Empirical Risk Minimization of Graphical Model Parameters Given Approximate Inference, Decoding, and Model Structure

Veselin Stoyanov **Alexander Ropson** **Jason Eisner**
Center for Language and Speech Processing, Johns Hopkins University
Baltimore, MD 21218

Abstract

Graphical models are often used "inappropriately," with approximations in the topology, inference, and prediction. Yet it is still common to train their parameters to approximately maximize training likelihood. We argue that instead, one should seek the parameters that minimize the empirical risk of the entire imperfect system. We show how to locally optimize this risk using back-propagation and stochastic meta-descent. Over a range of synthetic-data problems, compared to the usual practice of choosing approximate MAP parameters, our approach significantly reduces loss on test data, sometimes by an order of magnitude.

1 Introduction

Graphical models are widely used across AI. By modeling *joint* distributions, they permit structured prediction with arbitrary patterns of missing data (including latent variables and statistical relational learning). By explicitly representing how distributions are *factored*, they can expose problem-specific structure to be exploited by generic inference and learning algorithms.

However, several compromises are often made in practice. Predictive systems based on graphical models typically suffer from multiple approximations:

Mis-specified model structure. Usually the model structure is a guess or an oversimplification. We do not know that the true distribution of the data can be described by any setting of the model parameters θ.

MAP estimation. Even if the model structure is correct, we cannot infer the correct parameters θ from finite training data. A proper Bayesian approach would integrate over the posterior distribution of θ, but this can be expensive. It is common to choose a single θ vector via MAP estimation (i.e., empirical Bayes).

Approximate inference. Even if the model structure and parameters are both correct, we often cannot afford exact inference (in the usual sense of efficiently computing posterior marginals or partition functions). For model structures of high treewidth, we must fall back on approximations such as variational inference. Inference also plays a key role in parameter estimation, and Kulesza and Pereira (2008) show that approximate inference here can lead to pathological learning.

Approximate decoding. Even if we can perform exact inference, that is not the final goal. Ideally, a system would follow decision theory and emit the prediction, decision, or estimate that has lowest expected *loss* under the posterior distribution, i.e., the lowest Bayes risk. This is called a generalized Bayes rule, or a minimum Bayes risk (MBR) decoder. Alas, in structured prediction, global loss functions can make MBR decoding intractable even when exact inference is tractable.[1] So various heuristic procedures are used in place of MBR to extract structured predictions.

These approximations may have been forced by practical considerations—so we are stuck with some given model structure, approximate inference algorithm, decoding procedure, and parameter vector θ to optimize.[2] The loss function is also given.

Solution: Direct Risk Minimization. Our main observation is that at the end of the day, this is merely a discriminative learning setting. Just as when train-

Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15. Copyright 2011 by the authors.

[1] E.g., in an HMM, exact inference is tractable, yet it is intractable to predict the emission sequence (with nothing observed) that minimizes expected global 0-1 loss. I.e., finding the most probable emission sequence (summing out the states) is NP-hard (Casacuberta and Higuera, 2000).

[2] More generally, we may be willing to consider a *family* of model structures or inference/decoding procedures. In this case, θ will include extra parameters that select within these families. We try to choose the best θ at training time.

ing a linear classifier, we should simply set the model parameters θ to make the system perform accurately at test time. The entire system— approximations and all—can be treated as a black-box decision rule to be tuned via θ. The computation performed inside the black box may have been *motivated* by probabilistic inference (albeit with approximations). But ultimately it is just some parametric function constructed to suit the problem at hand, and one may accordingly train its parameters θ to minimize **risk** (expected loss).

Minimizing risk is the proper goal of any training, since it directly optimizes the evaluation measure. This is not to say that traditional training methods are *always* misguided. If one is lucky enough to enjoy correct model structure, exact inference, and MBR decoding, one's risk is minimized by choosing the true model parameters, which is accomplished by traditional maximum-likelihood or MAP estimation—at least in the limit of infinite training data. But trying to identify the true parameters of an approximate system makes no sense: no "true parameters" exist.

In this paper we focus on locally minimizing the *empirical* risk, i.e., the observed error of the system on supervised training data. (See section 10 for future work that goes beyond this setting.)

Recall how a feed-forward neural network is trained. One does not need to make any probabilistic interpretation of the neural network. It is simply a parametric function of the input, $y = f_\theta(x)$, that is used to predict y from x. The training procedure directly seeks a θ that works well in practice. In the terminology of decision theory, f_θ is a family of estimators. Traditionally, the parameters θ (synaptic weights) are tuned by gradient descent to minimize the *empirical risk* $\tilde{R}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$, which is the average loss ℓ of the chosen estimator f_θ over the training set $\{(x_i, y_i)\}$. The empirical risk is merely a differentiable function of θ, determined by the structure of the neural network, the inference method and the loss function.

Graphical models can be trained in the same way. When a practitioner builds a system around some approximate Bayesian technique, she is constructing a family of decision rules f_θ. The empirical risk is a function $\tilde{R}(\theta)$ (often differentiable) of the parameters θ. We can use methods such as gradient descent to tune θ to minimize the empirical risk. In general, it is fast to find the gradient by automatic differentiation—and we will show specifically how to do this when the inference algorithm is loopy belief propagation.

2 Modeling and Inference

An undirected graphical model, or **Markov random field** (MRF), is defined by a triple $(\mathcal{X}, \mathcal{F}, \Psi)$. $\mathcal{X} = (X_1, X_2, \ldots, X_n)$ is a sequence of n random variables; we use $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ to denote a possible assignment of values. $\mathcal{F}$ is a collection of subsets of $\{1, 2, \ldots, n\}$; for each $\alpha \in \mathcal{F}$, we write $\mathbf{x}_\alpha$ to denote a sub-assignment to the subset of variables $\mathcal{X}_\alpha$.

Finally, $\Psi = \{\psi_\alpha : \alpha \in \mathcal{F}\}$ is a set of **factors**, where each factor ψ_α is a function that maps each x_α to a **potential** value in $[0, \infty)$. Each of the factors ψ_α depends implicitly on a parameter vector θ.

The probability of a given assignment $\mathbf{x}$ is given by

$$p_\theta(\mathcal{X} = \mathbf{x}) = \frac{1}{Z} \prod_{\alpha \in \mathcal{F}} \psi_\alpha(x_\alpha) \tag{1}$$

where Z is chosen to normalize the distribution.

2.1 Loopy Belief Propagation

Inference in general MRFs is intractable. We focus in this paper on a popular approximate inference technique, **loopy belief propagation** (BP) (Pearl, 1988). BP is efficient provided that the domains of the factors ψ_α are small (i.e., no ψ_α has to evaluate too many local configurations x_α) and exact for "non-loopy" Ψ.

BP uses the following iterative update equations to solve for the **messages** $\mu_{i\to\alpha}$ and $\mu_{\alpha\to i}$. Both kinds of messages are unnormalized probability distributions over the possible values of X_i (initialized to uniform):

$$\mu_{i\to\alpha}(x_i) \;\leftarrow \prod_{\beta\in\mathcal{F}:\, i\in\beta, \beta\neq\alpha} \mu_{\beta\to i}(x_i) \tag{2}$$

$$\mu_{\alpha\to i}(x_i) \;\leftarrow \sum_{\mathbf{x}_\alpha:\, (\mathbf{x}_\alpha)_i = x_i} \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\alpha:\, j\neq i} \mu_{j\to\alpha}((\mathbf{x}_\alpha)_j) \tag{3}$$

Provided that the messages (or rather, normalized versions of them) converge, we may then compute

$$b_{x_i}(x_i) \;\leftarrow\; \frac{1}{Z_{b_{x_i}}} \prod_{\beta\in\mathcal{F}:\, i\in\beta} \mu_{\beta\to i}(x_i) \tag{4}$$

$$b_\alpha(\mathbf{x}_\alpha) \;\leftarrow\; \frac{1}{Z_{b_\alpha}} \psi_\alpha(\mathbf{x}_\alpha) \prod_{j\in\alpha} \mu_{j\to\alpha}((\mathbf{x}_\alpha)_j) \tag{5}$$

where the **beliefs** b_i and b_α respectively approximate the MRF's marginal distributions over the variable X_i and the set of variables X_α. $Z_{b_{x_i}}$ and Z_{b_α} are normalizing functions.

To approximate the *posterior* marginals given some observations, run BP over a modified MRF that enforces those observations. (If X_i has been observed to equal v, then set $\psi_{\{i\}}(x_i)$ to be 1 or 0 according to whether $x_i = v$ or not, first adding $\{i\}$ to $\mathcal{F}$ if $\{i\} \notin \mathcal{F}$.)

3 Training θ

We assume a supervised or semi-supervised learning setting. Each training example $\{(x^i, y^i)\}$ consists of a

set x^i of input random variables with observed values, and a set y^i of output random variables with observed values. (We do not assume that the same sets are designated as input and output in each example, and there may be additional hidden variables.)

For purposes of this paper, our system's goal is to predict y^i. $\ell(y, y^i)$ defines the task-specific loss that the system would incur by outputting y.

3.1 The Standard Learning Paradigm

MRFs are usually trained by maximizing the log-likelihood given training data (x_i, y_i):

$$\theta^* = \operatorname*{argmax}_\theta \mathrm{LogL}(\theta) = \operatorname*{argmax}_\theta \sum_i \log p_\theta(x^i, y^i) \quad (6)$$

The log-likelihood $\log p_\theta(x^i, y^i)$, or the conditional log-likelihood $\log p_\theta(y^i \mid x^i)$, can be found by considering two slightly different MRFs, one with (x^i, y^i) both observed and one with only the conditioning events (if any) observed. The desired result is the difference between the $\log Z$ values of the two MRFs.

To approximate the $\log Z$ of each MRF, one can run belief propagation. The resulting beliefs can be combined into a quantity called the **Bethe free energy** that approximates $-\log Z$ (Yedidia et al., 2000). The gradient of the Bethe free energy is very closely connected to the beliefs, making it easy to follow the gradient of the approximate log-likelihood. Training MRFs based on the Bethe free energy approximation has been shown to work relatively well in practice (Vishwanathan et al., 2006; Sutton and McCallum, 2005).

3.2 Decoding

The log-likelihood training aims to produce θ^* that causes the MRF to predict good beliefs about the output variables y^i. A **decoder** is a decision rule that converts these beliefs into a prediction. According to the minimum Bayes risk (MBR) principle, the procedure should pick the output that minimizes the expected risk under p_θ:

$$y^* \quad = \quad \operatorname*{argmin}_y \mathbb{E}_{p_\theta(y'|x_i)}[\ell(y, y')] \quad (7)$$

The MBR decoding procedure depends on the loss function ℓ and can be computed efficiently only in some cases. The MBR decoders that we use are described in more detail in Section 5.1.

3.3 Empirical Risk Minimization

The risk minimization principle says that if we have f_θ, a family of decision functions parameterized by θ, we should select θ to minimize the expected loss under the true data distribution over (x, y):

$$\theta^* \stackrel{\text{def}}{=} \operatorname*{argmin}_\theta \mathbb{E}[\ell(f_\theta(x), y)] \quad (8)$$

In practice, the true data distribution is unknown, but we can do **empirical risk minimization** and take the expectation over our sample of (x^i, y^i) pairs. In our setting, we explicitly evaluate the loss of a given θ on example (x^i, y^i) by computing from x^i our beliefs b, decoding the beliefs about y^i to obtain the prediction y^*, and returning $\ell(y^*, y^i)$.

4 Gradient of Empirical Risk

To carry out the above empirical risk minimization, we propose to use a gradient-based optimizer. The gradient indicates how slight changes to θ would affect the loss $\ell(y^*, y^i)$ via the beliefs b and prediction y^*.

4.1 Back-Propagation

To determine this gradient on example (x^i, y^i), we employ automatic differentiation in the reverse mode (Griewank and Corliss, 1991), a general technique for sensitivity analysis in computations. The intuition behind automatic differentiation is that our entire "black-box" predictor is nothing but a sequence of elementary differentiable operations. If intermediate results are recorded during the computation of the function (the forward pass), we can then compute the partial derivative of the loss with respect to each intermediate quantity, in the reverse order (the backward pass). At the end, we have accumulated the partials of the loss with respect to each parameter θ.

A well-known special case of this algorithm is back-propagation in feed-forward neural networks. Its extension to recurrent neural networks (Williams and Zipser, 1989) involves cyclic updates like those in BP. In this case we must consider an "unrolled" version of the forward pass, in which "snapshots" of a variable at times t and $t + 1$ are treated as distinct variables, with one perhaps influencing the other.

In our case the forward pass constitutes of running BP (equations (2) and (3)) until convergence is "good enough." Beliefs b are computed from the final messages using equations (4) and (5). The beliefs are then converted to a decision $y^* = d(b(y|x))$ using a decoder function d. Finally, the loss relative to the truth y^i is computed as $\ell(y^*, y^i)$. The forward pass includes inference in the model, but we record the messages that are sent and their order, as detailed in our Appendix.[3]

The backward pass computes the partials of loss with respect to the decision (differentiating the loss function) and next with respect to the marginal beliefs (differentiating the decoder). Subsequently, BP is replayed backwards in time, computing the partials with

[3] We do not assume that BP is run to convergence, so we must record what the forward pass actually accomplished.

respect to each message that was sent on the forward pass, and eventually with respect to the input parameters θ. The total time required by this algorithm is roughly twice the time of the forward pass, so its complexity is equivalent to approximate inference. The complete equations can be found in our Appendix.

4.2 Numerical Optimization

The above gradient could be used directly to optimize θ. While the objective function is locally rather bumpy (see below), stochastic gradient descent has some ability to escape small local optima.

However, we find that we obtain better optima when we collect second-order information about the optimization surface. Instead of stochastic gradient descent, we use the Stochastic Meta-Descent (SMD) method of Schraudolph (1999). SMD maintains a separate positive gain adaptation η_i for each optimization dimension θ_i. Parameter updates are scaled by η_i. Updates for the η_i themselves are computed using the product of the Hessian matrix with a vector. For this, we apply more automatic differentiation magic. It is not necessary to compute the full Hessian—as our Appendix explains, a Hessian-vector product can be computed by forward-mode automatic differentiation of the back-propagation pass (Pearlmutter, 1994; Griewank and Walther, 2008), without increasing the asymptotic complexity.

5 Experiments

To allow a proper factorial experimental design with a range of controlled and well-understood conditions, we experiment on artificially generated data. A companion paper shows improvements on 3 natural language tasks using real data (Stoyanov and Eisner, in review).

We randomly generate graphical models with known structure and parameters (the *true model*). We use 12 models consisting of a varying number of random variables and varying degree of connectivity in the model (shown in Table 1). We use binary random variables and functions ψ over pairs of random variables. Model structure is generated by picking edges at random. The true parameters θ_i are sampled IID from the standard normal, and each potential value $\psi_\alpha(\mathbf{x}_\alpha)$ is set to $\exp\theta_i$ for a different i. 1 lists all models used in the experiments. We generate training and test sets of 1000 examples from the *true model* by Gibbs sampling. Our experiments perform conditional training (i.e. we are training Conditional Random Fields). We select a random third of the random variables of the *true model* and designate them as *input* variables, another randomly selected third we designate as *hidden* variables and the rest we consider *output* variables. We then

num. vars (n)		50	100	150	200
num. edges	$2n$	50·100	100·200	150·300	200·400
	$4n$	50·200	100·400	150·600	200·800
	$n\ln(n)$	50·195	100·461	150·752	200·1051

Table 1: A listing of all models used in the experiments. 50·100 denotes a MRF with 50 binary variables and 100 binary edges.

learn a function of the input variables with the goal of minimizing loss on the output variables. Hidden variables are not observed even in training.

5.1 Test Settings

We experiment with different settings for the type of output the decoder is expected to produce and different loss functions.

Type of output:

- **Integer:** The algorithm is required to output a complete assignment for the output random variables (i.e., it has to commit to a specific value for each output random variable).
- **Fractional (soft):** The algorithm can output a fractional assignment for the output random variables. For example, it may hedge its bets by predicting 0.6 for a variable X with domain $\{0, 1\}$.
- **Distributional:** The output is a distribution over the output random variables. The algorithm outputs probability for a joint assignment of the input and output variables. In our work, this setting is used only for the APPR-LOGL loss function.

Loss functions (to be averaged over examples):

- **L1 loss:** L1 loss $L_1 = \frac{1}{k}\sum_i |y_i - y_i^*|$, where k is the number of output nodes. For *integer* outputs on our binary variables, it is proportional to Hamming loss or accuracy.
- **MSE:** Mean squared error $mse = \frac{1}{k}\sum_i (y_i - y_i^*)^2$. Equivalent to Hamming loss for *integer* outputs.
- **F-measure:** The harmonic average of precision and recall, $F = (2 * prec * rec)/(prec + rec)$. F-measure is defined for integer outputs.
- **Conditional log-likelihood:** The negative of the conditional log-likelihood of the test data under the predicted distribution of the model. Approximated in our case, since we use loopy MRFs.

As noted previously in the paper, the MBR decoder depends on the loss. It also depend on the type of output that the system is allowed to predict (i.e., integer vs. fractional). The MBR decoders for the loss functions that we will be using are discussed below and listed in Table 2:

L1 loss: Expected loss in both the integer and fractional case is minimized by placing all probability mass

Output	Loss	Decoder	Setting
integer	L1	max	int-L1
	MSE	max	int-L1
	F	apprF	int-F
fractional	L1	max	int-L1
	MSE	ident	frac-MSE
	F	apprF	int-F
distributional	LogL	distr	APPR-LOGL

Table 2: Experimental settings and their corresponding shortcut names. Some names appear in more than one cell in the table indicating that the particular conditions lead to equivalent settings of the algorithm. In total, we experiment with four unique settings.

on the most probable assignment for each output variable. In other words, the MBR decoder is the *argmax* function. The *argmax* is not differentiable, so when training, we enable back-propagation by using a soft version of the *argmax* function parameterized by temperature t: $\text{softargmax}(x,t) = x^{\frac{1}{t}} / \sum_{x'} x'^{\frac{1}{t}}$.

MSE: In the integer case this loss is equivalent to the L1 loss, so the same MBR decoder applies. If fractional outputs are allowed, the MBR decode of a binary-valued variable is its marginal probability. In other words, the MBR decoder is the *identity* function.

F-measure: This loss does not factorize over the output variables, making MBR decoding intractable (computing the expected loss includes summing over exponentially many settings for the random variables). Our approximate MBR decoder simply picks the threshold that would assign an equal number of variables to 0 and 1.[4] In preliminary experiments, the two approximations performed identically, so we chose the simple one in the interest of time. When minimizing F-score we again use *softargmax* during training.

log-likelihood: This loss requires no decoding.

Standard learning setting. Our baseline training setting (labeled APPR-LOGL below) follows Vishwanathan et al. (2006). It uses SMD to maximize the *conditional log-likelihood* of the training data ($\log p(y^i \mid x^i)$, as in CRFs) as approximated by loopy BP. (At test time, we do decode the beliefs using the proper MBR decoder matched to the evaluation loss function.)

Error Back-Propagation. Here we take the loss into account during training, using SMD this time to minimize the *empirical risk*. The gradient is computed as in Section 4.1. We implemented our algorithm in the libDAI framework (Mooij, 2010) extending the implementation of Eaton and Ghahramani (2009) (see Related Work Section).

[4]We also tried sampling from the posterior distribution and selecting the threshold value that minimizes the expected loss according to the samples. This is better-motivated but slower, and performed identically in preliminary experiments.

6 The Optimization Landscape

The advantage of using ERM is that it properly simulates test conditions. While it is not a convex objective, neither is the conventional choice of *approximate* log-likelihood—nor even exact log-likelihood in semi-supervised cases like ours.

To visualize the landscape of objective functions, we show in Figure 1 plots of the different losses in a particular direction. Plots show the continuum of loss on a line through the true parameters θ^* ($\alpha = 0$) and the parameters θ' found by some method ($\alpha = 1$). Each point shows the loss from an interpolated parameter vector $(1-\alpha)\theta^* + \alpha\theta'$. The plots are computed for a single model and are along only one dimension—the true optimization surface is high-dimensional.

Plots in Figure 1 show that approximate log-likelihood appears to be smoother than the other three loss functions. It is also clear, however, that the approximate log-likelihood function has a global minimum that does not occur at point θ^* (the true parameters), and the other three loss functions have other minima. Of the other loss functions, MSE appears smoothest and appears to closely resemble F and L1 loss.

7 Dealing with Non-Convexity

The previous section suggests that loss functions that we want to optimize can be non-convex and bumpy. In fact, initial experiments showed that the optimizing F-score and L1 loss was prone to getting stuck in local optima. We propose two continuation methods to deal with the non-convexity of the optimization function.

Interpolated Objective. We observed that *MSE* is smoother than F-score and L1 loss and the three losses have similar shapes. This motivates the use of a hybrid optimization function, which is a mix of the smoother loss and the function that ultimately needs to be optimized. By changing the balance between the two functions we can rely on the smooth function to get us to a good region and switch to optimizing the test loss. More formally, we define a hybrid loss function $\ell_{1,2}(y,y') = \lambda\ell_1(y,y') + (1-\lambda)\ell_2(y,y')$ between losses ℓ_1 and ℓ_2. The coefficient λ changes from 0 to 1 during training. Preliminary experiments on external models found that using a three value schedule $\lambda \in \{0, .5, 1\}$ and changing the value upon convergence works well in practice. In our experiments we use hybrids with *MSE* for F and L1 losses and label the corresponding runs with *-hyb*.

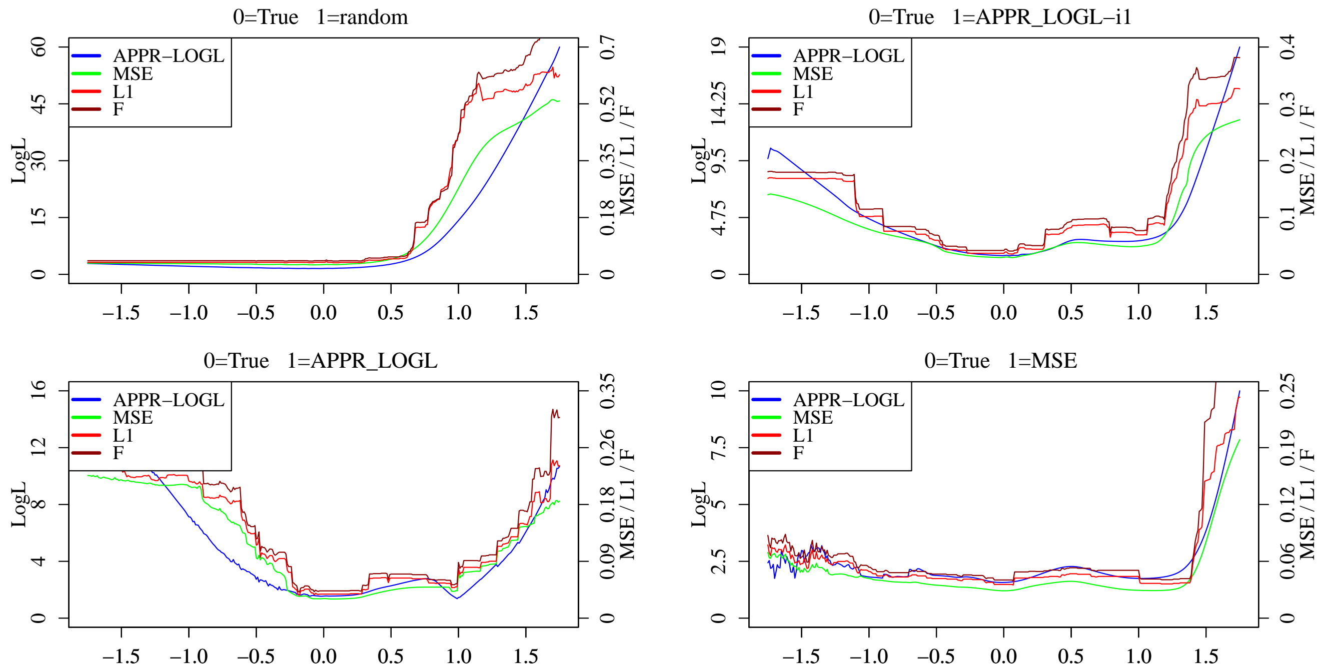

Figure 1: Plots of loss (objective) functions (on the $50 \cdot 200$ model) starting at the true generating parameters (θ^*) and moving toward: a random initialization (top left); the model found by one iteration of APPR-LOGL training (top right); APPR-LOGL training upon termination (bottom left); and, *MSE* training upon termination (bottom right). Note that the y axes have different scales.

Staged Training. The second strategy is motivated by the observation that the approximate log-likelihood is smooth and generally gets in the right region of parameter values. Thus, we can run a few iterations of APPR-LOGL (we use three) and use the learned parameters to initialize further tuning for loss, much as (Hinton et al., 2006) follow a unsupervised learning in a deep belief network with supervised tuning.

8 Results and Discussion

Results are averaged over the 12 models that we use in testing. We report the average of the difference between the loss of the training run and the loss of the corresponding true model using MBR decoding. Score of 0 indicates performance identical to the optimal.

All models were trained for 25 iterations of SMD with the exception of the hybrid models, which were trained until convergence of the optimization (in all cases convergence took < 25 iterations). We used 5 random restarts for each run with the exception of the *-in* runs where we used a single run. Parameters for the SMD algorithm (η_0, μ and λ) were tuned using grid search on supplemental models not used in the evaluation.

8.1 Overall Results

Table 3 lists the overall results of our experiments under "ideal" conditions—the exact model structure is known, there is sufficient amount of training data and BP is run to convergence. The table shows results for the four testing settings and for training runs that use hybrids (*-hyb*), staged training (*-in*) or both (*-in-hyb*) when applicable. Error back-propagation always outperforms the traditional APPR-LOGL setting on average. Both strategies for overcoming non-convexity appear to work, but improvements using only the hybrid loss are smaller and not statistically significant, while improvements using staged training are statistically significant ($p < 0.05$). Best results are obtained by combining the two strategies: staged training with a few iterations of APPR-LOGL followed by learning with hybrid loss. In the rest of the results, we only report this learning setting omitting the *-in-hyb* suffix.

Improvements are greatest when the loss function is MSE, which we empirically found to be relatively smooth. Error back-propagation is also very beneficial in the case of F-score where the MBR decoder is only approximate. By keeping the decoder fixed and learning parameters that minimize the loss of the decoder, EMR training can help the model learn parameters that optimize the particular approximate decoder.

Finally, we observe that the models trained on APPR-LOGL exhibit smaller approximate negative log-likelihood than the true model, which confirms our observation that the approximation induces a different global minimum of the log-likelihood function that is

test setting	train setting	Δloss	wins
frac-MSE (.04610)	APPR-LOGL	.00710	
	frac-MSE	.00482	5·0·7
	frac-MSE-in	**.00057**	12·0·0
int-F (.06425)	APPR-LOGL	.01170	
	int-F-hyb	.00411	7·0·5
	int-F-in	**.00115**	10·1·1
	int-F-hyb-in	**.00081**	11·0·1
int-L1 (.06385)	APPR-LOGL	.00751	
	int-L1-hyb	.00398	5·1·6
	int-L1-in	**.00137**	10·2·0
	int-L1-hyb-in	**.00079**	10·2·0
APPR-LOGL	APPR-LOGL	-.31618	

Table 3: Average loss for the different training settings. Δ*loss* lists the difference between the performance of the trained and the true model (negative loss indicates smaller loss than the true model). The average loss of the true model in a setting is shown in parentheses below the setting name. The *wins* column shows on how many models the setting wins/ties/loses vs. APPR-LOGL training. A bold number indicates a statistically significant improvement over APPR-LOGL ($p < 0.05$, paired permutation test).

not a minimum for the other loss functions.

Table 6 in Appendix B lists additional results for all pairs of test settings and training runs and shows that the smallest loss is achieved when training and test conditions are matched in all of our settings.

8.2 Model Structure Mismatch

In real problems, model structure matches the true structure of the process generating data only approximately. To represent this condition, we introduce mismatch between the structure of the true model and the structure of the model that we train by removing and adding at random a pre-specified percentage of the links of the graphical model.

Table 4 shows the results of the mismatched condition. Again, error back-propagation beats APPR-LOGL at all levels of structure noise, except for L1 loss, where it performs worse at the 30% level (not statistically significant) and improvement is statistically insignificant at the 40% level. Performance degradation in the error back-propagation runs is gradual. Interestingly, performance of the APPR-LOGL training slightly improves with a low level of structure noise. We speculate that the noise acts as a regularizer to prevent APPR-LOG from overfitting to the approximate criterion.

8.3 Approximation quality

Finally, to emulate the case in which the approximate algorithms may be forced to terminate early, we limit the run of BP to a fixed number of iterations. Table 5 shows the results of our experiments for different number of BP iterations. We use 100 iterations as our base case as most of the runs converge in that limit.

test setting	train setting	Perturbation 10%	20%	30%	40%
frac-MSE	APPR-LOGL	.00352	.00642	.00622	.01118
	frac-MSE	**.00101**	**.00316**	**.00312**	**.00534**
		12·0·0	11·0·1	11·0·1	10·0·2
int-F	APPR-LOGL	.01042	.01928	.01026	.02123
	int-F	**.00095**	**.00472**	**.00473**	**.00969**
		11·0·1	10·1·1	11·0·1	9·0·3
int-L1	APPR-LOGL	.00452	.00748	.00569	.01173
	int-L1	**.00147**	**.00442**	.00602	.00945
		9·2·1	9·0·3	9·0·3	9·0·3
APPR-LOGL	APPR-LOGL	-.3096	-.0180	-.0373	-.1169

Table 4: Results on varying degree of structure mismatch.

test setting	train setting	Num. of BP iterations 100	30	20	10
frac-MSE	APPR-LOGL	.00710	.00301	.00816	.02461
	frac-MSE	**.00057**	**.00072**	**.00063**	**.00064**
		12·0·0	11·0·1	12·0·0	12·0·0
int-F	APPR-LOGL	.01170	.00476	.01276	.03085
	int-F	**.00081**	**.00126**	**.00058**	**.00091**
		11·0·1	12·0·0	10·1·1	11·0·1
int-L1	APPR-LOGL	.00751	.00344	.01087	.02984
	int-L1	**.00079**	**.00101**	**.00078**	**.00096**
		10·2·0	10·0·2	10·2·0	12·0·0
APPR-LOGL	APPR-LOGL	-.3161	-.1823	-.2422	-.1104

Table 5: Results for different BP approximation quality.

As the quality of the approximation decreases, we see that back-propagation training remains quite robust, while the performance gap with APPR-LOGL widens. When using only 10 iterations, back-propagation training reduces the error by a factor of more than 30 in all testing settings. This shows that using back-propagation and ERM is very important when working with poor approximations. In general APPR-LOGL training appears to find parameters that require more iterations of BP to converge.

9 Related Work

ERM has been used with appropriate loss functions in speech recognition (Bahl et al., 1988), machine translation (Och, 2003), and energy-based models generally (LeCun et al., 2006). Our own contributions to this area included general algorithms for computing the gradient of annealed risk when dynamic programming is involved (Li and Eisner, 2009). In graphical models, methods have been proposed to directly minimize loss in tree-shaped or linear chain MRFs and CRFs (Kakade et al., 2002; Suzuki et al., 2006; Gross et al., 2007). All of these focus on exact inference. Our present paper can be seen as generalizing these methods to arbitrary graph structures, arbitrary loss functions and approximate inference.

Kulesza and Pereira (2008) show that within a fixed training method—the perceptron—substituting approximate inference may or may not allow the method to achieve low risk, depending on the inference method. In particular, loopy BP within a perceptron learner may lead to pathological results. They remark that that the empirical risk under loopy BP could be directly minimized by using grid search. Our gradient-based method is an improvement over grid search.

Guided by the intuition that the errors of approximate methods in the estimation and prediction phases may cancel one another, Wainwright (2006) provides theoretical analysis to show that it is beneficial with respect to end-to-end performance to learn the "wrong" model by using inconsistent methods for parameter estimation. This holds even in the infinite data limit.

Lacoste-Julien et al. (2011) also consider the effects of approximate inference on loss. Unlike us, they assume the parameters are given but propose an approximate inference algorithm that considers the loss function.

Simultaneous to us, Domke (2010) propose a finite-difference method that can compute the gradient of any loss that is a function of marginal inference results. His method relies on running the inference (forward) procedure three times, and, like our method, can be used with approximate inference. Compared to Domke's algorithm, our method has several advantages: it does not suffer from the numerical instabilities inherent in finite-difference methods; it does not require that inference runs to convergence; it can be used to compute gradients of additional parameters that are used in the inference algorithm (initial conditions, termination parameters, etc.). Furthermore, his algorithm imposes some additional (albeit mild) technical conditions on the choice of inference algorithm. The advantage of Domke's algorithm is that it is very easy to implement: in addition to computing the derivative of the loss function with respect to the marginals, it requires running inference two more times with perturbed parameters.

We are not aware of prior work that uses back-propagation to cope with approximate inference, approximate decoding, or arbitrary differentiable loss functions. Eaton and Ghahramani (2009) did independently apply back-propagation to BP for a quite different purpose: sensitivity analysis to find the "important" random variables in an MRF. They then conditioned the MRF on the important variables for subsequent runs of BP in the hope of finding better approximations. Their back-propagation algorithm is related to ours, but considers only the state where BP has converged, without saving intermediate messages; it could not handle our early stopping in section 8.3.

Many other learning methods have been proposed for when exact inference is intractable. Those include pseudolikelihood (Besag, 1975, 1977),[5], piecewise training (Sutton and McCallum, 2005), and many variational approaches. These training methods focus on approximately maximizing log-likelihood. They do not take into account the loss function or the choice of approximate inference or decoding procedure, nor do they try to compensate for model error.

10 Conclusions and Future Work

We have presented a new and well-motivated training objective for graphical models. Because the objective directly minimizes the empirical risk, it is robust to approximations in modeling, inference, and decoding. We show that this in fact leads to significant and substantial practical gains across a variety of distributions, models, inference procedures, and decoding procedures when evaluated on a range of synthetic data. Separately, we have found that the method also works well on real data (Stoyanov and Eisner, in review).

To optimize the objective, we have shown how to compute its gradient using automatic differentiation (see Appendix), and how to use a second-order optimization method. We have also experimented with two methods that mitigate the local optimum problem.

This line of work opens up many opportunities. Our sequel paper (in progress) will consider extensions

- to select also a graphical model *topology* (along with parameters) that gets good results despite loopy BP inference, still using back-propagation but on a modified objective (Lee et al., 2006);
- to re-incorporate a Bayesian prior (the need for which was pointed out by Minka (2000)—the present paper eliminates any role for a prior, and does not even regularize θ);
- to handle more complex patterns of missing data at training and test time; and
- to reparameterize the system to allow convex training (while the present paper copes with many approximations, it does not yet solve the significant approximation of local optimization).

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant # 0937060 to the Computing Research Association for the CIFellows Project.

[5] Vishwanathan et al. (2006) find that our baseline (SMD + loopy BP) outperforms pseudolikelihood training.

References

L. R. Bahl, P. F. Brown, P. V. de Souza, and R. L. Mercer. A new algorithm for the estimation of hidden Markov model parameters. In *Proceedings of ICASSP*, pages 493–496, 1988.

J. Besag. Statistical analysis of non-lattice data. *The Statistician*, 24(3):179–195, 1975. ISSN 0039-0526.

J. Besag. Efficiency of pseudolikelihood estimation for simple Gaussian fields. *Biometrika*, 64(3):616–618, 1977. ISSN 0006-3444.

F. Casacuberta and C. De La Higuera. Computational complexity of problems on probabilistic grammars and transducers. In *Proceedings of the 5th International Colloquium on Grammatical Inference: Algorithms and Applications*, pages 15–24, 2000.

J. Domke. Implicit Differentiation by Perturbation. In *Advances in Neural Information Processing Systems*, 2010.

F. Eaton and Z. Ghahramani. Choosing a variable to clamp: Approximate inference using conditioned belief propagation. In *Proceedings of AISTATS*, 2009.

A. Griewank and G. Corliss, editors. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, 1991.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

S. Gross, O. Russakovsky, C. Do, and S. Batzoglou. Training conditional random fields for maximum labelwise accuracy. *Advances in Neural Information Processing Systems*, 19:529, 2007.

G.E. Hinton, S. Osindero, and Y.W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. ISSN 0899-7667.

S. Kakade, Y.W. Teh, and S.T. Roweis. An alternate objective function for Markovian fields. In *International Conference on Machine Learning*, pages 275–282, 2002.

A. Kulesza and F. Pereira. Structured learning with approximate inference. In *Advances in Neural Information Processing Systems*, 2008.

S. Lacoste-Julien, F. Huszr, and Z. Ghahramani. Approximate inference for the loss-calibrated Bayesian. In *Proceedings of AISTATS*, 2011.

Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. A tutorial on energy-based learning. In G. Bakir, T. Hofman, B. Schlkopf, A. Smola, and B. Taskar, editors, *Predicting Structured Data*. MIT Press, 2006.

S.-I. Lee, V. Ganapathi, and D. Koller. Efficient structure learning of markov networks using l_1-regularization. In *Advances in Neural Information Processing Systems*, pages 817–824, 2006.

Z. Li and J. Eisner. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proc. of EMNLP*, pages 40–51, 2009.

T. Minka. Empirical risk minimization is an incomplete inductive principle. MIT Media Lab note, August 2000.

J. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, Aug 2010.

F. Och. Minimum error rate training in statistical machine translation. In *Proceedings of ACL*, 2003.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988. ISBN 1558604790.

B. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–160, 1994.

N.N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *Proceedings of the Ninth International Conference on Artificial Neural Networks*, volume 2, pages 569–574, 1999.

Veselin Stoyanov and Jason Eisner. Minimum-risk training of approximate CRF-based NLP systems, in review.

C. Sutton and A. McCallum. Piecewise training of undirected models. In *21st Conference on Uncertainty in Artificial Intelligence*, 2005.

J. Suzuki, E. McDermott, and H. Isozaki. Training conditional random fields with multivariate evaluation measures. In *Proceedings of COLING/ACL*, pages 217–224, 2006.

S.V.N. Vishwanathan, N.N. Schraudolph, M.W. Schmidt, and K.P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 969–976. ACM, 2006.

M. J. Wainwright. Estimating the "wrong" graphical model: Benefits in the computation-limited setting. *Journal of Machine Learning Research*, 7: 1829–1859, September 2006.

R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. ISSN 0899-7667.

J. Yedidia, W. Freeman, and Y. Weiss. Bethe free energy, Kikuchi approximations and belief propagation algorithms. Technical Report TR2001-16, Mitsubishi Electric Research Laboratories, 2000.

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):

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

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

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

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):

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

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

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:

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

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

---

unnormalized message $\tilde{\mu}_{\alpha\to i}^{(t-1)}$ is also needed. There are two possible solutions: to save the unnormalized version of the message $\tilde{\mu}_{\alpha\to i}^{(t-1)}$ 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 $\mu_{\alpha\to i}^{(t-1)}$, it is trivial to reconstruct the values for $\tilde{\mu}_{\alpha\to i}^{(t-1)}$ 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.