
CYBORG: Software for Max-Product Belief Propagation in High Order Factor Graphs

Danny Tarlow <dtarlow@cs.toronto.edu>

(with Inmar Givoni and Richard Zemel)

Last Updated: May 14, 2010

Abstract

There is a growing interest in building probabilistic models with high order potentials (HOPs), or interactions, over discrete variables. Message passing inference in such models generally takes time exponential in the size of the interaction, but in some cases, exact message updates for (generally approximate) max-product belief propagation can be computed much more efficiently.

This is documentation accompanying C++ code implementing the high order factors described in HOP-MAP: Efficient Message Passing with High Order Potentials (Tarlow, Givoni, & Zemel, 2010).

1 OVERVIEW

Max-product belief propagation (MPBP) is a general MAP inference routine that can be applied to arbitrary low-order and some high-order graphical models. Here, we focus on tractable instances of high order potentials (HOPs), providing efficient message computation routines as described in Tarlow et al. (2010). As in Tarlow et al. (2010), we work with a factor graph representation and exclusively use scalar messages and binary variables. To represent multinomial variables, we apply a basic transformation where one binary variable is created per state of the multinomial variable, and a one-of-N constraint is added over all states.

We call our software CYBORG, in reference to the fact that parts of the code use standard message passing routines, but a large part of the high order factor calculations use special-purpose combinatorial algorithms. CYBORG is written in object-oriented C++. The asymptotic runtime of high order message computations are optimized; we use the best known algorithms for each factor. However, we do not heavily optimize computation at a lower level. There are two primary goals for this software:

1. Be general. Allow a wide variety of models to be used without requiring new message derivations or new code (other than code to construct the model) to be written.
2. Be easily extensible. It should be easy to implement a new class of **Factor** and use it in combination with other factors.

A secondary goal is to make the code straightforward to understand, so that it is possible for a beginner to look through the code and understand how it works.

2 INSTALLATION AND RUNNING

2.1 Installation

CYBORG has only one external dependency, which is `boost::program_options`, which is often installed by default. If it is not installed, however, you will need to install it. Files for the full boost library are here:

<http://sourceforge.net/projects/boost/files/boost/1.38.0/>

After installing `boost::program_options`, make sure that `LDFLAGS` and `CCFLAGS` in the Makefile are set to properly reference the installation location. You may need to add this directory to your `LD_LIBRARY_PATH`.

Alternatively, see the Appendix for more detailed instructions on installing `boost`. Versions 1.38, 1.40, and 1.43 have been used successfully.

Afterwards, it should be sufficient to run `make` from within the CYBORG directory.

2.2 Running the Example

If all went well, you should be able to run `./cyborg` to call the example program. The example program uses a grid of binary variables of size `idim` by `jdim`. Outputs should look roughly as follows, where “X” represents a variable taking value 1, and “.” represents a variable taking value 0:

```
.....
.....
XX.....
XXXXXXXX.....
XXXXXXXX.....
XXXXXXXXXX.....
XXXXXXXXXXXX.....
XXXXXXXXXXXXX.....
XXXXXXXXXXXXX.....XXXXX.....
XXXXXXXXXXXXX.....XXXXX.....XXXXXXX.....
XXXXXXXXXXXXX.....XXXXX.....XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXX.....XXXXX.....XXXXXXXXXXXXXXXXXXXXXXXXXXXX
..XXXX.XX.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXXXXXXXXXX.....XXXXXXXXXXXXXXXXXXXX
.....XXXXXXXXXXXX.....
.....XXXXXXXX.....
.....
377 / 1500 (25.1333%) variables on
```

The following potentials are included, depending on command line arguments:

2.2.1 Unary Potentials

Each variable is assigned a random potential sampled uniformly at random in the range $[-1, 1]$. Positive potentials encourage a variable to take on value 1 (or label “X” in the visualization). Negative potentials encourage a variable to take on value 0 (or label “.” in the visualization).

2.2.2 Pairwise Potentials

(Control with `--use_pairwise 1` and `--pairwise_strength <strength>`)

Along each row, attractive pairwise potentials are added between left-right neighboring variables. There are no pairwise connections between above-below neighboring variables.

The strength of an individual potential is sampled uniformly at random in the range $[0, strength]$. The potential encourages neighboring pixels to both take on label 1 and gives value 0 for all other settings of the pair.

2.2.3 Cardinality Potential (Count Potential)

(Control with `--use_counts 1`, `--counts_strength <strength>`, and `--fraction_on <fractionOn>`)

A count potential is added that softly encourages `fractionOn` of the variables to take on label 1. The potential is of the form:

$$\theta_{\text{count}}(\mathbf{h}) = -\text{strength} \cdot (\text{idim} \cdot \text{jdim} \cdot \text{fractionOn} - \sum_{h_i \in \mathbf{H}} h_i)^2,$$

which penalizes the squared distance from the target number of pixels on.

2.2.4 Convexity Potential

(Control with `--use_convexity 1`)

Constrain each column to form a 1D convex set. That is, disallow all configurations where the variables in position $h_{i,j_1} = 1$, $h_{i,j_3} = 1$, $h_{i,j_2} = 0$, and $j_1 < j_2 < j_3$.

2.2.5 All Options

Importantly, any combination of these potentials can be used together. Try playing around with different combinations of active potentials and different relative strengths of each type of potential.

Run `./cyborg --help` to see all available options:

Allowed options:

<code>--help</code>	produce help message
<code>--idim arg (=20)</code>	size in i dimension
<code>--jdim arg (=180)</code>	size in j dimension
<code>--damping arg (=0.75)</code>	damping factor. higher damping means use less of new message.
<code>--iterations arg (=100)</code>	maximum number of iterations
<code>--seed arg (=1)</code>	random seed. use -1 for <code>srand(time(NULL))</code>
<code>--schedule arg (=0)</code>	message schedule: 0 - SYNCHRONOUS
<code>--use_counts arg (=1)</code>	use a counts factor in the model
<code>--counts_strength arg (=10)</code>	how much to scale penalty for deviating from desired count
<code>--fraction_on arg (=0.75)</code>	expected fraction of pixels that are 'on'
<code>--use_convexity arg (=0)</code>	use a convexity factor in the model
<code>--use_pairwise arg (=1)</code>	use pairwise factors between horizontally connected variables
<code>--pairwise_strength arg (=1)</code>	strength of potential between neighbors (positive, higher is stronger)

2.3 Examples to Try

Start with the simplest possible call, which uses only singleton potentials:

```
./cyborg --use_pairwise 0 --use_counts 0 --use_convexity 0
```

Now try adding pairwise potentials:

```
./cyborg --use_pairwise 1 --pairwise_strength 1 --use_counts 0 --use_convexity 0
```

Then stronger pairwise potentials:

```
./cyborg --use_pairwise 1 --pairwise_strength 10 --use_counts 0 --use_convexity 0
```

Then even stronger pairwise potentials:

```
./cyborg --use_pairwise 1 --pairwise_strength 100 --use_counts 0 --use_convexity 0
```

Keep the pairwise potentials and try adding a cardinality potential that encourages only 10% of the variables to be on (some of the arguments above were unnecessary and are set by default, so we drop them from here on):

```
./cyborg --pairwise_strength 10 --fraction_on .1
```

Try encouraging 90% of the pixels to be on instead:

```
./cyborg --pairwise_strength 10 --fraction_on .9
```

Finally, add in convexity constraints to each column:

```
./cyborg --pairwise_strength 10 --fraction_on .25 --use_convexity 1
```

This exercises the main functionality of the example. Feel free to experiment with other combinations of potentials and other relative potential strengths.

3 CONSTRUCTING AND USING FACTOR GRAPHS

To go beyond the example, you must programmatically construct new factor graph models. This section describes the essential components.

The outer `BinaryFactorGraph` class manages the connectivity between `Factor` instances and `BinaryVariable` instances. Its main storage includes variables, factors, and edges:

```
vector<BinaryVariable *> *variables_;
vector<Factor *> *factors_;
vector<Edge *> *edges_;
```

The `BinaryFactorGraph` constructor takes no arguments:

```
BinaryFactorGraph *fg = new BinaryFactorGraph();
```

3.1 Adding Variables

To create variables for use in the factor graph, there are two choices:

1. Create a `BinaryVariable` using `var = new BinaryVariable(string name)` constructor, then call `BinaryFactorGraph::AddVariable(var)`:

```
BinaryVariable *var = new BinaryVariable("x");
fg->AddVariable(var);
```

2. Create an arbitrary dimensional “grid” of variables, and automatically add it to `fg`:

```
vector<uint> dim(2);
dim[0] = 100;
dim[1] = 100;
fg->AddVariableGrid("foreground", dim);
```

Variables can then be retrieved using the layer name and an index of the proper dimension:

```
BinaryVariable *var = fg->GetVariableByGrid("foreground", index);
```

3.2 Adding Factors

To add factors to the factor graph, first add all variables in its scope to the factor. Suppose we have the variable grid as above and want to define a cardinality HOP over all variables in the grid:

```
CountsFactor *counts_factor = new CountsFactor("counts_factor_1");
vector<int> index(2);
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        index[0] = i;
        index[1] = j;
        BinaryVariable *var = fg->GetVariableByGrid("foreground", index);

        counts_factor->AddToScope(var);
    }
}
```

Next, define the potential using the factor's type-specific construction. For example, a `CountsFactor` needs a potential representation that assigns a log likelihood to each possible number of variables in its scope that are on. For example, one way to softly encourage half of the variables in the layer to be on would be to define the potential as follows:

```
vector<double> potential(100 * 100 + 1);
for (int i = 0; i <= 100 * 100; i++) {
    potential[i] = 100 * 100 / 2 - abs(i - 100 * 100 / 2);
}
counts_factor->SetPotential(potential);
```

(The factor will copy the entries, so `potential` can be deallocated at this point.)

In addition, we usually want to assign singleton potentials to each variable:

```
vector<int> index(2);
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        string factor_name = StringPrintf("s_%d_%d", i, j);
        SingletonFactor *s_ij = new SingletonFactor(factor_name);

        index[0] = i;
        index[1] = j;
        BinaryVariable *var = fg->GetVariableByGrid("foreground", index);

        s_ij->AddToScope(var);
        s_ij->SetPotential((i + j) % 2 == 0 ? 1: -1);

        fg->AddFactor(s_ij);
    }
}
```

Finally, the factor must be added to the factor graph:

```
fg->AddFactor(counts_factor);
```

3.3 Belief Propagation

A `BinaryBeliefPropagation` object takes as input a `BinaryFactorGraph` object. In the current implementation, only one `BinaryBeliefPropagation` object may be built for each `BinaryFactorGraph`. The following snippet can be used as the most basic way to construct a `BinaryBeliefPropagation` object, run inference, and decode beliefs:

```

BinaryBeliefPropagation *bp = new BinaryBeliefPropagation(fg);

bp->InitMessagePassing();
for (int it = 0; it < NUM_ITERATIONS; it++) {
    bp->PassMessages();
}

bp->AssignVariablesByBeliefs();
cout << "Likelihood:" << fg->Likelihood() << endl;

```

To access individual variable beliefs and assignments:

```

BinaryVariable *var = fg->GetVariableByGrid("foreground", index);
cout << var->Name() << " belief: " << var->Belief()
    << "(" << var->Assignment() << ")" << endl;

```

4 IMPLEMENTING NEW FACTORS

4.1 Factors

The `Factor` class is the core of the message passing implementation. All factors (both low- and high-order) subclass the base `Factor` class. The important instance variables that a `Factor` maintains are:

```

vector<BinaryVariable *> *scope_;
vector<double> *incoming_messages_;
vector<double> *outgoing_messages_;
vector<Edges *> *edges_;

```

Internally, each factor operates on input `incoming_messages_` and writes to `outgoing_messages_`, where the incoming (or outgoing) message at index `i` is the message incoming from (or outgoing to) the variable stored in `scope_->at(i)`.

Messages stored in `outgoing_messages_` are *not* immediately visible to the rest of the network. There is an intermediate storage object of class `Edge` between each variable and each factor, which stores `to_variable_` and `to_factor_` messages, which are visible to the rest of the network.

In order to pass a message, `UpdateOutgoingMessage(int position_in_factor)` must be called. This will copy a message from `outgoing_messages_->at(position_in_factor)` to `edges_->at(position_in_factor).to_variable_`. This somewhat redundant storage is an intentional design decision, which makes it easier to implement advanced message schedules and to cache previous factor computation results. Similarly, to move a message from `edges_->at(position_in_factor).to_factor_` to `incoming_messages_->at(position_in_factor)`, we call `UpdateIncomingMessage(position_in_factor)`.

4.2 Subclassing Factors

Each `Factor` subclass must implement the following methods:

```

void ComputeAllOutgoingMessages();
double Likelihood();

```

Importantly, any algorithm can be used to compute these messages, and factors persist throughout an execution of message passing, so they can cache results from previous message computations.

References

Tarlow, D., Givoni, I., & Zemel, R. (2010). HOP-MAP: Efficient message passing for high order potentials. In *Artificial Intelligence and Statistics (AISTATS)*.

Appendix: Installing Boost

Thanks to Nikola Karamanov for the following:

```
--Download
boost_1_43_0.tar.bz2 from
http://sourceforge.net/project/showfiles.php?group_id=7586&package_id=8041

--Extract boost_1_43_0.tar.bz2
You will have a boost_1_43_0 directory somewhere

--cd to the boost_1_43_0 directory and run
./bootstrap.sh
./bjam

--cd to cyborg_public directory

--In Makefile after EXTRA_INCLUDES you should have:
BOOST = <path/to/boost_1_43_0>
CCFLAGS = -g -Wall -Werror -Wno-deprecated -I $(BOOST)
LDLAGS = -lm -L$(BOOST)/stage/lib/ -lboost_program_options

--In the shell run (command is on one line):
setenv LD_LIBRARY_PATH <path/to/boost_1_43_0>/stage/lib/:$LD_LIBRARY_PATH

(OR if LD_LIBRARY_PATH is undefined leave out the ":$LD_LIBRARY_PATH")

--Now the command ./cyborg should work (you'll need to run the previous
command every time so consider putting it in a script).
```