.

3.4.1 Gaussian

Gaussian is a variable with a Gaussian prior distribution. The mean and variance parents are given for the constructor (mean is first, variance second). Variance is parametrised by Var = $\exp(-v)$, where v is the value of the second parent. Output is a scalar.

3.4.2 RecfifiedGaussian

RectifiedGaussian is a Gaussian with zero probability mass on the negative axis and with the positive axis scaled appropriately.

3.4.3 MoG

MoG is a variable consisting of a mixture of K Gaussians. It has $2K+1$ parents: one discrete variable, given in the constructor and K mean and variance parents. The component means and variances are given to the node after construction using the `AddComponent` method.

[1] The term clamping is used in biological neurosciences: clamping a potential of a cell means keeping the potential fixed by injecting a suitable current.

3.4.4 GaussNonlin

GaussNonlin is like Gaussian, but the output is a nonlinear function $\exp(-s^2)$ of the variables internal value s. Warning: if the prior mean is zero, there is a local minimum at the internal mean zero.

3.4.5 GaussRect

GaussRect is like Gaussian but it can be followed by a rectification nonlinearity $f(s) = \max(s, 0)$. The node should have at least one children, the Rectification node, which is followed by the children that receive the values of the Gaussian variable rectified. GaussRect can also have children that receive the values of the Gaussian variable without rectification. In that case the GaussRect node is directly used as the parent of the children. This is especially useful when building dynamical models.

3.4.6 Discrete

Discrete is a variable with discrete distribution over small integers. Its prior is given by soft-max distribution of Gaussians c_i: $p(d = i) = \exp(c_i)/\sum_j \exp(c_j)$.

3.4.7 DiscreteDirichlet

DiscreteDirichlet is a discrete variable like Discrete, but with Dirichlet prior for its weights. It has the Dirichlet variable as its only parent. When possible, this node should be favoured against the Discrete, since the posterior approximations are more accurate.

3.4.8 Dirichlet

Dirichlet variable is used as the parent for DiscreteDirichlet. It takes a ConstantV as its parent which specifies the prior observation counts.

3.4.9 GaussianV, RectifiedGaussianV, MoGV, GaussNonlinV, GaussRectV, DiscreteV and DiscreteDirichletV

These nodes are like their counterparts without V but their values are vectors. All parents can be scalars or vectors.

3.4.10 DelayGaussV

DelayGaussV is like GaussianV but the node includes a self-delay. The parents are (m, v, a, m_0, v_0). If the value of the node is $s(t)$, the mean is $m(t)+a(t)s(t-1)$ and variance is $\exp(-v(t))$ for $t > 1$. The prior for the first value ($t = 1$) is given by m_0 and v_0.

3.5 Special nodes for on-line learning

3.5.1 Memory nodes

In on-line learning, some of the nodes are time-dependent while others are time-independent. Memory node acts as an adapter between these two. It stores the gradients from time-dependent nodes.

Note that time-dependent and time-independent nodes must not be connected together without the memory node between them: the time-independent node will be silently converted into a time-dependent node and the conversion propagates to neighbours of the newly converted node.

3.5.2 OLDelayS, OLDelayD

The delay nodes are used to implement temporal models with on-line learning. They act as normal unit-delays, `OLDelayS` for scalars and `OLDelayD` for discrete values.

The delay nodes have two parents. The first tells the "initial" value of the delayed value and it is only used when the node is initially constructed. The second parent is the normal delayed value. The two parents of the delay node may also both be the same node. You will probably want to use a `Proxy` node as the second parent of a delay node.

3.6 Proxy node

`Proxy` is a special node that is allowed to seemingly violate the DAG structure of the network. Naturally these violations must not be real, but rather have a *delay* node somewhere in the loop. The proxy works as a placeholder parent for other (mainly delay) nodes and only connects to the true parent after the whole structure has been created.

When a `Proxy` is created, in addition to its own label, it only takes the label of its future parent as a parameter.

Before a proxy is connected to the true parent it blindly accepts all requests for different values but keeps a record of what it has promised. Once the proxy connects to the parent, usually as a result of a call to `ConnectProxies` in the `Net`, it checks whether the true parent can provide all the values it has promised and complains if this is not the case. After this, the proxy simply forwards all the requests to its true parent.

4 C++ library

The library provides the basic operations such as the creation of nodes, updates, (un)clamps, evaluations of the cost function, queries of the values of nodes, saving (and loading in the future), pruning etc.

Each node belongs to a network (class `Net`) which maintains the nodes. Each node has a string label which can be used to get the pointer to the node.

4.1 Class `Net`

This is the class which contains the nodes and provides functions for updating the net, evaluating the cost functions etc. The constructor is given the dimension of the vector nodes used in the network. Note that all vectors in one network need to be of equal length.

Net takes care of deleting the nodes. When net is deleted it deletes all nodes it contains. See 4.3.5 for information about removing nodes.

4.1.1 Connecting the proxies

The method `ConnectProxies` connects all the proxies to their true parents. It can safely be called several times as it does not affect previously connected proxies.

4.1.2 Updates

The method `UpdateAll` updates all variables of the net. A single variable node can be updated by the method `Update` (see 4.3.6).

For memory nodes the method `Update` propagates time. Therefore `UpdateAll` should not be used in on-line learning. See 4.4 for more details.

4.1.3 Decay hooks

The network supports so called decay hooks which allow fine control of decay of variance of the Evidence nodes. The hooks are called by name which is a string. `Decayer`s, which means so far just `Evidence` and `EvidenceV`, can be hooked to the desired hook with method `RegisterDecay`.

There are standard hooks for `UpdateAll`, `UpdateTimeDep`, `UpdateTimeInd` and `StepTime` which are processed at the end of corresponding functions. Additionally the user can specify his or her own hooks simply by giving a new string as a name of the hook. In this case, the hook must be activated by hand by calling `ProcessDecayHook` with the name of the hook as an argument. Items can be removed from the hooks by methods `UnregisterDecayFromHook` for single hook and `UnregisterDecay` for removing an item from all the hooks, but this is handled automatically when the evidence nodes die.

4.1.4 Evaluation of the cost function

The method `Cost` evaluates the cost function. Each node has a method which gives the cost for that node only.

4.1.5 Finding nodes and variables

The methods `GetNode` and `GetVariable` take a label as an argument and return the pointer to the node or variable with the matching label.

The methods `GetNodeByIndex` and `GetVariableByIndex` take an index as an argument and return the pointer to the corresponding node or variable. The number of nodes and variables can be obtained by the methods `NodeCount` and `VariableCount`.

4.1.6 Saving and loading

The method `SaveToXMLFile` saves the network in XML text format while `SaveToMatFile` uses Matlab format (the second argument is the name of the variable to be saved in the file). A single node can be saved using `SaveNodeToXMLFile`. This can be useful for debugging purposes. In addition to these, the function `SaveToPyObject` saves the network to a Python object. This can later be saved to a file using Pickle.

```
void SaveToXMLFile(string fname);
void SaveToMatFile(string, string);
void SaveNodeToXMLFile(string fname, Node *node);
PyObject *SaveToPyObject();
```

The library supports loading the network from a Matlab file or a Python object. The function `LoadFromMatFile` loads the network from given Matlab file (second string argument giving the name of the network) while `CreateNetFromPyObject` "loads" the network from given Python object.

The functions using Python objects can also be used for cloning the network:

```
temp = orignet.SaveToPyObject()
copynet = CreateNetFromPyObject(temp)
```

4.1.7 Pruning

When nodes are killed they need to be explicitly removed using the method `CleanUp`. See 4.3.5.

4.1.8 Miscellaneous

The method `Time` returns the length of the vectors (usually time is the vectorised dimension). The methods `NotifyDeath`, `Save`, `AddNode` and `AddVariable` are mainly for internal use. Methods `SetDebugLevel` and `GetDebugLevel` can be used to set the level of debugging messages required. Higher levels mean more messages. The default level is 0.

4.2 Class `NodeFactory`

The `NodeFactory` is used to create new nodes for the network. The constructors of different node classes can only be invoked by a `NodeFactory` that makes sure that the ownership of the nodes belongs to the network the nodes are part of. The `NodeFactory` has methods of type `GetXXX` for creating a node of type

XXX. They generally take the label of the node and references to its parents as arguments and return a pointer to a newly created node. The constructor of `NodeFactory` takes the `Net` it will create nodes to as a parameter.

4.3 Class `Node`

4.3.1 Creation

A node is created by calling the appropriate `GetNode` method of a `NodeFactory`. The arguments are usually the label and the parents of the node. `Constant` and `ConstantV` nodes do not have parents and instead the value is given. The `Proxy` node is given only the label of its future parent that is connected later by calling the method `ConnectProxies` from `Net`.

4.3.2 Clamping and unclamping

The value of a Gaussian or Discrete node can be set by the method `Clamp`. This tells the network that the node is observed. The node can again be made unobserved (latent, hidden, unknown variable) by the method `Unclamp`.

In a factor analysis type latent variable models, for instance, it is useful to initialise the linear mapping with random values in order to make the learning start. This can be done using `Clamp`. Once the factors have some nonzero values the linear mapping can be `Unclamp`ed. Alternatively the initialisation can be done using evidence nodes. If learning would be started without initialisations, the factors and linear mapping would all be zero and none of them would adapt because the whole system would be trapped in the local minimum where everything is zero.

4.3.3 Expectations

Each continuous node can be asked for its value using the method `GetReal` for scalars and `GetRealV` for vectors. The value is returned in the first argument which is `DSSet` for `GetReal` and `DVH` for `GetRealV` (see 4.5.1 and 4.5.6). The second argument is of class `DFlags` and indicates which expectations are requested (see 4.5.8).

These two methods are typically used by the children of a node to inquire the expectations required by the computation of gradients and cost function.

NOTA BENE: the expectations which are not specifically requested (no flags set in `DFlags`) may be left to their original values.

The corresponding methods for discrete nodes are `GetDiscrete` and `GetDiscreteV`. They take only one argument as there are no parameters to the request. The value is returned in the first argument which is a pointer to `DD` for `GetDiscrete` and `VDDH` for `GetDiscreteV` (see 4.5.4 and 4.5.7).

NOTA BENE: the structure returned by GetDiscrete *must not* be modified or destroyed by the caller as it is shared with the node returning it.

All the `Get*` methods return a boolean value indicating whether the requested expectation could be returned.

4.3.4 Gradients

A parent of a node asks the gradient w.r.t. the continuous expectations using the methods `GradReal` or `GradRealV` depending whether the parent is a scalar or vector. Both methods take two arguments. The first one is `DSSet` for `GradReal` and `DVSet` for `GradRealV` (see 4.5.1 and 4.5.3). The second argument is the pointer of the parent which is asking the gradient. It is used for identifying the role of the parent.

The gradients of the children of a node can be asked by the methods `ChildGradReal` and `ChildGradRealV`. They lack the second argument but are otherwise similar to the above methods.

NOTA BENE: the methods which give gradients do not initialise the first argument which is used for returning the value. Instead the gradient is added to the previous value. If the vector valued gradient requests do not touch a particular gradient w.r.t. an expectation, the vector for this gradient will be uninitialised.

The corresponding functions for the components of a discrete distribution are `GradDiscrete` and `GradDiscreteV`. Its first argument is a `DD` for `GradDiscrete` and `VDDH` for `GradDiscreteV` and the second argument is the pointer of the parent asking the gradient.

If a Node cannot provide a gradient of some sort, it should just silently return. This applies, for instance, to continuous nodes when they are asked for `GradDiscrete`.

4.3.5 Removing nodes

The method `Die` kills a node. This may cause a chain reaction where other nodes will be killed if they are left orphan or without children. This behaviour is controlled by the `persist` variable which can be accessed by the methods `GetPersist` and `SetPersist` (see 7.2).

The nodes are not actually removed from the network before calling `CleanUp` (see 4.1.7).

Do not try to `delete` a node: net has to know which nodes exist and it takes care of the deletion also.

4.3.6 Updates

A (variable) node can be updated by `Update`. A node can be outdated by `Outdate`. This propagates to children in computational nodes and outdates the cost function in variable nodes.

Another way to update a node is to use the set of functions `SaveState`, `SaveStep` and `RepeatStep`. These are used by `UpdateAllHookeJeeves` method in `PyNet` (see Sec. 5.1.4). `SaveState` saves the current value of the variable to be used as a base point for future changes. `SaveStep` saves the current change from the

state saved by `SaveState`. `RepeatStep` can then later be used to repeat the same step multiplied by a constant, starting from the original state saved by `SaveState`. These operations are at the moment only supported by `Gaussian` and `GaussianV` nodes.

4.3.7 Miscellaneous

The label and type of node can be queried by the methods `GetLabel` and `GetType`. The parents and children can be obtained by the methods `GetParent` and `GetChild`. They take an index as an argument and return null pointer if the index is out of range.

4.4 On-line learning

On-line learning works but is still experimental.

4.4.1 Time type

Each node is either

0: time independent,

1: time dependent or

2: memory node

3: delay node

The type of a node can be queried by `TimeType` method.

Time dependent nodes are assumed to have different values at each time instant while time independent nodes have only one value which does not depend on time. Memory nodes connect type 0 and type 1 nodes and gather the gradients when time is stepped.

4.4.2 Updates

Type 0 nodes are updated by the `UpdateTimeInd` method of `Net` class. Type 1 nodes are updated by the `UpdateTimeDep` method and memory nodes are instructed to save the gradients by `StepTime` method. This also instructs the net to memorise the cost function from type 1 nodes. The memorised cost is automatically added to the cost when queried by the `Cost` method. It can be directly accessed by `GetOldCost` and `SetOldCost` methods.

Do not use the `UpdateAll` method in on-line learning: it updates all nodes regardless of their type.

4.4.3 Controlling time type

Do not forget to use memory nodes. If two nodes are connected and one of them has type 1, then the other node is also converted to type 1. The conversion propagates so that the nodes connected to the newly converted node are also converted. Memory node blocks the conversion.

It is not necessary to give an explicit command about time type although the method `NotifyTimeType` could be used in principle. Children of memory nodes will be automatically set to type 1 and any node they are connected to will be set to type 1 as explained above.

4.4.4 Decay

In on-line learning, it is impossible to change the inferences made about past data. This means that the network could get stuck in false interpretations made in the early learning. To prevent this, old data is gradually forgotten. The decay rate is governed by a ratio parameter which is by default 0.5. This means that effectively the model uses only half of the data it has been given. Currently the decay rate is common to every memory node, but this may change in the future. The decay is controlled by a `DecayCounter` object maintained by the `Net` class. The latest decay factor can be queried by the `Decay` method of `Net`. Currently the the ratio parameter (and other relevant parameters) of `DecayCounter` need to be set by directly accessing the field `Net.dc.ratio`.

4.4.5 Usual operation

```
Connect the network
Iterate
    Clamp time dependent nodes
    Iterate
        UpdateTimeDep
    Sometimes
        UpdateTimeInd
    StepTime
```

4.5 Data structures

The network passes distributions from parents to children. For real values the distributions are summarised by a set of expectations (currently mean, variance and expectations of exponential are possible). The distributions of discrete variables are passed as lists of probabilities for each possible value, that is there are no summaries. The gradients of the quantities passed from parents to children are passed in the opposite direction from children to parents. Usually the user does not need to worry about the gradients.

The basic data structures needed by the user are `DSSet` (double scalar set), `DV` (double vector), `DD` (discrete distribution), `VDD` (vectorised discrete distribution), `DVH` (double vector handle), `VDDH` (vectorised discrete density handle) and `DFlags` (double flags).

4.5.1 Class `DSSet`

This is a set of scalar expectations. Currently there are members `DSSet.mean`, `DSSet.var` and `DSSet.ex` which stand for mean, variance and exponential. `DSSet` is used for asking the scalar values of nodes. It is also the structure used for the gradient of scalar values.

4.5.2 Class `DV`

This is the basic data structure for double vectors. The user mainly needs it for clamping the values of vectorised continuous valued variables.

4.5.3 Class `DVSet`

This is the vectorised equivalent of `DSSet`. It is used for the gradients of scalar vectors and is usually not needed by the user.

4.5.4 Class `DD`

The user needs this class for clamping discrete nodes. For now, `DD` is just a wrapper for a `DV` of suitable size. There are plans for adding support for sparse distributions, but the exact way of doing this is still unclear. The most important operation provided by `DD` is indexing provided by the operator "[]".

4.5.5 Class `VDD`

This is the vectorised equivalent of `DD`. It is needed for clamping vectorised discrete nodes.

4.5.6 Class `DVH`

This is a structure which can hold either `DSSet` or `DVSet`. It is used for asking the value of vector nodes. This data structure makes it possible for vector nodes to have either vector or scalar parents since scalar nodes can also be requested a vector. `DVH` can keep track whether the value it contains is a vector or scalar. The value can be accessed by indexing methods `DVH.Mean(i)`, `DVH.Var(i)` and `DVH.Exp(i)`. If the value of `DVH` is scalar, these methods return the scalar, otherwise they return the appropriate element of the vector. This way the child does not need to know whether the parent was vector or scalar.

4.5.7 Class `VDDH`

The relation between `VDDH` and `VDD` is mostly the same as that between `DVH` and `DV`. In `VDDH` also the scalar value is a pointer to `DD`. Neither of these structures must be deallocated as they are internal to the node whose values they represent.

4.5.8 Class `DFlags`

This is a set of boolean attributes, currently `DFlags.mean`, `DFlags.var` and `DFlags.ex`. It is used for indicating which expectations are requested when asking the value of a continuous valued node.

5 Python interface

The Python interface is generated directly from the C++ header files by SWIG (Simple Wrapper and Interface Generator)[2]. The basic idea is that all (or rather most) of the C++ classes have been wrapped so that they appear standard Python classes and can be accessed as such.

5.1 class PyNet

This class is derived from the C++ class Net. Useful functions have been added.

5.1.1 Sum trees

The methods `BuildSum2Tree` and `BuildSum2VTree` build a sum tree from the list of nodes given as argument. The other arguments form the base for label. The labeling works so that the root node gets exactly the label given to the function and other nodes get labels of the form `base_leaf(i0, i1, ..., in)`.

5.1.2 Pruning

The method `TryPruning` tests whether pruning a variable is beneficial and removes it if the cost function is predicted to decrease. The function can handle a list of nodes to test.

5.1.3 Accessing nodes

The methods `GetNodes` and `GetVariables` return lists of nodes or variables which match the regular expression given as the second argument. The first argument is the net.

5.1.4 Advanced updates

The method `UpdateAllDebug` does the updates as standard `UpdateAll` but also checks that none of the node updates increases the value of the cost function. This is useful for debugging the update methods of the nodes.

The method `UpdateAllHookeJeeves` does one round of optimisation according to the Hooke-Jeeves algorithm. It first does the standard updates for all

[2]For more information, see `http://www.swig.org`

variables, but then takes another step in the same direction defined by the individual updates. The length of this additional step is defined by performing a line search on the cost function values in that direction. The method also uses helper functions `RepeatAllSteps` and `FindOptimalStep`.

5.1.5 Miscellaneous

The method `GetConst0` returns a constant node with value zero. Its label is const0 and it is created if it does not already exist.

5.2 Helpers.py

This file contains useful functions.

5.2.1 Sum trees

The functions `AddSum2` and `AddSum2V` add a new entry to an existing sum tree or create a new tree. The argument list is net, rootnode, parind, newpar, label. The sum tree is in node rootnode in parent number parind. Newpar is the new node in the sum tree and label is the label for the `Sum2` of `Sum2V` node to be created. The function `FindLeaf` is used for finding the correct entry point. Caution: there are currently no checks for the DAG property.

5.2.2 Pruning

The function `CostDifference` returns a prediction of the change in cost function if a given variable node is removed from the network. Negative value means the cost function will decrease and thus removal of the variable is beneficial.

5.2.3 Adding a weighted connection

The function `AddWeight` creates a new weighted connection from one node and adds it to the sum tree of another vector node. It then tests whether the newly created connection is useful using `TryPruning` and returns the new weight if it was found useful. The arguments are: net, input node, output node, label of the weight, mean of the weight, variance of the weight and optionally additional cost. It is often useful to use negative cost because the importance of a new weight is easily under-estimated.

5.2.4 Loading data

The function `LoadAsciiData` returns a list of lists of the numbers in the given data file.

5.2.5 Accessing nodes

List of parents and children of a node can be obtained by the functions `ParentList` and `ChildList`. The values can be queried using the functions `GetMean`, `GetVar` and `GetExp`. They can handle lists of nodes.

5.3 Data types

SWIG does not understand operator overloading and therefore some of the indexing methods in the data types of C++ library do not always work. The indexing methods have been added to classes `DV` and `DD`. Indexing of a VDD is not yet supported due to difficulties in multidimensional indexing.

6 Examples

There are two small examples of the usage of the library in the main directory:

`main.C`: Factor analysis with dynamic model for factors

`main.py`: Factor analysis with pruning

7 Technicalities

7.1 Delayed variable check using Proxy nodes

In principle all nodes check that their parents are of the required type when they are created. This means that the parents must be created before their children. As the network is a DAG, this should in theory always be possible. Unfortunately the situation is not as simple in practice as vectorised nodes can cause problems in temporal models. In such cases one element of a vector somehow affects the model of the next element and the vector needs a delayed version of itself as a parent.

Such cases can be handled with the Proxy node. A proxy can be created without explicit reference to its parent, only a label of the future parent. When the proxy is further used as a parent to other nodes, it stores information of what it has been asked for. When the construction of the network is complete, all the proxies can be connected to their actual parents by calling the method `ConnectProxies` in the corresponding network. As part of this process, the proxies now check that their parents really can return the values they are required to.

Using a proxy creates a loop in the otherwise acyclic network graph. The method `SortNodes` in Net handles this by ignoring the parents of proxies.

7.2 Persist

The variable `persist` has flags which indicate the cases in which the node should die. Setting a flag allows dying. The dying can occur only when the node gets a

notification of the death of another node and it turns out to be either a parent or child.

The flags have the following meanings:

1 Die if a parent died

2 Die if there are no parents (all died or otherwise)

4 Die if there are no children (all died or otherwise)

8 Die if there is only one parent. Replace yourself with the parent in children.

By default the constant has `persist = 0` which means it never dies. Functions have the value 5 (= 1 + 4) which means that they need at least one child and all parents which they initially had. Variables have the value 1 which means that none of their parents must die. It may be useful to set the value to 5 for latent variables.

Exceptions are the `Switch` function nodes which never die (value 0) and `Sum2` and `Sum2V` function nodes which have the value 12 (= 4 + 8). They need at least one child and cut off if one of the parents dies. Cutting off means that before dying the node replaces the pointers to itself from its children by the pointers of its remaining parent. In other words, A + B → C will be replaced by A → C if B is killed.

7.3 Evidence node

The evidence node creates the virtual clamp by adding an artificial extra term to the cost function and providing gradients to its parent according to that. The extra term is essentially the same as that resulting from a observed Gaussian with single observation for the clamped value, mean given by the parent and variance given by the other value clamped to the node. As the network is updated and the decay increases the clamped variance, the clamp becomes weaker and gives smaller gradients. After the clamp has decayed completely, the evidence node kills itself. The time when the variance is decayed can be controlled by hooking the node to the appropriate decay hook as described in Sec. 4.1.3.

7.4 Memory node

7.5 SWIG

SWIG operates by taking an interface file (e.g. `Net.i`) which resembles standard C or C++ header file but with some additional directives. The "compiler" turns this file into a C/C++ source file (e.g. `Net_wrap.C`) and a corresponding Python source file (e.g. `Net.py`) containing the definitions of the wrappers and the shadow classes.

8 Error messages

Most messages should be self explanatory, such as "Double clamp not allowed", "Expected a vector parent in DelayGaussV" although sometimes the nodes just complain "Wrong type of parents in GaussNonlinV".

When `PyNet.UpdateAllDebug` is used, it will complain if the updates of some of the nodes result in increase of the cost function.

Sometimes there can be a cryptic error message as follows: (assume we are updating a Gaussian node s(29))

```
s(29) var: M=2.43445; V=0.02442; GEX=3.43523; GVA=34.24; GME=-43: diff = 3.424
```

It means that for some reason the minimisation of cost function failed in Gaussian node, but the node knows that. The reason is an ill behaved gradient which is typically caused by

1. degenerate solution (e.g. variances have gone very small)
2. invalid connections (e.g. multiple paths from a latent variable to another variable)
3. bug in gradient computation

If options 1 and 2 can be ruled out, please send a bug report.

The optimisation in Gaussian nodes is not fool-proof so it is possible that in some rare cases the computations genuinely fail, but so far there has always been some other reason for the error message. If you encounter one of these rare cases, we may improve the minimisation. The reason why it has not been improved so far is that this error message is often a useful indication of something else being suspicious.

References

[1] H. Valpola, T. Raiko, and J. Karhunen, "Building blocks for hierarchical latent variable models," in *Proc. Int. Conf. on Independent Component Analysis and Signal Separation (ICA2001)*, (San Diego, USA), pp. 710–715, 2001.