PGMs session 6 Inference: MAP Inference

So far, we have focused on marginal inference, which aims to calculate the marginal beliefs over a subset of random variables in a model. For this assignment, we turn to maximum a posteriori (MAP) inference, which aims to calculate the most likely combination of values for all the random variables in a model.

1 Preparation: Watch the Koller videos on MAP message passing

Watch the following videos in the Coursera course Probabilistic Graphical Models 2: Inference, Week 3:

- *MAP Message Passing: Max Sum Message Passing (20:27)*
- *MAP Message Passing: Finding a MAP Assignment (3:57)*
- *Optional: Other MAP Algorithms: Tractable MAP Problems (15:04)*

Notes on these videos:

- Using log factors changes factor products to factor summations.
- We can find the MAP values (instead of the normal marginal distributions) by replacing sum-marginalisation with max-marginalisation.
- The combination of the above two yields the max-sum algorithm. Apart from the change in the nature of the result, just about all the other properties and techniques we have seen before still holds. Those familiar with HMMs will recognise the the max-sum algorithm as the general case of the Viterbi algorithm used there.
- Not discussed in the videos: This can be very useful for continuous variables where the integrals required by the marginalisation do not have analytical solutions. By instead opting for the MAP solution in this manner, we are exchanging the need for integration for that of differentiation – in general a rather easier task (the derivative is zero at the maximum point).
- One can also build hybrid systems, where certain variables are being marginalised via summation/integration, and others by maximisation.

2 Preparation: Read Barber Section 5.2.1 on max-product inference

Barber does not use log factors, which results in a max-product algorithm compared to Koller's max-sum algorithm. Both algorithms are equivalent (in principle); we wil use Barber's max-product algorithm for the practical.

3 Information: Implementation of MAP inference in `emdw`

For MAP inference, `emdw` implements the max-product algorithm (i.e., Barber's flavour), not the max-sum algorithm (i.e., Koller's flavour). The only difference between belief propagation (the sum-product algorithm) and the max-product algorithm is that the marginalisation operation is replaced by the maximisation operation, and normalisation is performed differently. Using MAP inference in `emdw` simply involves replacing the default marginalisation operator with a maximisation operator, and replacing the default normalisation operator with one that is suitable for MAP inference. To do this for the `DiscreteTable` class, first create the appropriate operators as follows:

```
rcptr<Marginalizer> margPtr
   = uniqptr<Marginalizer>( new DiscreteTable_MaxMarginalize<T> );
rcptr<InplaceNormalizer> iNormPtr
   = uniqptr<InplaceNormalizer>( new DiscreteTable_InplaceMaxNormalize<T> );
rcptr<Normalizer> normPtr
   = uniqptr<Normalizer>( new DiscreteTable_MaxNormalize<T> );
```

The two normalisation operators scale the factor such that the maximum value of the table is 1. Each time you create a new `DiscreteTable` factor, you have to specify these operators for the factor. To illustrate this, consider the following factor that is created in the example file `example.cc`:

```
    rcptr<Factor> ptrXY =
      uniqptr<DT>(
        new DT(
          {X,Y},
          {binDom,binDom},
          defProb,
          {
            {{0,1}, 0.5},
            {{1,0}, 0.5},
          } ));
```

For MAP inference, this factor would be created as follows:

```
    rcptr<Factor> ptrXY =
      uniqptr<DT>(
        new DT(
          {X,Y},
          {binDom,binDom},
          defProb,
          {
            {{0,1}, 0.5},
            {{1,0}, 0.5},
          }
          margin, floor, false,
          margPtr, iNormPtr, normPtr
        ));
```

You can set the `double` arguments `margin` and `floor` to their default values of 0.0 (see `emdw/src/emdw-factors/discretetable.hpp` for their meanings). Now, when you call the member function `marginalize`, it will perform maximisation instead, and when you call the member functions `normalize` and `inplaceNormalize`, it will perform normalisation appropriate for MAP inference (i.e., the factor will be scaled such that the maximum value is 1). You can now perform message passing using exactly the same code as for marginal inference.

The result of MAP message passing is a set of so-called max-marginals – one for each cluster (this is similar to the cluster beliefs of belief propagation/update). A max-marginal contains a likelihood value for each combination of random variables in the scope of its cluster. If there is a single most likely global combination of random variable values, extracting this combination is easy: the most likely (local) combination of each max-marginal is consistent with the most likely global combination. However, if there are several most likely combinations, extracting a consistent combination can get tricky. For this, you have two options:

1. Similarly to the Viterbi algorithm, decoding can be done via a backtracking step. This is normally done via "back-pointers". Maximisation is the last step in constructing a new message. However, for each of the potential values in the message, you will now need to record the specific values of the variables you maximised over.

2. The easier alternative is to add a small amount of perturbation noise to all of the factors. This makes the maximal value in each cluster unique, and these maxima also synchronises over clusters.

4 Practical: Hamming (7,4) error-correction code (once again)

The objective of this practical is to implement MAP inference for the familiar Hamming (7,4) error-correction code problem.

Task 1: Manually implement the max-product algorithm for the junction tree of Session 5. Adapt your code for Task 1 of Session 5 for MAP inference, and extract the most likely combination of values for the sent bits $b_0, \dots, b_6$. Compare this to the full joint distribution you obtained in Session 3.

Task 2: Manually implement the max-product algorithm for the loopy cluster graph of Session 4. Adapt your code for Task 1 of Session 4 for MAP inference, and extract the most likely combination of values for the sent bits $b_0, \dots, b_6$. Compare this to the result of Task 1 above.

Task 3: Repeat Task 1 and Task 2 above, but now use `emdw`'s built-in functionality for MAP message passing. For both cases, extract the mostly likely combination and compare it with the results of Task 1 and Task 2 above.

5 Practical: Removing impulse noise from an image (optional)

This one is to exercise your ability to design a useful PGM. The design is much more open ended and therefore might take you somewhat longer to do. The resultant system will be loopy and, although still small (probably a few thousand factors), it will be *much* larger than the previous systems in this module. You need to:

- Design a set of factors that will reflect the semantics for the task at hand.
- Build the structure that will combine these factors into a graphical model with valid RIP.
- Do loopy inference (max-product) on this structure until it is calibrated.
- Do queries on the calibrated graph to determine whether a particular pixel position should be part of the foreground or background in the given image.

In the `noisy10.gz` file you will find the noisy version of a binarised 28x28 pixel hand-written character. It was obtained by randomly flipping 10% of the bits of the original clean image (available in `clean.gz`). Plot both (via `matplotlib` or similar) to get a feeling for the damage done. Your task is to build a PGM that will clean up this noisy image. For this you will have to think about the general properties of such handwritten characters (of course you should refrain from using the explicit detail of `clean.gz`). Do consider using factors larger than those of standard pairwise MRFs.

After you have designed and implemented the factors reflecting the semantics of this situation, you will have to consider how they can be compiled into a graphical model, and how inference will subsequently be done.

Using `emdw`: This code copes effortlessly with the task at hand. The vector of `Factors`, as well as the various observations (i.e., pixel values from the actual image as well as any other knowns) are used to automatically configure a valid `ClusterGraph`. This graph is calibrated by calling the loopy belief propagation function. The calibrated graph is then queried, after which one can extract the most likely combination of pixel foreground/background values.