A Back-Propagating the Error over the Belief Propagation Run

In this appendix, we will use a version of belief propagation where the messages and beliefs are normalized at every step. (Normalization is optional but is usually required for convergence testing and for decoding.)

In the main paper, μ (messages) and b (beliefs) refer to unnormalized probability distributions, but here they refer to the normalized versions. We use $\tilde{\mu}$ and $\tilde{b}$ to refer to the unnormalized versions, which are computed by the update equations below. The normalized versions are then constructed via

$$q(x) = \frac{\tilde{q}(x)}{\sum_{x'} \tilde{q}(x')} \tag{9}$$

A.1 Belief Propagation

The recurrence equations (2)–(5) for belief propagation are repeated below with normalization. To update a single message, the distribution $\mu_{i\to\alpha}$ or $\mu_{\alpha\to i}$, we compute the unnormalized distribution $\tilde{\mu}$, using equation (A.3) or (11) (respectively) for each value x_i in the domain of random variable X_i:

$$\tilde{\mu}_{i\to\alpha}(x_i) \leftarrow \prod_{\beta\in\mathcal{F}:\, i\in\beta, \beta\neq\alpha} \mu_{\beta\to i}(x_i) \tag{10}$$

$$\tilde{\mu}_{\alpha\to i}(x_i) \leftarrow \sum_{\mathbf{x}_\alpha:\, (\mathbf{x}_\alpha)_i = x_i} \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\alpha:\, j\neq i} \mu_{j\to\alpha}((\mathbf{x}_\alpha)_j) \tag{11}$$

We then compute the normalized version μ using equation (9). Upon convergence of the normalized messages or when another stopping criterion is reached (i.e., maximum number of iterations), beliefs are computed as:

$$\tilde{b}_{x_i}(x_i) \leftarrow \prod_{\beta\in\mathcal{F}:\, i\in\beta} \mu^{(T)}_{\beta\to i}(x_i) \tag{12}$$

$$\tilde{b}_\alpha(\mathbf{x}_\alpha) \leftarrow \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\alpha} \mu^{(T)}_{j\to\alpha}((\mathbf{x}_\alpha)_j) \tag{13}$$

The normalized beliefs b are used by a decoder d to produce a decode of the output variables. Finally, a loss function ℓ computes the "badness" of the decoded output as compared to the gold standard: $V = \ell(d(b), y^*)$. In this paper we work with decoders that are function of beliefs at the variables, but the method would also work with decoders that consider beliefs at the factors (e.g., variants of the Viterbi decoder).

The only requirement is that the decoder and loss function are differentiable functions of their inputs. Hence we replace non-differentiable functions, in particular *max*, with differentiable approximations such as *softargmax*, as described in the main paper.

A.2 Back-propagation

Let V be the loss of the system on a given example. We will use the notation $\eth y$ to represent $\partial V/\partial y$, called the **adjoint** of y. An adjoint is defined for each intermediate quantity that was computed during evaluation of V. If different quantities were assigned to the variable y at different times, then $\eth y$ will likewise take on different values during the algorithm, representing the various partials of V with respect to those various quantities.

Ultimately we are able to compute the adjoint $\eth\theta_j$ for each parameter θ_j, which gives us the gradient $\nabla_\theta V$.

We first compute V (the **forward pass**). This begins with belief propagation as described above. The only difference from standard loopy belief propagation is that we record an "undo list" (known as the tape) of the message values that are overwritten at each time step $t \in \{1, ..., T\}$. That is, if at time t the message $\mu_{\alpha\to i}$ was updated, then we save the **old value** as $\mu^{(t-1)}_{\alpha\to i}$ [1]. We then run the decoder over the resulting beliefs to obtain a prediction y, and compute the loss V with respect to the supervised answer y^*.

[1] In reality, the normalized version of the message $\mu^{(t-1)}_{\alpha\to i}$ alone is not sufficient for the backward pass because the

The **backward pass** begins by setting $\eth V = 1$. We then differentiate the loss function to obtain the adjoints of the decoded output: $\eth d(x_i) = \eth V \cdot \frac{\partial V}{\partial d(x_i)}$. (The actual formulas depend on the choice of loss function: if the decoded output for x_i were to change by an infinitesimal ϵ, how much would V change?) We use the chain rule again to propagate backward through the decoder to obtain the adjoints of the beliefs: $\eth b(x_j) = \sum_i \eth d(x_i) \cdot \frac{\partial d(x_i)}{\partial b(x_j)}$. (Again, the actual formula for this partial derivative depends on the decoder.)

From the belief adjoints, we can initialize the adjoints of the belief propagation messages by applying the chain rule to equations (12)–(13):

$$\eth \mu_{i\to\alpha}(x_i) \leftarrow \sum_{k\in\alpha: k\neq i} \psi_\alpha((x'_\alpha)_k) \prod_{j\in\alpha: j\neq i} \mu_{j\to\alpha}(x_j) \eth \tilde{b}_{x_i}(x_i) \tag{14}$$

$$\eth \mu_{\alpha\to i}(x_i) \leftarrow \prod_{\beta\in\mathcal{F}:\, i\in\beta, \beta\neq\alpha} \mu_{\beta\to i}(x_i) \eth \tilde{b}_\alpha(\mathbf{x}_\alpha) \tag{15}$$

$$\eth \psi^{(T)}_\alpha(x_\alpha) \leftarrow \prod_{i\in\alpha} \mu_{i\to\alpha}((\mathbf{x}_\alpha)_i) \eth \tilde{b}_\alpha(\mathbf{x}_\alpha) \tag{16}$$

Starting with these message adjoints, the algorithm proceeds to run the belief propagation computation backwards as follows. Loop for $t \leftarrow \{T, T-1, ..., 1\}$. If the message update at time t was to a message $\mu_{i\to\alpha}$ according to equation (A.3), we increment the adjoint for every β occurring in the right-hand side of equation (A.3), for each value x_i in the domain of X_i:

$$\eth \mu_{\beta\to i}(x_i) \leftarrow \eth \mu_{\beta\to i}(x_i) + \left(\prod_{i\in\gamma:\gamma\neq\alpha,\beta} \mu_{\gamma\to i}(x_i) \right) \eth \tilde{\mu}_{i\to\alpha}(x_i) \tag{17}$$

We then undo the update, restoring the old message (and initializing the adjoints of its components to 0):

$$\mu_{i\to\alpha} \leftarrow \mu^{(t-1)}_{i\to\alpha} \tag{18}$$

$$\eth \mu_{i\to\alpha}(x_i) \leftarrow 0 \tag{19}$$

Otherwise, the message update at time t was $\mu_{\alpha\to i}(x_i)$ according to equation (11). We increment the adjoints for all j and ψ_α occurring on the right-hand side of equation (11):

$$\eth \mu_{j\to\alpha}(x_j) \leftarrow \eth \mu_{j\to\alpha}(x_j) + \sum_{x_i} \left(\sum_{\mathbf{x}_\alpha:\, (\mathbf{x}_\alpha)_i = x_i \wedge (\mathbf{x}_\alpha)_j = x_j} \psi_\alpha(x_\alpha) \prod_{k\in\alpha: k\neq i,j} \mu_{k\to\alpha}(x_k) \right) \eth \tilde{\mu}_{\alpha\to i}(x_i) \tag{20}$$

$$\eth \psi_\alpha(x_\alpha) \leftarrow \eth \psi_\alpha(x_\alpha) + \sum_{i\in\alpha} \left(\prod_{j\in\alpha: j\neq i} \mu_{j\to\alpha}(\mathbf{x}_j) \right) \eth \tilde{\mu}_{\alpha\to i}(x_i) \tag{21}$$

And again, we undo the update:

$$\mu_{\alpha\to i} \leftarrow \mu^{(t-1)}_{\alpha\to i} \tag{22}$$

$$\eth \mu_{\alpha\to i}(x_i) \leftarrow 0 \tag{23}$$

unnormalized message $\tilde{\mu}^{(t-1)}_{\alpha\to i}$ is also needed. There are two possible solutions: to save the unnormalized version of the message $\tilde{\mu}^{(t-1)}_{\alpha\to i}$ and compute the normalized value as needed or to save the normalizing constant together with the message. We use the latter option in our implementation for efficiency. Given the normalization constants, and $\mu^{(t-1)}_{\alpha\to i}$, it is trivial to reconstruct the values for $\tilde{\mu}^{(t-1)}_{\alpha\to i}$ in the backward pass, so this computation is omitted from the rest of the discussion for brevity.

In either case, for each normalized distribution q whose adjoint was updated on the left-hand side of the above rules, we then update the adjoint of the corresponding normalized distribution $\tilde{q}$:

$$\eth \tilde{q}(x) = \frac{1}{\sum_{x'} \tilde{q}(x')} \left(\eth q(x) - \sum_{x'} q(x') \eth q(x') \right) \tag{24}$$

Finally, we compute the adjoints of the parameters θ (i.e., the desired gradient for optimization). Remember that each real-valued potential $\psi_\alpha(\mathbf{x}_\alpha)$ (where $\mathbf{x}_\alpha$ represents a specific assigment to variables X_α) is derived from θ by some function: $\psi_\alpha(\mathbf{x}_\alpha) = f(\theta)$. Adjoints of θ are computed from the final ψ adjoints as:

$$\eth \theta_i = \sum_{\alpha, \mathbf{x}_\alpha} \eth \psi_\alpha(\mathbf{x}_\alpha) \cdot \frac{\partial f(\psi_\alpha(\mathbf{x}_\alpha))}{\eth \theta_i} \tag{25}$$

A.3 Complexity

For nodes of high degree in the factor graph, the computations in the forward pass can be sped up using the "division trick." For example, the various messages $\prod_{\beta \in \mathcal{F}:\, i \in \beta, \beta \neq \alpha} \mu_{\beta \to i}(x_i)$ in (10) for different α can be found by found by first computing the belief (12) and then dividing out the respective factors $\mu_{\alpha \to i}(x_i)$.

To perform the backward pass, we must save all updated values during the forward pass, requiring space of *O(runtime)*.

The runtime of the backward pass is asymptotically the same as that of the forward pass (about three to four times as long). This is not the case for a straightforward implementation, since a single update $\tilde{\mu}_{i \to \alpha}$ in the forward pass results in n_i updates in the backward pass, where n_i is the number of functions (features) in which node i participates (see equation (17)). Similarly, a single update $\tilde{\mu}_{\alpha \to i}$ in the forward pass results in $2n_\alpha$ updates, where n_α is the number of nodes in the domain of ψ_α (from the updates in equations (20) and (21)). However, the computations can again be sped up using the division trick—for example, $\prod_{i \in \gamma : \gamma \neq \alpha, \beta} \mu_{\gamma \to i}(x_i)$ can be computed as $\left(\prod_{i \in \gamma : \gamma \neq \alpha} \mu_{\gamma \to i}(x_i) \right) / \mu_{\beta \to i}(x_i)$. Note from equation (10) that $\tilde{\mu}_{i \to \alpha}(x_i) \leftarrow \prod_{i \in \beta, \beta \neq \alpha} \mu_{\beta \to i}(x_i)$, so this quantity is already available and the whole product can be computed using a single division as $\tilde{\mu}_{i \to \alpha}(x_i) / \mu_{\beta \to i}(x_i)$. This optimization saves considerable amount of computation when nodes participate in many functions. The same optimization trick can be used for the updates in equations (20) and (21) and will lead to savings when domains of potential functions contain multiple nodes. This is not the case for the experiments in this paper as we work only with potential functions defined over pairs of nodes. In general, running inference in MRFs with functions over domains with large cardinalities is computationally expensive. Thus, MRFs used in practice can be expected to either have limited size potential function domains or use specialized computations for the $\mu_{\alpha \to i}$ messages. Therefore, speeding up the computation in equations (20) and (21) is less of a concern. Our implementation runs approximately three times slower than the forward pass.

A.4 Hessian-Vector Product

Section 4.2 of the main paper notes that for our Stochastic Meta-Descent optimization, we must repeatedly compute not only the *gradient* of the loss (with respect to the current parameters θ), but also the product of the *Hessian* of the loss (again computed with respect to the current θ) with a given vector v.

This requires a small adjustment to the algorithms above, using "dual numbers." Every quantity x computed by the forward or backward pass above should be replaced by an ordered pair of scalars $(x, \mathcal{R}\{x\})$, where $\mathcal{R}\{x\}$ measures the instantaneous rate of change of x as θ is moved along the direction v.

As the base case, each θ_i is replaced by (θ_i, v_i). For other quantities, if x is computed from y and z by some differentiable function, then it is possible to compute $\mathcal{R}\{x\}$ from y, z, $\mathcal{R}\{y\}$, and $\mathcal{R}\{z\}$. Thus, operations such as addition and multiplication can be defined on the dual numbers.

In practice, therefore, code for the forward and backward passes can be left nearly unchanged, using operator overloading to make them run on dual numbers. See Pearlmutter (1994) for details.

B Supplementary Results

test setting	train setting			
	frac-MSE	int-F	int-L1	APPR-LOGL
frac-MSE	**.00057**	.00122	.00115	.0071
int-F	.00109	**.00081**	.00106	.0117
int-L1	.00069	.00096	**.00079**	.00751
APPR-LOGL	.11141	.16153	.1508	**-0.31618**

Table 6: Results for all pairs of settings

As suggested by an anonymous reviewer, Table 6 lists the results (the Δ*loss* as in all previous tables) for all pairs of train and test settings. More specifically, the training loss is the option including staged training and hybrid loss where applicable (i.e., the *-in* and *-hyb* settings).

The results show that matching training and test conditions (the bolded diagonal of the table) leads to the best results in all settings. Only the last column was achievable with previously published algorithms, so our contribution is to provide the diagonal elements, which in each row do better than the last column.