Learning in Deep Factor Graphs with Gaussian Belief Propagation

Seth Nabarro [1] **Mark van der Wilk** [2] **Andrew J. Davison** [1]

Abstract

We propose an approach to do learning in Gaussian factor graphs. We treat all relevant quantities (inputs, outputs, parameters, activations) as random variables in a graphical model, and view training and prediction as inference problems with different observed nodes. Our experiments show that these problems can be efficiently solved with belief propagation (BP), whose updates are inherently local, presenting exciting opportunities for distributed and asynchronous training. Our approach can be scaled to deep networks and provides a natural means to do continual learning: use the BP-estimated posterior of the current task as a prior for the next. On a video denoising task we demonstrate the benefit of learnable parameters over a classical factor graph approach and we show encouraging performance of deep factor graphs for continual image classification.

1. Introduction

Deep learning (DL) has been transformative across many domains. However, its applicability is limited in cases where i) we require efficient, robust representations which can be trained incrementally; ii) supervision is sparse or irregular; and iii) learning must augment, or run alongside, hand-designed solvers. Pretrained models, when available, might mitigate these limitations, but struggle as train and test distributions diverge. How they should be updated online remains an open research question.

Concurrently, we reflect on how neural networks (NNs) are trained. Despite backpropagation (Rumelhart et al., 1985) being largely successful for DL, training NNs spread over multiple processors is made difficult by *backward locking*: processors for earlier layers sit idle after their part in the forward pass, awaiting the backward error signal. This challenge will become more pertinent with the growth of i) larger models which must be distributed over many devices, ii) new hardware architectures whose cores have significant local memory (Graphcore; Cerebras), and iii) parallel, distributed and heterogeneous embedded devices (Sutter, 2011). We thus expect a growing need for more flexible training algorithms which admit efficient model-parallelism.

We argue the above challenges essentially relate to the fusion of multiple signals: old and new (for incremental learning); hand-crafted vs learnt; and representations between different layers of a model (distributed training). Bayesian principles offer a clear answer on how to fuse: signals should be combined according to the rules of probability. We seek to exploit this fact in our probabilistic approach to DL. Specifically, our models are factor graphs (Fig. 1) with random variables for all quantities relevant to DL: inputs, outputs, activations and parameters. This representation enables continual learning via online updating of the parameter posterior, and interoperability through connections with other factor graphs. We seek to design the models so as to retain the properties which we believe make DL powerful. Namely random initialisation and over-parameterisation (Allen-Zhu et al., 2019); architectural motifs encoding good inductive biases; and non-linearities which switch to selectively activate and prune when exposed to data (Glorot et al., 2011).

Much recent work has shown that Gaussian BP (GBP) to be a robust and effective algorithm for distributed inference in factor graphs, even in the presence of non-linear and non-Gaussian factors (Davison & Ortiz, 2019; Ortiz et al., 2020; Murai et al., 2022; Patwardhan et al., 2022). It is thus our choice of inference engine here. By approximating the factors in our model as Gaussian, we can use GBP for training (inference over parameters, given observations) and prediction (inference of e.g. outputs, given parameters and inputs). The generality of GBP provides flexibility as to which variables are observed, meaning training and prediction are essentially the same computation, and partial observations or missing labels do not require fundamentally different treatment. As BP is inherently local and stateful, our training can be distributed and asynchronous.

Our work shares similar goals to Lucibello et al., (2022), who train MLP-like factor graphs using GBP with analytically derived message updates. For computational efficiency,

[1]Dyson Robotics Lab, Imperial College London, UK [2]Department of Computer Science, University of Oxford, UK. Correspondence to: <sdn09@ic.ac.uk>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

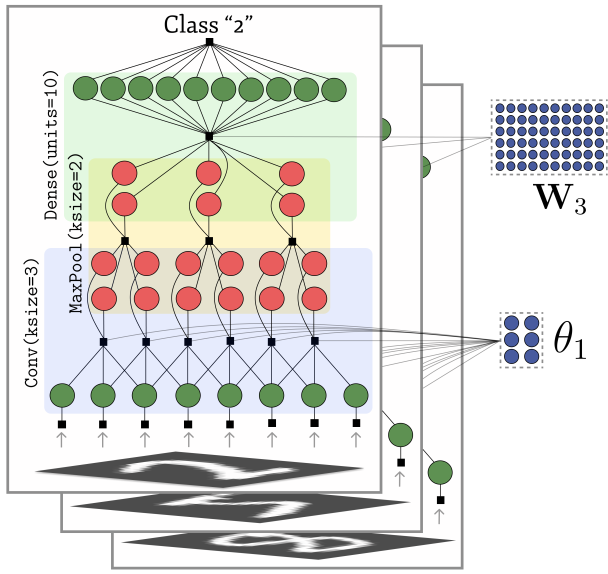


Figure 1: In **GBP Learning**, we design factor graphs whose structure mirrors common NN architectures, enabling distributed training and prediction with GBP. Learnable parameters are included as random variables (circles), as are inputs, outputs and activations. The parameters are shared over across all observations, where the other variables are copied once per observation. Factors (black squares) between layers constrain their representations to be locally consistent, while those attached to inputs and outputs encourage compatibility with observation. The inter-layer factors are non-linear to enable soft-switching behaviour. This example architecture for image classification comprises convolutional, max pooling and dense projection layers. The same architecture could be trained without supervision by removing the output observation factor.

they focus on architectures with binary weights and sign activation functions. In contrast, our approach enables training with GBP on arbitrary architectures, without rederivation of the message updates; and we focus on models with continuous weights in natural analogue to NNs. We demonstrate this generality, training convolutional architectures with our approach and showing we can outperform Lucibello et al., (2022) on image classification tasks.

We call our approach **GBP Learning**. Our models (Section 3.1) are factor graphs with architectures inspired by those in DL, and factors between layers to enforce multi-layer consistency as used in the predictive coding literature (Millidge et al., 2022). Within these models, GBP admits flexible and distributed training and prediction (Section 3.2), and learning can be done incrementally via Bayesian filtering over parameters (Section 3.4). Our experiments demonstrate the benefit of factor graphs with learnable parameters over hand-designed solvers in a video denoising task (Section 5.2). For image classification (Section 5.3), we evaluate our continual learning approach by single epoch training on MNIST. We achieve performance equivalent to an Adam-trained CNN with a replay buffer of 6×10^3 examples, and show this performance is robust to asynchronous training. Last, we compare to Lucibello et al., (2022), outperforming them on both MNIST (by 0.8%) and CIFAR10 (by 11.8%).

Our main contributions are as follows:

1. An **approach to train deep factor graphs with GBP**, which can be applied to any architecture and supports incremental learning.
2. **Experimental results** for convolutional architectures which show promise for continual image classification and video denoising.

2. Background

2.1. Factor graphs

A factor graph is a probabilistic graphical model which defines a joint distribution over variables $\mathbf{X}$ as the product of factors $\Phi = \{\phi_j\}$:

$$p(\mathbf{X}) = \frac{1}{Z} \prod_{j=1}^{|\Phi|} \phi_j\left(\mathbf{x}_{\phi_j}\right) \ . \tag{1}$$

Here $\mathbf{x}_{\phi_j}$ denotes the vector of all V_j variables in the neighbourhood of ϕ_j and Z is a normalising constant. A factor graph is bipartite: variables only connect to factors and vice versa. Each factor may encode an observation, or prior on one or many variables. Factor functions $\phi_j(\cdot)$ may be unnormalised distributions relating to their energy, $E_j(\cdot)$, as

$$\phi_j\left(\mathbf{x}_{\phi_j}\right) = \exp\left(-E_j\left(\mathbf{x}_{\phi_j}\right)\right). \tag{2}$$

A Gaussian factor graph is one in which all $\{E_j\}$ are quadratic in the related observation $\mathbf{y}_j$, i.e.

$$E_j\left(\mathbf{x}_{\phi_j}\right) = \frac{1}{2}\left(\mathbf{y}_j - \mathbf{h}\left(\mathbf{x}_{\phi_j}\right)\right)^\top \Lambda_{\mathbf{y}_j}\left(\mathbf{y}_j - \mathbf{h}\left(\mathbf{x}_{\phi_j}\right)\right) \tag{3}$$

where $\mathbf{h}(\cdot)$ is the measurement function, and $\Lambda_{\mathbf{y}_j}$ is the measurement precision. Note that $\mathbf{y}_j$ may be a "pseudo-observation", e.g. the mean of a prior on $\mathbf{x}_{\phi_j}$.

2.2. Belief Propagation

Belief Propagation (Pearl, 1988) is a message passing algorithm to perform inference on factor graphs via distributed, iterative computation. Each message is passed along the edge between one factor ϕ_j and one variable x_i. Messages travel in both directions, and we use the notations $m_{\phi_j \rightarrow x_i}$ and $m_{x_i \rightarrow \phi_j}$ for the two message types. Each message is a probability distribution in the space of the variable involved.

The following update rules are iterated until convergence:

$$m_{x_i \to \phi_j}(x_i) \leftarrow \prod_{k \in \mathrm{ne}(i) \setminus j} m_{\phi_k \to x_i}(x_i) \tag{4}$$

$$m_{\phi_j \to x_i}(x_i) \leftarrow \sum_{\{x_n\}_{\mathrm{ne}(j) \setminus i}} \phi_j\left(\mathbf{x}_{\phi_j}\right) \prod_{n \in \mathrm{ne}(j) \setminus i} m_{x_n \to \phi_i}(x_n) \ , \tag{5}$$

where $\mathrm{ne}(k) \setminus l$ denotes the indices for variables connected to factor ϕ_k, except l. After convergence the posterior marginal of a variable is estimated by the product of its incoming messages:

$$p(x_i) = \frac{1}{Z_i} \prod_{j \in \mathrm{ne}(i)} m_{\phi_j \to x_i}(x_i) \ , \tag{6}$$

where Z_i is straightforward to compute if the messages belong to a known parametric family. In this work, we assume the variable nodes are univariate, but BP can be extended for multivariate posterior inference by passing vector messages between sets of variables and factors.

BP updates (4), (5), (6) have a number of interesting properties. First, the required computations rely only on the local state of the graph; no global context is necessary. Second, for exponential family distributions under natural parameterisation, taking the product of a set of messages is reduced to the addition of their parameters.

The above routine (4), (5) is guaranteed to converge to the correct marginals in tree-structured graphs but not for graphs containing cycles. Despite a lack of guarantees, application of BP to loopy graphs has been successful in many domains, most notably for error-correcting codes (Gallager, 1962; MacKay & Neal, 1997).

2.2.1. Gaussian Belief Propagation

In Gaussian factor graphs, messages $m_{x_i \to \phi_j}(x_i)$ and $m_{\phi_j \to x_i}(x_i)$ are normal distributions, with natural parameters being the precision (inverse covariance) matrix Λ and information vector η. The products in the variable to factor message formula (4) become sums:

$$\begin{aligned} \Lambda_{x_i \to \phi_j} &\leftarrow \sum_{k \in \mathrm{ne}(i) \setminus j} \Lambda_{\phi_k \to x_i}(x_i) \ ; \\ \eta_{x_i \to \phi_j} &\leftarrow \sum_{k \in \mathrm{ne}(i) \setminus j} \eta_{\phi_k \to x_i}(x_i) \, . \end{aligned} \tag{7}$$

These updates can be implemented efficiently by computing the belief for x_i once (6) and then subtracting the incoming message from each factor to get the corresponding outgoing message. For a factor ϕ_j with precision matrix $\Lambda^{(\phi_j)} \in \mathbb{R}^{V_j \times V_j}$ and information vector $\eta^{(\phi_j)} \in \mathbb{R}^{V_j}$, the factor to variable messages are:

$$\begin{aligned} \Lambda_{\phi_j \to x_i} &\leftarrow \Lambda^{(\phi_j)}_{i,i} - \Lambda^{(\phi_j)}_{i,\setminus i} \Sigma^{(\phi_j + m)}_{\setminus i, \setminus i} \Lambda^{(\phi_j)}_{\setminus i, i} \ ; \\ \eta_{\phi_j \to x_i} &\leftarrow \eta^{(\phi_j)}_i - \Lambda^{(\phi_j)}_{i,\setminus i} \Sigma^{(\phi_j + m)}_{\setminus i, \setminus i} \eta^{(\phi_j + m)}_{\setminus i} \ , \end{aligned} \tag{8}$$

where we have used $\Sigma^{(\phi_j+m)}_{\setminus i,\setminus i} = \left(\Lambda^{(\phi_j)}_{\setminus i,\setminus i} + \left(\mathbf{D}_{\phi_j}\right)_{\setminus i,\setminus i}\right)^{-1}$, $\eta^{(\phi_j+m)} := \eta^{(\phi_j)} + \mathbf{e}_{\phi_j}$, $\mathbf{D}_{\phi_j}$ for the matrix of precision messages coming into ϕ_j, and $\mathbf{e}_{\phi_j}$ for the vector of incoming information messages. Subscript $_{\setminus r}$ indicates all elements except that for variable r. Note that $\mathbf{D}_{\phi_j}$ is diagonal for graphs with scalar variable nodes.

Convergence and correctness of GBP and similar algorithms has been studied extensively (e.g. Malioutov et al., 2006; Moallemi & Van Roy, 2009; 2010). As for general models, BP is not guaranteed to converge in Gaussian models with cycles. However, if it does converge in linear-Gaussian models, it is guaranteed to converge to the correct posterior means (Weiss & Freeman, 1999). Further, many iterative pre-conditioning schemes have been proposed which guarantee convergence in the linear-Gaussian case (Johnson et al., 2009; Ruozzi & Tatikonda, 2013). The guarantees do not hold for models with non-linear factors. We refer the reader to Ortiz et al., (2021) for an intuitive introduction to GBP.

2.3. Non-linear Factors

GBP supports the use of factors which include non-linear transformations of their connected variables, and this is crucial in our proposed use here for representation learning. Every time a message from a non-linear factor is computed, the factor is linearised around the current variable estimates, $\mathbf{X}_0$, i.e. for each factor $\mathbf{h}\left(\mathbf{x}_{\phi_j}\right) \approx \mathbf{h}\left(\mathbf{x}_{\phi_j,0}\right) + \mathbf{J}_j^\top \left(\mathbf{x}_{\phi_j} - \mathbf{x}_{\phi_j,0}\right)$ where $\mathbf{J}_j := \partial \mathbf{h}\left(\mathbf{x}_{\phi_j}\right) / \partial \mathbf{x}_{\phi_j}\big|_{\mathbf{x}_{\phi_j,0}}$. The resulting approximated factor has a quadratic (Gaussian) energy, and the following factor precision and information can be derived

$$\begin{aligned} \Lambda^{(\phi_j)} &\approx \mathbf{J}_j^\top \Lambda_{\mathbf{y}_j} \mathbf{J}_j \ ; \\ \eta^{(\phi_j)} &\approx \mathbf{J}_j^\top \Lambda_{\mathbf{y}_j} \left(\mathbf{J}_j^\top \mathbf{x}_{\phi_j,0} + \mathbf{y}_j - \mathbf{h}\left(\mathbf{x}_{\phi_j,0}\right)\right) \ . \end{aligned} \tag{9}$$

(9) can be substituted into the message update rules (8), enabling GBP inference in the linear-approximated model.

To do approximate inference in the original model, one can alternate between i) setting the linearisation point $\mathbf{X}_0$ to be the MAP estimate of the variables given current messages, and ii) updating messages given the current linearisation point. We emphasise that only non-linear factors need to be iteratively approximated in this way, and linear-Gaussian factors remain static throughout inference. We refer the reader to Section 3.3 of (Davison & Ortiz, 2019) for further details on GBP in non-linear models.

3. GBP Learning

Our aim is to produce factor graphs which have similar architectural inductive biases to NNs, that can be overparameterised in a similar way, but can be trained with GBP.

We will now describe the key factors in our model and our efficient GBP routine for training and prediction. The factor energies of form (3) contain $\mathbf{h}(\cdot)$ and $\mathbf{y}$ which, along with $\Lambda_{\mathbf{y}}$, are sufficient to deduce the linearised factor (9). This, in turn, defines the GBP message updates (7), (8). It is thus sufficient to describe our model in terms of factor energies. We emphasise that our model parameters are included as random variables in the factor graph and inferred with GBP.

3.1. Deep Factor Graphs

We design networks to find representations based on local consistency, i.e. the activations $\mathbf{x}_l \in \mathbb{R}^{D_l}$ in a layer l should "predict" those in either the previous layer $\mathbf{x}_{l-1} \in \mathbb{R}^{D_{l-1}}$ or subsequent layer $\mathbf{x}_{l+1} \in \mathbb{R}^{D_{l+1}}$, via a parametric non-linear transformation $\mathbf{f}(\cdot, \mathbf{\Theta}_l)$. Applying this principle in e.g. the generative direction, together with the Gaussian assumption, suggests the following form for the factor energy:

$$E(\mathbf{x}_l, \mathbf{x}_{l-1}, \mathbf{\Theta}_l) = \frac{\|\mathbf{x}_{l-1} - \mathbf{f}(\mathbf{x}_l, \mathbf{\Theta}_l)\|_2^2}{2\sigma_l^2}\ , \tag{10}$$

which is low when $\mathbf{f}(\mathbf{x}_l, \mathbf{\Theta}_l)$ matches the input[1] $\mathbf{x}_{l-1}$ using parameters $\mathbf{\Theta}_l$ and output $\mathbf{x}_l$. σ_l is the factor strength ($\Lambda_{\mathbf{y}_l} = \frac{1}{\sigma_l^2}\mathbf{I}_{D_{l-1}}$). Through choice of $\mathbf{f}(\cdot,\cdot)$ we can encode different operations and inductive biases.

CNNs, which have been successfully applied across a range of computer vision tasks, have a sparse connectivity structure which suggests efficient factor graph analogues. In our convolutional layers, a factor at spatial location (a, b) connects to i) the $K_l \times K_l$ patch of the input within its receptive field, $\mathbf{X}_{l-1}^{(a,b)} \in \mathbb{R}^{K_l \times K_l \times C_{l-1}}$, ii) the corresponding activation variable for an output channel c, $\mathbf{x}_l^{(a,b,c)} \in \mathbb{R}^{C_l}$ and iii) the parameters: filters $\theta_l^{(c)} \in \mathbb{R}^{K_l \times K_l \times C_{l-1}}$ and bias $b_l^{(c)}$ for output channel c, which are shared across the layer. The energy can be written as:

$$E_{\text{conv}}^{(a,b,c)} = \frac{\left(x_l^{(a,b,c)} - r\left(\mathbf{X}_{l-1}^{(a,b)}, \theta_l^{(c)}, b_l^{(c)}\right)\right)^2}{2\sigma_l^2}\ , \tag{11}$$

$$r\left(\mathbf{A}, \mathbf{B}, s\right) := g\left(\text{vec}\left(A\right) \cdot \text{vec}\left(B\right) + s\right)\ . \tag{12}$$

$g(\cdot)$ is an elementwise non-linear activation function, and we have removed the functional dependence of $E_{\text{conv}}^{(a,b,c)}$ on the connected variables for brevity. The total energy of the layer is found by summing $E_{\text{conv}}^{(a,b,c)}$ over spatial locations (a, b) and output channels c.

We can similarly define a transposed convolution layer. In this case, each filter is weighted by the output variable of its corresponding channel, and the weighted sum reconstructs the inputs. Note that for a stride smaller than the kernel size, each input will belong to multiple receptive fields which are summed over to give its reconstruction. For example, for a stride of one and neglecting edge effects, these contributions can be combined according to:

$$E_{\text{convT}}^{(a,b,c)} = \frac{\left(x_{l-1}^{(a,b,c)} - r\left(\mathbf{X}_l^{(a,b)}, \theta_l^{(c)}, b_l^{(c)}\right)\right)^2}{2\sigma_l^2}\ . \tag{13}$$

In this case, the parameters have dimension $\theta_l^{(c)} \in \mathbb{R}^{K_l \times K_l \times C_l}$ and $b_l^{(c)} \in \mathbb{R}$.

For image classification, we use a dense projection layer which translates the activations in the preceding layer $\mathbf{X}_{L-1} \in \mathbb{R}^{H_{L-1} \times W_{L-1} \times C_{L-1}}$ to a vector $\mathbf{x}_L \in \mathbb{R}^{C_L}$, where C_L is the number of classes. The dense factor energy is:

$$E_{\text{dense}} = \frac{\left\|\mathbf{x}_L - g\left(\mathbf{W}_L^\top \text{vec}\left(\mathbf{X}_{L-1}\right) + \mathbf{d}_L\right)\right\|_2^2}{2\sigma_L^2}\ . \tag{14}$$

where the activation function $g(\cdot)$ is included for generality, but usually chosen to be identity for a last-layer classifier.

In addition to convolutional and dense factors, we design factors akin to other common CNN layers. For example, we use max-pooling factors to reduce the spatial extent of the representation, upsampling layers to increase it and softmax observation factors for class supervision. We include the energies for these factors in App. A. Note the the set of layers described here is non-exhaustive, we leave the exploration of other layer types as future work.

These "layer" abstractions can be composed to produce deep models with similar design freedom to DL. Further, our models may be overparameterised by introducing large numbers of learnable weights and we believe that the inclusion of non-linear activation functions $g(\cdot)$ in our factor graph can aid representation learning as in DL. In particular, we note that for non-linear factors, the factor Jacobian is a function of the current variable estimates $\mathbf{J}_j = \mathbf{J}_j\left(\mathbf{x}_{\phi_j,0}\right)$, which will cause the strength of the factors (9) to vary depending on the input data. We expect this to produce a similar "soft switching" of connections as observed in non-linear NNs.

3.2. Learning and Predicting with GBP Inference

We have described the components of deep factor graph models, in which inputs, outputs, activations and parameters are random variables. As these models include non-linear factors such as (11), (13) and (14), we use the iterative linearisation scheme described in Section 2.3 to do approximate inference with GBP. We apply this inference engine to estimate posteriors over all latent variables. Note that there is no fundamental difference between training and prediction — only a difference in which variables are observed.

[1]We use "inputs" to mean the activation variables within a layer which are closer to the pixels, and "outputs" those which are further away. However, BP is bidirectional so these quantities are not equivalent to the inputs of a function.

In training, we infer a posterior over parameters given observed inputs and, where supervision is available, outputs. To then make predictions on new examples, we run GBP to predict the unobserved inputs/outputs given the observed inputs/outputs and parameters.

Our message schedule proceeds as follows unless stated otherwise. For a given batch, we initialise the graph and update messages by sweeping forward and backward through the layers: first nearest the input observations, progressing to the deepest layer and back again. We repeat this for a specified number of iterations. Within each layer, we compute all factor to variable updates in parallel, and likewise for the variable to factor messages of each variable type (inputs, outputs, parameters as applicable). In addition, we experimentally show our approach works well with layerwise asynchronous message schedules (Section 5.3.1). While we have demonstrated these schedules work, they are likely suboptimal and we leave exploration for future work. We find that applying damping (Murphy et al., 2013) and dropout to the factor to variable messages is sufficient for stable GBP.

3.3. Efficient GBP

Efficient inference is necessary for our models to be useful in practice. However, the inversion of a $(V_j - 1) \times (V_j - 1)$ matrix to compute $\Sigma^{(\phi_j + m)}_{\setminus i, \setminus i}$ in (8) has $O\left((V_j - 1)^3\right)$ complexity, bottlenecking GBP. To alleviate this, we exploit the structure of the matrix being inverted. In particular, i) for factors with observation dimension $M = \dim(\mathbf{y}) < V_j$, the precision $\Lambda^{(\phi_j)}$ (9) is low-rank, and ii) for graphs with scalar variable nodes, $\mathbf{D}$ is diagonal. Thus the sum $\Lambda^{(\phi_j)} + \mathbf{D}_{\phi_j}$ may be efficiently inverted via the Woodbury identity (Woodbury, 1950). Further savings come from reusing intermediates when computing messages to multiple variables (9). These optimisations change the complexity of updating messages from a factor to all V variables, from $O(V\left(V-1\right)^3)$ to $O\left(VM^3\right)$. Space complexity is changed from $O(V^2)$ to $O(VM + M^2)$. See App. B for details.

These results constitute significant savings when $M << V$, raising the question of typical values of observation dimension M. For feedforward factors such as (11) and (14), M is equal to the number of output variables connected to the factor. Thus for layers with many outputs, the cubic complexity of the factor to variable update with M may be prohibitive. However, we note that such factors can be decomposed along the output dimension. For example, a dense factor with energy (14) may be decomposed into M smaller factors, one per output variable. As the energy of the original dense factor is recovered by summing the decomposed factor energies, $E_{\text{dense}} = \sum_j E_{\text{dense,j}}$, the model is remains unchanged. However, each message update is M^2 more efficient because one factor with M outputs has been replaced by M factors each with one output.

We now consider how the factor to variable message update complexity translates to the complexity of updating *all* the messages in a model. As an illustrative example, we consider a model comprising L dense layers, each with C input units and C output units. As described, we can decompose the factor between each pair of layers into C smaller factors, each with $M = 1$ and connected to $2 \cdot (C + 1)$ variables. The complexity of factor to variable message updates in a layer is therefore $O(C^2)$, the same as the variable to factor message updates. In the non-linear case, we must also account for the computation of the factor Jacobian $\mathbf{J}$ and information vector η each time the factor is relinearised. This requires the matrix-vector product $\mathbf{W}_L^\top \text{vec}\left(\mathbf{X}_{L-1}\right)$, which is also complexity $O(C^2)$. We thus conclude that the overall complexity for updating all messages in a model of L such layers, with a batch size of B, is $O(BLC^2)$. This is the same complexity as a forward or backwards pass of backpropagation in the equivalent MLP, and the same as that for the approach of Lucibello et al., (2022).

3.4. Continual Learning and Minibatching

We now describe how we can do continual learning of model parameters with Bayesian filtering. For generality, we use $\mathbf{\Psi} = \{\psi_l\}_{l=1}^L$ to denote the set of all parameters where ψ_l is the vector of those for layer l. After initialising parameter priors $p_{t=1}\left(\psi_{l,i}\right) \leftarrow \mathcal{N}\left(0, \sigma\right)$ we perform the following for each task t in a sequence of datasets $[\mathbf{z}_1, \ldots, \mathbf{z}_T]$:

1. construct a copy of the graph with task dataset $\mathbf{z}_t$,
2. connect a unary prior factor to each parameter variable, equal to the marginal posterior estimate from the previous task $p_t\left(\psi_{l,i}\right) \leftarrow p\left(\psi_{l,i} | \mathbf{z}_{1:t-1}\right)$,
3. run GBP training to get an estimate of the updated posterior $p(\mathbf{\Psi}) = \prod_l \prod_{i=1}^{|\psi_l|} p\left(\psi_{l,i} | \mathbf{z}_{1:t}\right)$.

This method is equivalent to doing message passing in the combined graphical model for all tasks, but where messages between tasks are only passed forward in t. The advantage however, is that datapoints can be discarded after processing, and the combined graphical model for all tasks does not have to be stored in memory. As such, we also use this routine for memory-efficient training by dividing the dataset into minibatches and treating each minibatch as a task.

4. Related Work

Our models can be viewed as probabilistic energy-based models (EBMs; LeCun et al., 2006; Du & Mordatch, 2019) whose energy functions are the sum of the quadratic energies for all factors in the graph. The benefit of using a Gaussian factor graph is a model which normalises in closed form without resorting to expensive MCMC sampling. While the quadratic energy may seem constraining, we add capacity with the introduction of non-linear factors and overparameterisation. For most EBMs, parameters are attributes of the factors, where we include them as variables in our graph. As a result, learning and prediction are the same procedure in our approach, but two distinct stages for other EBMs.

Of the EBM family, restricted Boltzmann machines (RBMs Smolensky, 1986) are of particular relevance. Their factor graph resembles a single layer, fully connected version of our model, however RBMs are models over discrete variables which are unable to capture the statistics of continuous natural images. While exponential family generalisations exist, such as Gaussian-Bernoulli RBMs (Welling et al., 2004), scaling RBMs to multiple layers remains a challenging problem. To our knowledge there are no working examples of training stacked RBMs jointly, instead they are trained greedily, with each layer learning to reconstruct the activations of the trained and fixed previous layer (Hinton et al., 2006; Hinton & Salakhutdinov, 2006). As an artefact of this, later layers use capacity to learn artificial correlations induced by the early layers, rather than correlations present in the data. Global alignment is then found by fine tuning with either backprop (Hinton & Salakhutdinov, 2006) or wake-sleep (Hinton et al., 2006). We have found GBP Learning to work in multiple layer models without issue.

Our approach also relates to Bayesian DL (Neal, 2012) as we seek to infer a distribution over parameters of a deep network. Common methods to train Bayesian NNs are based on e.g. variational inference (Blundell et al., 2015; Gal & Ghahramani, 2016), the Laplace approximation (MacKay, 1992; Ritter et al., 2018), Hamiltonian Monte Carlo (Neal, 2012), Langevin-MCMC (Zhang et al., 2019). Each of these comes with benefits and drawbacks, but we note two distinguishing features of our method. Computationally, prior Bayesian DL methods rely on backprop, and so inherit its restrictions to distributed and asynchronous training, which do not limit GBP Learning. Further, our activations are random variables, allowing us to model (and resolve) disagreement between bottom-up and top-down signals.

The factor graphs we consider relate to multi-layer predictive coding (PC) models (Millidge et al., 2022; Rao & Ballard, 1999; Friston, 2005; Buckley et al., 2017; Alonso et al., 2022). Designed as a hierarchical model of biological neurons in the brain, multi-layer PC networks are trained by minimising layerwise-local, Gaussian prediction errors similar to our inter-layer consistency factors. Like ours, they also include likelihood factors which ensure observed input/output variables remain close to the observation value. Moreover, some works have noted the suitability of PC for layerwise-parallelism and emerging hardware platforms (Salvatori et al., 2023). Our framework provides a new method to train PC models with GBP, an alternative to the standard approach using gradient descent of a variational free energy (Millidge et al., 2022). The work of Parr et al., (2019) is an exception, and uses BP for inference in a model of biological neurons. However, they assume the model parameters are known, where we infer parameters jointly with activations. Moreover, we consider larger scale machine learning applications. Some PC works perform *active inference* (Buckley et al., 2017; Friston et al., 2009), where an agent may use a PC model to choose actions in uncertain environments. We do not consider this here, but highlight that GBP Learning could be extended for action selection in a similar manner.

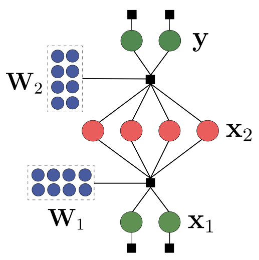
Much work has focused on augmenting BP with DL (Nachmani et al., 2016; Satorras & Welling, 2021; Yoon et al., 2019; Lázaro-Gredilla et al., 2021). In contrast, our method does DL *within* a BP framework. George et al., (2017) propose a hybrid vision system in which visual features and graph structure are learnt in a separate process to the discrete BP routine used to parse scenes (given features and graph). The work of Lázaro-Gredilla et al., (2016) is more similar to ours in that parameters are treated as variables in the graphical model and updated with BP. However, their factor graph comprises only binary variables, making it ill-suited to natural images. In addition, max-product BP is used to find a MAP solution, and not a marginal posterior estimate. This precludes incremental learning via Bayesian filtering.

Most relevant to our work is that of Lucibello et al., (2022), who use GBP to train MLPs. Their method relies on analytically derived message updates which only apply to dense architectures, and they focus on models with binary weights and sign activations. In contrast, our approach is straightforward to apply to any network structure without rederivation of messages, assuming the appropriate factor energies can be specified. Further, we focus on continuous weights in direct analogy to NN parameters. Though they similarly minibatch by filtering over parameters, they visit each datapoint multiple times and run a small number of GBP iterations during each visit. After so few iterations, the resulting messages are unlikely to constitute an accurate posterior, and ad-hoc "forgetting" factors are necessary to avoid overcounting data seen multiple times. In contrast, we train with only a single epoch, running GBP to convergence on each batch, enabling straightforward filtering.

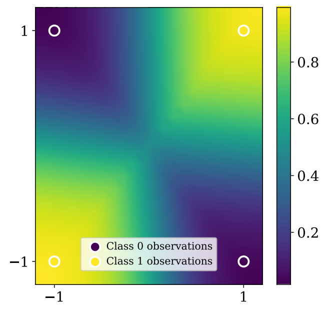
5. Results

We demonstrate our approach with three experiments: some small regression tasks, sequential video denoising and image classification. Our TensorFlow (Abadi et al., 2015) implementation is made available[2].

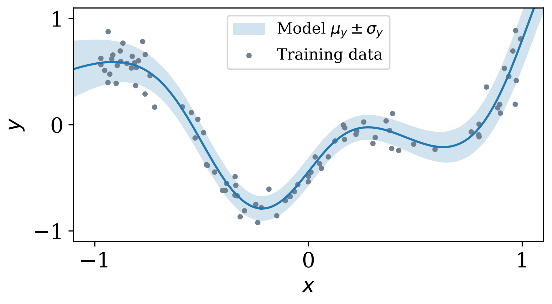
[2]github.com/sethnabarro/gbp_learning/



(a) Architecture



(b) XOR, $p(y = 1|\mathbf{x}_1, \mathbf{W}_1, \mathbf{W}_2)$



(c) Nonlinear regression

Figure 2: GBP Learning in MLP-like factor graphs (a) can solve nonlinear regression and classification tasks. (b) was generated with 8 hidden units, (c) with 16 hidden units.

5.1. Toy Experiments

We start by verifying that our model can solve tasks requiring nonlinear modelling. To this end, we run GBP Learning with single hidden layer, MLP-like factor graphs (Fig. 2a) in two settings. The first is "Exclusive-OR", which is a well-known minimal test for nonlinear modelling (Fig. 2b). The second is a nonlinear regression problem (Fig. 2c) with 90 training points. In both cases, we employ a nonlinear activation function in the first dense layer. Full details are provided in App. C. It is clear that GBP Learning can solve both tasks, confirming that the linearisation method described in Section 2.3 is sufficient to capture nonlinear dependencies.

5.2. Video Denoising

We now ask whether the learnable components in our model can improve performance over a hand-designed solver. We apply our method to the task of denoising the "bear" video from the DAVIS dataset[3] (Perazzi et al., 2016), downsampled to 258×454 with bilinear interpolation. We sample 10% of the pixels in each frame of the video, replacing their intensities with noise drawn from $U(0, 1)$. Performance is assessed by how well the denoised image matches the ground truth under the peak signal-to-noise ratio (PSNR).

[3]Creative Commons Attributions 4.0 License

The estimated pixel intensities may explained by the noisy pixel observations and/or the model reconstruction. Both the pixel observation and reconstruction factors have robust energies (see Section 5 of Davison & Ortiz, (2019)) to enable the pixel variables to "switch" between these explanations.

We evaluate two types of reconstruction model: i) Gaussian factor graphs with learnable parameters, trained with GBP Learning; and ii) a hand-designed baseline, with no learnable parameters. In the learnable models, we examine the impact of model depth by comparing: i) a single transposed convolution layer (factor energies as per (13)) with four filters, and ii) a five layer model comprising transposed convolution and upsampling layers (for details see App. D). In the baseline model, neighbouring pixels are encouraged to have similar intensities via shared smoothness factors (see e.g Ortiz et al., (2021)). We refer to this as "pairwise smoothing". Robust energies on the smoothness factors aid the preservation of edges present in the original image.

In both the pairwise smoother and GBP Learning, we do inference with GBP. Note that in our models, GBP is jointly estimating the true pixel intensities *and* the model parameters. We emphasise that our model learns parameters *from the noisy images only* and without supervision. It is incentivised to do so by the reconstruction factors in noiseless regions. To evaluate our continual learning approach, we try two routines to infer parameters: i) learn them from scratch on each frame and ii) learn them incrementally, by doing filtering on the parameters as described in Section 3.4.

The hyperparameters for all models were tuned using the first five frames of the video. Further details of the final models can be found in App. D. Denoising the entire 82-frame video with the single layer model took $\sim$ 8mins on a NVIDIA RTX 3090 GPU, and the five layer model took $\sim$ 27mins. Pairwise smoothing took $\sim$ 2mins.

The PSNR results are presented in Fig. 3 (and examples of denoised frames in Fig. D1). All methods exhibit a similar relative evolution of PSNR over the course of the video, slowly rising until the fiftieth frame and then falling again. We conjecture that this is due to varying image content, with some frames being easier to denoise than others.

The models with learnable parameters (orange and green) significantly outperform the classical baseline (blue). We attribute this to the learned models being able to capture image structure, which enables more high-frequency features to be retained while removing the corruption.

We see a clear benefit of depth, with significantly higher PSNR scores for the five layer model (orange/green dotted) over the single layer (orange/green dash). Despite the single layer model having only four filters, we find that it sometimes learns to reconstruct noise when learning continually over the video. To counteract this, we set the prior for each

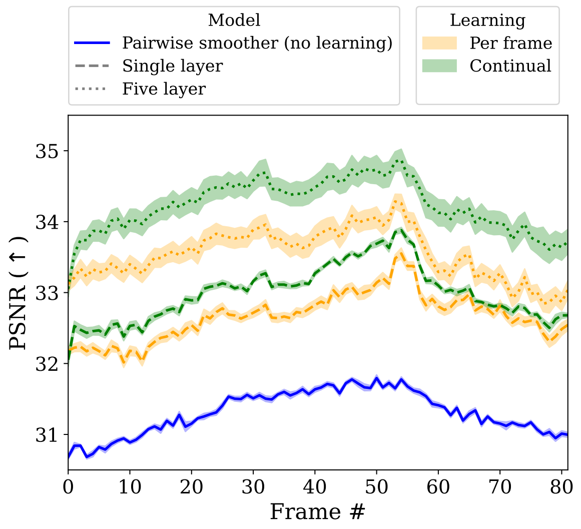


Figure 3: **Video denoising results.** Factor graphs with learnable components outperform a hand-specified pairwise smoother. Continual learning of parameters over the video further improves the PSNR over per-frame learning, and the deep model outperforms the single layer. Shading is ± 1 standard error (SE) over 10 seeds.

frame to be an interpolation between the previous posterior and the original prior (plotted; see App. D). In contrast, the deeper model does not require additional regularisation when trained continually. These results imply i) deep factor graphs have better inductive biases, capturing higher-level patterns in the data; ii) GBP Learning is can find aligned multi-layer representations with only local message updates.

In general we see that continual learning (green) leads to an improvement over learning per-frame (orange), suggesting that we can effectively fuse what has been learnt in previous frames and use it to better denoise the current frame.

5.3. Image Classification

Next, we assess our method in a supervised learning context, evaluating it on MNIST[4]. The goal here is to gauge both sample efficiency and the ability to learn continually with only one pass through the training set.

We train a convolutional factor graph with architecture similar to Fig. 1: a convolutional layer (with energy as per (11)), followed by a max-pool (15), dense layer (14) and (at train time) a class observation factor (17) on the output logit variables. Training is minibatched via the Bayesian filtering approach of Section 3.4, so the model only sees each datapoint once after which it can be discarded. To assess sample efficiency, we train on subsets of varying sizes, as well as the full training set. We compare against two baselines:

[4]yann.lecun.com/exdb/mnist/. Creative Commons Attribution-Share Alike 3.0 license.

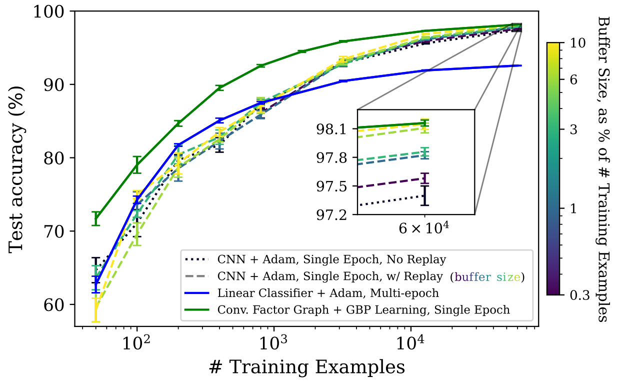


Figure 4: **Single epoch MNIST results.** GBP Learning outperforms other methods in the small data regime, and performs similarly to a CNN with a replay buffer of 6×10^3 examples on the full training set. Error bars cover $\pm$1SE over 5 seeds.

1. a linear classifier baseline which predicts logits via a dense projection of the image vector, trained with Adam over multiple epochs, and
2. the CNN equivalent of our factor graph, trained with Adam (Kingma & Ba, 2014) for a single epoch, both i) with a FIFO replay buffer (varying sizes) and ii) without replay.

The first baseline allows us to verify our method can do nonlinear learning on images. Further, varying the size of the replay buffer in the second baseline gives an indication of how much information our model can fuse via filtering. All model hyperparameters, including factor strengths σ, were tuned on validation sets generated by randomly subsampling 15% of the training set. Further details of the models and hyperparameter selection are included in App. F.

In the low data regime, GBP Learning comprehensively outperforms all baselines (Fig. 4), likely due to regularising priors and marginalising over uncertain activations. With the full training set, GBP Learning achieves a test accuracy of $98.16 \pm 0.03\%$, significantly higher than the linear classifier. Further, we outperform most CNN + Adam variants except those with large replay buffers (3600 examples/6% of training data, 6000 examples/10% training data), to which we perform similarly. We are thus encouraged that our model can learn complex relationships incrementally, consolidating new examples into its parameter posterior online.

Full dataset training with GBP Learning takes $\sim$ 3 hours on NVIDIA RTX3090 GPU. While this is much slower than the few minutes to train the CNNs on CPU, we note that the software to train NNs with backprop has benefited from decades of optimisation, as have well-established GPU hardware platforms. In contrast, our current GBP Learning implementation runs on a hardware and software stack optimised for DL. We believe orders of magnitude efficiency could be gained by developing on a more tailored setup with optimised software and hardware with on-chip memory. See Section 6 for discussion.

5.3.1. Asynchronous Training

To demonstrate the robustness to different update schedules, we also trained in a layer-wise asynchronous manner. At each iteration, we uniformly sample a sequence of $L = 4$ layers with replacement to determine the layer update order. Sampling with replacement means some layers may not be updated in an iteration, where some may be updated multiple times. Such an approach yields a final accuracy of $98.11 \pm 0.04\%$ ($\pm$1SE over 5 seeds), close to $98.16 \pm 0.03\%$ achieved by the same model trained with synchronous forward/backward sweeps (details and plot of convergence in App. F.4). This result implies that GBP Learning can work well when the model is distributed over multiple processors, without the need for global synchronisation.

5.3.2. Comparison with Lucibello et al., (2022)

To understand how our method compares to previous work, we evaluate on the same image classification benchmarks presented in Lucibello et al., (2022): MNIST, FashionMNIST and CIFAR10. Lucibello et al., (2022) only support dense models where our method works with general architectures. We thus compare the convolutional model described in Section 5.3 with the dense factor graph results from Lucibello et al., (2022). For GBP Learning, we retain the same architecture and hyperparameters used for MNIST — we do not conduct any task-specific tuning.

The results are given in Table 1. The approaches perform similarly for FashionMNIST, but our method achieves higher accuracy for both MNIST and CIFAR10. In the latter case, we outperform Lucibello et al., (2022) by more than 10%. Moreover, we obtain such performance with incremental training, passing over the training set once only, where Lucibello et al., (2022) train for 100 epochs.

We note a significant difference in the relative improvement between FashionMNIST (0%) and CIFAR10 (11.8%), which we conjecture may result from CIFAR10 benefiting more from translation equivariance of the convolutional layer. In FashionMNIST, the objects of interest are mostly centred and fill the frame, where in CIFAR10 the objects are smaller and occur in different regions of the images.

Table 1: Image classification test accuracy ($\%$). Ranges cover 1SE either side of the mean for 5 seeds.

Dataset	Lucibello et al., (2022)	Ours
MNIST	97.40 ± 0.04	$\mathbf{98.16 \pm 0.03}$
FashionMNIST	88.2 ± 0.1	88.2 ± 0.1
CIFAR10	41.3 ± 0.1	$\mathbf{53.1 \pm 0.3}$

6. Conclusion

We have introduced GBP Learning, a method for learning in Gaussian factor graphs. Parameters are included as variables in the graph and learnt using the same BP inference procedure used to estimate all other latent variables. Inter-layer factors encourage representations to be locally consistent, and multiple layers can be stacked in order to learn richer abstractions. Experimentally, we have shown a shallow, learnable model improves over a hand-crafted method on a video denoising task, with further performance gains coming from stacking multiple layers. We have also trained convolutional factor graphs for image classification with GBP Learning. On MNIST, we demonstrate encouraging sample efficiency, and reach comparable single epoch performance to a CNN with a replay buffer of 6×10^3 examples. GBP Learning outperforms Lucibello et al., (2022) by 11.8% on CIFAR10 and 0.8% on MNIST.

While we find these initial results encouraging, we highlight scaling up GBP Learning to bigger, more complex models and datasets as an exciting future direction. Our current implementation is built on software and run on hardware heavily optimised for DL. Scaling up GBP Learning will require a bespoke hardware/software system, which can better leverage the distributed nature of GBP inference. In particular, we believe processors with memory local to the compute cores (Graphcore; Cerebras) to be a promising platform for BP. Different sections of the factor graphs could be mapped to different cores. Local message updates, between factors and variables on the same core, would be cheap and could be updated at high frequency, with occasional inter-core communication to ensure alignment between different parts of the model. Low-level GBP primitives, written in e.g. Poplar[5], could be used to ensure the message updates make best use of high-bandwidth, local memory. A similar system was applied to bundle adjustment problems with GBP (Ortiz et al., 2020), and was found to be around $24\times$ faster on IPU (Graphcore) than a state-of-the-art CPU solver.

Further, this work has only considered GBP for marginal inference of scalar variables. We note that this is restrictive and prevents capturing the rich correlation structure if the energy landscape of deep networks. By including multidimensional variables in the factor graph, we would expect to mitigate this, and possibly enable more stable BP due to a reduced number of loops in the model. As higher dimensional variables incur greater computational cost per iteration, only variables which are likely to be highly correlated, e.g. the activations or weights within a layer, should be combined.

[5]graphcore.ai/products/poplar

Acknowledgements

We are grateful to many researchers for helpful discussion relating to this work. In particular, we thank the members of the Dyson Robotics Lab, especially Riku Murai, Joseph Ortiz, Ignacio Alzugaray, Talfan Evans; members of MvdW's research group, particularly Anish Dhir; and the Vicarious team at Google DeepMind.

SN and AJD are funded by EPSRC Prosperity Partnerships (EP/S036636/1) and Dyson Techonology Ltd.

Impact Statement

Computationally, our method learns via local update rules, which could enable training to be parallelised over arrays of low-power devices in place of GPUs which have high power consumption. In addition, our flexibility to different message schedules could further boost efficiency by executing intra-processor updates at high frequency and inter-processor updates less often, thus reducing communication.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems. `http://download.tensorflow.org/paper/whitepaper2015.pdf`, 2015.

Allen-Zhu, Z., Li, Y., and Liang, Y. Learning and generalization in overparameterized neural networks, going beyond two layers. *Advances in neural information processing systems*, 32, 2019.

Alonso, N., Millidge, B., Krichmar, J., and Neftci, E. O. A theoretical framework for inference learning. *Advances in Neural Information Processing Systems*, 35:37335–37348, 2022.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. Weight uncertainty in neural network. In *International conference on machine learning*, pp. 1613–1622. PMLR, 2015.

Buckley, C. L., Kim, C. S., McGregor, S., and Seth, A. K. The free energy principle for action and perception: A mathematical review. *Journal of Mathematical Psychology*, 81:55–79, 2017.

Cerebras. Cerebras. URL https://www.cerebras.net/.

Davison, A. J. and Ortiz, J. FutureMapping 2: Gaussian Belief Propagation for Spatial AI. *arXiv preprint arXiv:1910.14139*, 2019.

Du, Y. and Mordatch, I. Implicit generation and modeling with energy based models. *Advances in Neural Information Processing Systems*, 32, 2019.

Friston, K. A theory of cortical responses. *Philosophical transactions of the Royal Society B: Biological sciences*, 360(1456):815–836, 2005.

Friston, K. J., Daunizeau, J., and Kiebel, S. J. Reinforcement learning or active inference? *PloS one*, 4(7):e6421, 2009.

Gal, Y. and Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pp. 1050–1059. PMLR, 2016.

Gallager, R. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

George, D., Lehrach, W., Kansky, K., Lázaro-Gredilla, M., Laan, C., Marthi, B., Lou, X., Meng, Z., Liu, Y., Wang, H., et al. A generative vision model that trains with high data efficiency and breaks text-based captchas. *Science*, 358(6368):eaag2612, 2017.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings, 2011.

Graphcore. Graphcore. URL https://www.graphcore.ai/.

Hinton, G. E. and Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *science*, 313 (5786):504–507, 2006.

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

Johnson, J. K., Bickson, D., and Dolev, D. Fixing convergence of gaussian belief propagation. In *2009 IEEE International Symposium on Information Theory*, pp. 1674–1678. IEEE, 2009.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lázaro-Gredilla, M., Liu, Y., Phoenix, D. S., and George, D. Hierarchical compositional feature learning. *arXiv preprint arXiv:1611.02252*, 2016.

Lázaro-Gredilla, M., Lehrach, W., Gothoskar, N., Zhou, G., Dedieu, A., and George, D. Query training: Learning a worse model to infer better marginals in undirected graphical models with hidden variables. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 8252–8260, 2021.

LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., and Huang, F. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.

Lucibello, C., Pittorino, F., Perugini, G., and Zecchina, R. Deep learning via message passing algorithms based on belief propagation. *Machine Learning: Science and Technology*, 3(3):035005, 2022.

MacKay, D. J. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

MacKay, D. J. and Neal, R. M. Near shannon limit performance of low density parity check codes. *Electronics letters*, 33(6):457–458, 1997.

Malioutov, D. M., Johnson, J. K., and Willsky, A. S. Walksums and belief propagation in gaussian graphical models. *The Journal of Machine Learning Research*, 7:2031–2064, 2006.

Millidge, B., Tschantz, A., and Buckley, C. L. Predictive coding approximates backprop along arbitrary computation graphs. *Neural Computation*, 34(6):1329–1368, 2022.

Moallemi, C. C. and Van Roy, B. Convergence of minsum message passing for quadratic optimization. *IEEE Transactions on Information Theory*, 55(5):2413–2423, 2009.

Moallemi, C. C. and Van Roy, B. Convergence of minsum message-passing for convex optimization. *IEEE Transactions on Information Theory*, 56(4):2041–2050, 2010.

Murai, R., Ortiz, J., Saeedi, S., Kelly, P., and Davison, A. J. A robot web for distributed many-device localisation. *arXiv preprint arXiv:2202.03314*, 2022.

Murphy, K., Weiss, Y., and Jordan, M. I. Loopy belief propagation for approximate inference: An empirical study. *arXiv preprint arXiv:1301.6725*, 2013.

Nachmani, E., Be'ery, Y., and Burshtein, D. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 341–346. IEEE, 2016.

Neal, R. M. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

Ortiz, J., Pupilli, M., Leutenegger, S., and Davison, A. J. Bundle adjustment on a graph processor. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Ortiz, J., Evans, T., and Davison, A. J. A visual introduction to gaussian belief propagation. *arXiv preprint arXiv:2107.02308*, 2021.

Parr, T., Markovic, D., Kiebel, S. J., and Friston, K. J. Neuronal message passing using mean-field, bethe, and marginal approximations. *Scientific reports*, 9(1):1889, 2019.

Patwardhan, A., Murai, R., and Davison, A. J. Distributing collaborative multi-robot planning with gaussian belief propagation. *arXiv preprint arXiv:2203.08040*, 2022.

Pearl, J. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan kaufmann, 1988.

Perazzi, F., Pont-Tuset, J., McWilliams, B., Van Gool, L., Gross, M., and Sorkine-Hornung, A. A benchmark dataset and evaluation methodology for video object segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 724–732, 2016.

Rao, R. P. and Ballard, D. H. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87, 1999.

Ritter, H., Botev, A., and Barber, D. A scalable laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning, 2018.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Ruozzi, N. and Tatikonda, S. Message-passing algorithms for quadratic minimization. *The Journal of Machine Learning Research*, 14(1):2287–2314, 2013.

Salvatori, T., Mali, A., Buckley, C. L., Lukasiewicz, T., Rao, R. P., Friston, K., and Ororbia, A. Brain-inspired computational intelligence via predictive coding. *arXiv preprint arXiv:2308.07870*, 2023.

Satorras, V. G. and Welling, M. Neural enhanced belief propagation on factor graphs. In *International Conference on Artificial Intelligence and Statistics*, pp. 685–693. PMLR, 2021.

Smolensky, P. Information processing in dynamical systems: Foundations of harmony theory. Technical report, Colorado Univ at Boulder Dept of Computer Science, 1986.

Sutter, H. Welcome to the jungle. URL https://herbsutter.com/welcome-to-the-jungle, 2011.

Tukey, J. A survey of sampling from contaminated distributions. *Contributions to Probability and Statistics.*, 1960.

Weiss, Y. and Freeman, W. Correctness of belief propagation in gaussian graphical models of arbitrary topology. *Advances in neural information processing systems*, 12, 1999.

Welling, M., Rosen-zvi, M., and Hinton, G. E. Exponential family harmoniums with an application to information retrieval. In Saul, L., weiss, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004. URL `https://proceedings.neurips.cc/paper/2004/file/0e900ad84f63618452210ab8baae0218-Paper.pdf`.

Woodbury, M. A. *Inverting modified matrices*. Department of Statistics, Princeton University, 1950.

Yoon, K., Liao, R., Xiong, Y., Zhang, L., Fetaya, E., Urtasun, R., Zemel, R., and Pitkow, X. Inference in probabilistic graphical models by graph neural networks. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pp. 868–875. IEEE, 2019.

Zhang, R., Li, C., Zhang, J., Chen, C., and Wilson, A. G. Cyclical stochastic gradient mcmc for bayesian deep learning. *arXiv preprint arXiv:1902.03932*, 2019.

A. Energy Functions for Additional Factors

In Section 3.1, we described the energies for factors connecting convolutional and transposed convolutional layers. Here, we provide the energies for additional factors we use in our models.

For an input patch from channel c, $\mathbf{X}_{l-1}^{(a,b,c)} \in \mathbb{R}^{K\times K}$, centred at (a,b), a connected max-pooling factor has energy:

$$E_{\text{maxpool}}^{(a,b,c)} = \frac{1}{2\sigma_l^2}\left(\max\left(\mathbf{X}_{l-1}^{(a,b,c)}\right) - x_l^{(a,b,c)}\right)^2 . \tag{15}$$

To increase the spatial extent of a representation from layer l to its input $l-1$, we use upsampling layers. These factors connect a single activation variable in l, $x_l^{(a,b,c)}$ to the patch within its receptive field in $l-1$, $\mathbf{X}_{l-1}^{(a,b,c)} \in \mathbb{R}^{K\times K}$. Their energy is

$$E_{\text{upsample}}^{(a,b,c)} = \frac{1}{2\sigma_l^2}\sum_{i,j=-\lfloor K/2\rfloor}^{\lfloor K/2\rfloor}\left(x_{l-1}^{(a-i,b-j,c)} - x_l^{(a,b,c)}\right)^2 . \tag{16}$$

Last, class supervision may then be incorporated by treating last layer activations $\mathbf{x}_L$ as logits and connecting them to a observation factor with energy

$$E_{\text{softmax}} = \frac{\left\|\text{softmax}\left(\mathbf{x}_L\right) - \mathbb{1}_y\right\|_2^2}{2\sigma_{\text{softmax}}^2} , \tag{17}$$

where $\mathbb{1}_y$ is a one-hot encoding of the observed class y.

B. Factor to Variable Message Update Optimisation

We aim to reduce the complexity of message update computations (8). Naiv̈e inversion of the sum of factor and message precisions $\Sigma_{\setminus i,\setminus i}^{(\phi_j+m)} = \left(\Lambda_{\setminus i,\setminus i}^{(\phi_j)} + \left(\mathbf{D}_{\phi_j}\right)_{\setminus i,\setminus i}\right)^{-1}$ has complexity $O\left(\left(V_j-1\right)^3\right)$. This can be reduced to $O\left(\left(V-1\right)M^2 + M^3\right)$ ($M := \dim \mathbf{y}$) by substituting $\Lambda^{(\phi_j)}$ for the linearised factor precision (9) and applying the Woodbury identity (Woodbury, 1950):

$$\mathbf{S}_{\setminus i} = \mathbf{D}_{\setminus i,\setminus i}^{-1} - \mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\right)^{-1}\mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1} , \tag{18}$$

where we have dropped subscripts and superscripts relating to ϕ_j for brevity.

Further efficiencies come from substituting (18) back into the factor to variable message update (8),

$$\Lambda_{\phi\to x_i} \leftarrow \Lambda_{i,i} - \Lambda_{i,\setminus i}\left(\mathbf{D}_{\setminus i,\setminus i}^{-1} - \mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\right)^{-1}\mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\right)\Lambda_{\setminus i,i} \tag{19}$$

$$\eta_{\phi\to x_i} \leftarrow \eta_i - \Lambda_{i,\setminus i}\left(\mathbf{D}_{\setminus i,\setminus i}^{-1} - \mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\right)^{-1}\mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\right)\eta_{\setminus i}^{(\phi+m)} , \tag{20}$$

and noting that $\Lambda_{i,\setminus i} = \left(\mathbf{J}_{:,i}\right)^{\top}\Lambda_{\mathbf{y}}\mathbf{J}_{:,\setminus i}$. We can then write:

$$\Lambda_{\phi\to x_i} \leftarrow \Lambda_{i,i} - \left(\mathbf{J}_{:,i}\right)^{\top}\Lambda_{\mathbf{y}}\mathbf{J}_{:,\setminus i}\left(\mathbf{D}_{\setminus i,\setminus i}^{-1} - \mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\right)^{-1}\mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\right)\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\Lambda_{\mathbf{y}}\mathbf{J}_{:,i} \tag{21}$$

$$\eta_{\phi\to x_i} \leftarrow \eta_i - \left(\mathbf{J}_{:,i}\right)^{\top}\Lambda_{\mathbf{y}}\mathbf{J}_{:,\setminus i}\left(\mathbf{D}_{\setminus i,\setminus i}^{-1} - \mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\left(\mathbf{J}_{:,\setminus i}\right)^{\top}\right)^{-1}\mathbf{J}_{:,\setminus i}\mathbf{D}_{\setminus i,\setminus i}^{-1}\right)\eta_{\setminus i}^{(\phi+m)} . \tag{22}$$

Right-multiplying the $\mathbf{J}_{:,\setminus i}$ before the parentheses, and left multiplying the $\left(\mathbf{J}_{:,\setminus i}\right)^{\top}$ after gives

$$\Lambda_{\phi\to x_i} \leftarrow \Lambda_{i,i} - \left(\mathbf{J}_{:,i}\right)^{\top}\Lambda_{\mathbf{y}}\left(\mathbf{U}_i - \mathbf{U}_i\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{U}_i\right)^{-1}\mathbf{U}_i\right)\Lambda_{\mathbf{y}}\mathbf{J}_{:,i} , \tag{23}$$

where we have defined $\mathbf{U}_i := \mathbf{J}_{:,\backslash i}\mathbf{D}^{-1}_{\backslash i,\backslash i}\left(\mathbf{J}_{:,\backslash i}\right)^\top$. For the information update, we instead right-multiply $\eta^{(\phi+m)}_{\backslash i}$

$$\eta_{\phi\to x_i} \leftarrow \eta_i - \left(\mathbf{J}_{:,i}\right)^\top \Lambda_{\mathbf{y}}\left(\mathbf{T}_i - \mathbf{U}_i\left(\Lambda_{\mathbf{y}}^{-1} - \mathbf{U}_i\right)^{-1}\mathbf{T}_i\right) \ . \tag{24}$$

where $\mathbf{T}_i := \mathbf{J}_{:,\backslash i}\mathbf{D}^{-1}_{\backslash i,\backslash i}\eta^{(\phi+m)}_{\backslash i}$. Given $\mathbf{U}_i$ and $\mathbf{T}_i$, both message updates are $O\left(M^3\right)$, i.e. independent of V. However direct computation of $\mathbf{U}_i$ or $\mathbf{T}_i$ has complexity $O\left((V-1)M^2\right)$ for each outgoing variable i ($\mathbf{D}^{-1}$ is diagonal), so the overall complexity is quadratic in V. We achieve linear complexity by exploiting $\mathbf{U}_i = \mathbf{J}\mathbf{D}^{-1}\mathbf{J}^\top - \mathbf{J}_{:,i}\mathbf{D}^{-1}_{i,i}\left(\mathbf{J}_{:,i}\right)^\top$ where $\mathbf{J}\mathbf{D}^{-1}\mathbf{J}^\top$ can be computed once for all connected variables. Similarly, we use $\mathbf{T}_i = \mathbf{J}\mathbf{D}^{-1}\eta^{(\phi+m)} - \mathbf{J}_{:,i}\mathbf{D}^{-1}_{i,i}\eta^{(\phi+m)}_i$ for the information update. This reduces the complexity for updating all outgoing messages from a factor from $O\left(V\left((V-1)M^2 + M^3\right)\right)$ to $O\left(VM^3\right)$. In addition, these optimisations require memory $O\left(VM + M^2\right)$ which, in most cases, is a saving relative to $O\left(V^2\right)$ needed to store the full factor precision.

C. Toy Experiment Details

C.1. XOR

We use the MLP-inspired factor graph architecture illustrated in Fig. 2a, with 8 units in the hidden layer and a Leaky ReLU activation in the first-layer dense factor. The full model is described in Table 2.

We run GBP in the training graph (with 4 input/output observations) for 600 iterations. We then fix the parameters, remove the softmax class observation layer and run GBP for 300 iterations on a grid of 20×20 test input points. The last layer activation variables are then treated as the logits for class predictions. At both train and test time, we use a damping factor of 0.7 applied to the factor to variable message updates from the dense factors.

Layer #	1	2	3
Layer type	Dense	Dense	Softmax
Input dim.	2	8	2
Output dim.	8	2	2
Inc. bias	✓	✓	-
Weight prior σ	3.0	3.0	-
Activation prior σ	5.0	2.0	-
Input obs. σ	0.02	-	-
Class obs. σ	-	-	0.1
Dense recon. σ	0.1	0.1	-
Activation function $g(\cdot)$	Leaky ReLU	Linear	-

Table 2: The MLP-like factor graph used for the XOR experiment.

C.2. Regression

To generate the 1D regression data, we sample the 90 input points uniformly in the interval $(-1.0, 1.0)$. For each input point x_i, the output y_i is generated according to

$$z_i = 5 \cdot \sin\left(6.7 \cdot x_i\right) + \left(10 \cdot x_i\right)^2 \cdot 0.15 + \epsilon_i \ , \tag{25}$$

$$y_i = \frac{z_i - \mathrm{mean}\left(z\right)}{2 \cdot \mathrm{std}\left(z\right)} \tag{26}$$

where

$$\epsilon_i \sim \mathcal{N}(0., 1.5 + x_i^2) \tag{27}$$

and $\mathrm{mean}\left(\cdot\right)$, $\mathrm{std}\left(\cdot\right)$ are empirical estimates over the 90 training points.

To fit the data, we use a similar MLP-inspired factor graph architecture to that illustrated in Fig. 2a. However, this architecture has only 1 input variable and 1 output variable. We use 16 units in the hidden layer and a sigmoid activation in the first-layer dense factor. The full model is described in Table 3.

We run GBP in the training graph (with 90 input/output observations) for 4000 iterations. We then fix the parameters, remove the output observation layer and run GBP for 1000 iterations on 225 uniformly spaced test input points. The last layer activation variables are then treated as the predictions for these query points. At both train and test time we use a damping factor of 0.8 and a dropout of 0.6 in the factor to variable message updates from the dense factors.

Layer #	1	2	3
Layer type	Dense	Dense	Output observation
Input dim.	1	16	1
Output dim.	16	1	1
Inc. bias	✓	✓	-
Weight prior σ	6.0	1.5	-
Activation prior σ	10.0	3.0	-
Input obs. σ	0.02	-	-
Output obs. σ	-	-	0.05
Dense recon. σ	5×10^{-3}	1×10^{-2}	-
Activation function $g(\cdot)$	Sigmoid	Linear	-

Table 3: The MLP-like factor graph used for the regression experiment.

D. Video Denoising Experiment Details

D.1. Single Layer Convolutional Factor Graph

We use the transposed convolution model described in Table D1 for both per-frame and continual learning experiments. We run for 300 GBP iterations on each frame with damping factor of 0.8 and dropout factor of 0.6 applied to the factor to variable messages. We used robust factor energies similar to the Tukey loss (Tukey, 1960): quadratic within a Mahalanobis distance of N_{rob} from the mean, and flat outside.

Layer #	1
Layer type	Transposed Conv.
Number of filters	4
Inc. bias	✓
Kernel size	3×3
Conv. recon. σ	0.1
Recon N_{rob}	1.4
Weight prior σ	0.018
Activation prior σ	0.5
Pixel obs. σ	0.2
Pixel obs N_{rob}	0.2
Activation function $g(\cdot)$	Linear

Table D1: The single layer convolutional factor graph model used for the video denoising experiments. N_{rob} is the "robust threshold": the Mahalanobis distance beyond which the factor energy is flat rather than quadratic.

D.1.1. Preventing overfitting in single-layer model

We find the single-layer convolutional model described above can overfit to the salt-and-pepper noise when the filters are learnt continually over the course of the video. We find that overfitting can be reduced by choosing the parameter prior at each frame to be an interpolation of the previous parameter posterior and the original parameter prior.

More concretely, for a parameter $\theta_{l,i}$ with original prior (before the first frame) $\mathcal{N}(\theta_{l,i}; \mu_{\theta_l}, \sigma^2_{\theta_l})$ and posterior from previous frames $\mathcal{N}(\theta_{l,i}; \mu^{(t-1)}_{l,i}, (\sigma^{(t-1)}_{l,i})^2)$ we set its prior for a frame t to be

$$p_t(\theta_{l,i}) \leftarrow \mathcal{N}\left(\theta_{l,i}; \mu_{l,i}, (\sigma_{l,i})^2\right) \ , \tag{28}$$

where

$$\mu_{l,i} = \alpha \cdot \mu_{\theta_l} + (1-\alpha) \cdot \mu^{(t-1)}_{l,i} \tag{29}$$

$$\sigma_{l,i} = \alpha \cdot \sigma_{\theta_l} + (1-\alpha) \cdot \sigma^{(t-1)}_{l,i} \ . \tag{30}$$

We used $\alpha = 0.5$ for the single-layer, continual learning video denoising experiment.

D.2. Five Layer Convolutional Factor Graph

We use the transpose convolution model described in Table D2 for both per-frame and continual learning experiments. We run for 500 GBP iterations on each frame with damping factor of 0.8 and dropout factor of 0.6 applied to the factor to variable messages. We used robust factor energies similar to the Tukey loss (Tukey, 1960): quadratic within a Mahalanobis distance of N_{rob} from the mean, and flat outside.

To increase the spatial extent of the activations, we use upsampling layers within which each output $x_l^{(a,b,c)}$ connects to a $K \times K$ patch of the input $\mathbf{X}_{l-1}^{(a,b,c)}$ via a factor with energy (16).

Layer #	1	2	3	4	5
Layer type	Transposed Conv.	Upsample	Transposed Conv.	Upsample	Transposed Conv.
Number of filters	4	-	8	-	8
Inc. bias	✓	-	✓	N/A	✓
Kernel size	3×3	2×2	3×3	2×2	3×3
Conv. recon. σ	0.12	0.03	0.07	0.03	0.07
Recon N_{rob}	2.5	-	-	-	-
Weight prior σ	0.15	-	0.3	-	0.3
Activation prior σ	0.5	-	0.5	-	0.5
Pixel obs. σ	0.2	-	-	-	-
Pixel obs N_{rob}	0.2	-	-	-	-
Activation func. $g(\cdot)$	Linear	-	Leaky ReLU	-	Leaky ReLU

Table D2: The five-layer convolutional factor graph model used for the video denoising experiments. N_{rob} is the "robust threshold": the Mahalanobis distance beyond which the factor energy is flat rather than quadratic.

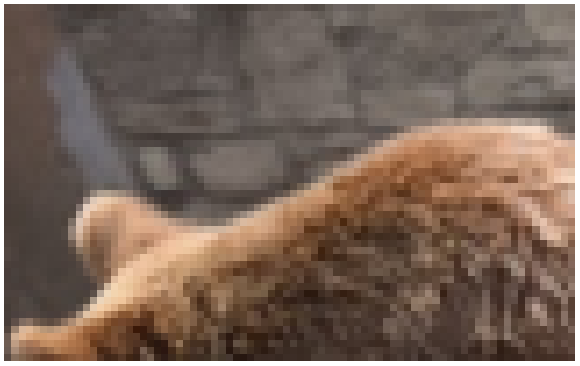
D.3. Pairwise Factor Graph

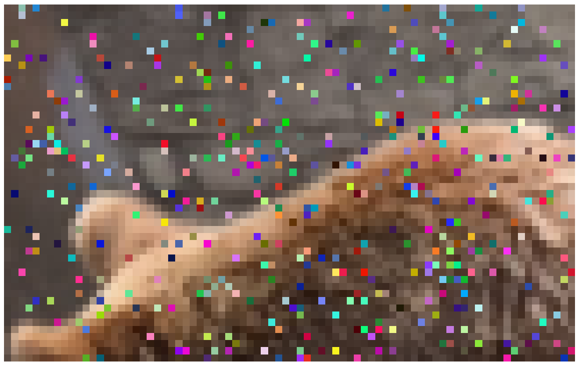
The factor hyperparameters for the pairwise smoother baseline are given in Table D3. We denoise by running 200 GBP iterations on each frame with damping factor of 0.7 applied to the factor to variable messages. We used robust factor energies similar to the Tukey loss (Tukey, 1960): quadratic with a Mahalanobis distance of N_{rob} from the mean, and flat outside.

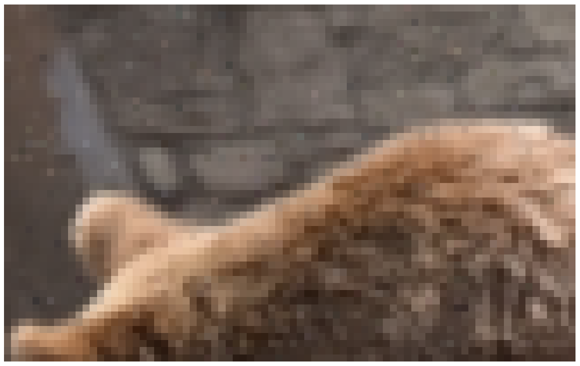
Layer #	1
Layer type	Pairwise smoothing
Pixel obs. σ	0.2
Pixel obs N_{rob}	0.14
Pairwise σ	1.3
Pairwise N_{rob}	0.35

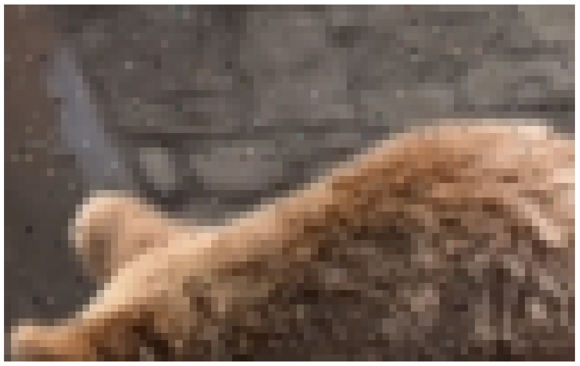
Table D3: The pairwise smoothing baseline model used for the video denoising experiments. N_{rob} is the “robust threshold”: the Mahalanobis distance beyond which the factor energy is flat rather than quadratic.

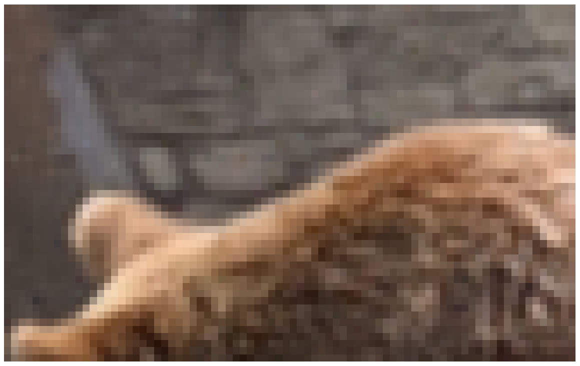
E. Denoised Video Frame Example

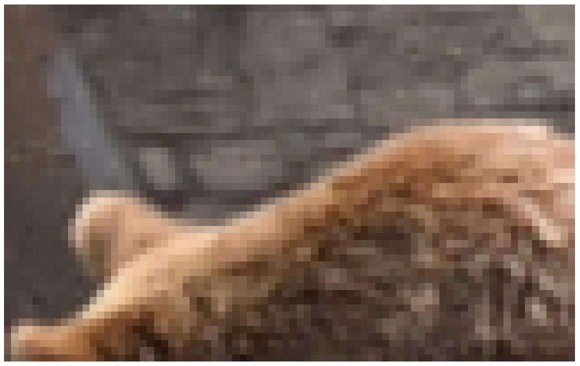

(a) Clean image


(b) Corrupted image


(c) Per-frame learning, single layer


(d) Continual learning, single layer


(e) Per-frame learning, five layer


(f) Continual learning, five layer

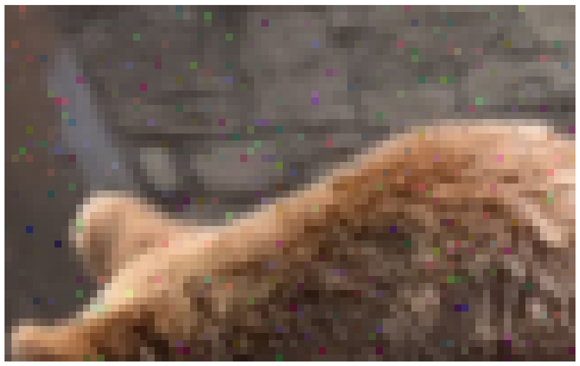
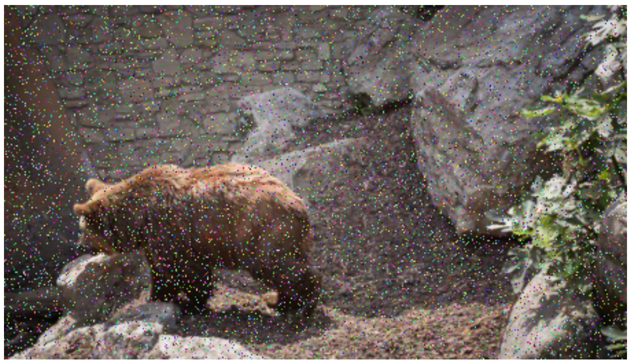

(g) Pairwise smoothing

Figure D1: **A crop from frame 5.** The learnt models are able to remove more noise while retaining more high-frequency signal.

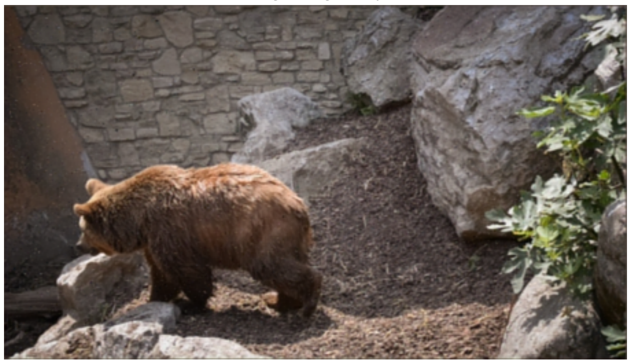
(a) Clean image


(b) Corrupted image

(c) Per-frame learning, single layer, PSNR $= 32.6$

(d) Continual learning, single layer, PSNR $= 32.9$

(e) Per-frame learning, five layer, PSNR $= 33.4$


(f) Continual learning, five layer, PSNR $= 34.3$

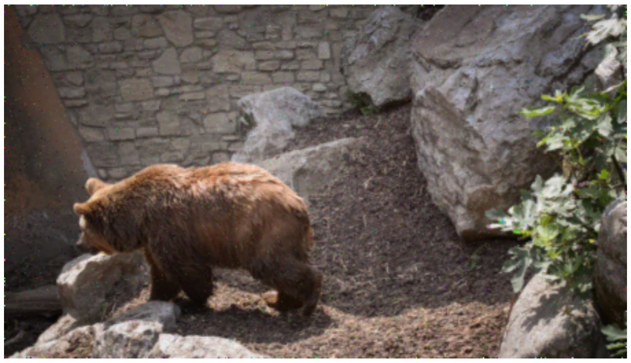

(g) Pairwise smoothing, PSNR $= 31.1$

Figure D2: **Frame 5 denoised by each method.**

F. MNIST experiment details

F.1. Convolutional Factor Graph

For the MNIST experiment we tuned the factor graph architecture and the other hyperparameters on a validation set of $9,000$ examples sampled uniformly from the training set. We then used the same tuned model for every training set size.

The final convolutional factor graph model is summarised in Table D4. Pixel variables are fixed at their observed values. To produce the results shown in Fig. 4, we train with continual learning, with a batchsize of 50 and run 500 GBP iterations on each batch. At test time, we fix the parameters and run GBP for 300 iterations per test batch of 200 examples. We apply a damping factor of 0.9 and a dropout factor of 0.5 to the factor to variable messages at both train and test time. Note that each iteration includes the updating of the messages in each layer, sweeping from image layer to classification head, and then sweeping back to image.

Layer #	1	2	3	4[†]
Layer type	Conv.	Max pool	Dense	Softmax
Num. filters	16	-	-	-
Kernel size	5×5	2×2	-	-
Dense num. inputs	-	-	2304	-
Dense num. outputs	-	-	10	-
Inc. bias	✓	-	✓	-
Weight prior σ	0.1	-	0.15	-
Activation prior σ	3.0	3.0	2.0	-
Recon. σ	0.01	0.01	0.01	-
Activation func. $g(\cdot)$	Leaky ReLU	Linear	Linear	-
Class observation σ	-	-	-	0.01

Table D4: The convolutional factor graph model used for the MNIST experiment. The dimensions of the layers are chosen such that no padding is necessary. Note that "Recon σ" denotes the strength of the factors which connect one layer to the next. [†]Softmax layer only included at training time, when class observation is available. As test time, the final state of the last layer variables after GBP inference is treated as the logit prediction.

F.2. Linear Classifier Baseline

As a baseline we used a linear classifier trained with Adam to minimise the cross-entropy loss. We tuned the step size and number of epochs on a randomly sampled validation set comprising $9,000$ training set examples. We tuned a separate linear classifier configuration for each training set size. As with the factor graph model, we used a batchsize of 50 for all experiments in Fig. 4.

F.3. CNN + Replay Buffer Baseline

We train a CNN with Adam as a comparison against our convolutional factor graph trained with GBP Learning. To make this comparison as fair as possible we

- Use a CNN architecture as close as possible to our factor graph (Table D4). The CNN architecture is summarised in Table D5.
- Train for only one epoch, but equip the CNN with a FIFO replay buffer to reduce forgetting. A fixed number of elements from each batch are randomly selected and added to the buffer. The size of the buffer necessary to match the performance of GBP Learning then provides an indication as to the efficacy of our continual learning approach. We evaluate buffers of various sizes. To allow comparison across different dataset sizes, we express buffer size as a fraction of the training set size.

We tune the following hyperparameters of the CNN + replay buffer method, finding a different configuration *for each combination of buffer size* (as fraction of training set size) *and training set size*:

- Number of elements in each batch added to the buffer

- Number of steps to take on each training set batch
- Number of steps to take on batches sampled from the replay buffer, for each training set batch
- Step size for training set batch updates
- Step size for replay buffer updates.

These hyperparameters were selected based on a validation set of $9,000$ examples, sampled uniformly from the training set. The Adam optimiser (Kingma & Ba, 2014) was used for all parameter updates.

Layer #	1	2	3
Layer type	Conv.	Max pool	Dense
Num. filters	16	-	-
Kernel size	5×5	2×2	-
Dense num. inputs	-	-	2304
Dense num. outputs	-	-	10
Inc. bias	✓	-	✓

Table D5: The baseline CNN architecture.

F.4. Asynchronous Training

To test the robustness of our method to asynchronous training regimes, we evaluated training the model in Table D4 on MNIST using random layer schedules. At each iteration, we uniformly sample 4 integers in the interval $[1, 4]$ with replacement. These integers index the different layers of the network. We then update the messages within each of the sampled layers, in the sampled order. Note that we sample with replacement, meaning that some layers may not be updated at all during a given iteration, and some may be updated multiple times.

The progression of test accuracy for both asynchronous and synchronous training is presented in Fig. D3. Both regimes exhibit similar performance throughout training, suggesting that GBP Learning can be executed in a distributed and asynchronous manner with little loss in performance.

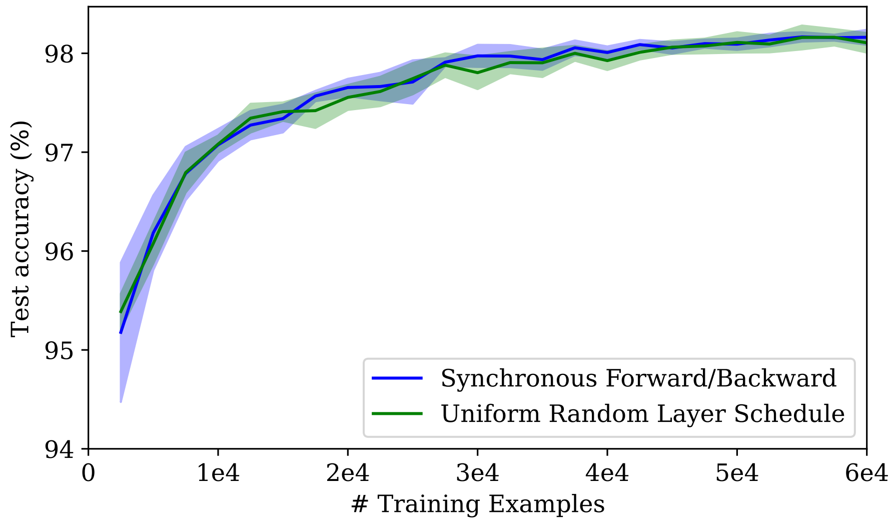


Figure D3: The evolution in test accuracy over the course of training, for a convolutional factor graph (Table D4) trained on MNIST, with different message schedules: i) synchronous forward/backward sweeps and ii) random layer ordering. Intervals represent ± 1 standard error either side of the mean, over 5 random seeds.

F.5. Dependence on number of iterations

We seek to understand how the performance of our models depends on different levels of compute at train and test time. We train replicas of the convolutional factor graph model (summarised in Table D4) on MNIST, each with a different number of GBP iterations per training batch. We then evaluate the test accuracy of each of the trained models multiple times: for differing numbers of test time GBP iterations. All configurations used a batch size of 50 at train time and 200 at test time. We ran our experiments with $50, 100, 200, 400, 800, 1600$ iterations per batch at both train time and test time.

The results are presented in Fig. D4. They show that good test time performance can be achieved with relatively few GBP iterations per training batch (~ 200), as long as a sufficient number of iterations is run at test time ($\geqslant 200$). However, the best performance is achieved with 1600 iterations per train batch.

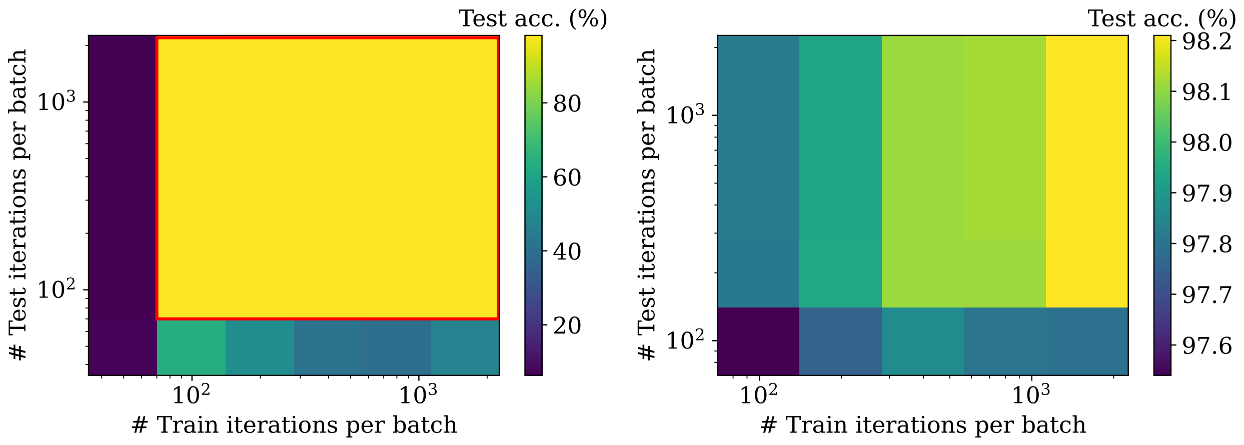


Figure D4: Dependence of MNIST test accuracy on the number of GBP iterations at train and test time. The right-hand plot covers the range of iterations marked by the red box in the left-hand plot, with the colours rescaled accordingly.