

# Contents

---

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Research Problem . . . . .	2
1.3 Outline . . . . .	4
<b>2 Preliminaries</b>	<b>5</b>
2.1 Notation . . . . .	5
2.2 Neural Networks and Bayesian Inference . . . . .	5
2.3 Factor Graphs . . . . .	6
2.4 Message Passing . . . . .	8
2.5 Approximate Message Passing . . . . .	9
2.6 Properties of Gaussians . . . . .	11
2.7 Related Work . . . . .	12
<b>3 Theoretical Model</b>	<b>15</b>
3.1 Modeling Neural Networks as a Factor Graph . . . . .	15
3.2 Factors . . . . .	16
3.2.1 Linear Algebra . . . . .	16
3.2.2 Activation Functions . . . . .	22
3.2.3 Training Signals . . . . .	26
<b>4 Implementation</b>	<b>33</b>
4.1 High-Level Overview . . . . .	33
4.2 Architecture of the Factor Graph . . . . .	35
4.2.1 Theory to Implementation . . . . .	35
4.2.2 Batched Training . . . . .	36
4.2.3 Software Architecture and Interfaces . . . . .	39
4.3 Message Equations . . . . .	42
4.3.1 Linear Algebra . . . . .	42
4.3.2 Layers . . . . .	44
4.4 Refactorings . . . . .	46
4.4.1 Numerical Stability . . . . .	46
4.4.2 Performance Improvements . . . . .	47
4.4.3 GPUs and CUDA . . . . .	49
<b>5 Results and Discussion</b>	<b>53</b>
5.1 Synthetic Datasets . . . . .	53
5.2 Evaluation on MNIST . . . . .	56

5.3	Comparison to Related Work . . . . .	62
5.4	Runtime Analysis . . . . .	64
<b>6</b>	<b>Conclusion</b>	<b>67</b>
6.1	Summary . . . . .	67
6.2	Limitations and Future Work . . . . .	68
	<b>Bibliography</b>	<b>71</b>
<b>A</b>	<b>Performance Optimization in Julia</b>	<b>77</b>
<b>B</b>	<b>Choosing the FactorGraph’s Priors</b>	<b>83</b>
<b>C</b>	<b>Experimental Setup</b>	<b>87</b>

## 1.1 Motivation

In the last decade, deep learning models have achieved state-of-the-art results in various fields such as natural language processing [Dub+24], speech recognition [Rad+22], recommender systems [CAS16], and computer vision [Rav+24]. As deep learning became more successful, its application has also extended into more sensitive fields such as medical image analysis and diagnostics [Rad19] or autonomous vehicle control [Boj+16]. In these domains, overreliance on models with no inherent ability to express uncertainty in their predictions poses significant risks. For instance, language models like GPT-4 can generate confident-sounding but incorrect responses, making it difficult for users to discern when the model’s output can be trusted and when it is hallucinating [XJK24]. Conversely, observing errors of AI systems may also cause users to stop trusting their outputs entirely, leading to underreliance. As a consequence, the combination of humans and AI has often performed worse in controlled studies than either of them alone [Zha+24].

Both overreliance and underreliance hinder the adoption of effective AI systems in fields where decisions cannot or should not be fully automated. When near-perfect performance is required, the system must either be able to delegate difficult decisions reliably or solve them sufficiently by itself. By producing overconfident decisions, reliably delegating becomes impossible and the only remaining option is to achieve near-perfect performance with the AI system alone, which can often be prohibitively difficult to attain. An example of this dynamic is autonomous driving, where self-driving systems easily manage routine tasks but can struggle with unusual situations. As long as autopilots cannot either handle all such situations or reliably return control to the human driver, their adaption will continue to face difficulties with regulation and acceptance [Ata+24].

To address these challenges, the need for transparency and explainability has been recognized in regulatory frameworks. The EU AI Act (released in 2024) requires that high-risk AI systems in safety-critical fields provide sufficient transparency so that users can interpret and use the model’s output appropriately [Int24]. Though no specific technical standard has been established, this legislation reflects the growing societal and political emphasis on the importance of model interpretability, reliability, and calibration [Chu+24]. Ensuring that neural networks can reliably indicate uncertainty is therefore not just useful but also a regulatory obligation under upcoming legislation.

In the context of machine learning, two types of uncertainty are distinguished: epistemic and aleatoric uncertainty [HW21]. Epistemic uncertainty reflects the model’s lack of knowledge about the underlying data distribution. This uncertainty can be reduced by gathering more data or by improving the model’s architecture. On the other hand, aleatoric uncertainty arises from the inherent randomness or noise in the data, such as sensor errors or measurement inaccuracies, and cannot be reduced even with additional data. For example, the predicted outcome of a coin flip is

inherently stochastic and the model could predict with high epistemic certainty that the chance of revealing tail is 50%. The aleatoric uncertainty can be expressed easily by using probabilistic predictors [HW21]. For classification, this simply means predicting a (conditional) probability distribution over classes (given the input), as is commonly done with softmax probability scores. For regression, this could mean predicting a density function over  $\mathbb{R}$ , for example by predicting the mean and variance of a normal distribution (over continuous outputs). While these methods are often inaccurate [HW21], they allow the straightforward consideration of aleatoric uncertainty. It is much more difficult to quantify epistemic uncertainty, which is why we focus on this task in this project.

In traditional deep learning, the goal is to find one set of weights  $\hat{W}$  that minimizes some loss function. In contrast, Bayesian neural networks (BNNs) offer a principled approach to capture epistemic uncertainty by reasoning about the (posterior) distribution of weights given training data,  $p(W | D)$ . This posterior distribution directly captures the uncertainty (variance) of each network parameter. Furthermore, given a distribution over weights, the network outputs also become random variables with a distribution  $p(y_p | x_p, D)$ , the posterior predictive. This allows BNNs to express varying levels of confidence in their predictions and to better distinguish in-domain and out-of-domain samples. Furthermore, the marginalization over weights in the predictive posterior makes BNNs significantly less prone to overfitting, making them more suitable for data-constrained settings or tasks such as continual learning and active learning. By avoiding overfitting, they are also more likely to be robust against adversarial attacks [Osa+19], another necessary characteristic in safety-critical domains. Therefore, addressing the challenge of accurately quantifying epistemic uncertainty is not only an emerging regulatory requirement but a critical step towards the effective application of machine learning systems in sensitive real-world tasks.

## 1.2 Research Problem

The primary challenge in BNNs lies in computing the posterior distribution over weights,  $p(W | D)$ . For neural networks, this posterior distribution is inherently complex and impractical to represent exactly. Evaluating it requires integration over all possible parameter configurations, which quickly becomes computationally intractable for non-trivial models. Approximate methods are therefore essential for evaluating the posterior (or posterior predictive) distribution. Existing approaches largely fall into two main categories: sampling-based approaches like Markov Chain Monte Carlo (MCMC) and parametric approximations such as variational inference (VI). While MCMC methods such as Hamiltonian Monte Carlo (HMC) can produce asymptotically exact posterior samples, they are generally too inefficient for large-scale deep learning applications [KR24]. In contrast, VI has become increasingly scalable [She+24]. However, it is prone to overconfident predictions [Pap+24] and can struggle to break symmetry when multiple modes of the posterior distribution are close [Zha+18]. Additionally, mean-field approaches, commonly used in VI, are susceptible to posterior collapse [Cok+22; Kur+22], and complex hyperparameter tuning [Osa+19] adds further complexity to the practical deployment of VI in real-world settings. These challenges motivate the search for alternative approaches that can potentially address some of the shortcomings of VI while maintaining its scalability.

Message passing (MP) in factor graphs presents a promising alternative for scalable Bayesian

inference. For a proper introduction of MP and factor graphs, refer to [Chapter 4](#). The core idea behind many MP algorithms is belief propagation [\[KFL01\]](#), which integrates over variables of a joint density  $p(\mathbf{a})$  that factorizes into a product of functions  $f_i$  on subsets of the random variables  $a_1, \dots, a_n$ . The corresponding factor graph is a bipartite graph that connects factors with the variables they depend on. When this graph is acyclic, belief propagation yields an efficient recursive algorithm to compute all (true) marginals  $p(a_i)$ :

$$p(a) = \prod_{f \in N_a} m_{f \rightarrow a}(a) \quad m_{f \rightarrow a}(a) = \int f(N_f) \prod_{b \in N_f \setminus \{a\}} m_{b \rightarrow f}(b) \, d(N_f \setminus \{a\}),$$

where  $N_v$  denotes the neighborhood of vertex  $v$ . For each factor  $f$ ,  $N_f$  is the set of variables that  $f$  depends on, and for variables  $a$ ,  $N_a$  are the factors that depend on  $a$ . The variable-to-factor message  $m_{b \rightarrow f}$  is defined as the product of all other incoming messages:  $m_{b \rightarrow f}(b) = \prod_{f' \in N_b \setminus \{f\}} m_{f' \rightarrow b}(b)$ . In acyclic graphs, belief propagation yields exact marginals under the true joint density, but when the graph contains cycles (as in almost all neural networks), exact solutions are intractable. Nonetheless, MP can still be applied by approximating messages  $m_{f \rightarrow x}$  and marginals  $p(x)$  with a tractable family of distributions like Gaussians [\[Min01\]](#).

MP has several advantages over traditional methods like VI. By employing moment matching instead of minimizing the reverse KL divergence, MP avoids the mode-seeking behavior of VI [\[Min05\]](#), leading to more balanced uncertainty estimates. Additionally, MP frameworks often require fewer hyperparameters and offer modularity in modeling. Multiplying messages into a marginal product also makes MP particularly suited for distributed settings. Despite these advantages, MP methods for BNNs are underexplored, perhaps primarily due to the difficulty of deriving factor-specific message equations and implementing scalable, bug-free systems.

In this project, we explore Gaussian MP as a scalable framework for BNNs and address these challenges by deriving message equations and providing a scalable implementation in Julia. Our previous work on this topic during the 2023/24 Master's project laid the groundwork for our framework [\[Ada+24\]](#), and this project expands on that foundation. Unlike the Master's project, which was limited to one-dimensional regression with feature functions, we now demonstrate the scalability of our approach to convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs) with up to 5.6 million parameters. We summarize the key contributions of this project as follows:

1. Improved numerical stability through new message equations with better numerical properties, comprehensive edge-cases handling, and enhanced code quality.
2. A complete rewrite of the Julia code, featuring an improved architecture, CUDA support, and individually optimized message equations. End-to-end tests on MNIST showed up to 309x speed improvements.
3. Optimized prior parameter selection with an analysis of its effect on the prior predictive distribution.
4. A new batching approach to manage memory requirements during training on large datasets.
5. Derivation and implementation of new factors corresponding to convolutional, softmax, argmax, and MaxPool layers.

To the best of our knowledge, this is the first MP method to handle CNNs and avoid double-counting training data, thereby preventing overconfidence and, eventually, posterior collapse into a point estimate. By focusing on scalable, efficient message passing for Bayesian inference, this work represents a promising step toward developing a viable alternative to VI for Bayesian inference on large-scale neural networks.

## 1.3 Outline

So far, we have discussed the motivation for developing scalable BNNs to accurately quantify uncertainty in sensitive applications, and introduced MP as a promising alternative to traditional BNN methods like VI. This project aims to advance MP for BNNs by developing and testing a scalable MP framework. The remainder of this project is structured as follows:

In [Chapter 2](#), we establish our conventions for notation and terminology and introduce fundamental concepts in Bayesian inference, factor graphs, and message passing. Following that, we review related work in MCMC, VI, and MP for BNNs, providing the necessary background to contextualize our contributions.

We then begin by describing our theoretical model in [Chapter 3](#), which represents the exact predictive posterior of a neural network as a factor graph. Due to computational intractability of this model, we then derive our approximate message equations for all relevant factors, laying the groundwork for its practical application.

Next, [Chapter 4](#) presents the implementation details of our MP framework. Here we discuss how our software architecture addresses challenges related to scalability, numerical stability, memory management, and GPU acceleration. This was a significant challenge in the development of our framework, and our final architecture aligns closer with traditional deep learning frameworks than with factor graphs.

In [Chapter 5](#), we evaluate our framework across multiple datasets and discuss the experimental results. On synthetic data, we show illustrative examples of posteriors and evaluate the uncertainty calibration outside the training range. We then measure accuracy and uncertainty metrics on MNIST, comparing our results with related work on MNIST and CIFAR-10. We also perform a runtime analysis and compare our implementation to the Master’s project version and Torch.

Finally, [Chapter 6](#) provides a summary of our findings and discusses the limitations of our approach. We conclude with potential directions for future research in applying MP to large-scale BNNs.

## 2.1 Notation

We denote column vectors in bold lowercase letters, such as  $\mathbf{x}$ , with their elements represented as  $x_i$ . A term  $\mathbf{x}_i$  refers to the  $i$ -th vector in a collection. A matrix is represented by a capital letter  $X$ , while  $c \cdot I$  is a diagonal matrix with all diagonal elements set to  $c$ . The transpose of a vector or matrix is denoted by  $\mathbf{x}^\top$  or  $X^\top$ . With  $W$ , we often denote the entire set of weights of a neural network, although  $W$  may not necessarily be a single matrix.

An element  $x \sim \mathcal{N}(\mu, \sigma^2)$  is a random variable that follows a normal distribution with mean  $\mu$  and variance  $\sigma^2$ . We express the Gaussian density (pdf) as  $\mathcal{N}(x; \mu, \sigma^2)$  and its cumulative distribution function (cdf) as  $\Phi(x; \mu, \sigma^2)$ , or simply  $\phi(x) = \phi(x; 0, 1)$ . In [Section 2.6](#), we introduce the natural parametrization of Gaussian distributions  $\mathcal{G}(\tau, \rho)$  where  $\rho = \frac{1}{\sigma^2}$  and  $\tau = \frac{\mu}{\sigma^2}$ . If a normal distribution is defined with moment parameters  $\mu_x, \sigma_x^2$ , we also treat the corresponding natural parameters  $\tau_x, \rho_x$  as defined and vice-versa. When working with a random variable  $x$ , we sometimes use  $\mathbb{E}[f(x)]$  to describe the expected value of a function  $f$  over the distribution of  $x$ . Probabilities are written as  $p(x)$ , and conditional probabilities as  $p(x|y)$ . We may also use  $p(x = c)$  to denote the probability of  $x = c$  under the (discrete) distribution of  $x$ .

Finally, we categorize messages in the factor graph as forward or backward messages, depending on their direction in the underlying neural network. Since we always factorize all distributions into independent one-dimensional Gaussians, we define forward messages element-wise. For a factor  $f$  and a vector-valued  $\mathbf{a}$ , each element of the forward or backward message is defined as

$$\begin{aligned} m_{f \rightarrow a_i}(a_i) &= \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2) \text{ (Forward Message)} \\ m_{a_i \rightarrow f}(a_i) &= \mathcal{N}(a_i; \overleftarrow{\mu}_i, \overleftarrow{\sigma}_i^2) \text{ (Backward Message)} \end{aligned}$$

If  $\mathbf{a}$  is the input of a layer represented by  $f$ , then  $m_{a_i \rightarrow f}(a_i) = \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2)$  would be the forward message, and  $m_{f \rightarrow a_i}(a_i) = \mathcal{N}(a_i; \overleftarrow{\mu}_i, \overleftarrow{\sigma}_i^2)$  would be the backward message. The arrows indicate direction, while  $m_{f \rightarrow a_i}$  could denote either a forward or backward message, depending on the relationship between  $a_i$  and  $f$  (e.g., if  $a_i$  is the output of the layer).

## 2.2 Neural Networks and Bayesian Inference

For the purpose of this thesis, we define a neural network as a parametric function  $f_W : \mathbf{x} \mapsto \mathbf{y}$  that maps some input  $\mathbf{x}$  to some output  $\mathbf{y}$ . The input  $\mathbf{x}$  may take different forms, such as an image with dimensions  $d_w \times d_h \times d_c$  (width, height, and channels). In a regression problem,  $\mathbf{y}$  is a real-valued



vector, whereas in a classification problem, we expect  $\mathbf{y}$  to be a vector of class probabilities that sum to 1. We only consider supervised learning where a training dataset  $D = (X, Y)$  with inputs  $X$  and targets  $Y$  is available.

The output of  $f_W$  is determined by its weights  $W$  as well as its architecture, which consists of a stack of layers. We consider the neural network a multi-layer perceptron (MLP) if it consists of alternating linear layers and activation functions. An activation layer applies an element-wise (or vector-valued) activation function to transform inputs  $\mathbf{a}$  into activations  $\mathbf{z}$ . Common activations include ReLU, LeakyReLU, and softmax. A linear layer with weight matrix  $B$  and bias  $\mathbf{c}$  transforms some input  $\mathbf{a}$  into a pre-activation  $\mathbf{z} = \mathbf{a}^\top \cdot B + \mathbf{c}$ . This choice of  $\mathbf{a}^\top \cdot B$  over the more typical  $B^\top \cdot \mathbf{a}$  is made to optimize performance based on Julia's column-major memory layout for matrices (see [Section 4.3.1](#) for details). A neural network is considered a convolutional neural network (CNN) when it contains convolutional layers that map an input of size  $d_{w_1} \times d_{h_1} \times d_{f_1}$  to an output of size  $d_{w_2} \times d_{h_2} \times d_{f_2}$  with a trainable kernel of size  $d_{w_k} \times d_{h_k} \times d_{f_1} \times d_{f_2}$ . For more information on CNNs, see [\[KSH12; LKF10; ON15\]](#).

In deep learning, gradient-based optimization methods are used to find an optimal  $\hat{W}$  that minimizes a loss function:  $\hat{W} = \operatorname{argmin}_W l(f_W(x), D)$ . In Bayesian inference, however, we are instead interested in the following distributions:

$$\begin{aligned} p(W) & \text{ (Prior)} \\ p(Y | W, X) & \text{ (Likelihood)} \\ p(W | D) & \propto p(Y | W, X) \cdot p(W) \text{ (Posterior)} \\ p(y_p | x_p, D) & = \int_W p(y_p | x_p, W) \cdot p(W | D) \, dW \text{ (Posterior Predictive)} \end{aligned}$$

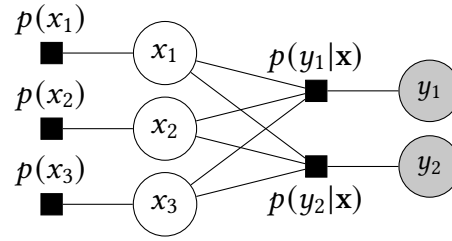
Specifically, we *choose* the prior and likelihood and aim to *evaluate* the posterior and posterior predictive. As the posterior over weights is generally intractable due to the high dimensionality of  $W$ , it is common to select an approximating distribution  $q(W)$  and minimize a divergence measure between  $q(W)$  and  $p(W | D)$ . The weights  $W$  and training inputs  $X$  are usually assumed to be independent.

## 2.3 Factor Graphs

Factor graphs are probabilistic models useful for computing marginals in factorized distributions. Let  $p$  be a joint probability distribution over variables  $\{a_1, \dots, a_n\}$  that factorizes into terms  $p(\mathbf{a}) = \prod_{i=1}^m f_i(\mathbf{a}_i)$ , where each factor  $f_i$  depends only on a subset of variables  $\mathbf{a}_i \subseteq \mathbf{a}$ . We can represent  $p$  as an undirected, bipartite graph where each term  $f_i$  and variable  $a_i$  are represented as nodes connected by an edge if  $f_i$  depends on  $a_i$ . In this factor graph, the joint probability  $p$  corresponds to the product of all factors  $f_i$ . [Figure 2.1](#) illustrates a simple factor graph showing the factorization of  $p$  into independent prior and likelihood terms.

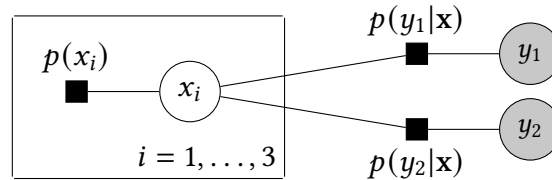
The representation of  $p$  in a factor graph is not unique; factors can be combined, and additional connections or nodes can be added depending on the modeling requirements. For example,  $y_1$  and  $y_2$  could be represented as a single vector-valued variable  $\mathbf{y}$ , or we could include parameters in





**Figure 2.1:** Factorization of  $p$  into independent prior and likelihood terms. Factor nodes are shown as black rectangles, while variables are represented as circular nodes. Constants, such as the label  $y$  of a training example, are visualized as known variables.

the prior, drawing them as known variables. For efficient inference, it is generally beneficial to work with a fully factorized version of the graph while using a simplified visual representation for clarity. Plate notation helps summarize repeated subgraphs:

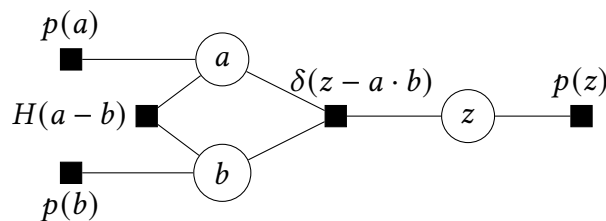


**Figure 2.2:** Plate notation summarizes three identical subgraphs for  $x_i$ .

Factors are also not limited to probability densities; they can be arbitrary functions. This flexibility allows modeling of deterministic relationships using functions like Dirac's delta  $\delta$  or step functions  $H$ :

$$H(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0. \end{cases}$$

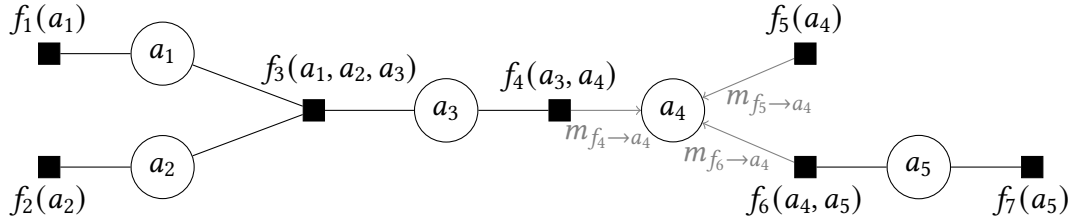
Figure 2.3 shows a factor graph that uses such deterministic constraints to enforce  $z = a \cdot b$  and  $a \geq b$ . In the following chapters, we typically use symbolic representations of factors when the underlying functions are straightforward and unambiguous.



**Figure 2.3:** The factor graph induces a joint distribution where  $P(ab = z) = 1$  and  $P(a > b) = 1$ . Each variable is also subject to a prior belief.

## 2.4 Message Passing

To perform inference in a factor graph, we are often interested in the marginal distribution of unobserved variables (e.g., weights) given observed variables (data). This requires integrating over all other unobserved variables.



**Figure 2.4:** Marginal computation in a factor graph that leverages its tree structure. The marginal of  $a_4$  is the product of messages from connected factors.

Let  $p$  be a joint probability over variables  $a_1, \dots, a_5$  that factorizes as follows:

$$p(\mathbf{a}) = f_1(a_1)f_2(a_2)f_3(a_1, a_2, a_3)f_4(a_3, a_4)f_5(a_4)f_6(a_4, a_5)f_7(a_5).$$

The corresponding factor graph is shown in Figure 2.4. Since the graph is acyclic, the marginal computation for  $p(a_4)$  can be separated into independent terms:

$$\begin{aligned} p(a_4) &= \int_{a_1} \int_{a_2} \int_{a_3} \int_{a_5} f_1(a_1)f_2(a_2)f_3(a_1, a_2, a_3)f_4(a_3, a_4)f_5(a_4)f_6(a_4, a_5)f_7(a_5) da_1 da_2 da_3 da_5 \\ &= \underbrace{\int_{a_1} \int_{a_2} \int_{a_3} f_1(a_1)f_2(a_2)f_3(a_1, a_2, a_3)f_4(a_3, a_4) da_1 da_2 da_3}_{m_{f_4 \rightarrow a_4}(a_4)} \cdot \underbrace{f_5(a_4)}_{m_{f_5 \rightarrow a_4}(a_4)} \cdot \underbrace{\int_{a_5} f_6(a_4, a_5)f_7(a_5) da_5}_{m_{f_6 \rightarrow a_4}(a_4)} \end{aligned}$$

These independent terms can be expressed as *messages*. We can further simplify the first term:

$$\begin{aligned} m_{f_4 \rightarrow a_4}(a_4) &= \int_{a_3} f_4(a_3, a_4)m_{a_3 \rightarrow f_4}(a_3) da_3, \\ m_{a_3 \rightarrow f_4}(a_3) &= m_{f_3 \rightarrow a_3}(a_3), \\ m_{f_3 \rightarrow a_3}(a_3) &= \int_{a_1} \int_{a_2} f_3(a_1, a_2, a_3)m_{a_1 \rightarrow f_3}(a_1)m_{a_2 \rightarrow f_3}(a_2) da_1 da_2. \end{aligned}$$

From this example, we derive general rules for inference in factor graphs. These message passing rules are known as the sum-product algorithm [KFL01] for acyclic factor graphs, or as belief propagation in Bayesian networks [Pea88]. In acyclic graphs, computation starts at the leaves, where  $N_a \setminus \{f\}$  or  $N_f \setminus \{a\}$  are empty. Messages at inner nodes are sent once all required incoming messages are available. We can similarly compute all marginals at once by continuing until messages have been passed in both directions along every edge.

## Inference in Factor Graphs

The three fundamental rules for message passing in factor graphs are as follows:

1. Marginals: For any variable  $a$  and its adjacent factors  $N_a \subseteq \{f_1, \dots, f_m\}$ , the marginal of  $a$  is the product of messages from  $N_a$ :

$$p(a) = \prod_{f \in N_a} m_{f \rightarrow a}(a).$$

2. Variable-to-factor messages: For a variable  $a$  and an adjacent factor  $f \in N_a$ , the message from  $a$  to  $f$  is the product of incoming messages from all other adjacent factors:

$$m_{a \rightarrow f}(a) = \prod_{f' \in N_a \setminus \{f\}} m_{f' \rightarrow a}(a).$$

3. Factor-to-variable messages: For a factor  $f$  and its adjacent variables  $N_f \subseteq \{a_1, \dots, a_n\}$ , the message from  $f$  to a variable  $a \in N_f$  integrates over all other adjacent variables:

$$m_{f \rightarrow a}(a) = \int_{N_f \setminus \{a\}} f(N_f) \cdot \prod_{a' \in N_f \setminus \{a\}} m_{a' \rightarrow f}(a') d(N_f \setminus \{a\}).$$

Therefore, for any variable  $a$  and factor  $f$ , the marginal  $p(a)$  is equal to the product of the message from  $a$  to  $f$  and the message from  $f$  to  $a$ :

$$p(a) = m_{a \rightarrow f}(a) \cdot m_{f \rightarrow a}(a).$$

This allows us to compute any of these quantities given the other two; for example, by storing  $p(a)$  and  $m_{f \rightarrow a}$  and recomputing  $m_{a \rightarrow f}$  when needed.

In graphs with loops, such as those in [Section 2.3](#), cyclic dependencies prevent the direct application of the sum-product algorithm. Instead, we initialize messages and iteratively update them as described. This iterative approach, called loopy belief propagation, can approximate the true marginals if it converges, though multiple fixed points can exist and convergence is not guaranteed [[MWJ99](#)]. This algorithm is especially relevant for most real-world applications where factor graphs often contain loops.

## 2.5 Approximate Message Passing

Implementing message passing with continuous variables is challenging, as messages can be complicated functions, and marginal distributions may have complex, multi-modal shapes. In practice, parameterized distributions are often used as approximations, including methods like variational message passing, mean-field approximation, (approximate) loopy belief propagation, and expectation propagation. Minka [[Min05](#)] provides a unified framework for these algorithms as approximate message passing.

The goal of approximate message passing is to approximate an intractable joint distribution  $p$  using a parameterized distribution  $q$  by minimizing a divergence measure  $D_\alpha(p \parallel q_\theta)$ . More details on divergence measures are available in Minka [Min05]. Two common cases are  $\alpha = 1$ , which yields the inclusive KL-divergence  $\text{KL}(p \parallel q_\theta)$ , and  $\alpha = 0$ , which gives the exclusive (reverse) KL-divergence  $\text{KL}(q_\theta \parallel p)$ . To minimize the global divergence between  $p$  and  $q$ , we choose message approximations that minimize the local divergence between the marginal under  $q$  and the corresponding marginal under  $p$ .

A common choice for  $q$  is a factorized Gaussian, where each variable is independent<sup>1</sup>. Below is a general description of the approximate message passing algorithm, following Minka [Min05].

### Approximate Message Passing with Factorized Gaussians

In this definition, the true (non-Gaussian) messages are denoted as  $m_{f \rightarrow a}$ , while Gaussian densities are represented as  $\hat{m}_{f \rightarrow a}$ . In contrast, later we will assume messages to be Gaussian unless specified otherwise. The general message passing algorithm then is as follows:

1. Initialize all factor-to-variable messages  $m_{f \rightarrow a}$ , typically using  $\mathcal{G}(0, 0)$ .
2. Repeat for a fixed number of steps (or until convergence):
  - a) Choose a variable  $a$  and an adjacent factor  $f \in N_a$ . Assume that

$$\hat{m}_{a \rightarrow f}(a) = \prod_{f' \in N_a \setminus \{f\}} \hat{m}_{f' \rightarrow a}(a) \approx m_{a \rightarrow f}(a)$$

is available as a sufficiently accurate approximation of the actual message.

- b) Approximate the updated marginal of  $a$  with a Gaussian  $g_a$  by minimizing the local divergence:

$$D_\alpha(\hat{m}_{a \rightarrow f}(a) \cdot m_{f \rightarrow a}(a) \parallel g_a).$$

- c) Compute the approximate message:

$$\hat{m}_{f \rightarrow a}(a) = \frac{g_a(a)}{\hat{m}_{a \rightarrow f}(a)} \approx m_{f \rightarrow a}(a).$$

In summary, we approximate factor-to-variable messages by minimizing the local divergence between the actual marginal and a Gaussian approximation. We then divide by the variable-to-factor message to obtain the message approximation. Our approach combines three local approximations: variational approximation, moment matching on marginals, and moment matching on messages. Variational approximation minimizes  $\text{KL}(q \parallel p)$ , while moment matching minimizes  $\text{KL}(p \parallel q)$ . For moment matching on messages, we minimize the KL divergence directly between a message  $m_{f \rightarrow v}$  and a Gaussian  $g$ , which is useful when no good marginal approximation is available, or when the actual message is almost Gaussian. Directly matching the moments of message could also be seen as marginal approximation when the incoming messages are not available yet.

<sup>1</sup> Partial factorization can also be used to model covariances within subsets of variables, such as weights within the same layer. The same algorithm applies if each subset is treated as a multidimensional variable.

Let  $s(x)$  be an actual marginal or message, and let  $m_0 = \int_x s(x) dx$  be its normalization constant. Define  $Z \sim s/m_0$  as a random variable distributed according to  $s$ . To approximate  $s/m_0$  using moment matching with a Gaussian  $q$ , we minimize the KL-divergence by setting the moments of  $q$  equal to the moments of  $s/m_0$  [Min05]. Specifically:

$$\begin{aligned}\mu_q &= \mathbb{E}[Z] \\ \sigma_q^2 &= \mathbb{V}[Z] = \mathbb{E}[Z^2] - \mathbb{E}[Z]^2.\end{aligned}$$

This method is called moment matching as it equates the moments of the approximation with the moments of the target.

## 2.6 Properties of Gaussians

Lastly, we highlight a few properties of Gaussian distributions that are essential for our approximate Gaussian message passing framework. A Gaussian density is defined as:

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

It is characterized by its mean  $\mu$  and variance  $\sigma^2$  parameters, also known as location and (squared) scale. We also use an alternative parametrization consisting of the precision  $\rho = 1/\sigma^2$  and the precision-mean  $\tau = \mu/\sigma^2$ , which we refer to as the Gaussian's natural parameters. We denote the Gaussian distribution using these parameters as  $\mathcal{G}$  and the traditional moment parameters as  $\mathcal{N}$ . For every pair  $(\mu, \sigma^2) \in \mathbb{R} \times \mathbb{R}_{>0}$ , there is a corresponding pair  $(\tau, \rho)$ . Thus, we use

$$\mathcal{G}(x; \tau, \rho) = \mathcal{N}(x; \mu, \sigma^2)$$

interchangeably, switching between moment and natural parameters without additional notice. Moment parameters are often more intuitive and useful for operations like adding Gaussian noise. On the other hand, natural parameters simplify multiplication and division of Gaussian densities:

$$\mathcal{G}(x; \tau_1, \rho_1) \cdot \mathcal{G}(x; \tau_2, \rho_2) = c \cdot \mathcal{G}(x; \tau_1 + \tau_2, \rho_1 + \rho_2).$$

The product of two Gaussian densities is itself Gaussian [Ada+24], with parameters equal to the sum of the natural parameters from the two input densities. The normalization constant  $c$  is independent of  $x$  and, while computable, is irrelevant in the context of approximate Gaussian message passing<sup>2</sup>. This result makes it straightforward to compute marginals as the product of multiple Gaussians by simply adding their natural parameters. Similarly, if  $\rho_1 > \rho_2$ , we can express the division of Gaussian densities as:

$$\mathcal{G}(x; \tau_1, \rho_1) / \mathcal{G}(x; \tau_2, \rho_2) = c \cdot \mathcal{G}(x; \tau_1 - \tau_2, \rho_1 - \rho_2).$$

<sup>2</sup> The normalization constant does not affect the parameters of the Gaussian, allowing us to discard it while still obtaining correctly normalized marginal distributions [Ada+24].

Natural parameters also allow us to represent a distribution that contains no information while remaining within the Gaussian framework, as the multiplication rule mentioned above naturally treats  $\mathcal{G}(0, 0)$  as an identity element:

$$\mathcal{G}(x; 0, 0) \cdot \mathcal{G}(x; \tau, \rho) = \mathcal{G}(x; \tau, \rho) = \mathcal{G}(x; 0, 0) \cdot \mathcal{G}(x; 0, 0).$$

We therefore use  $\mathcal{G}(0, 0)$  as the initialization for unknown messages in the approximate message passing framework.

## 2.7 Related Work

Evaluating the posterior distribution of neural networks has been a longstanding challenge in machine learning research. Since the posterior is not available in closed form for most practical problems, it can be approached either through sampling-based methods or by approximating it with parameterized distributions. While Markov chain Monte Carlo (MCMC) methods can produce asymptotically exact posterior samples, they often struggle to efficiently produce high-likelihood samples within a reasonable time frame [Cok+22]. Additionally, computing the posterior predictive from a collection of weight samples substantially increases computational demands during inference. As a result, parameterized approximations are often more practical. Among these, variational inference (VI) has gained significant traction and has been successfully applied to large-scale models [She+24]. This section therefore focuses on message passing approaches in relation to our method, with VI as the dominant alternative in the field.

### Message Passing for Neural Networks

Bayesian graphical models provide a flexible framework that can express a wide range of inference techniques. The original sum-product algorithm by Kschischang et al. [KFL01] generalized algorithms like Viterbi, "turbo" decoding, Kalman filters, and the fast Fourier transform. Subsequent work revealed that several decoding algorithms were instances of loopy belief propagation [FM97]. Following this discovery, several approximate message passing algorithms emerged, such as expectation propagation (EP) and variational message passing, which use tractable approximations for the true marginal distributions [Min01; WB05]. Minka [Min05] provides a unifying framework for these algorithms.

Expectation propagation has been shown to behave similarly to Newton updates when searching for the mode of a function [DB16], making it a promising candidate for Bayesian inference in neural networks. Soudry et al. [SHM14] introduced expectation backpropagation (EBP), which adapts message passing to multi-layer perceptrons (MLPs). EBP minimizes the reverse KL-divergence between a factorized approximation  $q$  and the true posterior  $p$  using backward propagation of gradients. The predictions in EBP are deterministic, and they are obtained either by averaging over multiple posterior samples or using maximum a posteriori (MAP) estimates. Furthermore, their experiments are limited to 3-layer MLPs with binary weights, and their approach yields inferior results when extended to continuous weights, without providing a posterior variance estimate [HA15]. Similarly, Jylänki et al. [JNV14] proposed an adaptation of EP, but their method

was restricted to 2-layer MLPs, required numerical quadrature, and failed to scale to larger models and datasets [GDY16].

Another message passing approach is probabilistic backpropagation (PBP), introduced by Hernández-Lobato and Adams [HA15]. PBP also models a factorized Gaussian posterior for deterministic neural networks  $f_W$ , updating the weight distributions through alternating forward and backward passes, similar to EBP. During the forward pass, PBP uses moment matching to approximate a distribution over network outputs, similar to our method. In the backward pass, it computes the log-likelihood gradients and uses moment matching to update beliefs. A key distinction between PBP and EBP is that PBP supports continuous weights and single-target regression, not just binary classification. However, a critical limitation of PBP is that it treats each epoch's data as new, instead of dividing out previous message updates from the marginal<sup>3</sup>. This limitation can lead to underestimated posterior variance and cause the posterior to collapse into a point estimate. Ghosh et al. [GDY16] extended both EBP and PBP to multi-class classification with continuous weights and ReLU activations, finding that PBP generally outperformed EBP with more accurate posterior approximations.

In another extension of message passing for neural networks, Lucibello et al. [Luc+22] introduced focused belief propagation (fBP). They modeled MLPs as explicit factor graphs, derived approximate message equations, and updated weight distributions through batch-wise updates with a discount factor. It is worth mentioning that, unlike the other methods discussed, fBP explicitly constructs a factor graph instead of relying solely on implicit message passing. Their experiments included MLPs that were larger than those used in EBP and PBP, and they evaluated the method on vision datasets like MNIST, FashionMNIST, and CIFAR-10. However, their approach suffered from posterior collapse due to the double counting of batch updates. Additionally, most of their experiments were limited to 3-layer MLPs with binary weights and no bias terms, which restricted the generalizability of their results.

## Variational Inference

An alternative to message passing is variational inference (VI). The goal is, again, to approximate an intractable posterior  $p(W | \mathcal{D})$  by some parameterized distribution  $q$ , referred to as the variational posterior. While message passing algorithms often optimize local divergences iteratively, VI directly minimizes a global divergence, namely the KL-divergence from the variational to the true posterior,  $\text{KL}[q(W) || p(W | \mathcal{D})]$ . Up to a constant, this is equivalent to:

$$\mathcal{L}(q) = \mathbb{E}_{W \sim q(W)} [-\log(p(Y | W, X))] + \text{KL}[q(W) || p(W)],$$

which reveals that the objective in VI is to maximize the expected (log-)likelihood of the data under the variational posterior while remaining close to the prior.

Recent derivatives like IVON [She+24] minimize a more general objective:

$$\mathcal{L}(q) = \lambda \mathbb{E}_{W \sim q(W)} [l(W)] + \text{KL}[q(W) || p(W)],$$

<sup>3</sup> In approximate message passing, the marginal of a variable should be updated by dividing out the old message before multiplying in the new one.



where  $\lambda > 0$  is a hyperparameter, typically set to the size of the dataset (or batch), representing the reliability of the data. The function  $l$  is not restricted to be the negative log-likelihood of the data but can be a more general loss function.

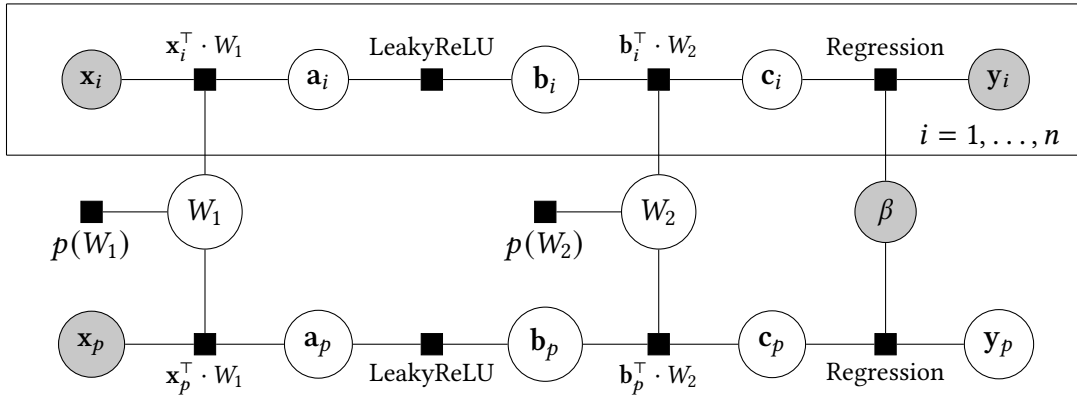
The first scalable approach for VI was presented by Graves [Gra11], which directly optimizes  $\mathcal{L}$  using stochastic gradient estimators with respect to the parameters of the variational posterior approximation  $q$ . This method was extended by Blundell et al. [Blu+15] as Bayes By Backprop, which obtains unbiased gradients by applying a reparametrization trick similar to that in Kingma and Welling [KW22]. However, these methods often require extensive hyperparameter tuning and suffer from slow convergence and severe underfitting, particularly in large models or when the dataset size is small [GYD18; Osa+19; She+24].

Further advancements in VI have been made using natural gradients, a second-order optimization technique that takes into account the curvature of the loss function [ZMG19]. Since computing these second-order terms exactly is computationally expensive, practical implementations rely on approximations. Khan et al. [Kha+18] introduced VOGN, which uses a Gauss-Newton approximation to the diagonal of the Hessian matrix. In a subsequent pioneering large-scale experiment using 128 GPUs, VOGN achieved Adam-like performance on ImageNet LSVRC, a dataset with 1.28 million training examples [Osa+19]. However, despite the improved performance, their runtime was 2-5 times longer than Adam, and the hyperparameter search for VOGN was described as a complex and challenging process that involves initially mirroring Adam's steps with a simplified version of VOGN.

A more recent improvement over VOGN, called IVON, was recently introduced by Shen et al. [She+24]. It leverages a Hessian estimator from Lin et al. [LSK20] that only requires storing mini-batch gradients instead of the gradient of each individual example. With additional techniques such as gradient clipping, they demonstrated Adam-like performance on large-scale networks, such as GPT-2, with nearly the same computational cost as Adam but improved predictive uncertainty. Despite these impressive results, the implementation still involves numerous hyperparameters. Furthermore, variational methods often yield overly confident posterior estimates [Zha+18]. Therefore, it remains valuable to explore alternative methods for approximate Bayesian inference.

### 3.1 Modeling Neural Networks as a Factor Graph

We first introduce the theoretical model underlying our approach, which forms the basis for our later implementation in [Chapter 4](#). It is important to note that our actual implementation differs significantly from this theoretical model.



**Figure 3.1:** The factor graph for a simple regression network. Each training example is modeled with its own variables and factors, while the regression layer represents the likelihood terms. A separate set of variables and factors is used for predicting  $y_p$  for an unlabeled input  $x_p$ .

Our goal is to efficiently approximate the posterior predictive distribution  $p(y_p | x_p, D)$ , which represents the distribution over outputs given a new example  $x_p$  and the training data  $D$ . Let  $f_W$  be the parametric function corresponding to the neural network architecture with weights  $W$ . The posterior predictive distribution can then be expressed as:

$$\begin{aligned}
 p(y_p | x_p, D) &= \int_W p(y_p | x_p, W) \cdot p(W | D) dW \\
 &\propto \int_W p(y_p | x_p, W) \cdot p(Y | W, X) \cdot p(W) dW \\
 &= \underbrace{\int_W p(y_p | f_W(x_p), W) dW}_{\text{Conditional Predictive}} \cdot \underbrace{p(W)}_{\text{Prior}} \cdot \underbrace{\prod_i p(y_i | f_W(x_i), W)}_{\text{Training Data Likelihood}} dW.
 \end{aligned}$$

We model  $p(y_p | x_p, D)$  directly using a factor graph. [Figure 3.1](#) illustrates the structure of the factor graph for a regression dataset. In the case of classification, the labels would be a class  $y_i \in \{1, \dots, d_y\}$  instead of a target vector  $y_i \in \mathbb{R}^{d_y}$ . The factor graph consists of the following components:

1. **Deterministic Neural Network:** We represent the deterministic function  $f_W$  with corresponding factors and hidden variables. This representation is duplicated once per training example  $\mathbf{x}_i$ , referred to as the *training branches*, and once for the predictive input  $\mathbf{x}_p$ , called the *predictive branch*. The inputs are modeled as known variables.
2. **Variables:** The neural network's parameters  $W$  are modeled as variables that connect to each training branch. They are also connected to a factor that represents their prior  $p(W)$ .
3. **Training Data Likelihood:** We assume that the training data likelihood  $p(Y | W, X)$  factorizes over individual training examples. Each likelihood term  $p(y_i | f_W(\mathbf{x}_i), W)$  is represented by a factor at the end of each training branch, and the training labels are modeled as known variables.
4. **Conditional Predictive:** Similarly, the term  $p(f_W(\mathbf{x}_p) | W)$  is represented by a factor in the predictive branch. For regression, this would be additive noise with variance  $\beta^2$ .

The marginal of  $y_p$  in our factor graph directly corresponds to the true posterior predictive distribution. However, the factor graph contains loops between the predictive branch and the training branches, leading to a dependency where the messages from  $W$  to the predictive branch are influenced by  $\mathbf{x}_p$  as well as the training data. Consequently, if  $\mathbf{x}_p$  changes, it necessitates iterating through the entire factor graph to accurately approximate the posterior predictive distribution. In neural network terms, this translates to retraining the whole network for every test input.

As this would be prohibitively expensive, we first approximate the posterior  $p(W | D)$  through approximate Gaussian message passing on the training branches of the factor graph. This results in a factorized Gaussian  $q(W) \approx p(W | D)$ , which we refer to as the marginal of  $W$ . The approximation of the predictive posterior then becomes:

$$p(y_p | \mathbf{x}_p, D) \approx \int_W p(y_p | f_W(\mathbf{x}_p), W) \cdot q(W) dW.$$

By applying approximate Gaussian message passing on the predictive branch, we obtain an approximate predictive posterior  $q(y_p) \approx p(y_p | \mathbf{x}_p, D)$ . Since the predictive branch is acyclic, we only need to run a single forward pass from  $\mathbf{x}_p$  to  $y_p$ . This approach is equivalent to moment propagation but can be implemented using the same factor approximations as those used for the training branches.

## 3.2 Factors

### 3.2.1 Linear Algebra

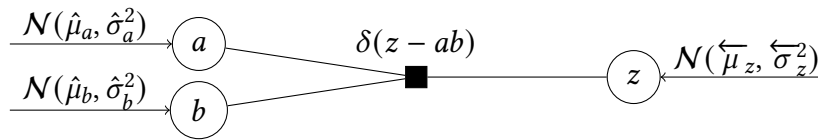
Many operations in a neural network can be expressed as linear algebra operations. By developing a library of these operations, we ensure modularity and extensibility in our approach. In this section, we introduce the following operations:

1. (1D) Product:  $a \cdot b$
2. Weighted sum:  $\mathbf{a} \cdot \mathbf{b}$  for constants  $\mathbf{a}$
3. Inner product:  $\mathbf{a} \cdot \mathbf{b}$

We then briefly outline higher-order multiplication, such as vector-matrix, matrix-matrix, and batched operations.

## Product

Let  $a$  and  $b$  be two incoming variables, such as an input and a weight variable. We want to model a factor  $f$  such that a variable  $z = a \cdot b$  is the product of  $a$  and  $b$ , as shown in Figure 3.2. Since the true message densities are bimodal, we use a (mode-seeking) variational approximation, which also requires approximate marginals as inputs (instead of forward messages) [SHG09].



**Figure 3.2:** The (one-dimensional) product of two variables  $a$  and  $b$ . We denote the parameters of approximate marginals with a hat  $\hat{\mu}$  and message parameters with a directed arrow  $\vec{\mu}$  or  $\overleftarrow{\mu}$ .

Let  $\mathcal{N}(\vec{\mu}_a, \vec{\sigma}_a^2)$  and  $\mathcal{N}(\vec{\mu}_b, \vec{\sigma}_b^2)$  be the forward messages from  $a$  and  $b$ , and let  $\mathcal{N}(\hat{\mu}_a, \hat{\sigma}_a^2)$  and  $\mathcal{N}(\hat{\mu}_b, \hat{\sigma}_b^2)$  be the approximate marginals. If no backward message is available yet, we initialize the approximate marginal with the forward message. Let  $\mathcal{N}(\overleftarrow{\mu}_z, \overleftarrow{\sigma}_z^2)$  be the backward message from  $z$ . We then compute the forward message  $m_{f \rightarrow z}$  and the backward messages  $m_{f \rightarrow a}$  and  $m_{f \rightarrow b}$ . The equations for these messages are presented in Stern et al. [SHG09]. As noted in the MP report [Ada+24], the forward message uses moment matching, while the backward message is a variational approximation.

### Forward Message (Product)

The forward message is  $m_{f \rightarrow z}(z) = \mathcal{N}(z; \vec{\mu}_z, \vec{\sigma}_z^2)$ , where

$$\begin{aligned}\vec{\mu}_z &= \hat{\mu}_a \cdot \hat{\mu}_b \\ \vec{\sigma}_z^2 &= (\hat{\sigma}_a^2 + \hat{\mu}_a^2) \cdot (\hat{\sigma}_b^2 + \hat{\mu}_b^2) - \hat{\mu}_a^2 \hat{\mu}_b^2.\end{aligned}$$

The backward message to  $b$  is defined as  $m_{f \rightarrow b}(b) = \mathcal{N}(b; \overleftarrow{\mu}_b, \overleftarrow{\sigma}_b^2)$ , where

$$\begin{aligned}\overleftarrow{\sigma}_b^2 &= \frac{(\overleftarrow{\sigma}_z^2 + \overleftarrow{\mu}_z^2) - \overleftarrow{\mu}_z^2}{\hat{\sigma}_a^2 + \hat{\mu}_a^2} \\ \overleftarrow{\mu}_b &= \frac{\overleftarrow{\mu}_z \cdot \hat{\mu}_a}{\hat{\sigma}_a^2 + \hat{\mu}_a^2}.\end{aligned}$$

All messages are stored in natural parameters, so if some message equation uses  $\sigma^2 = \frac{1}{\rho}$ , division by  $\rho$  is required. This becomes problematic when  $\rho$  approaches zero, which frequently occurs for backward messages. It is therefore preferable to compute the natural parameters of the backward message directly instead of first computing its moment parameters and then converting to natural parameters.

The backward precision  $\overleftarrow{\rho}_b$  can be computed as

$$\begin{aligned}\overleftarrow{\rho}_b &= \frac{\hat{\sigma}_a^2 + \hat{\mu}_a^2}{(\overleftarrow{\sigma}_z^2 + \overleftarrow{\mu}_z^2) - \overleftarrow{\mu}_z^2} \\ &= \overleftarrow{\rho}_z \cdot (\hat{\sigma}_a^2 + \hat{\mu}_a^2),\end{aligned}$$

and  $\overleftarrow{\tau}_b$  is computed as follows:

$$\begin{aligned}\overleftarrow{\tau}_b &= \frac{\overleftarrow{\mu}_z \cdot \hat{\mu}_a}{\hat{\sigma}_a^2 + \hat{\mu}_a^2} \cdot \overleftarrow{\rho}_z \cdot (\hat{\sigma}_a^2 + \hat{\mu}_a^2) \\ &= (\overleftarrow{\mu}_z \cdot \overleftarrow{\rho}_z) \cdot \hat{\mu}_a = \overleftarrow{\tau}_z \cdot \hat{\mu}_a.\end{aligned}$$

Computing the natural parameters directly avoids parameter conversions when storing the backward message. Additionally, the transformed message equation becomes simpler and depends only on the natural parameters of  $m_{z \rightarrow f}$ , which reduces numerical instability.

#### Backward Message (Product)

The backward message is  $m_{f \rightarrow b}(b) = \mathcal{G}(b; \overleftarrow{\tau}_b, \overleftarrow{\rho}_b)$ , where

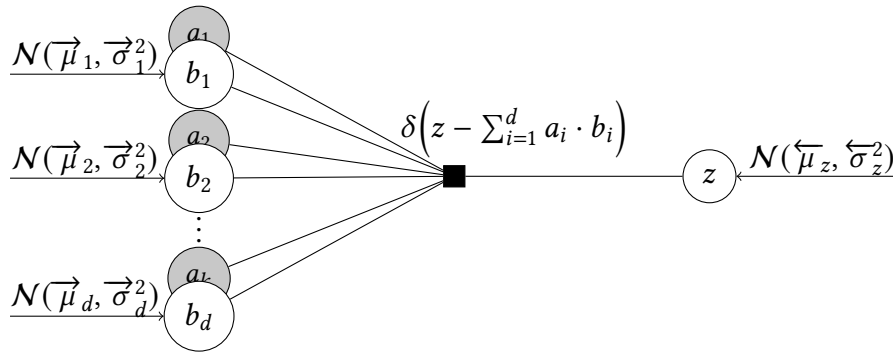
$$\begin{aligned}\overleftarrow{\tau}_b &= \overleftarrow{\tau}_z \cdot \hat{\mu}_a \\ \overleftarrow{\rho}_b &= \overleftarrow{\rho}_z \cdot (\hat{\sigma}_a^2 + \hat{\mu}_a^2).\end{aligned}$$

The message to  $a$  can be computed equivalently using  $\hat{\mu}_b$  and  $\hat{\sigma}_b^2$  instead of  $\hat{\mu}_a$  and  $\hat{\sigma}_a^2$ .

### Weighted Sum

Let  $\mathbf{a}$  be a  $d$ -dimensional vector of constants, and let  $\mathbf{b}$  be a vector of variables. We aim to model a factor  $f$  such that a scalar variable  $z = \sum_{i=1}^d a_i \cdot b_i$  is the weighted sum of the elements  $b_i$ . [Figure 3.3](#) illustrates the factor graph of this operation.

Let  $m_{b_i \rightarrow f}(b_i) = \mathcal{N}(b_i; \overrightarrow{\mu}_i, \overrightarrow{\sigma}_i^2)$  be the incoming forward messages from the summands, and let  $m_{z \rightarrow f}(z) = \mathcal{N}(z; \overleftarrow{\mu}_z, \overleftarrow{\sigma}_z^2)$  be the backward message from  $z$ . The forward message can then be derived using rescaling and summation of Gaussian variables.



**Figure 3.3:** The weighted sum of a  $d$ -dimensional vector  $\mathbf{b}$ .

#### Forward Message (Weighted Sum)

The forward message is  $m_{f \rightarrow z}(z) = \mathcal{N}(z; \vec{\mu}_z, \vec{\sigma}_z^2)$ , where

$$\vec{\mu}_z = \sum_{i=1}^d a_i \vec{\mu}_i,$$

$$\vec{\sigma}_z^2 = \sum_{i=1}^d a_i^2 \vec{\sigma}_i^2.$$

The backward message can be derived similarly, using scalar constants  $\frac{-a_j}{a_i}$  for  $j \neq i$  and  $\frac{1}{a_i}$  for  $z$ . The equations for the backward mean and variance are:

$$\overleftarrow{\mu}_i = a_i^{-1} \cdot \left( \overleftarrow{\mu}_z - \sum_{j=1}^d a_j \overleftarrow{\mu}_j + a_i \overleftarrow{\mu}_i \right),$$

$$\overleftarrow{\sigma}_i^2 = a_i^{-2} \cdot \left( \overleftarrow{\sigma}_z^2 + \sum_{j=1}^d a_j^2 \overleftarrow{\sigma}_j^2 - a_i^2 \overleftarrow{\sigma}_i^2 \right),$$

$$m_{f \rightarrow b_i}(b_i) = \mathcal{N}(b_i; \overleftarrow{\mu}_i, \overleftarrow{\sigma}_i^2).$$

To improve numerical accuracy and performance, we enhance the message equations for the backward message by: (1) storing and reusing  $m_{f \rightarrow z}$ , and (2) computing the backward message directly in natural parameter space.

**(1) Reusing the forward message:** The parameters of each backward message  $m_{f \rightarrow b_i}$  only depend on  $b_{j \neq i}$  through the sums  $\sum_{j=1}^d a_j \overleftarrow{\mu}_j$  and  $\sum_{j=1}^d a_j^2 \overleftarrow{\sigma}_j^2$ , which are already known as  $\overleftarrow{\mu}_z$  and  $\overleftarrow{\sigma}_z^2$ . Storing the forward message to  $z$  therefore allows us to compute each backward message  $m_{f \rightarrow b_i}$  in-place using only  $m_{b_i \rightarrow f}$ ,  $m_{f \rightarrow z}$ , and  $m_{z \rightarrow f}$ .

**(2) Direct computation in natural parameter space:** Similar to the product factor, we directly compute the natural parameters  $\overleftarrow{\tau}_i$  and  $\overleftarrow{\rho}_i$  instead of  $\overleftarrow{\mu}_i$  and  $\overleftarrow{\sigma}_i^2$ . Furthermore, we again aim to find an expression that relies on the precision of  $m_{z \rightarrow f}$  rather than its variance.

We first substitute the variance of the forward message:

$$\overleftarrow{\rho}_i = \frac{a_i^2}{\overleftarrow{\sigma}_z^2 + \overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2}$$

and then factor out  $\overleftarrow{\sigma}_z^2$ :

$$\begin{aligned} \overleftarrow{\rho}_i &= \frac{a_i^2}{\overleftarrow{\sigma}_z^2 \cdot \left(1 + \frac{1}{\overleftarrow{\sigma}_z^2} (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2)\right)} \\ &= \frac{a_i^2 \overleftarrow{\rho}_z}{1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2)}. \end{aligned}$$

This equation avoids dividing by  $\overleftarrow{\rho}_z$ , allowing the computation of  $\overleftarrow{\rho}_i$  even when  $\overleftarrow{\rho}_z = 0$ . A similar transformation can be applied to compute  $\overleftarrow{\tau}_i = \overleftarrow{\mu}_i \cdot \overleftarrow{\rho}_i$ :

$$\begin{aligned} \overleftarrow{\tau}_i &= \frac{\overleftarrow{\mu}_z - \overrightarrow{\mu}_z + a_i \overrightarrow{\mu}_i}{a_i} \cdot \frac{a_i^2 \overleftarrow{\rho}_z}{1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2)} \\ &= a_i \cdot \frac{\overleftarrow{\rho}_z \overleftarrow{\mu}_z - \overleftarrow{\rho}_z (\overrightarrow{\mu}_z - a_i \overrightarrow{\mu}_i)}{1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2)} \\ &= a_i \cdot \frac{\overleftarrow{\tau}_z - \overleftarrow{\rho}_z (\overrightarrow{\mu}_z - a_i \overrightarrow{\mu}_i)}{1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2)}. \end{aligned}$$

These equations form the updated backward message:

#### Backward Message (Weighted Sum)

We first pull out the denominator:

$$\rho_i^* = 1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - a_i^2 \overrightarrow{\sigma}_i^2).$$

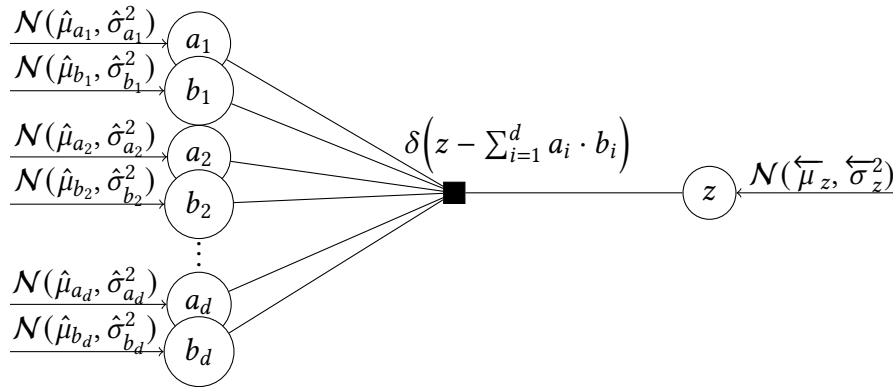
Then, the backward message is  $m_{f \rightarrow b_i}(b_i) = \mathcal{G}(b_i; \overleftarrow{\tau}_i, \overleftarrow{\rho}_i)$ , with

$$\begin{aligned} \overleftarrow{\tau}_i &= a_i \cdot \frac{\overleftarrow{\tau}_z - \overleftarrow{\rho}_z \cdot (\overrightarrow{\mu}_z - a_i \overrightarrow{\mu}_i)}{\rho_i^*}, \\ \overleftarrow{\rho}_i &= \frac{a_i^2 \overleftarrow{\rho}_z}{\rho_i^*}. \end{aligned}$$



## Inner Product

The product and weighted sum factors are already sufficient for modeling higher-order products. However, it is worthwhile to combine them into a new factor to reduce the number of intermediate variables and improve efficiency.



**Figure 3.4:** The inner product of two  $d$ -dimensional variables  $\mathbf{a}$  and  $\mathbf{b}$ .

Let  $\mathbf{a}$  and  $\mathbf{b}$  be two  $d$ -dimensional vectors. We model a factor  $f$  such that a scalar variable  $z = \sum_{i=1}^d a_i \cdot b_i$  represents the inner product (dot product) of  $\mathbf{a}$  and  $\mathbf{b}$ . Note that the main difference between an inner product and a weighted sum is that now,  $\mathbf{a}$  is an unobserved variable. This requires us to work with approximate marginals instead of input messages due to the variational approximation chosen for the product. Figure 3.4 shows the setup.

Let  $\mathcal{N}(\hat{\mu}_{a_i}, \hat{\sigma}_{a_i}^2)$  and  $\mathcal{N}(\hat{\mu}_{b_i}, \hat{\sigma}_{b_i}^2)$  be the approximate input marginals, and let  $\mathcal{N}(\overleftarrow{\mu}_z, \overleftarrow{\sigma}_z^2)$  be the backward message from  $z$ . We first perform an element-wise product followed by an unweighted sum, without storing the intermediate results. The forward message can then be computed by combining the message equations derived above.

### Forward Message (Inner Product)

The forward message is  $m_{f \rightarrow z}(z) = \mathcal{N}(z; \overrightarrow{\mu}_z, \overrightarrow{\sigma}_z^2)$ , where

$$\begin{aligned} \overrightarrow{\mu}_z &= \sum_{i=1}^d \hat{\mu}_{a_i} \hat{\mu}_{b_i}, \\ \overrightarrow{\sigma}_z^2 &= \sum_{i=1}^d (\hat{\sigma}_{a_i}^2 + \hat{\mu}_{a_i}^2) \cdot (\hat{\sigma}_{b_i}^2 + \hat{\mu}_{b_i}^2) - \hat{\mu}_{a_i}^2 \hat{\mu}_{b_i}^2. \end{aligned}$$

For the backward message, combining the two separate backward messages directly does not work because the backward message from the weighted sum factor would require the mean and variance of the (intermediate) forward message from  $a_i \cdot b_i$ . Although it would be possible to store this forward message, it is simpler to recompute the product factor when needed.

### Backward Message (Inner Product)

Compared to the weighted sum factor, the added complexity is mostly in the denominator:

$$\rho_i^* = 1 + \overleftarrow{\rho}_z \cdot (\overrightarrow{\sigma}_z^2 - (\hat{\sigma}_{a_i}^2 + \hat{\mu}_{a_i}^2) \cdot (\hat{\sigma}_{b_i}^2 + \hat{\mu}_{b_i}^2) + \hat{\mu}_{a_i}^2 \hat{\mu}_{b_i}^2),$$

The parameters of  $m_{f \rightarrow b_i}(b_i) = \mathcal{G}(b_i; \overleftarrow{\tau}_{b_i}, \overleftarrow{\rho}_{b_i})$  are then:

$$\begin{aligned} \overleftarrow{\tau}_{b_i} &= \hat{\mu}_{a_i} \cdot \frac{\overleftarrow{\tau}_z - \overleftarrow{\rho}_z \cdot (\overrightarrow{\mu}_z - \hat{\mu}_{a_i} \hat{\mu}_{b_i})}{\rho_i^*}, \\ \overleftarrow{\rho}_{b_i} &= (\hat{\sigma}_{a_i}^2 + \hat{\mu}_{a_i}^2) \cdot \frac{\overleftarrow{\rho}_z}{\rho_i^*}. \end{aligned}$$

The backward messages  $m_{f \rightarrow a_i}$  to  $a$  can be computed similarly.

### Higher-order Multiplication

Combining one-dimensional multiplication and weighted sums introduces unnecessary intermediate variables. We eliminate these by combining the factors into the inner product factor. In contrast, combining inner products to model higher-order multiplications does not introduce additional variables. For example, modeling a matrix-matrix multiplication simply results in a collection of independent inner products with no interaction.

While it can still be more efficient to combine these inner products into a single operation, no new message equations are required. In [Section 4.3.1](#), we will introduce our library for efficient general multiplication that makes linear algebra factors available under a uniform interface.

### 3.2.2 Activation Functions

We investigated multiple activation functions, but ultimately developed message equations only for LeakyReLU and MaxPool (for convolutional networks). We begin by describing our message equations for LeakyReLU, where our current approach combines moment-matching either the marginal or directly the message—both of which were presented individually in the MP report [\[Ada+24\]](#). Below, we provide these equations and describe our procedure for combining them into a robust approximation.

#### LeakyReLU



**Figure 3.5:** The non-linear transformation of a scalar  $a$  using the *LeakyReLU* activation function.

We model the relationship between two variables  $a$  and  $z$  such that  $z = \text{LeakyReLU}_\iota(a)$ , as shown in Figure 3.5. The LeakyReLU function is defined as:

$$\text{LeakyReLU}_\iota(a) = \begin{cases} a & \text{if } a \geq 0, \\ \iota \cdot a & \text{if } a < 0. \end{cases}$$

For  $\iota = 0$ , we obtain  $\text{ReLU}(a) = \text{LeakyReLU}_0(a)$ . For all other leaks  $\iota \neq 0$ ,  $\text{LeakyReLU}_\iota$  is bijective, and its inverse is  $\text{LeakyReLU}_{\iota^{-1}}$ . This technically allows us to develop only a forward message and apply it with an inverted leak for the backward message. We start by presenting these generic LeakyReLU message equations—taken from the MP report [Ada+24]—and then discuss small modifications that address numerical edge cases in the forward and backward messages.

The equations for LeakyReLU depend on two building blocks introduced in the MP report [Ada+24]. Note that  $\phi$  represents the cdf of the standard normal distribution:

$$\begin{aligned} \text{ReLUMoment}_k(\mu, \sigma^2) &= \begin{cases} \mathbb{E}[\text{ReLU}(a)] \text{ where } a \sim \mathcal{N}(\mu, \sigma^2) & \text{for } k = 1, \\ \mathbb{E}[\text{ReLU}^2(a)] \text{ where } a \sim \mathcal{N}(\mu, \sigma^2) & \text{for } k = 2. \end{cases} \\ &= \begin{cases} \sigma \mathcal{N}(x; 0, 1) + \mu \phi(x) & \text{for } k = 1, \\ \sigma \mu \mathcal{N}(x; 0, 1) + (\sigma^2 + \mu^2) \phi(x) & \text{for } k = 2. \end{cases} \end{aligned}$$

and

$$\begin{aligned} \text{Block52}_k(\mu_1, \sigma_1^2, \mu_2, \sigma_2^2) &= \mathcal{N}(\mu_1; \mu_2, \sigma_1^2 + \sigma_2^2) \\ &\cdot \begin{cases} \text{ReLUMoment}_k(\mu_m, \sigma_m^2) & \text{for } k = 1, 2, \\ \phi(\mu_m / \sigma_m) & \text{for } k = 0, \end{cases} \end{aligned}$$

where

$$\tau_m = \frac{\mu_1}{\sigma_1^2} + \frac{\mu_2}{\sigma_2^2}, \quad \rho_m = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}, \quad \mu_m = \frac{\tau_m}{\rho_m}, \quad \sigma_m^2 = \frac{1}{\rho_m}.$$

The  $\text{Block52}_k$  was introduced as Building Block 5.2 in the MP report [Ada+24] as an approximation for integrals of the form:

$$\int_0^\infty a^k \mathcal{N}(a; \mu_1, \sigma_1^2) \mathcal{N}(a; \mu_2, \sigma_2^2) da.$$

Using these two building blocks, we can reproduce the message equations for moment matching and marginal approximation of LeakyReLU:

#### Moment Matching (LeakyReLU)

The LeakyReLU forward message  $m_{f \rightarrow z}(z) = \mathcal{N}(z; \vec{\mu}_z, \vec{\sigma}_z^2)$  can be approximated using the ReLUMoment building block:

$$\begin{aligned} \vec{\mu}_z &= (1 - \iota) \cdot \text{ReLUMoment}_1(\vec{\mu}_a, \vec{\sigma}_a^2) + \iota \cdot \vec{\mu}_a, \\ \vec{\sigma}_z^2 &= \left( (1 - \iota^2) \cdot \text{ReLUMoment}_2(\vec{\mu}_a, \vec{\sigma}_a^2) + \iota^2 \cdot (\vec{\sigma}_a^2 + \vec{\mu}_a^2) \right) - \vec{\mu}_z^2. \end{aligned}$$

Refer to the MP report [Ada+24] for the detailed derivation.

### Marginal Approximation (LeakyReLU)

The marginal approximation of the forward message is:

$$m_{f \rightarrow z}(z) = \frac{\mathcal{N}(z; \frac{\vec{m}_1}{\vec{m}_0}, \frac{\vec{m}_2}{\vec{m}_0} - \left(\frac{\vec{m}_1}{\vec{m}_0}\right)^2)}{m_{z \rightarrow f}(z)} = \mathcal{N}(z; \vec{\mu}_z, \vec{\sigma}_z^2),$$

with the moments:

$$\begin{aligned} \vec{m}_k = & (-1)^k \cdot \text{Block52}_k(-\vec{\mu}_a, \vec{\sigma}_a^2, -\iota \cdot \overleftarrow{\mu}_z, \iota^2 \cdot \overleftarrow{\sigma}_z^2) \\ & + \text{Block52}_k(\vec{\mu}_a, \vec{\sigma}_a^2, \overleftarrow{\mu}_z, \overleftarrow{\sigma}_z^2). \end{aligned}$$

We clamp  $\vec{\sigma}_z^2$  to a maximum of  $10^8$ . Refer to the MP report [Ada+24] for the derivation of these moments.

Both LeakyReLU approximations use moment parameters rather than natural parameters. While forward messages typically have well-behaved moment parameters, the LeakyReLU approximations may become problematic for backward messages with near-zero precision, which then affects the marginal approximations of both the forward and the backward message.

To avoid dysfunctional messages in edge cases, we developed the following guardrails to ensure a well-defined marginal approximation message for LeakyReLU:

1. The resulting natural parameters  $\vec{\tau}_z$  and  $\vec{\rho}_z$  must be finite and non-NaN.
2. Forward Message: The precision of  $m_{f \rightarrow z}$  must be at least as high as the precision of  $m_{a \rightarrow f}$ , as LeakyReLU cannot increase the variance when  $\iota > 0$ . We also require a normalization constant of at least  $\vec{m}_0 > 10^{-8}$ .
3. Backward Message: If  $m_{z \rightarrow f}$  has low precision, the approximate backward message is likely to be numerically unstable. In experiments, it has been effective to enforce  $(\overleftarrow{\tau}_z > 0) \vee (\overleftarrow{\rho}_z > 2 \cdot 10^{-8})$ .

Under the assumption that  $m_{a \rightarrow f}$  is generally well-behaved while  $m_{z \rightarrow f}$  may be degenerate, we define the following message equation for LeakyReLU:

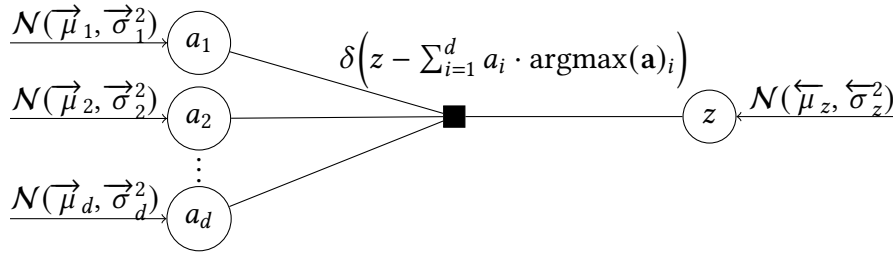
### Forward / Backward Message (LeakyReLU)

We use the marginal approximation whenever it complies with the guardrails above. Recall that a marginal approximation of the backward message can be obtained by setting  $\iota_{\text{back}} = \iota^{-1}$  and swapping  $m_{a \rightarrow f}$  and  $m_{z \rightarrow f}$  in the marginal approximation equation.

If the marginal approximation falls outside the guardrails, the moment-matching forward message is used. However, for the backward message, moment matching is usually also unreliable when the marginal approximation is unstable. Therefore, we use  $m_{f \rightarrow a} = \mathcal{G}(0, 0)$  as the backward message's fallback.

## MaxPool

In CNNs, MaxPool layers are commonly used to apply a patch-wise maximum function. While it would be straightforward to construct a factor  $f(z) = \delta(z - \max(\mathbf{a}))$  that directly mirrors this behavior, using moment matching on the resulting marginals is challenging. A promising direction for future work on direct approximations would be the use of moments of order statistics [BG59]. However, for now, we have chosen to approximate the effect of MaxPool using a slightly adapted factor.



**Figure 3.6:** Our approximation to MaxPool uses a weighted sum with coefficients set to argmax probabilities, which are treated as constants during the computation of the backward message.

As shown in Figure 3.6, we approximate the max operation through a weighted sum, where the coefficients  $p_i$  are the argmax probabilities of the inputs. To simplify the message equations, we treat  $p_i$  as constants during the backward pass. This means that we send a weighted-sum backward message with coefficients  $p_i$  but do not propagate a backward message "through" the coefficients. Although this deviates from a strict message passing framework, this approximation has demonstrated good empirical results.

The argmax weights  $p_i$  are computed as follows:

$$p_i^* = \prod_{j \neq i} \phi(0; \vec{\mu}_j - \vec{\mu}_i, \vec{\sigma}_i^2 + \vec{\sigma}_j^2),$$

$$p_i = \frac{p_i^*}{\sum_{j=1}^d p_j^*},$$

where  $\phi$  is the cdf of the standard normal distribution. For more background on this approximation, refer to the argmax training signal in Section 3.2.3. We now define the forward and backward messages of our MaxPool approximation:

### Forward / Backward Message (MaxPool)

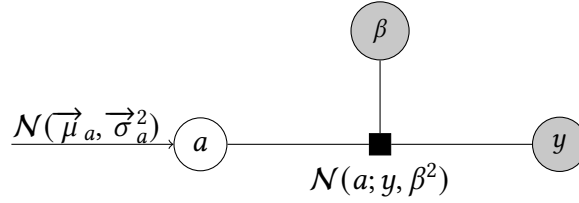
Let  $\mathcal{N}(\vec{\mu}_i, \vec{\sigma}_i^2)$  be the forward messages from a patch of a feature map, and let  $\mathcal{N}(\overleftarrow{\mu}_z, \overleftarrow{\sigma}_z^2)$  be the backward message of the MaxPool output. We compute the coefficients  $p_i = \text{argmax}(\mathbf{a}_i)$  and use the weighted sum factor to compute the forward and backward messages.

### 3.2.3 Training Signals

So far, we have presented factors involved in modeling the layers of a neural network. Now, we introduce three additional factors that incorporate training labels into the factor graph. Each of these factors is associated with a specific training example and connects to the output variables of the final layer. The combined messages from all training signal factors represent the likelihood term of the neural network. We first cover a simple factor for regression, followed by a discussion of softmax and argmax functions for classification.

#### Regression

Similar to Bayesian linear regression, we define the likelihood in regression problems as a normal distribution centered around the true observation. Let  $\mathbf{a}$  be the predictions,  $\mathbf{y}$  the labels, and  $\beta^2$  the noise constant. We then define the likelihood as a multivariate normal distribution with a diagonal covariance matrix:  $\mathcal{N}(\mathbf{a}; \mathbf{y}, \beta^2 \cdot \mathbf{I})$ . Notably, the maximizer of this likelihood,  $\mathbf{a} = \mathbf{y}$ , also minimizes the mean squared error between predictions and labels. Since the covariance matrix is diagonal, the likelihood term can be split into a product of individual likelihood terms. Figure 3.7 illustrates the factor graph for one such one-dimensional likelihood term.



**Figure 3.7:** The regression factor applies a  $y$ -centered normal distribution as the likelihood term, using a noise hyperparameter  $\beta^2$ . During prediction,  $y$  is unknown.

During training, when the label  $y$  is known, only the backward message  $m_{f \rightarrow a}$  is required to initiate the backward pass. However, when predicting  $y$  for new inputs, a forward message  $m_{f \rightarrow y}$  is also necessary. Both messages are straightforward to derive. Let  $m_{a \rightarrow f}(a) = \mathcal{N}(a; \vec{\mu}_a, \vec{\sigma}_a^2)$  be the forward message from  $a$ .

#### Forward Message for Prediction (Regression)

During prediction,  $y$  is unknown, and a forward message  $m_{f \rightarrow y}$  can be defined. Because the factor is a Gaussian density function, it can be rearranged as  $\mathcal{N}(a; y, \beta^2) = \mathcal{N}(y; a, \beta^2)$ . The forward message  $m_{a \rightarrow f}$  then serves as a conjugate prior on  $a$ , and the message to  $y$  is:

$$m_{f \rightarrow y}(y) = \mathcal{N}(y; \vec{\mu}_a, \vec{\sigma}_a^2 + \beta^2).$$

**Backward Message (Regression)**

When  $y$  is known, the backward message to  $a$  is directly equivalent to the factor function:

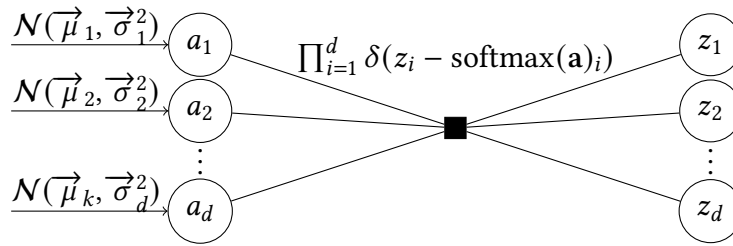
$$m_{f \rightarrow a}(a) = \mathcal{N}(a; y, \beta^2).$$

**Softmax Classification**

The softmax function is an activation function that transforms a  $d$ -dimensional input vector  $\mathbf{a}$  into a vector  $\mathbf{z}$  where all  $z_i \in (0, 1)$  and  $\sum_{i=1}^d z_i = 1$ . The definition of softmax is:

$$z_i = \text{softmax}(\mathbf{a})_i = \frac{\exp(a_i)}{\sum_{j=1}^d \exp(a_j)}.$$

In classification, softmax is typically applied to the network output  $\mathbf{a}$  (known as logits), and the softmax outputs  $\mathbf{z}$  are interpreted as predicted probabilities for each class.



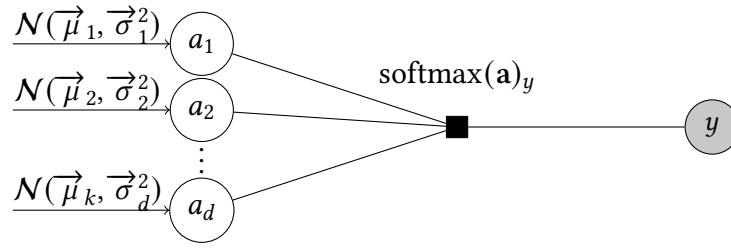
**Figure 3.8:** Softmax as an activation function with an explicit model of  $\mathbf{z}$  as variables.

We create a factor that models the softmax relationship between variables  $\mathbf{a}$  and  $\mathbf{z}$  directly, as shown in Figure 3.8. In this model, each element of  $\mathbf{z}$  is treated as a variable, and we reason about their marginal distributions, as expected from an activation function. However, when using the outputs to predict the class of an unseen input, it is more practical to obtain a vector of class probabilities  $\mathbf{p}$  that sum to 1 and integrate over all possible values of  $\mathbf{a}$ . Given forward messages  $\mathcal{N}(\vec{\mu}_i, \vec{\sigma}_i^2)$ , we aim to solve the integral:

$$p_i = \int_{\mathbf{a}} \text{softmax}(\mathbf{a})_i \cdot \prod_{j=1}^d \mathcal{N}(a_j; \vec{\mu}_j, \vec{\sigma}_j^2) d\mathbf{a},$$

where  $p_i$  is the probability that  $i \in \{1, \dots, d\}$  is the true class. This approach can be elegantly expressed as a factor that allows us to derive appropriate backward messages for a training likelihood, as shown in Figure 3.9. An approximation of this integral is available in section B.4.6 of Daxberger et al. [Dax+22], referred to as the "probit approximation."





**Figure 3.9:** Softmax as a classification output. Here,  $y$  is a discrete variable in  $\{1, \dots, d\}$ , which is known during training and unknown during prediction.

#### Forward Message for Prediction (Softmax Classification)

Let  $\mathcal{N}(\vec{\mu}_i, \vec{\sigma}_i^2)$  be the forward messages. We first compute the probit inputs  $\mathbf{t}$  as:

$$t_i = \frac{\vec{\mu}_i}{\sqrt{1 + \frac{\pi}{8} \vec{\sigma}_i^2}},$$

and then set:

$$p_i = \text{softmax}(\mathbf{t})_i.$$

During training, the label  $y$  is known, and we need to find backward messages  $m_{f \rightarrow a_i}$  for a suitable likelihood factor. A natural likelihood term for classification is the predicted softmax probability of the true class, as maximizing this likelihood also maximizes the predicted probability of the true class. We thus want to solve the integral:

$$m_{f \rightarrow a_i}(a_i) = \int_{\mathbf{a} \setminus a_i} \text{softmax}(\mathbf{a})_y \cdot \prod_{j \neq i} \mathcal{N}(a_j; \vec{\mu}_j, \vec{\sigma}_j^2) da_1 \cdots da_{i-1} da_{i+1} \cdots da_d.$$

This integral also perfectly aligns with the backward message of the factor in Figure 3.9. Two key differences from the forward message are: the backward message  $m_{f \rightarrow a_i}$  evaluates  $\text{softmax}_y$  instead of  $\text{softmax}_i$ , and  $a_i$  is not integrated out, meaning its density is not evaluated. We can approximate this integral similarly to the forward message by setting  $t_i = a_i$ . However, moment matching is not feasible because for all  $i \neq y$ ,  $\lim_{a_i \rightarrow -\infty} m_{f \rightarrow a_i}(a_i)$  converges to a constant  $c > 0$ , which causes the message to have infinite mass.

As this message is not integrable, we apply marginal approximation. We then aim to solve the integral:

$$m_{i,k} = \int_{a_i} a_i^k \cdot \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2) \cdot m_{f \rightarrow a_i}(a_i) da_i.$$

We currently compute the marginals by finding  $m_{i,0}, m_{i,1}, m_{i,2}$  for each  $i$  using numerical integration. This approach has shown good results on MNIST, but solving a large number of numerical integrations is computationally expensive. Despite performance optimizations, numerical integration remains a limiting factor in scaling when using softmax.

A preferable approach would be to combine these integrals:

$$m_{i,k} = \int_{\mathbf{a}} a_i^k \cdot \text{softmax}(\mathbf{a})_y \cdot \prod_{j=1}^d \mathcal{N}(a_j; \vec{\mu}_j, \vec{\sigma}_j^2) d\mathbf{a},$$

and to find an approximation for the entire expression. One promising method could be the use of a Laplace approximation similar to the "Laplace Bridge" in Daxberger et al. [Dax+22], as the difference between the marginal integral and the forward softmax is only that we evaluate  $a_i^k$  with  $\text{softmax}_y$  instead of  $\text{softmax}_i$ . This direction shows promise for future work, but currently, numerical integration is our only method. We compute the backward message as follows:

#### Backward Message (Softmax Classification)

Let  $\mathcal{N}(\vec{\mu}_i, \vec{\sigma}_i^2)$  be the forward messages, and let  $y$  be the class label. We compute the backward messages using marginal approximation:

$$m_{f \rightarrow a_i}(a_i) = \frac{\mathcal{N}(a_i; \frac{\overleftarrow{m}_1}{\overleftarrow{m}_0}, \frac{\overleftarrow{m}_2}{\overleftarrow{m}_0} - \left(\frac{\overleftarrow{m}_1}{\overleftarrow{m}_0}\right)^2)}{m_{a_i \rightarrow f}(a_i)} = \mathcal{N}(a_i; \overleftarrow{\mu}_i, \overleftarrow{\sigma}_i^2).$$

Similar to the forward message, we compute the probit inputs  $\mathbf{t}$  as:

$$t_i = \frac{\overrightarrow{\mu}_i}{\sqrt{1 + \frac{\pi}{8} \overrightarrow{\sigma}_i^2}},$$

and obtain the three moments using numerical integration:

$$\begin{aligned} m_{i,k} &= \int_{a_i} a_i^k \cdot \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2) \cdot m_{f \rightarrow a_i}(a_i) da_i \\ &= \int_{a_i} a_i^k \cdot \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2) \cdot \text{softmax}(t_1, \dots, t_{i-1}, a_i, t_{i+1}, \dots, t_d)_y da_i. \end{aligned}$$

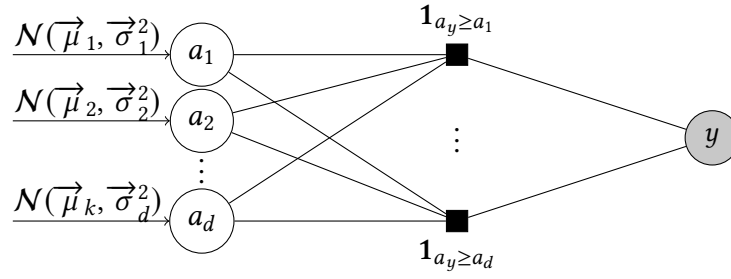
We then improve the numerical stability by computing log-softmax and log-pdf:

$$m_{i,k} = \int_{a_i} a_i^k \cdot \exp\left(\log(\mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2)) + \log(\text{softmax}(t_1, \dots, t_{i-1}, a_i, t_{i+1}, \dots, t_d)_y)\right) da_i.$$

#### Argmax Classification

Since we have no efficient approximations for the backward messages in softmax classification, we explore an alternative approach. Here, we directly enforce that the predicted class scores  $\mathbf{a}$  for a given label  $y$  satisfy  $\text{argmax}(\mathbf{a}) = y$ . Figure 3.10 shows how we model this constraint with individual factors.

We begin by deriving the message equations for the backward messages when  $y$  is known. Each individual  $\geq$  factor depends only on two variables,  $a_y$  and  $a_i$ , so the backward messages to all other



**Figure 3.10:** Factor arrangement for the backward messages of a single training example. When  $y$  is known, the  $\geq$  factors enforce that  $a_y$  is pairwise larger than all other variables. Each  $a_i$  with  $i \neq y$  then has only two non-constant backward messages, while  $a_y$  has  $d - 1$  such backward messages.

connected variables are constant and can be ignored. Consequently,  $a_y$  has  $d - 1$  non-constant backward messages, while all other variables have only one.

Assume that  $a_1$  and  $a_2$  are two variables connected to a factor enforcing  $a_1 \geq a_2$ , and let  $\mathcal{N}(\vec{\mu}_1, \vec{\sigma}_1^2)$  and  $\mathcal{N}(\vec{\mu}_2, \vec{\sigma}_2^2)$  be the respective forward messages. The message equation becomes apparent by rearranging the constraint: if  $a_1 \geq a_2$ , then  $a_1 - a_2 \geq 0$ . Since the forward messages are independent Gaussians, the difference  $a_\Delta = a_1 - a_2$  is a weighted sum, which has a forward message:

$$\mathcal{N}(a_\Delta, \mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2).$$

The marginal of  $a_\Delta$  is the product of this forward message and the backward message from the constraint  $a_\Delta \geq 0$ , which is a step function. The result is a truncated Gaussian  $\text{trunc}(\mathcal{N}(\mu_1 - \mu_2, \sigma_1^2 + \sigma_2^2))$ . By modeling the  $\geq$  factor as two separate factors,  $f_1$  (weighted sum) and  $f_2$  ( $\geq 0$ ), connected by the variable  $a_\Delta$ , we can derive the backward message as the combination of  $\text{trunc}$  and the weighted sum factor.

To increase the stability of the factor, we add some noise  $\epsilon \sim \mathcal{N}(0, \beta^2)$  to the equation:  $a_\Delta = a_1 - a_2 + \epsilon$ . The forward message to  $a_\Delta$  is still computed as a weighted sum. Let  $m_{f_1 \rightarrow a_\Delta}(a_\Delta) = \mathcal{N}(a_\Delta; \vec{\mu}_\Delta, \vec{\sigma}_\Delta^2)$  be the resulting forward message. The backward message from the truncation factor  $f_2$  to  $a_\Delta$  is then computed as:

$$m_{f_2 \rightarrow a_\Delta}(a_\Delta) = \frac{\text{trunc}(m_{f_1 \rightarrow a_\Delta})(a_\Delta)}{m_{f_1 \rightarrow a_\Delta}(a_\Delta)} = \mathcal{N}(a_\Delta; \overleftarrow{\mu}_\Delta, \overleftarrow{\sigma}_\Delta^2).$$

The moments of a zero-truncated normal can be derived from Greene [Gre03]. For a normal variable  $x \sim \mathcal{N}(\mu, \sigma^2)$  we are interested in  $\mu_{\text{trunc}} = \mathbb{E}[\text{trunc}(x)]$  and  $\sigma_{\text{trunc}}^2 = \mathbb{E}[\text{trunc}(x)]$ . These truncated moments are:

$$\begin{aligned} t &= \frac{\mu}{\sigma}, \\ v_t &= \frac{\mathcal{N}(t; 0, 1)}{\phi(t)}, \\ \mu_{\text{trunc}} &= \mu + \sigma \cdot v_t, \\ \sigma_{\text{trunc}}^2 &= \sigma^2 \cdot (1 - v_t \cdot (v_t + t)). \end{aligned}$$

To improve numerical stability, we directly compute the natural parameters of a moment-matched Gaussian again, and we also use logpdf and logcdf for intermediate steps:

$$\begin{aligned}
 t &= \frac{\tau}{\sqrt{\rho}}, \\
 v_t &= \exp(\log(\mathcal{N}(t; 0, 1)) - \log(\phi(t))), \\
 \tau_{\text{trunc}} &= \frac{\tau + v_t \cdot \sqrt{\rho}}{1 - v_t \cdot (v_t + t)}, \\
 \rho_{\text{trunc}} &= \frac{\rho}{1 - v_t \cdot (v_t + t)}.
 \end{aligned}$$

### Backward Message (Argmax Classification)

To compute the backward message for a factor  $\mathbf{1}_{a_1 \geq a_2}$ :

1. Apply the forward weighted sum to obtain  $m_{f_1 \rightarrow a_\Delta}(a_\Delta) = \mathcal{N}(a_\Delta; \vec{\mu}_\Delta, \vec{\sigma}_\Delta^2)$ .
2. Use trunc to compute:  $m_{f_2 \rightarrow a_\Delta}(a_\Delta) = \frac{\text{trunc}(m_{f_1 \rightarrow a_\Delta})(a_\Delta)}{m_{f_1 \rightarrow a_\Delta}(a_\Delta)} = \mathcal{N}(a_\Delta; \overleftarrow{\mu}_\Delta, \overleftarrow{\sigma}_\Delta^2)$ .
3. Compute  $m_{f_1 \rightarrow a_1}$  and  $m_{f_1 \rightarrow a_2}$  as weighted sum backward messages.

The overall backward messages of the argmax factor are then obtained as follows. The score  $a_y$  of the target class  $y$  is connected to  $d - 1 \geq$ -factors, and its backward message is the product of these individual messages. The backward message to all other class scores comes directly from one of the  $\geq$ -factors.

We found this factor to be unstable during training and propose two "hacks". First, the variance of all  $m_{f_1 \rightarrow a_i}$  for  $i \neq y$  is regularized by multiplying it with  $\phi(0; -\vec{\mu}_{a_\Delta}, \vec{\sigma}_{a_\Delta}^2)$  to prevent extreme precisions when  $a_y \ll a_i$ . Second, we add a regression-like term with a mean  $t_i = -1$  for  $i \neq y$  and  $t_y = 1$ . The updated backward message then becomes:

$$m'_{f \rightarrow a_i}(a_i) = m_{f \rightarrow a_i}(a_i) \cdot \mathcal{N}(a_i; t_i, \gamma^2).$$

These stabilizations are justified based on experimental results.

For prediction, we want to find  $p_i = p(\text{argmax}(\mathbf{a}) = i)$ . This probability is expressed as:

$$p_i = \int_{a_i} \mathcal{N}(a_i; \vec{\mu}_i, \vec{\sigma}_i^2) \cdot \prod_{j \neq i} \phi(a_j; \vec{\mu}_j, \vec{\sigma}_j^2) da_i.$$

For  $d = 4$ , this integral appears in Owen [Owe80] as integral "30,010.3," which requires evaluating the cdf of a multivariate Gaussian. Since no closed form exists, we use a simple approximation for  $p_i$  by computing a product of pairwise comparisons. This approximation again matches the message from the training factor in Figure 3.10 when  $y$  is unknown.

## Forward Message for Prediction (Argmax Classification)

We compute an unnormalized argmax probability  $p_i^*$  as:

$$\begin{aligned} p_i^* &= \prod_{j \neq i} p(a_i \geq a_j) \\ &= \prod_{j \neq i} \phi(0; \vec{\mu}_j - \vec{\mu}_i, \vec{\sigma}_i^2 + \vec{\sigma}_j^2), \end{aligned}$$

and then normalize  $p_i^*$  to approximate  $p_i = p(\text{argmax}(\mathbf{a}) = i)$ :

$$p_i \approx \frac{p_i^*}{\sum_{j=1}^d p_j^*}.$$

Our implementation of the factor graph simplifies the theoretical model presented in [Section 3.1](#) for practical purposes. We first present the file structure of our repository and a high-level code example. The following subsections will then explain the different levels of abstraction and their specific responsibilities.

## 4.1 High-Level Overview

root	
— archived	Archived code, mostly not operational anymore.
— utils	
— datasets.jl	Wrapper for datasets (downloaded or synthetic).
— factor_graph.jl	Implementation of the Trainer, FactorGraph, and layers.
— gaussian.jl	Library for operations on 1-dimensional Gaussian distributions.
— message_equations.jl	Message equations for the different layers.
— message_gaussian_mult.jl	Library for linear algebra message equations.
— plotting.jl	Plotting utilities.
— ...	
— mnist.jl	Example script for training a FactorGraph on MNIST.
— mnist_regression.jl	Training on MNIST as a one-hot encoded regression.
— mnist.py	Python script for comparable PyTorch experiments.
— <more scripts>	

**Figure 4.1:** The file structure of our repository. The core library is implemented in `utils/`, while the top level contains runnable training scripts or experiments. The `archived/` directory contains previous experiments that are unmaintained and often not operational anymore.

Our codebase is available on GitHub upon request and can be accessed [here](#)<sup>4</sup>. The relevant code is

<sup>4</sup> <https://github.com/HPI-Masterproject-Bayesian-Neural-Nets/Exploration>

currently on the `scale-gpu` branch of the linked repository. A simplified version with a slightly different file layout is also available [here](#)<sup>5</sup>.

Figure 4.1 shows the repository structure. All library code resides in the `utils/` directory, while scripts like `mnist_regression.jl` serve as good entry points for understanding the usage of our library. All code related to factor graphs is based on Julia v1.10. The README of the `scale-gpu` branch explains how to set up an environment with the required libraries.

After reviewing the general file structure, we now present a minimal working code example (Listing 1) for training on MNIST. This example demonstrates the high-level interface of our implementation and provides an initial understanding of its components. These components include:

1. Functions for loading MNIST and transforming it into a normalized regression dataset.
2. A domain-specific language for creating `FactorGraph` objects. In this case, the architecture can be translated as `[Linear(784, 100), LeakyReLU(0.01), Linear(100, 10)]`.
3. A `Trainer` object that manages training epochs, batching, and continued training.
4. Simple functions for `train` and `evaluate`.

The following sections delve into the inner workings of these components, as well as the various abstraction layers used by our implementation.

```
1  # Setup the training
2  batch_size = 320
3  d = as_regression_dataset(normalize_X!(MNIST()))
4  fg = create_factor_graph([
5      size(d.X_train)[1:end-1],
6      (:Flatten,),
7      (:Linear, 100),
8      (:LeakyReLU, 0.1),
9      (:Linear, 10),
10     (:Regression, 0.01)
11     ], batch_size
12 )
13 trainer = Trainer(fg, d.X_train, d.Y_train)
14
15 # Train and evaluate
16 train(trainer; num_epochs=5, num_training_its=2)
17 evaluate(fg, d.X_val, d.Y_val; as_classification=true)
```

**Listing 1:** Training a `FactorGraph` on MNIST.

<sup>5</sup> [https://github.com/iclr2025-7302/iclr2025\\_7302](https://github.com/iclr2025-7302/iclr2025_7302)



## 4.2 Architecture of the Factor Graph

The central object in [Listing 1](#) is the `FactorGraph`. Its definition is as follows:

```

1  mutable struct FactorGraph
2      layers::Vector{Factor} # The layers of the FactorGraph
3      batch_size::Int        # The number of training examples represented
4      device_array::UnionAll # Array type for CPU or GPU
5  end

```

The most notable aspect of this definition is the absence of variables. There are two key reasons why variables are not explicitly modeled: (1) contraction of the branches into one list of layers, and (2) the message passing schedule (iteration order).

### 4.2.1 Theory to Implementation

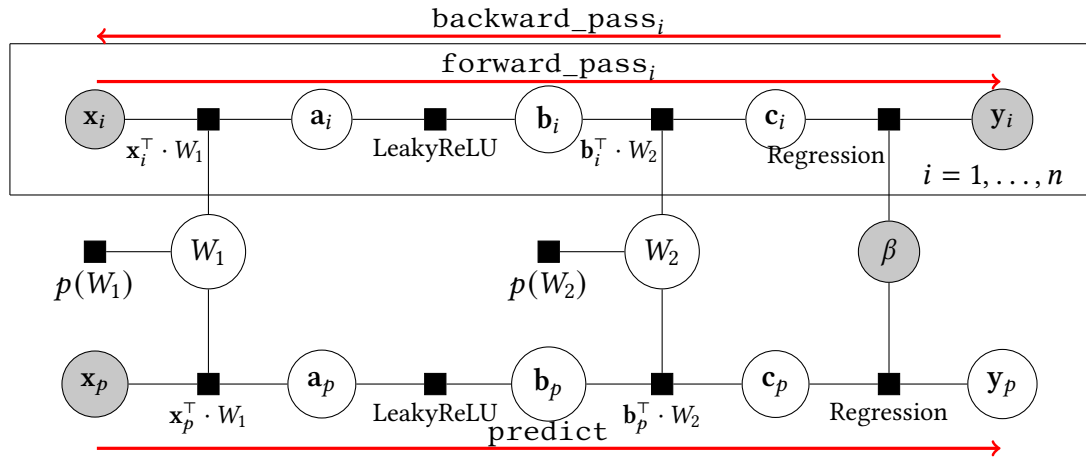
Let  $n$  be the number of training examples. The factor graph from [Section 3.1](#) for a neural network architecture with  $l$  layers then consists of:

1.  $n$  known-variable nodes for the inputs and outputs
2.  $n + 1$  duplicates of the hidden variables that connect two consecutive layers
3.  $l$  weight variables
4.  $n + 1$  duplicates of layer factors that represent each layer's function.

Since factors store no state, we can contract the  $n + 1$  layer factors into a single instance that represents the layer. As a result, each of the  $l$  weight variables is associated with only one layer object. Therefore, we can pull the weight variables into the layers to reduce the need for additional objects, references, and function calls.

After eliminating the separate weight variables and duplicate factors, we focus on the known-variable nodes and hidden variables. The factor graph could theoretically be iterated in any order. However, it is most efficient to pass messages sequentially front-to-back or back-to-front along an entire branch, as each new message builds on the information contained in the previous message. [Figure 4.2](#) illustrates the three iteration orders supported by our `FactorGraph`: `forward_passi`, `backward_passi`, and `predict`. Each of these iterates through an entire branch of the factor graph, starting at one end and proceeding through each layer until the other end.

In this iteration order, the message passing always begins with one of the known variables ( $x_i$  or  $y_i$ ). As this is always true, we can simply pass the known variables as parameters without needing any structural information about the specific layers. This pattern continues throughout the entire forward or backward pass: For two consecutive layers  $l_1$  and  $l_2$  connected by a hidden variable  $a$ , the message  $m_{a \rightarrow l_2}$  is equal to  $m_{l_1 \rightarrow a}$ . Thus, each layer's input is simply the output from the previous layer. The code for the `forward_passi` demonstrates this process:



**Figure 4.2:** The three iteration orders supported by our FactorGraph: a forward/backward pass for a training example or a forward pass for prediction.

```

1  # Perform a forward pass on branch i
2  function forward_pass(fg: FactorGraph, x_in: AbstractArray{Float64},
3      i_in: Int)
4      # Start with the input
5      m_forward = forward_message(fg.layers[1], x_in, i_in)
6      for layer in fg.layers[2:end]
7          # Pass messages through each layer
8          m_forward = forward_message(layer, m_forward, i_in)
9      end
10     return
11 end

```

In summary, we have reduced the factor graph from [Section 3.1](#) to the following responsibilities:

1. The FactorGraph passes the known variables and the messages from hidden variables as message parameters.
2. Each layer represents itself across all  $n + 1$  branches and stores the messages to its weights, if required.

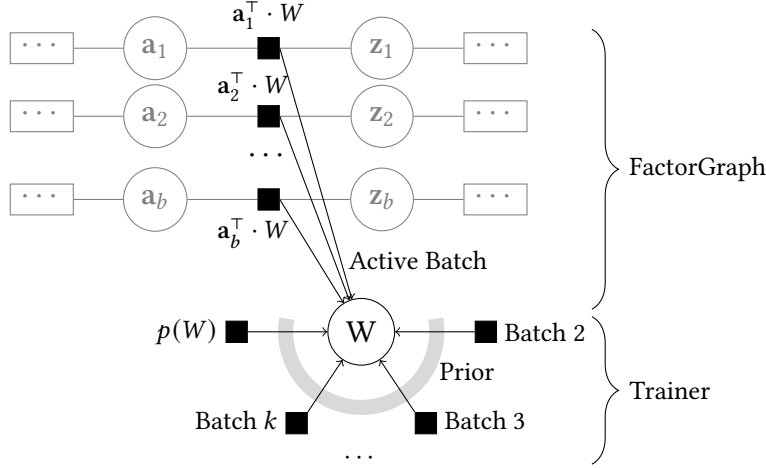
This allows us to represent the  $3n + l + 2$  elements from [Section 3.1](#) with  $l$  layers and one FactorGraph. In contrast, the implementation described in the MP report used one-dimensional variables that were each modeled as an instance. A linear layer mapping from width  $d_1$  to  $d_2$  would then require  $(d_1 + 1) \cdot d_2$  weight variables, including biases, and  $d_1 + d_2$  hidden variables.

## 4.2.2 Batched Training

So far, we have considered a factor graph that contains the entire dataset. At a minimum, we would need to store messages from each branch to the weights. If there are  $p$  parameters and  $n$  training examples, this results in storing  $p \cdot n$  messages. While it is possible to keep most messages

in CPU memory and transfer them to the GPU in smaller batches, the CPU memory requirements also become prohibitively large as either  $p \rightarrow \infty$  or  $n \rightarrow \infty$ .

We propose modeling a FactorGraph for one batch of examples and storing aggregated messages for batches that are not currently being processed. Theoretically, there are  $n$  messages to the weights  $m_{f_i \rightarrow W}$ . However, we limit the FactorGraph to updating only  $b$  examples at a time, which reduces the number of messages to  $W$  that we need to model to  $b$ . For the remaining examples, we summarize their messages to  $W$  in batch-wise products and store them in a Trainer object. Figure 4.3 shows the resulting setup.



**Figure 4.3:** A full FactorGraph models all messages for one batch of training examples. To iterate through the FactorGraph, we only need one joint message that summarizes the prior and all other examples. When switching to the next batch, we multiply all messages from the batch to  $W$  together into a single batch message and store it in the Trainer.

Let  $k$  be the number of batches, and let  $b$  be the number of training examples per batch. We then model a factor graph of size  $b$ , including the  $b$  messages to  $W$ . However, each message  $m_{W \rightarrow f_i}$  from  $W$  to those  $b$  branches is the product of all other messages:

$$m_{W \rightarrow f_i}(W) = \underbrace{\left( p(W) \cdot \prod_{j=b+1}^n m_{f_j \rightarrow W}(W) \right)}_{m_{o \rightarrow W}(W)} \cdot \underbrace{\left( \prod_{j=1}^b m_{f_j \rightarrow W}(W) \right)}_{m_{* \rightarrow W}(W)} / m_{f_i \rightarrow W}(W).$$

Since we iterate over only one batch at a time, the remaining incoming messages to  $W$  stay constant until the batch changes. From the perspective of the active batch, these can be treated as a single prior message  $m_{o \rightarrow W}$ , as shown above. When switching batches, we first compute the joint message  $m_{* \rightarrow W}$  from the FactorGraph to  $W$ , then store it in the Trainer:

$$m_{* \rightarrow W}(W) = \prod_{i=1}^b m_{f_i \rightarrow W}(W).$$

The FactorGraph then resets all internal messages to  $\mathcal{G}(0, 0)$ , and a new batch prior  $m_{o \rightarrow W}(W)$  is

computed from all batches except the new active one. When iterating through the FactorGraph, the inputs  $x_i$  and labels  $y_i$  corresponding to the new batch are used. From a theoretical perspective, this constructs a completely new factor graph with the prior  $m_{o \rightarrow W}(W)$ . In practice, the FactorGraph only needs to reset caches and store the new batch prior computed by the Trainer. Listing 2 shows pseudo-code for how batch-wise training is performed.

```

1  # Run batched message passing: Iterate over one batch, then retain only
2  # their joint likelihood and discard individual messages.
3  function pseudo_train(trainer::Trainer; num_epochs::Int,
4      num_training_its::Int)
5
6      for _ in 1:num_epochs
7          for (batch_number, X, y) in batched(trainer.X, trainer.y)
8              # Compute priors
9              for (l_i, m_batches) in zip(1:num_layers,
10                  trainer.batch_messages)
11
12                  marginal = prod(m_batches)
13                  batch_prior = marginal ./ m_batches[l_i, batch_number]
14                  trainer.fg.layers[l_i].prior = batch_prior
15              end
16
17              # Reset layer caches, if needed
18              reset_batch_caches.(trainer.fg.layers)
19
20              # Iterate through the FactorGraph
21              train_batch(trainer.fg, X, y; num_training_its)
22
23              # Compute new joint message from the batch
24              for l_i in 1:num_layers
25                  joint_message = prod(trainer.fg.m_to_weights)
26                  trainer.m_batches[l_i, batch_number] = joint_message
27              end
28          end
29      end
30  end

```

**Listing 2:** Pseudo-code for batch-wise training.

In conclusion, training in batches allows us to reduce the number of stored messages from  $p \cdot (n+1)$  to  $p \cdot (b + k + 1)$ , where  $p$  is the number of parameters,  $b$  is the batch size, and  $k$  is the number of batches. For the 60,000 examples in MNIST with a batch size of 320,  $(n + 1) = 60,001$  and  $(b + k + 1) = 509$ , resulting in a 118-fold reduction.

The trade-off of this approach is that examples cannot be freely interchanged between batches. When computing the batch prior  $m_{o \rightarrow W}$ , the messages from all examples outside the active batch must be multiplied together. If some examples from an old batch are active while others are not,

we would need individual messages for the inactive examples. However, since we store only their combined message, we cannot compute  $m_{o \rightarrow W}$  for partial batches. A solution to this problem is suggested in the future work section.

Furthermore, we have considered batching primarily as a means to reduce memory usage so far. However, traditional machine learning uses batching as a way to compute forward or backward passes for multiple examples simultaneously. In contrast, our batching approach still runs `forward_pass` and `backward_pass` on individual branches (training examples). While passing multiple messages in parallel could increase GPU utilization, preliminary experiments showed a decline in performance with an increasing number of messages passed concurrently. This suggests that, similar to shuffling, the problem may warrant further investigation in future work.

### 4.2.3 Software Architecture and Interfaces

Having covered the conceptual basis and design of our `FactorGraph` implementation, we now describe the interfaces and interactions of the main components. To keep the explanations concise, we usually simplify the function definitions, particularly by omitting type annotations. For the full, detailed definitions, we refer directly to the code.

1. **Trainer:** The `Trainer` encapsulates a `FactorGraph` instance and the entire training dataset. Its role is to store the necessary artifacts for training, allowing the process to be interrupted and resumed, for example, to measure validation accuracy. It also manages batch training, as described in [Section 4.2.2](#). Its only function is:

```
train(trainer::Trainer; num_epochs, num_training_its, silent=false)
```

An epoch is completed after passing once through all batches, while `num_training_its` specifies the number of forward-backward passes on a batch before proceeding to the next one.

2. **FactorGraph:** As discussed, the `FactorGraph` is essentially a list of layers. However, it offers a rich interface for high-level operations, such as in particular the message-passing interface shown in [Figure 4.2](#) that enforce the allowed iteration orders:

```
a) forward_pass(fg::FactorGraph, x_in, i_in::Int)
```

```
b) backward_pass(fg::FactorGraph, y, i_in::Int)
```

```
c) predict(fg::FactorGraph, X_in)
```

The `forward_pass` and `backward_pass` methods work on individual training examples, while the `predict` method processes a batch of examples. The `FactorGraph` also provides a `train_batch` function that combines these calls:

```
train_batch(fg::FactorGraph, X, Y; num_training_its, rng=nothing)
```

This method first shuffles the examples (branches) and then performs multiple iterations

of forward-backward passes, while iterating over the examples in the same order for each pass. Within an iteration, it calls `forward_pass_i` and then `backward_pass_i` for each training example before moving on to the next one.

Between iterations, we call `recompute_marginals(fg: :FactorGraph)` to ensure the marginal distributions remain accurate<sup>6</sup>. This function recalculates the marginal as the product of all incoming messages, countering any drift caused by sequential updates.

The `FactorGraph` also offers an `evaluate` function for computing metrics like mean squared error (MSE) or classification accuracy and an `Adapt.adapt_structure` method for transferring the entire `FactorGraph` between the CPU and GPU.

Additionally, the `FactorGraph` provides a domain-specific language for its construction:

```
create_factor_graph(layer_templates: :Vector{<:Tuple}, batch_size)
```

An example usage is shown in [Listing 1](#). The interface largely mirrors conventional deep learning libraries. However, our function automatically tracks the input shape, so that only the output size of each layer needs to be specified. The currently supported layers include:

- a) `(:Linear, d2)`: A fully connected layer (with bias) that maps from  $d_1$  to  $d_2$  nodes.
  - b) `(:Conv, k, f2)`: A convolutional layer with kernel size  $k \times k$ , mapping from  $d_{f_1}$  input to  $d_{f_2}$  output feature dimensions.
  - c) `(:LeakyReLU, l)`: An element-wise LeakyReLU activation with leak  $l$ .
  - d) `(:MaxPool, k)`: A max-pooling function for CNNs that is applied to  $k \times k$  patches for each feature map separately.
  - e) `(:Flatten, )`: Flattens the input into a vector.
  - f) `(:Regression, b2)`: A regression training signal with noise parameter  $b^2$ .
  - g) `(:Softmax, )`: A softmax training signal (see [Section 3.2.3](#)).
  - h) `(:Argmax, r)`: An argmax training signal with a boolean  $r$  to toggle adding regression-like labels to the backward message (see [Section 3.2.3](#)).
3. **Layer (Factor)**: The `FactorGraph` implicitly stores the factor graph structure, while the layers maintain the state. Each layer stores messages to its weights and other necessary information for message passing. Typically, layers store the last forward message and all backward messages, which may be needed to compute input marginals. Layers implement a shared interface for consistency with the `FactorGraph`'s operations:

```
a) forward_message(layer, m_in, i_in)
```

<sup>6</sup> Messages from the weights are computed by dividing the incoming message out of the weight marginal, the product of all incoming messages. When a message to the marginal changes, we compute the new marginal by dividing out the old message and multiplying the updated one in. However, this leads to drift - the marginal is eventually not equal to the product of messages anymore.

- b) `backward_message(layer, m_back, i_in)`
- c) `forward_predict(layer, M_in)`

The `forward_message` and `backward_message` methods compute messages for an individual training branch  $i$ , while `forward_predict` handles batches. While all layers implement the same interface, the input types can vary based on the layer's position—first layers use constants instead of variables. We implemented specialized layers like `FirstGaussianLinearLayer` (for constants) and `GaussianLinearLayer` (for variables) accordingly.

Layers with weights inherit from a subtype `WeightFactor`. Although Julia does not allow enforcing attributes on abstract types, each weight-based layer implementation is expected to contain the attributes `m_to_weights`, `last_W_marginal`, `m_to_biases`, and `last_bias_marginal`. Layers with weights also provide two additional methods:

- a) `recompute_marginal(layer::WeightFactor)`
- b) `reset_for_batch(layer, new_prior, new_bias_prior)`

The first method recalculates the marginal, while the second resets messages and sets a new prior, which is usually chosen by the `Trainer` to be the combined message from the true prior and other batches (Figure 4.3). Layers may also implement `reset_batch_caches` for resetting state before batch switching, and we provide a generic template implementation of `Adapt.adapt_structure` for putting any layer on GPU.

4. **Gaussian1d:** As discussed in the MP report [Ada+24], we implemented a one-dimensional Gaussian representation using natural parameters. We also considered multivariate Gaussians with diagonal covariance matrices as an alternative, but obtained superior performance with non-mutable one-dimensional arrays.

`Gaussian1d` offers a comprehensive function interface, categorized as follows:

- a) Constructors (e.g., for natural parameters, moment parameters, arrays, zero constructors)
- b) Conversion to moment parameters (as the representation uses natural parameters)
- c) Density operations (e.g., multiplication, division, truncation)
- d) Linear transformations of the random variable
- e) Utilities (e.g., thresholding and printing)
- f) Sampling and pdf/cdf functions

For further details, refer to the implementation.

The `Trainer`, `FactorGraph`, layers, and `Gaussian1d` are objects, while the message equations from Section 3.2 are implemented as stateless functions. This modular structure ensures that the



system is flexible and scalable, while allowing for easy extension or modification of components as needed. In the next section, we will examine some of the more interesting layer implementations, highlighting how they contribute to the overall efficiency and adaptability of the framework.

## 4.3 Message Equations

This section mirrors [Section 3.2](#) in both name and structure. We first present our library for linear algebra operations and then show how these operations are utilized in defining various layers.

### 4.3.1 Linear Algebra

In [Section 3.2](#), we already introduced the message equations required for products  $\mathbf{a}^\top \cdot \mathbf{b}$ , where  $\mathbf{b}$  is a variable and  $\mathbf{a}$  can be either constant (weighted sum) or a variable (inner product). We now construct a library for generalized operations  $Z = A \cdot B + C$  based on these factors.

Our design is guided by the following principles:

1. **Reverse multiplication order:** For vector-matrix products, we prefer computing  $\mathbf{a}^\top \cdot B$  over the customary  $A \cdot \mathbf{b}$ . If the matrix is the right operand, each output element becomes a product of  $\mathbf{a}^\top$  and one column of  $B$ , aligning with Julia's column-major storage format. As the weight matrix is the right operand in our reverse multiplication order and a backward message to the weights is always required, the backward functions in our library default to returning backward messages to the right operand unless specified otherwise.
2. **Uniform interface for constants and variables:** We designed our interface to abstract the differences in message equations when transitioning from the product of a variable and a constant to the product of two variables. Each element of  $A \cdot B + C$  can be freely switched between constants and variables. In practice, our `FactorGraph` mainly requires this flexibility for  $A$ , while  $B$  and  $C$  are always variables.
3. **Support for various dimensionalities:** The interface should handle scalars, vectors, matrices, and tensors. We have implemented the combinations needed by our code, but the interface can be easily extended to accommodate other signatures.
4. **Minimized memory allocation:** To minimize memory usage, most methods require pre-allocated arrays for their outputs. This approach not only reduces ambiguity but also increases efficiency. Furthermore, it is generally faster to use the weight marginals and factor-to-weight messages directly in the equations rather than computing weight-to-factor messages first. This can be viewed as "inlining" the computation where weight-to-factor messages are needed, thereby avoiding an additional array operation upfront.

With these principles in mind, we present three function signatures for vector-matrix multiplication:



```

1  # 1: Prediction
2  function forward_mult(
3      a::AbstractVector{T}, B::AbstractMatrix{Gaussian1d},
4      c::AbstractVector{Gaussian1d}, out::AbstractVector{Gaussian1d}
5  ) where {T<:Union{Float64,Gaussian1d}}
6
7  # 2: Variable a
8  function forward_mult(
9      marginal_a::AbstractVector{Gaussian1d},
10     marginal_B::AbstractMatrix{Gaussian1d},
11     marginal_c::AbstractVector{Gaussian1d}, m_to_c::AbstractVector{Gaussian1d},
12     out::AbstractVector{Gaussian1d}
13 )
14
15 # 3: Constant a
16 function forward_mult(
17     a::AbstractVector{Float64},
18     marginal_B::AbstractMatrix{Gaussian1d}, m_to_B::AbstractMatrix{Gaussian1d},
19     marginal_c::AbstractVector{Gaussian1d}, m_to_c::AbstractVector{Gaussian1d},
20     out::AbstractVector{Gaussian1d}
21 )

```

**Listing 3:** Forward messages for vector-matrix multiplications  $\mathbf{a}^\top \cdot \mathbf{B} + \mathbf{c}$ .

The differences between these functions are subtle yet important. All three assume  $B$  and  $\mathbf{c}$  to be variables, while  $\mathbf{a}$  may be a constant or a variable. For a constant  $\mathbf{a}$ , the multiplication's message equations requires the factor-to-weight messages of  $B$  (to compute weight-to-factor messages inline), while for a variable  $\mathbf{a}$ , only the marginals of  $\mathbf{a}$  and  $B$  are needed. During prediction, these are equivalent<sup>7</sup>, allowing the first method to handle both cases with a single input per operand.

The other two functions distinguish between constants and variables. The second function handles a variable  $\mathbf{a}$  and requires only the marginal of  $B$ , while the third handles a constant  $\mathbf{a}$  and depends on the message from  $B$ . Both take the message from  $\mathbf{c}$  as addition always uses messages. Recall that instead of directly working with the message from a variable, we inline its computation  $\text{marginal}_W/\text{message\_to}_W$  when it is needed.

For the backward message of vector-matrix multiplication, there are only two corresponding implementations because predictions require no backward messages. Listing 4 shows the function signature of the backward message for the case when  $\mathbf{a}$  is a constant. The backward message requires all information from the forward message, as well as the corresponding forward and backward messages for the result. The  $\text{m\_pred}$  parameter is the stored forward message to the result (computed by `forward_mult`), while  $\text{m\_back}$  is the backward message from the result. Two output arrays are provided for the backward messages towards  $B$  and  $\mathbf{c}$ . There is a similar method for computing the backward message when  $\mathbf{a}$  is a variable, which additionally also produces a backward message to  $\mathbf{a}$  and does not require  $\text{m\_to}_B$  as input.

<sup>7</sup> We never model messages from the prediction branch to the weights, so there is no need to divide out the marginal to obtain the message for prediction.

```

1  # Constant a
2  function backward_mult(
3      a::AbstractVector{Float64},
4      marginal_B::AbstractMatrix{Gaussian1d},
5      m_to_B::AbstractMatrix{Gaussian1d},
6      marginal_c::AbstractVector{Gaussian1d},
7      m_to_c::AbstractVector{Gaussian1d},
8      m_pred::AbstractVector{Gaussian1d},
9      m_back::AbstractVector{Gaussian1d},
10     out_B::AbstractMatrix{Gaussian1d},
11     out_c::AbstractVector{Gaussian1d}
12 )

```

**Listing 4:** Backward message for vector-matrix multiplications  $\mathbf{a}^\top \cdot \mathbf{B} + \mathbf{c}$  with a constant  $\mathbf{a}$ .

Overall, we provide the following categories of operations:

1.  $\text{Vector}^\top \times \text{Vector}$  (plus bias)
2.  $\text{Vector}^\top \times \text{Matrix}$  (plus bias)
3.  $\text{Matrix} \times \text{Matrix}$  (plus bias)
4. Batched  $\text{Vector}^\top \times \text{Vector}$  (for each column of matrices  $A$  and  $B$ )
5. Batched  $\text{Matrix} \times \text{Matrix}$  (for a 3D tensor  $A$  and a matrix  $B$ )

These operations all rely on low-level functions that compute parameters for a single forward or backward message. While we don't delve into the details here, we recommend examining the code for one of the vector-matrix functions as all others follow a similar pattern. The high number of functions is owed to the fact that we optimized for specific interfaces over generic implementations, as a significant part of the runtime is spent on linear algebra operations.

### 4.3.2 Layers

We now want to demonstrate how the linear algebra library facilitates efficient and concise layer construction by looking at the code for the `GaussianLinearLayer`'s function to compute the `forward_message`. As shown in [Listing 5](#), we essentially simply have to compute the input marginals and then call `forward_mult` with appropriate parameters. Also notice that only `last_m_to_inputs` depends on `i_in`, while `last_in_marginal` and `last_m_out` only store the current branch's messages. This example demonstrates how our linear algebra library enables us to write concise code for the message equations of many layers. The `FirstGaussianLinearLayer` can be implemented almost identically by simply working with constants directly (instead of the input marginal).

```

1  # Variable input
2  function forward_message(node::GaussianLinearLayerFactor,
3      m_in::AbstractVector{GaussianId}, i_in::Int)
4
5      # Compute input marginals
6      m_to_inputs = @view node.last_m_to_inputs[:, i_in]
7      @tullio node.last_in_marginal[i] = m_in[i] * m_to_inputs[i]
8
9      # Compute product-sum factor
10     m_to_biases = selectdim(node.m_to_biases, 2, i_in)
11     forward_mult(node.last_in_marginal, node.last_W_marginal,
12         node.last_bias_marginal, m_to_biases, node.last_m_out)
13     return node.last_m_out
14 end

```

**Listing 5:** Using linear algebra for the GaussianLinearLayer.

We also built the convolutional layers on our linear algebra library by applying the `im2col` algorithm as a pre-processing step [CPS06], which transforms convolutions into matrix-matrix multiplications. Initially, we implemented convolutions directly on tensors, but switching to `im2col` significantly improved performance and maintainability. For forward messages, the message equation begins by applying the `im2col` function from FluxML/NNlib [Inn+18] and then performs the matrix-matrix operation using our library, before finally reshaping the output back into the appropriate three-dimensional tensor. The backward message for convolutions is more complex. Each input and weight participates in multiple multiplications, so the backward messages are the products of the individual backward messages from these operations. While the backward message for the weights is straightforwardly handled by our linear algebra library, the backward message for the inputs requires an additional step, as the `im2col` transformation duplicates input elements. Therefore, after computing the backward messages for each (unfolded) input matrix element, these duplications must be merged back into the original image tensor. Although FluxML/NNlib offers a `fold` operation for this purpose, we observed that it occasionally produces non-deterministic results with noise magnitudes around  $10^{-14}$ . This can cause slight variations in training results even despite seeding. However, we use the `fold` operation nonetheless. Pooling layers such as `MaxPool` can also leverage the `im2col` transformation alongside our linear algebra functions, enabling efficient reuse of our matrix operations. Thanks to the linear algebra library, only the `LeakyReLU`, `Regression`, `Softmax`, and `Argmax` layers required direct implementations of their respective message equations.

Overall, our linear algebra library serves as the foundation for many of our layer implementations, significantly improving both performance and code quality. By centralizing these operations, we were able to create a consistent and reusable interface that simplifies the layer development. Future work could focus on further abstracting our interfaces to balance code generality with performance. Nonetheless, the modularity and efficiency of the current design have proven effective in building and maintaining a robust and flexible architecture for message passing.

## 4.4 Refactorings

We have already detailed many aspects of our implementation, including design rationale and optimizations. However, a significant portion of the work after the Master's project [Ada+24] involved bug fixing, performance optimization, and enhancing numerical stability. This section highlights these efforts by summarizing the optimizations that enabled the project to evolve from handling linear regression on feature transforms [Ada+24] to building (feature-function-free) CNNs with competitive MNIST performance.

### 4.4.1 Numerical Stability

The improvements of our codebase can be summarized as "making it work" and "scaling up." These two tasks are inherently connected, as bug-free code with high numerical stability was a concern throughout the entire development process. However, we here want to emphasize the changes that first allowed us to train MLPs without relying on feature functions. This happened only after addressing several critical issues, which include:

1. **Handling edge cases:** We introduced specific handling for edge cases, such as setting minimum precisions for backward messages or adding epsilon tolerances before zero checks during Gaussian divisions. These adjustments were crucial for ensuring that the training process works without crashing or triggering assertions.
2. **LeakyReLU approximations:** The marginal approximation for LeakyReLU can become unstable in certain scenarios. To address this, we defined guardrails for well-defined marginal approximations and, when necessary, use message approximation as a fallback, as described in [Section 3.2](#). Combining marginal and direct message approximations gave us the best results, as only relying on direct message matching also degraded the training results.
3. **Marginal drift:** As we continue updating the running sums used as variable marginals throughout the training, they may diverge from the actual product of messages. Initially, we recomputed the marginal each time it was needed, but later reduced to recomputing it once per iteration<sup>8</sup>.
4. **Spectral parameterization:** Setting the prior mean and variance significantly impacts the stability of the training process. More details on our prior parameters can be found in [Chapter B](#).

These changes were instrumental in achieving the first successful training of a feature-free MLP that did not rely on polynomials or shifted Gaussian densities to generate input features. After this breakthrough, our focus shifted to scaling up the approach and introducing new layer types. Nevertheless, we further improved the numerical stability during these developments, through:

1. **Message equations in natural parameters:** As outlined in [Section 3.2](#), we derived message

<sup>8</sup> An iteration, as defined earlier, involves running one forward and one backward pass for each training example in the batch.

equations that directly compute natural parameters, thereby avoiding divisions by the precision of incoming messages. This approach significantly enhanced the stability and numerical precision of our training.

2. **Log-space softmax:** For larger models, the softmax training layer occasionally threw errors for highly unlikely configurations. We resolved this by computing the softmax and  $\mathcal{N}$  values in log-space and only applying exp after multiplying these two (or rather adding, in log-space), as also described in [Section 3.2](#).
3. **Randomized training order:** Although we cannot reassign training examples to new batches (as discussed in [Section 4.2.2](#)), shuffling batches and randomizing the order within batches already improved the stability of the training as well.
4. **Message damping:** We found that applying an exponential moving average to the natural parameters of the batch-wise factor-to-weight message stabilized the training. Importantly, this damping is applied at the Trainer level but not within the FactorGraph, as doing so degraded performance in our experiments.

Beyond these conceptual changes, we also consistently focused on eliminating bugs and programming errors. Debugging was made particularly challenging by the fact that message passing can be robust enough to work despite certain bugs, yet sensitive enough to become unstable with minor errors. Throughout the project, we extensively debugged the codebase, as most new layers and message equations did not work correctly out of the box. One of the most effective debugging strategies involved comparing each layer's output against a previous version of that layer. This approach allowed us to confidently refactor code, catch bugs early, and directly observe the numerical effects of improved equations.

### 4.4.2 Performance Improvements

We want to start by introducing a key tool in our refactoring process: Tullio [[Abb+23](#)], a Julia macro for Einstein summation (Einsum) notation. Tullio allows for highly expressive code, making complex operations easier to read and understand. For instance, in [Listing 6](#), we illustrate a simple example where ten 20-dimensional vectors are mapped into ten  $20 \times 30$ -dimensional matrices. While this operation can also be expressed using broadcasting, the Tullio version is not only more

```

1  # Setup
2  function f(a, b) = a + b
3  A, B = rand(10, 20), rand(20, 30)
4
5  # Broadcasting to create a (10, 20, 30) output
6  out1 = f.(reshape(A, (10, 20, 1)), reshape(B, (1, 20, 30)))
7
8  # Tullio version
9  @tullio out2[i,j,k] := f(A[i,j], B[j,k])

```

**Listing 6:** Using Tullio for efficient matrix operations.

explicit but also performs faster<sup>9</sup>. For additional examples of optimizing functions using Tullio, we highly recommend our [Chapter A](#).

After extensive refactoring, we have sped up our CPU code by more than two orders of magnitude compared to the MP report [Ada+24]. Even after overcoming numerical issues, our code initially only managed small-scale MLPs with a few hundred parameters on one-dimensional data. In contrast, the optimizations below allowed us to scale up to MLPs with over 5.6 million parameters trained on 60,000 examples with 784 input dimensions. Below, we summarize the major lessons learned during this scaling process:

1. **The initial approach** (MP report [Ada+24]):

- a) Direct implementation of a factor graph. Each training example had its own copy of factors and variables, creating a separate training branch for each.
- b) Factors were connected to vectors of one-dimensional variables for inputs and outputs, using `get_message`, `send_message`, or `get_marginal` operations for message passing, which each involved multiple divisions of Gaussian distributions.

2. **First major optimization - a new FactorGraph implementation:**

- a) **Centralized message passing:** We began by eliminating the explicit modeling of variables between factors. Messages are now passed directly by the training loop rather than through decentralized edges, which also improves numerical stability by avoiding redundant multiplications and divisions. Previously, we had to divide at least once for each time we passed a message.
- b) **Single factor instance per layer:** Instead of creating a factor per data example, we now use only one instance per layer, thereby significantly reducing memory usage.
- c) **Batching** (as described in [Figure 4.3](#)): Rather than building a FactorGraph for the entire training dataset, we model only the active batch, with batch likelihoods for other batches stored externally in the `Trainer`. This reduced the memory footprint to  $\text{size}(\text{weights}) \cdot (1 + \text{num\_batches} + \text{batch\_size})$ .
- d) **LeakyReLU optimization:** Combined computations of  $\vec{m}_0$ ,  $\vec{m}_1$ , and  $\vec{m}_2$  (the normalization constant, first, and second moments of the marginal). By reusing intermediate results, we improved LeakyReLU performance by approximately 3-4x.
- e) **Improved message equations:** Transitioning from moment-parameter-based message equations to those derived using natural parameters (see [Section 3.2](#)) not only enhanced numerical stability, but also reduced the number of required divisions and intermediate calculations.
- f) **Marginal updating strategy:** As discussed in [Section 4.4.1](#), we reduced the frequency of marginal recomputation to once per iteration after experimenting with different updating strategies.

<sup>9</sup> In a `@btime` benchmark, the broadcasting approach took around  $5.1\mu\text{s}$ , while the Tullio implementation took between  $4.0$  and  $4.6\mu\text{s}$ .



### 3. Further performance optimization - memory management and dispatching:

- a) **Minimizing allocations:** We now avoid creating intermediate `Gaussian1d` objects where possible, as they negatively impact performance. We reduced memory usage and the need for garbage collection through:
  - i. Fusing operations like `division_mean(g1, g2)`, which returns the mean parameter of the result of a Gaussian division without allocating a `Gaussian1d`.
  - ii. Refactoring linear algebra code to inline weight-to-factor message computations when needed, thereby eliminating unnecessary allocations and copying.
  - iii. Reusing the layer's existing storage arrays within message equations to avoid allocating and copying data into separate output arrays. Further speed-ups could be achieved by pre-allocating arrays for intermediate results, but we prioritize maintaining a smaller memory footprint.
- b) **Single training example updates:** We simplified our methods to process only one training branch at a time. Previously, our interface allowed message passing on multiple branches simultaneously, which introduced inefficient 3D array operations even when only updating one branch. This change resulted in a tenfold speed-up for some sub-operations.
- c) **Minimizing the number of dispatches:** Dispatches occur, for example, when applying element-wise functions to arrays. We significantly improved performance by combining multiple consecutive array operations into a single kernel call, often using Tullio. For instance, the updated backward messages in [Section 3.2](#) for linear algebra are now implemented as an element-wise map function, resulting in substantial performance gains.
- d) **Multi-threading:** Rather than refactoring the code, configuring Julia to utilize multiple threads was the key step in leveraging multi-threading. Our other optimizations, such as utilizing Tullio and minimizing dispatches, naturally enabled multi-threading to provide significant performance gains without extensive modifications.

Beyond these principles, implementation details played a crucial role in achieving optimal performance. Many message implementations underwent extensive benchmarking, with multiple versions (often 5-10) tested to find the most efficient approach. In [Chapter A](#), we present an illustrative guide that showcases our optimization strategies through practical code comparisons. We highly recommend this guide as a general walk-through of performance optimization in Julia. Furthermore, we measure the runtime performance of our implementation in [Section 5.4](#).

#### 4.4.3 GPUs and CUDA

Leveraging GPUs is crucial for training large models efficiently. In this section, we discuss how to deploy CUDA kernels from Julia and integrate CUDA code with CPU code. To begin using Julia's CUDA library `CUDA.jl`, import it as the first library:

```
using CUDA, KernelAbstractions, CUDA.CUDAKernels
```

Many libraries will compile to CUDA-compatible code if CUDA is already loaded before they are imported. Standard functions such as `sum`, `prod`, and broadcasting also leverage CUDA by default. Furthermore, Tullio statements compile to GPU kernels without any modifications, allowing us to write efficient CUDA code without resorting to low-level CUDA-specific kernels.

The lessons from the previous section, particularly minimizing the number of dispatches, are equally applicable when optimizing performance on GPUs, and it is often possible to write code that performs well on both CPU and GPU. However, we found that the most efficient versions for each are not always the same. In such cases, we provide two versions of the code, implemented either through conditional statements within a function or by using specialized dispatch based on function parameter types, depending on the degree of overlap between the CPU and GPU code.

**Listing 7** demonstrates these two different versions of the forward message for vector-matrix multiplication. On the CPU, it is efficient to run each inner product on a single thread, as this avoids memory allocation. On the GPU, however, it is faster to separate the multiplication and summation into two steps.

```

1  if !isCUDA(a)
2      # For each column of B, compute the vector product
3      Threads.@threads for j in eachindex(out)
4          out[j] = forward_mult(a, (@view B[:, j]), c[j])
5      end
6  else
7      # Compute a full product, then sum out one dimension
8      @tullio temp[i, j] := forward_mult_μ(a[i], B[i, j])
9      μ = sum(temp, dims=1)
10
11     @tullio temp[i, j] = forward_mult_σ2(a[i], B[i, j])
12     σ2 = sum(temp, dims=1)
13
14     # Construct the result vector and add bias
15     @tullio out[j] = Gaussian1d(
16         μ=μ[j] + mean(c[j]),
17         σ2=σ2[j] + variance(c[j])
18     )
19
20     # Free CUDA memory
21     free_if_CUDA!.(temp, μ, σ2)
22 end

```

**Listing 7:** CPU and GPU code for the forward message of vector-matrix multiplication.

A challenge with `CUDA.jl` is that it cannot be installed on devices without a GPU, causing imports to fail. To address this, we developed small macros and functions to safely integrate CUDA code without sacrificing CPU compatibility. These functions can be used even if CUDA is not installed:

1. `isCUDA(a)`: Tests if a given array is located on the GPU.



2. `free_if_CUDA!(a)`: Similar to `CUDA.unsafe_free!(a)`, but it releases the memory allocated by `a` only if `isCUDA(a)` is true.
3. `@CUDA_RUN`: Executes a given statement only if CUDA is available. For example:

```
@CUDA_RUN using CUDA, KernelAbstractions, CUDA.CUDAKernels
```

4. `@NOT_CUDA_RUN`: The opposite of `@CUDA_RUN`; it runs a statement only when CUDA is not installed.

While adding GPU support did not significantly increase the overall code complexity, debugging became more challenging, as obtaining sufficiently specific error messages from CUDA can be difficult. Additionally, CUDA code may not be thread-safe when dispatched from multiple CPU threads simultaneously. Despite these challenges, Julia's CUDA library enabled us to write high-level code that achieves high GPU utilization, and, although our code is likely memory-bound<sup>10</sup>, using GPUs still greatly accelerates the training of larger models and has enabled us to run large-scale experiments efficiently.

<sup>10</sup> While GPU utilization approaches 100%, power consumption remains moderate, suggesting that memory access may be the bottleneck. A detailed analysis would be required for a definitive conclusion.

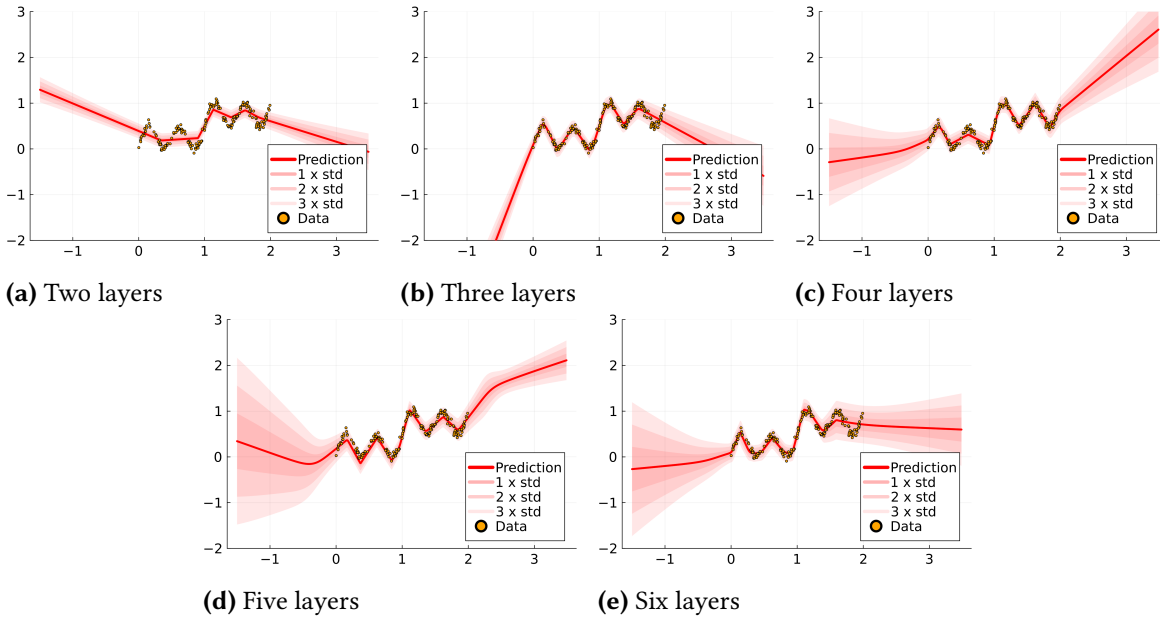


## 5.1 Synthetic Datasets

We begin by presenting results on a synthetic dataset to provide an understanding of the posterior distributions generated by our approach. The dataset is created using an underlying function  $g : \mathbb{R} \rightarrow \mathbb{R}$ , defined as:

$$g(x) = 0.5x + 0.2 \cdot \sin(2\pi x) + 0.3 \cdot \sin(4\pi x).$$

We generate a 1D regression dataset by sampling inputs  $x$  in the range  $[0, 2]$  and collecting observations  $y_i = g(x_i) + \epsilon_i$ , where  $\epsilon_i \sim \mathcal{N}(0, \beta^2)$  represents additive observation noise. For more details on this training dataset and the experimental setups used throughout this chapter, refer to [Chapter C](#).



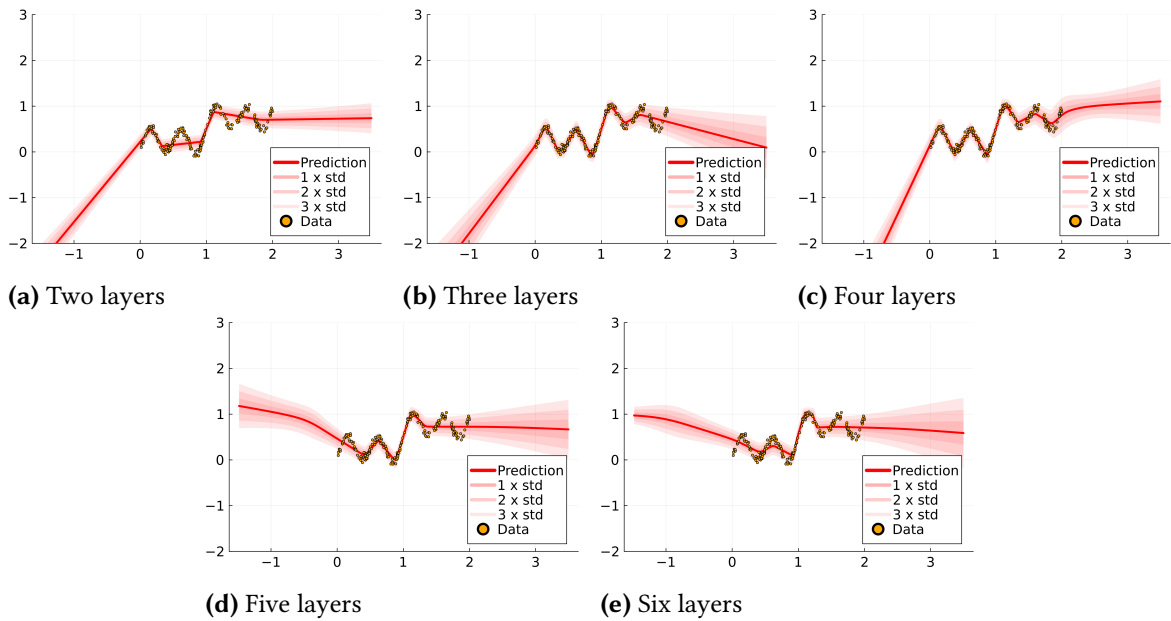
**Figure 5.1:** Predictive posteriors of MLPs with increasing depth. The two-layer MLP has the architecture  $[(1, 16), (\text{LeakyReLU}, 0.1), (16, 1)]$ , with each additional layer adding  $[(16, 16), (\text{LeakyReLU}, 0.1)]$  in the middle. As depth increases, the network becomes more expressive but also harder to fit.

[Figure 5.1](#) shows the results of fitting MLPs of increasing depth to this 1D regression dataset. Since the dataset is small, the FactorGraph contains all training data at once, without batching, and each MLP was trained for 500 iterations<sup>11</sup>. Each subplot shows the predictive posterior of an MLP trained with our MP framework. The dark red line represents the mean prediction, while

<sup>11</sup> Recall that an "iteration" refers to one forward and backward pass for each training branch in the FactorGraph.

the shaded areas show the  $1\sigma$ ,  $2\sigma$ , and  $3\sigma$  intervals of the posterior predictive distribution. The orange scatter points indicate the training data.

The smallest network lacks the complexity required to fit the data properly, while the three-layer MLP already provides a reasonable fit. Adding more layers (to a depth of four or five) improves the fit further and also broadens the predictive uncertainty outside the training range. However, with six layers, the model becomes difficult to train, and achieving a stable fit is challenging. These results align with our findings on traditional (non-Bayesian) MLPs using Torch, as reported in [Ada+24]. Figure 5.2 presents the same experiment but with a different random seed. In this case, the predictive posterior is overly narrow, even for deeper networks that fit the data well (e.g., the four-layer network). This indicates that the predictive uncertainty is sensitive to the random initialization of the network, suggesting the need for a more quantitative evaluation of posterior uncertainty. However, before doing so, we demonstrate the effect of scaling the width of the MLP.

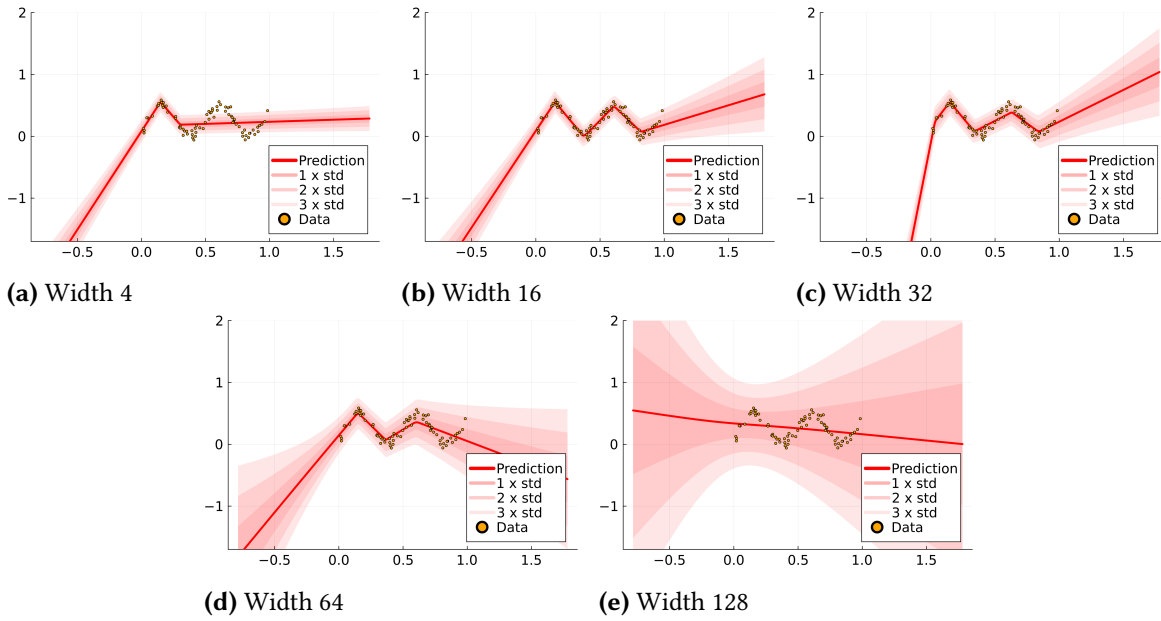


**Figure 5.2:** Predictive posteriors with a different random seed. The setup is identical to Figure 5.1, but the results show a worse fit.

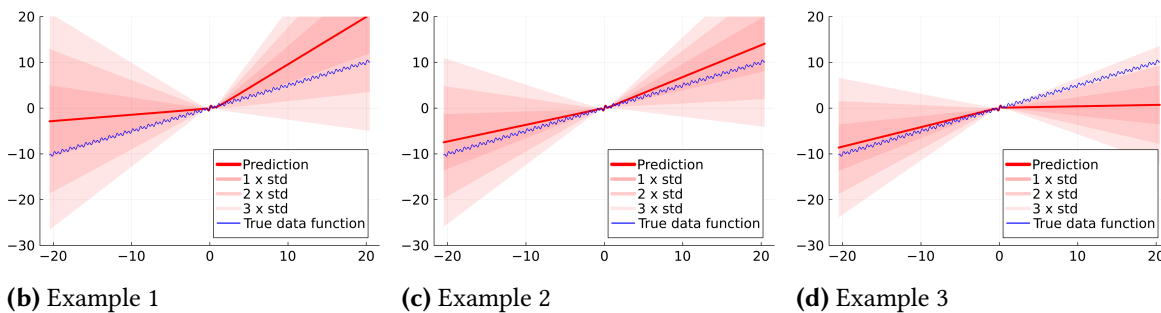
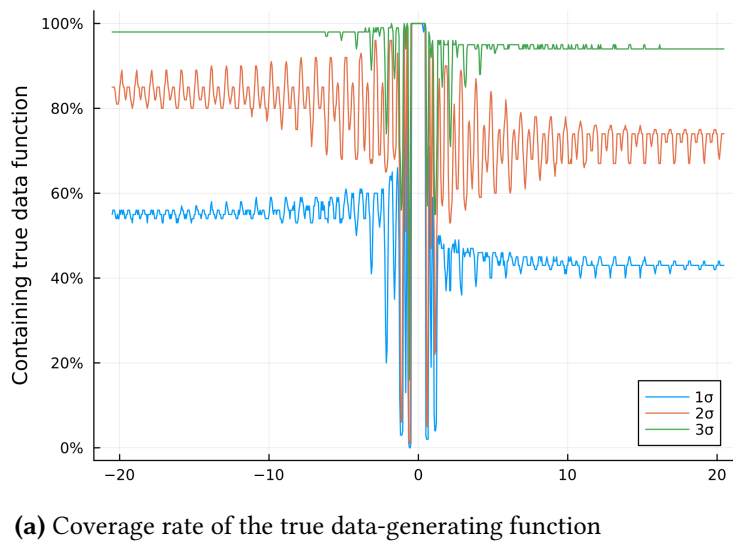
In the width-scaling experiment in Figure 5.3, we again observe that certain architectures provide good fits, while narrower or wider networks struggle to achieve good fit. However, we can directly improve the fit at width 128 by adding more data or double-counting the existing data, whereas simply increasing the dataset size to improve fit has a less pronounced effect in depth scaling.

These experiments build an intuitive understanding of the generated posteriors. Next, we evaluate the predictive uncertainty for out-of-distribution (OOD) data more quantitatively. While we lack a ground truth for the true posterior over network predictions, we compare our uncertainty estimates to the true data-generating function by training 100 networks and measuring how often the true function lies within the  $\sigma$ -intervals of the predictive posterior, as shown in Figure 5.4.

For negative  $x$ , the coverage rates were 55%, 83%, and 98% for the  $1\sigma$ ,  $2\sigma$ , and  $3\sigma$  intervals, respectively. For positive  $x$ , they were 43%, 74%, and 94%. Within the training range ( $x \in (-0.5, 0.5)$ ), all  $\sigma$  intervals covered the true data-generating function. While these results suggest



**Figure 5.3:** Predictive posteriors of a two-layer MLP with increasing width. Narrow networks lack expressiveness, while wider networks require more data to converge to a posterior with a good fit.



**Figure 5.4:** Analysis of OOD uncertainty. We trained 100 three-layer networks with width 32 and measured the percentage of predictive posteriors where a 1/2/3- $\sigma$  interval covers the true data-generating function. In the three examples, the data-generating function is shown in blue.

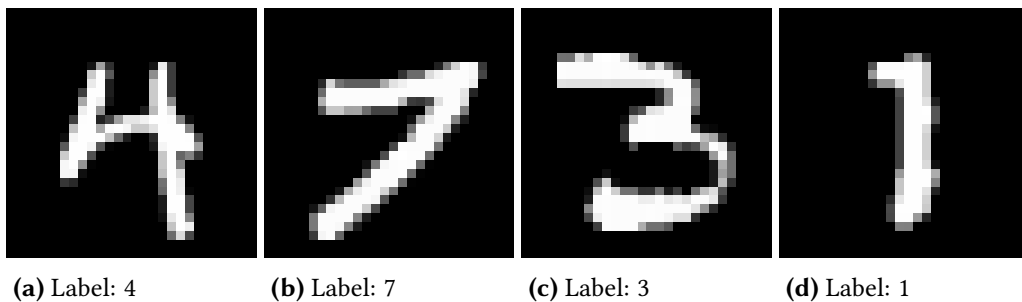
generally good calibration, assessing whether these measurements reliably indicate a true posterior approximation remains challenging.

If the predictive posterior were interpreted as credible intervals, 68% of  $1\sigma$  intervals should cover the data<sup>12</sup>. However, our predictive posterior approximates the predictive uncertainty over networks of the architecture  $f_W$  rather than credible intervals over arbitrary data-generating functions. This is further highlighted by the coverage probabilities when the true data-generating function is sampled from the neural network's own prior: then, 90% of  $1\sigma$  intervals covered the true function, even far outside the training range.

Nonetheless, we further analyzed the correlation between posterior intervals from Figure 5.4 and their coverage rates. Across 101  $p$  values from 0 to 1, the correlation between the  $p$ -credible intervals and their coverage rates was 0.94 overall, with 0.96 for positive  $x$  and 0.98 for negative  $x$ . While we cannot form a definitive conclusion about the accuracy of our predictive posterior without a direct ground truth, these results suggest generally good posterior calibration. Notably, many related approaches (e.g., [Blu+15; Gra11; Osa+19]) did not evaluate posterior behavior for OOD data. Korattikara et al. [Kor+15] compared their method to Hamiltonian Monte Carlo, which is computationally expensive but regarded as a gold standard for ground truth comparisons.

## 5.2 Evaluation on MNIST

We now evaluate our posterior quality on MNIST, a commonly used image dataset for digit classification. It is divided into 60,000 training examples and 10,000 test examples, and each input is a grayscale (single-channel) image of size  $28 \times 28$  showing a digit from 0 to 9. The class label corresponds to the depicted digit. Figure 5.5 shows four representative training examples. For our setup, we normalize the data by subtracting the mean input and dividing by the sample variance<sup>13</sup>. For further details, refer to Chapter C again.



**Figure 5.5:** Training examples from the MNIST dataset.

Although MNIST is a classification dataset, we evaluate all three training layers introduced in Section 3.2. For regression, we use one-hot encodings of the class labels. We set the batch size to 320 and run the training for 90 iterations over 18 epochs using a "staircase" pattern. Specifically,

<sup>12</sup> Even under this interpretation, we would need to generate  $n$  true data functions and train one network per function for a more accurate comparison.

<sup>13</sup> The test inputs are normalized using the mean and variance of the training data, not the mean and variance of the test data.

in the first two epochs, we perform a single forward-backward pass for each training example before moving to the next batch. In the following two epochs, we iterate through each batch twice before continuing, and so on. The number of iterations per epoch increases every two epochs, starting at 1 and ending at 9. This setup has performed well in our experiments, perhaps because it allows the model to quickly process all data a few times in early epochs and then achieve finer convergence in later epochs.

We set the hyperparameters as follows: for regression,  $\beta^2 = 0.01$ ; for argmax, noise variance  $\beta^2 = 0.3$  and the regularization variance  $\gamma^2 = 0.1$ . Softmax requires no additional hyperparameters. These values were selected through small-scale test runs without extensive hyperparameter tuning, as the training appeared robust across a wide range of settings. For comparison, we trained deterministic models using stochastic gradient descent (SGD) in PyTorch (Torch) [Pas+19], using softmax with negative log-likelihood loss and regression (one-hot encoding) with mean squared error loss. Here, we also only applied minimal hyperparameter tuning and kept the SGD learning rate at its default value of 0.001.

	Num Data	80	160	320	640	1,280	2,560	5,120	10,240	60,000
3-layer MLP	<b>R-MP</b>	30.01	<b>61.79</b>	77.61	<b>85.69</b>	<b>88.95</b>	<b>91.72</b>	<b>94.85</b>	<b>96.25</b>	<b>98.33</b>
	<b>AM-MP</b>	<b>31.88</b>	61.08	<b>80.79</b>	85.50	87.92	91.56	94.08	95.72	98.21
	<b>AMNR-MP</b>	15.61	10.10	10.28	9.82	10.10	8.92	9.80	-	-
	<b>SM-MP</b>	27.70	28.21	37.97	77.41	85.75	89.20	91.23	-	-
	<b>R-SGD</b>	10.11	11.45	14.89	29.66	49.78	67.01	76.41	83.00	92.22
	<b>SM-SGD</b>	21.47	30.38	46.18	58.83	76.67	85.55	89.10	91.17	96.36
LeNet-5	<b>R-MP</b>	<b>27.75</b>	<b>25.58</b>	<b>38.02</b>	<b>94.72</b>	95.36	96.32	97.40	<b>98.12</b>	<b>99.02</b>
	<b>AM-MP</b>	17.32	10.42	10.28	93.48	<b>96.19</b>	<b>96.44</b>	<b>97.70</b>	98.05	98.95
	<b>AMNR-MP</b>	11.35	10.10	-	-	-	-	-	-	-
	<b>SM-MP</b>	9.11	10.10	10.28	8.92	10.32	10.10	10.32	-	-
	<b>R-SGD</b>	14.06	14.51	14.07	13.99	16.02	31.16	49.43	69.84	94.12
	<b>SM-SGD</b>	18.57	19.54	21.03	22.15	39.36	82.30	90.92	95.04	98.55

**Table 5.1:** Comparison of test accuracies on MNIST (percent correct). Our method (MP) consistently achieves better accuracy than SGD (implemented in Torch). The abbreviations stand for Regression (R), Argmax (AM), Argmax-No-Regression (AMNR), and Softmax (SM).

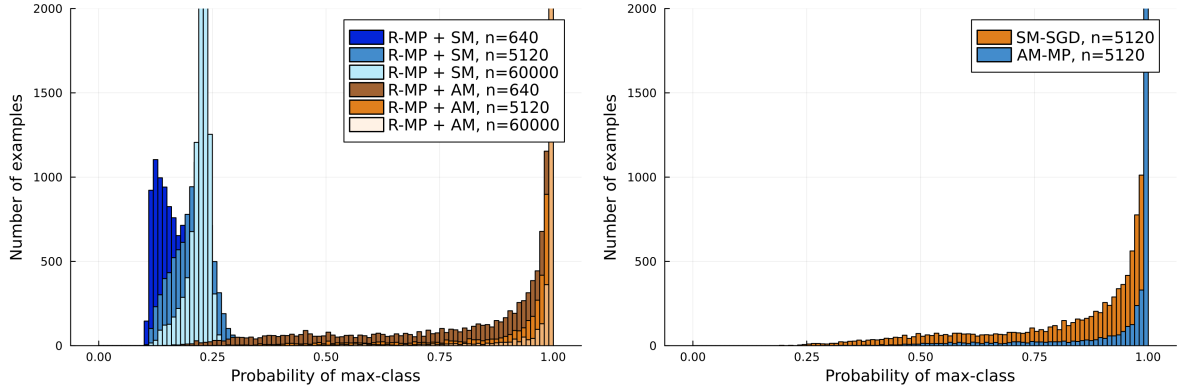
Our experiments use two different architectures:

1. 3-layer MLP: The architecture is [Linear(1, 256), LeakyReLU(0.1), Linear(256, 256), LeakyReLU(0.1), Linear(256, 10)].
2. LeNet-5 [Lec+98]: A simple CNN network with the architecture [Conv(5x5, 1→6), LeakyReLU(0.1), MaxPool(2x2), Conv(5x5, 6→16), LeakyReLU(0.1), MaxPool(2x2), Linear(400, 120), LeakyReLU(0.1), Linear(120, 84), LeakyReLU(0.1), Linear(84, 10)].

For each architecture and training approach, we compare the performance over various training dataset sizes ranging from 80 randomly chosen images to the full dataset of 60,000 training images. Table 5.1 shows the test accuracies after training. The highest accuracy achieved in this experiment is 99.02% with LeNet-5 trained using R-MP (see Table 5.1 for the legend of abbreviations). Even

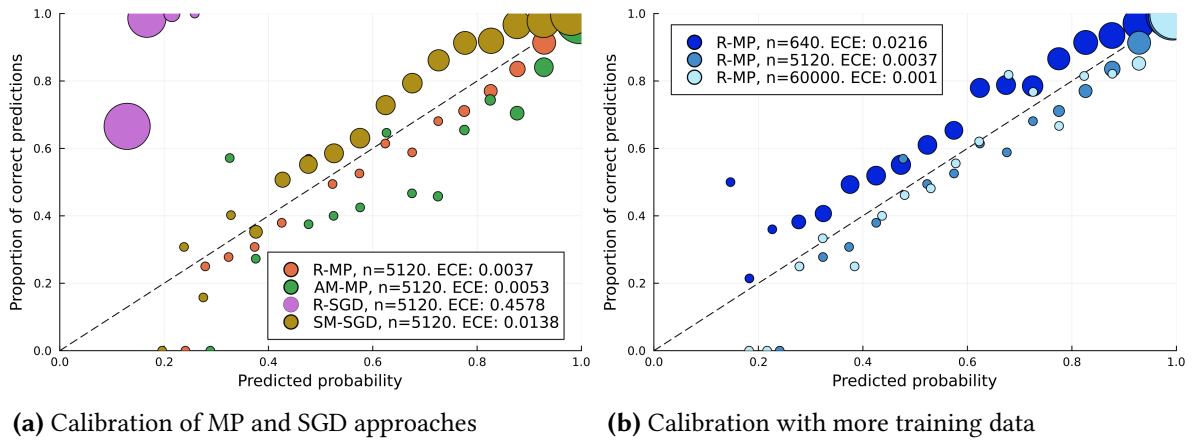
on accuracy alone, our MP framework outperforms its SGD counterparts, especially in data-constrained settings.

The results also show that our message passing framework successfully trains CNNs to achieve higher accuracy than MLPs. Notably, SM-MP and AMNR-MP crashed with larger datasets, likely due to numerical instability when a prediction is overconfident in incorrect cases. In contrast, enhancing argmax with regression labels (AM-MP) appears numerically stable and performs well. In the following, we will therefore focus on comparing R-MP, AM-MP, R-SGD, and SM-SGD.



**Figure 5.6:** Histograms of max-class probabilities for different approaches. Applying softmax to the predictions of a net trained with regression loss leads to overly uncertain predictions. In contrast, applying argmax yields probabilities with a distribution similar to AM-MP. The probabilities of SM-SGD are also similar but slightly less peaked.

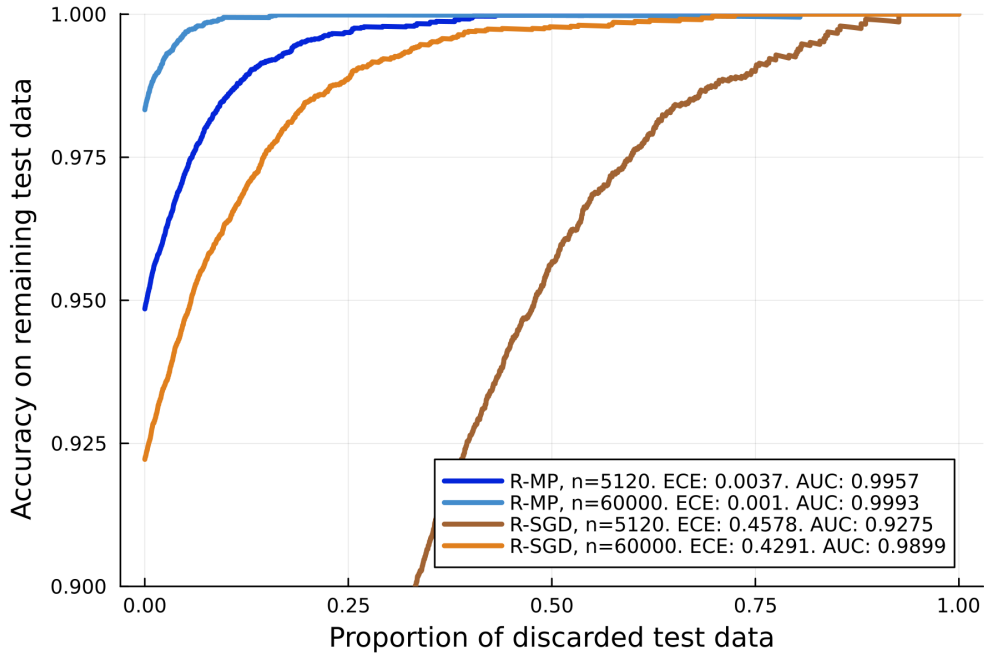
If we want to compare the predictions from the two regression models (R-MP and R-SGD) with other approaches, we need to transform their predicted class scores into a discrete probability distribution over classes. For R-MP, we applied either the softmax forward message or the argmax forward message to the predicted class scores. Figure 5.6 shows that argmax recovers probabilities with a distribution similar to models trained with classification-based losses. However, our argmax forward message cannot be applied to the SGD regression model, as it requires the posterior predictive over class scores to approximate the argmax probability distribution. Therefore, R-SGD’s class scores are transformed using (non-probabilistic) softmax, while R-MP leverages (probabilistic) argmax.



**Figure 5.7:** Calibration curves for different approaches. Each point represents the accuracy in one of 20 bins with uniform spacing, while its size indicates the bin size. The expected calibration error (ECE) is computed using 20 dynamic bins obtained by iteratively splitting the bin with the highest variance.



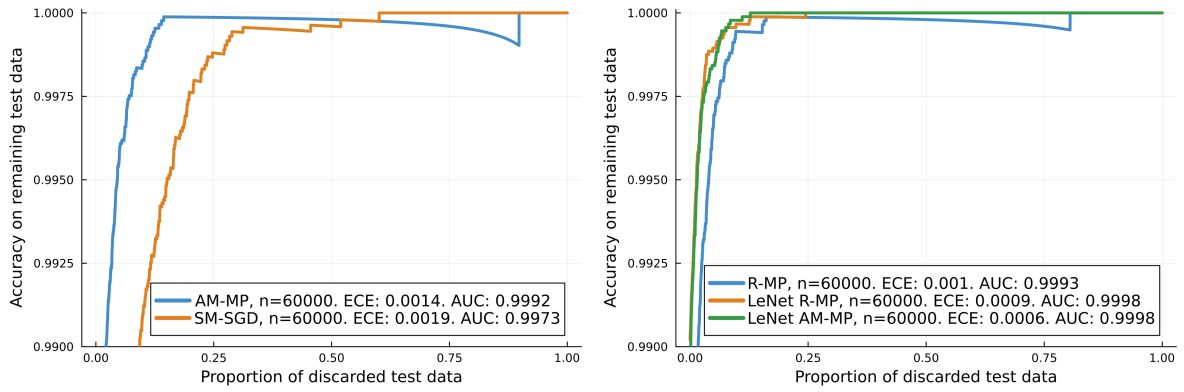
The advantage of R-MP over R-SGD is clearly reflected in their calibration plots, as shown in Figure 5.7 (a). With an expected calibration error (ECE) of 0.0037, R-MP even has a better calibration error than AM-MP (0.0053) and SM-SGD (0.0138). In contrast, R-SGD has an ECE of 0.4578 and never predicts a probability higher than 0.35. Figure 5.7 (b) shows how the ECE improves with more training data. However, examining the calibration curves over the full range  $[0, 1]$  can be misleading, as 90% of the predicted probabilities are over 0.87, and 50% are even over 0.995 (for R-MP at  $n = 5, 120$ , see also Figure 5.6). For this reason, plotting relative calibration can be more insightful. These plots address the question, "Given two examples, is the one with more uncertainty truly more likely to be incorrect?" Figure 5.8 shows a relative calibration plot and describes the method used to create it.



**Figure 5.8:** Relative calibration plots for regression models. Each point in the graph represents the accuracy on the remaining test data after discarding  $x\%$  of predictions with the highest uncertainty (lowest predicted max-class probability). The AUC score is the area under the relative calibration curve.

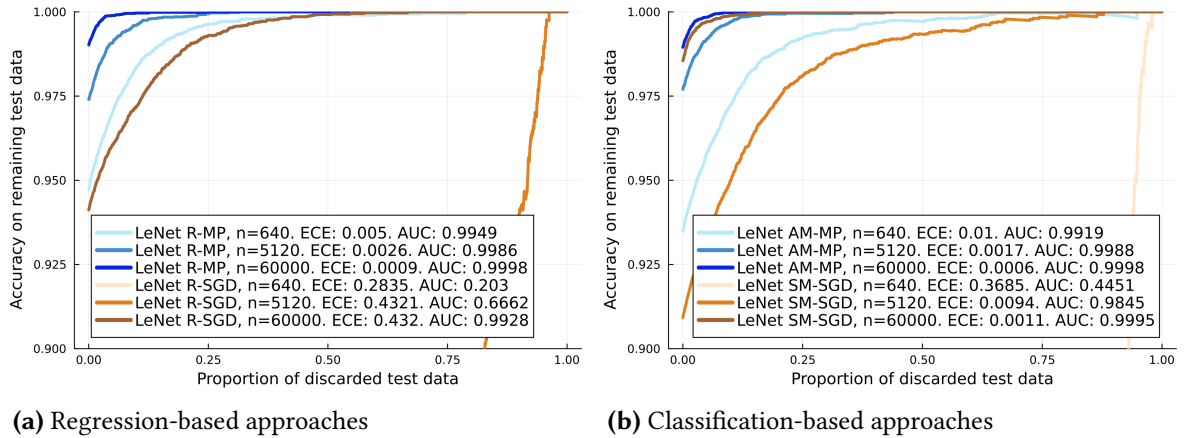
The relative calibration curve of R-SGD in Figure 5.8 reveals that the posterior uncertainties do provide useful information, despite the high ECE and low predicted max-class probabilities. While the overall accuracy of R-SGD at  $n = 5, 120$  is 76.41%, discarding the half of predictions with higher predictive uncertainty increases accuracy to over 95%. For R-MP, the accuracy rises quickly to almost 100% with an AUC of 0.9993, whereas R-SGD only achieves 0.9899. The difference is smaller for AM-MP (AUC 0.9992) and SM-SGD (AUC 0.9973), but MP consistently outperforms SGD across all our experiments.

An interesting property that is visible in the calibration plots is that the MLPs trained with MP exhibit a few overly certain wrong predictions—more so than SGD. Figure 5.9 shows the resulting dip in the relative calibration curve, albeit at a  $y$ -scale close to 1. For ideal relative calibration, the list of predictions sorted by uncertainty would begin with all incorrect predictions and end with all correct ones. While predictions from all models contain out-of-order pairs, R-MP and AM-MP on the MLP are particularly affected. In contrast, the relative calibration curves for LeNet-5 models with MP are excellent (see Figure 5.9).



**Figure 5.9:** Relative calibration plots. Both R-MP and AM-MP are overly certain on a few incorrect predictions, leading to a decreasing relative calibration curve for overconfident examples. SM-SGD shows only minor declines but still achieves a lower AUC score. Message passing with LeNet-5 does not exhibit this issue and achieves an AUC score very close to 1 (0.9998).

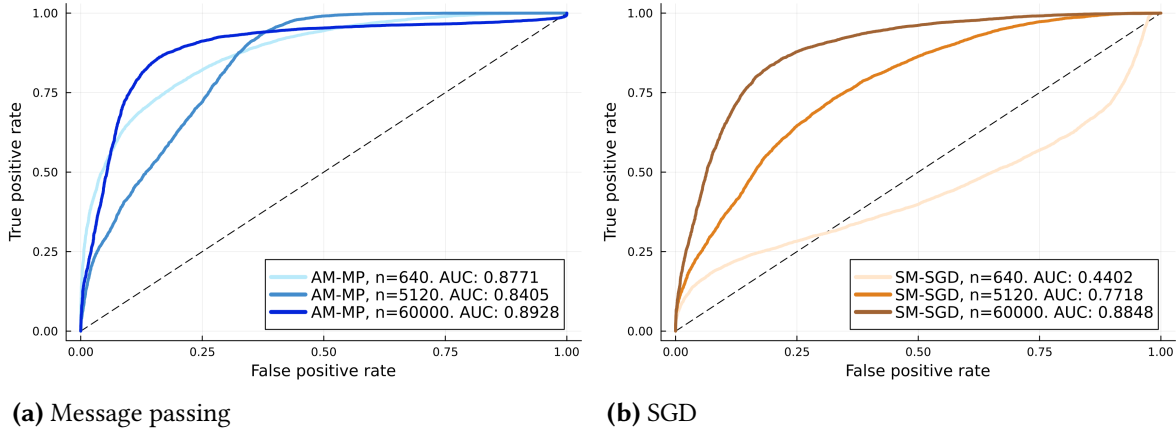
Figure 5.10 compares the relative calibration curves of MP and SGD for LeNet-5. Similar to the MLPs, the regression models trained with SGD perform poorly. For  $n = 640$ , the R-SGD curve is not even visible within the chosen  $y$ -range, while R-MP achieves a better AUC at  $n = 640$  than R-SGD at  $n = 60,000$ . For classification, SM-SGD at  $n = 60,000$  is only slightly behind AM-MP, but AM-MP achieves significantly higher AUC scores for smaller dataset sizes. For example, at  $n = 640$ , AM-MP achieves an AUC of 0.9919 while SM-SGD reaches only 0.4451.



**Figure 5.10:** Relative calibration plots for LeNet-5. Regression models perform significantly better (on classification datasets) with MP than with SGD. For classification, AM-MP achieves slightly better relative calibration than SM-SGD at  $n = 60,000$  but is significantly better for smaller training sizes.

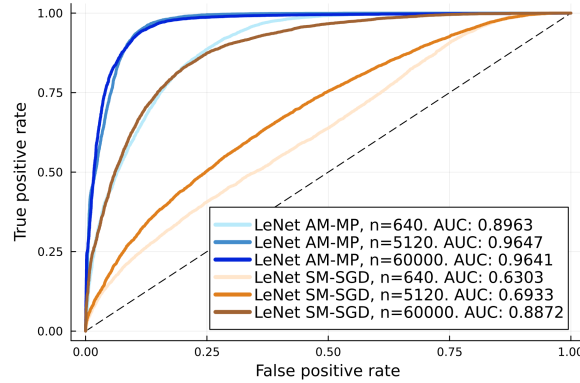
Lastly, we evaluate the out-of-distribution (OOD) recognition of the MNIST models using FashionMNIST as the OOD dataset. Its input format and dataset size are identical to MNIST, but the images show clothing items of 10 different classes. After training a model on MNIST's training data, we collect its predictions on the combined test sets of MNIST and FashionMNIST. We then classify each prediction as MNIST vs. FashionMNIST by thresholding the negative entropies of all predictions. Since FashionMNIST images are entirely different from MNIST digits, we expect the predictive distribution of the trained digit classifier model to be more uniform for FashionMNIST images than for MNIST digits. Figure 5.11 shows the receiver operating characteristic (ROC) curve,

which compares the true positive and false positive rates at different thresholds. If each decision were random, the ROC curve would resemble a straight line from (0, 0) to (1, 1).



**Figure 5.11:** Comparison of ROC curves for out-of-distribution recognition. MP performs substantially better than SGD at small dataset sizes and slightly better for  $n = 60,000$ .

For the MLP model in Figure 5.11, MP achieves an OOD-recognition AUC score of 0.8771 at  $n = 640$ , compared to 0.4402 for SM-SGD. As more data is added, the advantage becomes smaller, but AM-MP still shows better OOD recognition at  $n = 60,000$  than SM-SGD (AUC of 0.8928 vs. 0.8848). Interestingly, SM-MP’s AUC score at  $n = 5,120$  is lower than at either smaller or larger dataset sizes. While this phenomenon affects both R-MP and AM-MP, we did not investigate it further. For LeNet-5 models, OOD recognition improved consistently with more data, and AM-MP’s AUC score (0.9641) is considerably higher than SM-SGD’s (0.8872) even at  $n = 60,000$ . Figure 5.12 shows the ROC curves at different training dataset sizes.



**Figure 5.12:** ROC curves for out-of-distribution recognition with LeNet-5. Unlike for the MLPs, AM-MP performs considerably better than SM-SGD for all dataset sizes, even at  $n = 60,000$ .

While AM-MP already outperforms SM-SGD (and R-SGD) for OOD recognition, R-MP performs even better. When trained on the full dataset, R-MP achieved an AUC score of 0.9018 with the MLP and 0.9675 with the CNN. Although this advantage over AM-MP is small, it is consistent across all MNIST experiments.

Overall, our results demonstrate that our MP framework achieves better calibration, relative calibration, and out-of-distribution recognition than SGD while maintaining similar or better

accuracy. MP models performed especially well when dataset sizes were limited, where R-MP and AM-MP significantly outperformed SGD-based methods in all measured metrics. While AM-MP shows slightly lower performance than R-MP, it still serves as a robust alternative to softmax training (SM-MP). Interestingly, both MP models showed a slight tendency towards overconfidence in rare cases, as observed in the relative calibration curves of the MLP models in Figure 5.9. However, this issue was not observed in the CNN (LeNet-5) models, which achieved near-perfect relative calibration and excellent out-of-distribution recognition.

### 5.3 Comparison to Related Work

After comparing our MP approach against SGD, we attempt to replicate two experimental setups from other studies. Blundell et al. [Blu+15] compared their VI algorithm, Bayes By Backprop (BBB), against SGD by training MLPs on MNIST using a three-layer architecture with hidden layers of varying widths (400, 800, 1200). We applied R-MP and AM-MP to train MLPs with the same architecture, although we used LeakyReLU(0.1) instead of ReLU as the activation function. Table 5.2 shows the results of Blundell et al. [Blu+15] and our experiments. Since accuracy was the only reported performance metric, a comprehensive comparison of posterior quality is not possible. Nevertheless, the results show that our R-MP approach achieves similar or better accuracy than both their SGD and BBB with Gaussian priors, while BBB with a mixture prior

Model	Width of Hidden Layers	Num Parameters	Accuracy	AUC
R-MP	400	500k	98.21	0.9994
	800	1.3m	98.13	0.9993
	1200	2.4m	98.11	0.9994
AM-MP	400	500k	98.11	0.9993
	800	1.3m	97.85	0.9989
	1200	2.4m	97.91	0.9992
SGD	400	500k	98.17	-
	800	1.3m	98.16	-
	1200	2.4m	98.12	-
BBB, Gaussian	400	500k	98.18	-
	800	1.3m	98.01	-
	1200	2.4m	97.96	-
BBB, Mixture	400	500k	98.64	-
	800	1.3m	98.66	-
	1200	2.4m	98.68	-
BPL	501	650k	97.4	-

**Table 5.2:** Classification accuracies on MNIST, presented similarly to Blundell et al. [Blu+15]. The results for SGD and BBB (with two different priors) are taken from their work. The BPL row contains results from the belief propagation algorithm by Lucibello et al. [Luc+22]. Our R-MP results match or exceed those of SGD and BBB with Gaussian priors, while BBB with mixture priors achieves slightly higher accuracy. BPL achieved the lowest accuracy among the compared methods.

achieves slightly higher accuracies. Additionally, we included results from the belief propagation algorithm proposed by Lucibello et al. [Luc+22], referred to as BPL here. With an accuracy of 97.4%, BPL performs slightly but consistently worse than all other methods in this comparison. Interestingly, increasing the width beyond 400 did not result in relevant performance gains for any of the methods.

To enable a broader comparison beyond just accuracy, we selected a second benchmark: the CIFAR-10 dataset. This dataset consists of 3-channel color images across 10 classes (animals and vehicles), making it significantly more challenging than MNIST. We applied R-MP and AM-MP to CIFAR-10 using the LeNet-5 architecture and compared our results with those of Adam, BBB, and VOGN, as reported in Osawa et al. [Osa+19]. As shown in Table 5.3, R-MP and AM-MP underperform compared to these methods. Since we reused hyperparameters and the training schedule from our MNIST experiments without further optimization, this underperformance suggests that further tuning, numerical improvements, or longer training time would be needed to match the performance of VOGN on CIFAR-10.

Optimizer	Train/Val Accuracy (%)	Validation NLL	Epochs	Time/epoch (s)	ECE	AUC
R-MP	56.95 / 49.67	1.847	90	(118.2)	0.105	0.669
AM-MP	58.03 / 51.56	2.351	90	(185.1)	0.106	0.688
Adam	71.98 / <b>67.67</b>	<b>0.937</b>	210	(6.96)	<b>0.021</b>	0.797
BBB	66.84 / 64.51	1.018	800	(11.43)	0.045	0.784
VOGN	70.79 / <b>67.32</b>	<b>0.938</b>	210	(18.33)	0.046	<b>0.800</b>
BPL (MLP)	- / 41.3	-	100	-	-	-
R-MP (MLP)	52.12 / 46.76	1.871	90	-	0.094	0.641
AM-MP (MLP)	51.77 / 47.50	2.489	90	-	0.140	0.644

**Table 5.3:** Performance comparison on CIFAR-10 using LeNet-5. The results for Adam, BBB, and VOGN were taken from Osawa et al. [Osa+19]. BBB stands for Bayes By Backprop, as presented in Blundell et al. [Blu+15]. In comparison, our approaches (R-MP and AM-MP) show underfitting but still perform better than BPL by Lucibello et al. [Luc+22] (BPL). The runtime measurements are not directly comparable due to different parallelization setups.

We also included results from Lucibello et al. [Luc+22], labeled as BPL. Since BPL only works with MLPs, we trained a comparable MLP with 3 layers and a width of 501. The results show that both R-MP and AM-MP outperform BPL in accuracy, the only metric reported for BPL.

Finally, to test the scalability of our framework, we trained a larger MLP model with 5.6 million parameters on MNIST, achieving an accuracy of 98.04%, similar to the smaller MLPs. While this is not a direct comparison to related work, it demonstrates our MP framework’s capability to handle significantly larger MLPs than those tested by Lucibello et al. [Luc+22]. In the next section, we evaluate the runtime performance of our framework in more detail.

## 5.4 Runtime Analysis

In addition to improving posterior quality, optimizing the runtime performance of our FactorGraph has been a major focus of our work. At a minimum, this optimization was essential for running experiments on MNIST, as the original implementation was too slow for practical use. [Section 4.4.2](#) already contains a detailed description of our performance improvements; here, we compare the runtime of our current implementation to the original FactorGraph from the MP report [[Ada+24](#)]<sup>14</sup>. For more details about the experimental setup, we again refer to [Chapter C](#).

As shown in [Table 5.4](#), our new implementation achieves up to 9x faster performance on CPU for individual messages on linear layers. Additionally, our implementation leverages GPUs effectively, yielding up to a 62x speedup compared to the old version. Notably, the training signals (e.g., softmax and argmax) do not show similar GPU speedups, which is expected. The softmax implementation relies on numerical integration, which is slower on GPU because the computation is performed on the CPU anyways but the data must be transferred to and from the GPU. For the argmax operation, the slowdown is caused by a scalar array access, which also requires data transfer to the CPU. While this array access is likely avoidable, the increase in runtime ( $\leq 2.7$  ms) is minimal and scales slowly with the number of classes. For example, the argmax layer took 2.4 ms on CPU and 3.4 ms on GPU for 1,000 classes.

Layer	Old (CPU)	New (CPU)	New (GPU)
FirstGaussianLinearLayer	817.0 ms	90.3 ms	13.0 ms
GaussianLinearLayer	871.0 ms	147.8 ms	23.9 ms
LeakyReLU	5.8 ms	3.9 ms	3.1 ms
FirstConv2d	-	40.8 ms	9.1 ms
Conv2d	-	102.1 ms	15.5 ms
MaxPool2d	-	10.6 ms	5.7 ms
Regression	0.0 ms	0.0 ms	0.1 ms
Softmax	-	12.5 ms	15.2 ms
Argmax	-	0.2 ms	2.7 ms

**Table 5.4:** Forward-backward passes over 10 examples. The new version is up to 9x faster on CPU and 62x faster on GPU compared to the old version from the MP report [[Ada+24](#)]. Input sizes were similar to MNIST: 784 for the first group,  $28 \times 28 \times 4$  for the second group, and 100 for the last group.

The results already show substantial improvements for isolated forward-backward passes in linear layers. However, the runtime difference is even more pronounced for end-to-end training on MNIST. [Table 5.5](#) shows that the old approach is much slower than the layer-wise performance suggests. In this experiment, there are 3,200 training examples and two forward-backward passes, which require 640x the amount of computation as in the previous experiment in [Table 5.4](#). However, the old version is 25x slower than expected by extrapolating the layer-wise results<sup>15</sup>. In contrast,

<sup>14</sup> The version tested here is not identical to the one in the MP report; it is a snapshot from week 3 or 4 of my master’s thesis. The primary changes relate to numerical stability and enabling the code to handle MNIST-sized experiments. Unlike the original, we also test this version with multi-threading enabled, though it is not optimized for it. Without multi-threading, the runtime would only be 40-50 ms higher.

<sup>15</sup> The runtime is dominated by the FirstGaussianLinearLayer, so we extrapolate the expected runtime based on its isolated performance. As the first experiment used Linear(784, 1000) while the current experiment uses Linear(784, 100) with 640x more forward-backward passes, we estimate the runtime as  $817 \text{ ms} \times \frac{640}{10}$ .



the new version is only 1.5x slower than its extrapolated estimate. Consequently, the gap in runtime widens for the MNIST benchmark: the new version is now up to 309x faster. Table 5.5 also indicates that memory usage was a key bottleneck in the old approach, as it allocates (and deallocates) 1.3 TiB of memory, causing significant overhead and noticeable pauses due to garbage collection. In contrast, the new version allocates just 110.8 MiB.

Layer	Old (CPU)	New (CPU)	New (GPU)	Torch (CPU)	Torch (GPU)
Runtime	1297.8 s	8.9 s	4.2 s	0.8 s	0.7 s
Memory	1.3 TiB	110.8 MiB	500.2 MiB	-	-

**Table 5.5:** Two forward-backward passes (or epochs for Torch) on 3,200 training examples from MNIST. The new version is up to 309x faster due to its improved layer performance (Table 5.4) and vastly reduced memory usage. The Torch implementation is 6-11x faster than MP.

Table 5.5 also reveals that SGD in Torch remains 6-11x faster than our FactorGraph. However, while the measured runtime is lower, the runtime complexity remains the same. Each linear algebra factor has a complexity linear to the number of atomic addition and multiplication factors used, and the message equations for each atomic factor have a constant runtime<sup>16</sup>, which is significantly higher than an actual addition or multiplication. The LeakyReLU, regression, and argmax forward and backward messages also scale linearly with  $n$  inputs ( $O(n)$ ). The same applies to softmax, but with a particularly high constant cost due to the numerical integration used.

Despite having the same runtime complexity as Torch, training with MP remains slower overall. Furthermore, the runtime of Torch is nearly identical on CPU and GPU in Table 5.5, suggesting that the experiment size is too small for Torch to fully utilize the GPU. To better understand the impact of scaling we therefore conducted another experiment with LeNet-5, a slightly larger network. The results are shown in Table 5.6.

	MP (CPU)	MP (GPU)	Torch (CPU)	Torch (GPU)
Training	261.4 s	96.4 s	3.3 s	2.3 s
Inference FP64	464.3 s	72.4 s	-	-
Inference FP32	392.7 s	26.5 s	17.6 s	15.9 s

**Table 5.6:** Comparison of runtimes for LeNet-5, trained on 3,200 examples over 6 epochs. Inference measures the time taken to predict the classes of 200,000 examples. The FactorGraph only implements FP64 training but supports inference in both FP64 and FP32. Torch was only tested in FP32.

We estimate the computation required for Table 5.6 to be approximately 67.5x greater than that for Table 5.5 based on the following rough estimate. We only consider linear algebra operations, as these are the most computationally expensive (see Table 5.4). The computational cost of a CNN layer with output size  $d_w \times d_h \times d_{f_2}$  and kernel size  $k \times k \times d_{f_1}$  is estimated as  $k^2 \cdot d_{f_1} \cdot d_w \cdot d_h \cdot d_{f_2}$ , while for a linear layer with input size  $d_1$  and output size  $d_2$ , the cost is  $d_1 \times d_2$ . Using these estimates, LeNet-5 requires approximately 1,787,400 operations per forward or backward pass, while the

<sup>16</sup> While our implementation does not contain atomic factors, they simplify runtime reasoning. Since each addition or multiplication has constant costs, higher-order multiplication factors maintain similar complexity as their traditional counterparts.

MLP requires only 79,400 operations. Since the LeNet-5 experiment performs 6 iterations instead of 2, the total cost estimate becomes  $3 \cdot \frac{1,787,400}{79,400} \approx 67.5$  times higher.

As shown in Table 5.6, the runtime of our approach is roughly consistent with this estimate, increasing by 23-29x over Table 5.5, which falls within the expected order of magnitude. However, the runtime of Torch increased by only a factor of 3-4, suggesting that much of the increased computation is offset by more efficient parallelization on the GPU. Another factor is the difference in floating-point precision: our approach uses Float64 (FP64), while Torch uses Float32 (FP32) by default. Since GPUs can be 32-64x slower for FP64 operations [Tecnda; Tecndb], this likely has a significant effect on the performance comparisons, which is not accounted for in this experiment.

To further explore this hypothesis, we developed an FP32 version of our MP framework, although only for inference. As shown in Table 5.6, the FP32 version is approximately 3x faster than the FP64 version on GPU, while the CPU runtime is reduced by around 15%. Although this 3x speedup is significant, it falls short of the 32x speedup suggested by the GPU performance data [Tecnda; Tecndb]. However, the comparison with Torch shows that the 26.5 s runtime is likely near the minimum achievable for MP, as it is only 1.7x slower than Torch despite working with twice the number of parameters (mean and variance). It is also important to note that the experiment size may still be too small for Torch to fully leverage the GPU's capabilities.

In conclusion, while larger-scale experiments would be needed for a comprehensive conclusion, the scaling behavior of our approach is promising. We have demonstrated that the overhead for probabilistic inference with our MP framework is relatively small once the posterior over weights has been approximated, which is especially important for deploying inference on consumer or edge devices. Overall, our results show competitive runtimes for FP32 inference and acceptable runtimes for training, which suggests that further optimizations could yield even greater speedups that could potentially narrow the training performance gap with Torch.



## 6.1 Summary

Traditional deep learning techniques, such as SGD, produce only point estimates for the weights of a neural network, which do not account for epistemic uncertainty (uncertainty over parameters) and its effect on predictions. This makes it challenging to differentiate between predictions strongly supported by the training data and those unconstrained by the likelihood, such as hallucinations in large language models. BNNs aim to capture this uncertainty by approximating a posterior distribution over parameters or predictions using methods such as VI. However, despite recent advances like IVON [She+24], VI is generally believed to produce overly confident posterior approximations [Zha+18] and often requires complex hyperparameter tuning [Osa+19].

In this work, we introduced a novel approach based on approximate MP in a fully factorized factor graph that represents the predictive posterior distribution. We described the theoretical foundation of our approach, derived approximate message equations, outlined our implementation, and discussed the optimizations and improvements made for numerical stability and performance. Compared to the MP report [Ada+24], our implementation is 309 times faster and we have expanded our framework from feature-function (one-layer) regression on one-dimensional toy data to CNNs on MNIST with 60,000 training images. After training, the final model has only twice the number of parameters as a conventional model and achieves inference runtimes just 1.7 times slower than Torch. However, a significant gap remains in training costs, likely due to more effective parallelization in Torch.

In our experiments on the MNIST dataset, we demonstrated that our approach outperforms SGD in terms of accuracy (98.33% vs. 96.36%), calibration (ECE of 0.0037 vs. 0.0138), relative calibration (AUC of 0.9993 vs. 0.9973), and out-of-distribution recognition (AUC of 0.9018 vs. 0.8848) for a 3-layer MLP. We extended our approach to CNNs and successfully trained the LeNet-5 architecture on MNIST, achieving a test accuracy of 99.02% compared to 98.55% for SGD, alongside improved uncertainty measures. Specifically, the out-of-distribution recognition AUC for LeNet-5 was much higher with our approach: 0.9675 for R-MP versus 0.8872 for SM-SGD.

Our method proved particularly effective with limited training data. We compared MP against SGD on eight increasingly larger subsets of the MNIST training data. On a subset of 640 randomly chosen examples, LeNet-5 trained with MP achieved 94.72% accuracy and a relative calibration AUC of 0.9949, while SGD only reached 22.15% accuracy and a 0.4451 relative calibration AUC. This aligns with the expectation that Bayesian methods are more effective when data is scarce. We also found that our approach performs well on MNIST without significant hyperparameter tuning. Coupled with its strong performance in low-data scenarios, this makes MP a promising technique for end devices that require learning without human intervention.

Interestingly, the regression-based training factor (R-MP) outperformed the classification-based factors (AM-MP and SM-MP) in all experiments except for CIFAR-10. In the CIFAR-10 experiment, we compared our approach against Adam (Torch) and two VI techniques (BBB and VOGN [Blu+15; Osa+19]), finding that MP achieved lower accuracy than the other three methods (51.56% for MP compared to 64.51-67.67% for the others). However, CIFAR-10 was the largest dataset we tested, and our evaluation of MP was based on a small number of trials. Further tuning, numerical improvements, or longer training might have improved the results to the level of other approaches. Nonetheless, MP already performed better on CIFAR-10 than the belief propagation approach by Lucibello et al. [Luc+22]. On MNIST, BBB’s performance was comparable to ours.

We also investigated the quality of posterior approximations using a synthetic dataset to explore how the posteriors change as network width and depth increase. Our experiments suggest that there is an optimal model complexity for a given dataset: smaller models fail to capture the required complexity, while deeper models, although more expressive, are harder to fit—a challenge also seen with traditional MLPs in Torch [Ada+24]. Wider models also underfit but show rapid improvement when more training data is added (or existing data is double-counted). We also conducted a quantitative evaluation of the posterior uncertainty outside the training range and found that our posteriors are generally well-calibrated. Although we lack a ground truth for the exact posterior, we observed a strong correlation (0.94) between posterior credible intervals and their probability of covering the data-generating function.

Compared to previous MP-based approaches, our framework has demonstrated the largest BNNs trained with MP to date. Soudry et al. [SHM14] applied their EBP method to 3-layer MLPs with sign activation functions, and Hernández-Lobato and Adams [HA15] used PBP on small regression datasets. Both approaches also rely on gradients instead of pure message passing. Lucibello et al. [Luc+22] were the first to train neural networks on MNIST using only MP, similar to our work, but our method achieves higher accuracy (98.21% vs. 97.4%) and provides a comprehensive evaluation of uncertainty, whereas Lucibello et al. [Luc+22] only report accuracy. Furthermore, we demonstrate that our approach works effectively on CNNs with continuous weights, while they only present experiments with MLPs that have binary weights and activations. They also reported that their posterior collapses to a point estimate, which negates many of the advantages of Bayesian models [Luc+22]. Our approach is therefore the first to successfully train both MLPs and CNNs at this scale using MP in factor graphs. Overall, our work addresses the two main challenges in MP (deriving message equations and building scalable, bug-free implementations) as we created a comprehensive library of approximate message equations and demonstrated a scalable, effective Julia implementation that we tested on CNNs and MLPs with up to 5.6 million parameters.

## 6.2 Limitations and Future Work

There are several limitations that we want to address. Most importantly, our runtime performance remains slower than Torch’s, even after improving the previous version by a factor of 309. In Table 5.5, Torch is still 6-11 times faster than MP. While this gap is somewhat expected, given that our approach was implemented from scratch in Julia and Torch is a heavily optimized mainstream machine learning library, the results from Table 5.6 indicate that our method still

lags behind significantly in terms of parallelization efficiency. However, despite the performance improvements we’ve achieved, further gains are plausible. For example, the CUDA code is currently auto-generated using Tullio and other high-level Julia functions, so manually writing CUDA kernels could allow for significantly better optimization.

Another factor affecting performance is our use of FP64 data types. We initially chose FP64 as it was the default precision in Julia, but GPU performance for FP64 is known to be 32-64 times slower than for FP32 [Tecnda; Tecndb]. When we re-implemented inference in FP32, we observed a 3x speed-up on GPU without further changes, reducing inference time to just 1.7x slower than Torch in Table 5.6. An FP32 re-implementation of the entire framework, coupled with more efficient GPU utilization, could therefore hope to significantly narrow the training performance gap with Torch.

Our approach is also inherently limited by its lack of batch processing. Currently, batching in MP only serves to manage memory requirements, as we still process forward-backward messages sequentially for individual examples. Overcoming this constraint is a key concern for future work that aims to scale our method further. One potential solution could be to iterate through branches in multiple batch FactorGraphs simultaneously<sup>17</sup>, or alternatively, processing multiple branches within the same batch in parallel. The latter would be closer to traditional batching but might require additional strategies to ensure stable convergence.

The second major limitation is the weak performance on CIFAR-10, as reported in Table 5.3. Unfortunately, the authors of VOGN did not publish MNIST results, so CIFAR-10 is the only available dataset for comparison against VOGN [Osa+19]. The reason for MP’s underperformance on CIFAR-10 remains unclear, but we conducted only a few experiments due to time constraints. Besides investigating the specific training behavior on CIFAR-10, we have identified several next steps for generally improving the quality of results of our framework:

1. **Different prior parameters:** During development, we found that the choice of priors significantly affects training outcomes. The current priors are detailed in Chapter B, but further work is needed to determine the most effective parameters, especially for deep networks, where naive parameterization can lead to exploding variances during forward passes. Orthogonal initialization might be a promising direction [HXP20].
2. **Using Torch weights as initialization:** It could be more effective to apply our method after some initial training using traditional methods. These maximum likelihood weights could serve as prior means or as initial backward messages<sup>18</sup>. Whether this strategy improves or deteriorates the posterior approximation remains to be tested.
3. **Initializing backward messages for new batches:** When a batch becomes active, we reset all messages in the FactorGraph to  $\mathcal{G}(0, 0)$  to prevent biasing the new batch with the messages of the previous one. While storing the actual message history for all examples is prohibitively expensive, maintaining a single exponential moving average of backward messages could be a compromise that provides a better starting point at a low cost.

<sup>17</sup> With  $k$  active batches, messages could be updated in parallel while iterating sequentially within each batch. This would allow  $k$  messages to be computed simultaneously, one for each active batch.

<sup>18</sup> We could initialize the factor-to-weight messages from inactive batches with the maximum likelihood weights so that the initial posterior estimate is centered around them while preserving the true prior.

4. **Mixing examples between batches:** Currently, we cannot change batch assignments between epochs since we only store combined factor-to-weight messages for each batch (see [Section 4.2.2](#)). As a solution, we propose splitting batches in half and combining them with halves from other batches. This approach only requires us to store two additional messages per weight<sup>19</sup> and could help prevent overfitting.
5. **Input augmentation:** Just like in traditional approaches, input augmentation could enhance generalization. The augmented inputs could either be added as new examples (double-counting information) or replace the original inputs upon each revisit in the active batch.

Furthermore, in the context of deep learning, CIFAR-10 with the LeNet-5 architecture is still a relatively small experiment. Future work should therefore explore larger datasets and architectures. Another important direction would be to evaluate our posterior approximations against ground truths obtained through Hamiltonian Monte Carlo or other MCMC methods, as done in Korattikara et al. [[Kor+15](#)]. We are also excited to apply our MP framework to use cases that benefit specifically from uncertainty estimation and Bayesian methods, such as active learning, continual learning, sparse networks, and tasks with limited training data, as these tasks will provide a better understanding of the practical usability of our method.

A general limitation of MP frameworks is the need to derive, implement, and test approximate message equations manually. Our linear algebra factor library addresses this limitation by offering reusable and extensible message equations for constructing various layers, such as linear and CNN layers. Future work could extend our approach to larger CNNs with skip connections (e.g., ConvNeXt [[Liu+22](#)]), self-attention mechanisms, or transformer models (e.g., LLMs). Exploring more activation functions, such as GeLU, Softplus, or Squareplus, as well as different prior functions may also improve the quality of our posterior approximation.

We aimed to adapt factor graphs to deep learning in a scalable, robust, and performant manner. To achieve this, we used a fully-factorized posterior approximation (Gaussians with diagonal covariance matrices), as eliminating covariances simplifies message equations and reduces the computational complexity. Despite this simplification, our results have far surpassed our expectations. In future work, it will be interesting to selectively reintroduce covariances when the runtime impact is small or the quality improvement is substantial. For example, replacing the vector-matrix multiplication factor with a multivariate approximation is particularly promising, and message equations might be available in related work on bilinear models [[Akr+20](#); [FT97](#)]. The modularity of factor graphs is a big advantage here, as it allows factors to be chosen individually for each layer, example, or use case, offering flexibility in extending our approach.

In summary, we have advanced the application of MP to BNNs by demonstrating a framework that scales up to CNNs and MLPs with up to 5.6 million parameters. While further refinements are needed to match VI's scale and performance, MP's potential for balanced uncertainty estimates positions it as a strong alternative. With continued exploration of larger datasets, architectures, and real-world applications, MP thereby holds the promise of bringing uncertainty-aware deep learning to broader use cases and of, perhaps, unlocking the comprehensive application of AI in safety-critical domains.

<sup>19</sup> When processing the first batch of an epoch, backward messages for each half are stored. For subsequent batches, we split again and combine the stored half with one new half. The very last half is combined with the very first half. By randomizing batch orders, this approach could provide a compromise between storage and randomization.

# Bibliography

---

- [Abb+23] Michael Abbott, Dilum Aluthge, N3N5, Vedant Puri, Chris Elrod, Simeon Schaub, Carlo Lucibello, Jishnu Bhattacharya, Johnny Chen, Kristoffer Carlsson, and Maximilian Gelbrecht. *mcabbott/Tullio.jl: v0.3.7*. Version v0.3.7. Oct. 2023. DOI: [10.5281/zenodo.10035615](https://doi.org/10.5281/zenodo.10035615). URL: <https://doi.org/10.5281/zenodo.10035615> (see page 47).
- [Ada+24] Janina Adamcic, Benjamin Fürst, Christian Helms, Janine Kluge, Henok Lachmann, and Romeo Sommerfeld. **Bayesian Neural Networks**. Hasso Plattner Institut, Feb. 2024 (see pages 3, 11, 17, 22–24, 41, 46, 48, 54, 64, 67, 68).
- [Akr+20] Mohamed Akrouf, Anis Housseini, Faouzi Bellili, and Amine Mezghani. *Bilinear Generalized Vector Approximate Message Passing*. 2020. arXiv: [2009.06854](https://arxiv.org/abs/2009.06854) [cs.IT]. URL: <https://arxiv.org/abs/2009.06854> (see page 70).
- [Ata+24] Shahin Atakishiyev, Mohammad Salameh, Hengshuai Yao, and Randy Goebel. *Explainable Artificial Intelligence for Autonomous Driving: A Comprehensive Overview and Field Guide for Future Research Directions*. 2024. arXiv: [2112.11561](https://arxiv.org/abs/2112.11561) [cs.AI]. URL: <https://arxiv.org/abs/2112.11561> (see page 1).
- [BG59] R. C. Bose and Shanti S. Gupta. **Moments of Order Statistics from a Normal Population**. *Biometrika* 46:3/4 (1959), 433–440. ISSN: 00063444, 14643510. URL: <http://www.jstor.org/stable/2333540> (visited on 10/06/2024) (see page 25).
- [Blu+15] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. *Weight Uncertainty in Neural Networks*. 2015. arXiv: [1505.05424](https://arxiv.org/abs/1505.05424) [stat.ML]. URL: <https://arxiv.org/abs/1505.05424> (see pages 14, 56, 62, 63, 68).
- [Boj+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. *End to End Learning for Self-Driving Cars*. 2016. arXiv: [1604.07316](https://arxiv.org/abs/1604.07316) [cs.CV]. URL: <https://arxiv.org/abs/1604.07316> (see page 1).
- [CAS16] Paul Covington, Jay Adams, and Emre Sargin. **Deep Neural Networks for YouTube Recommendations**. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys '16. Boston, Massachusetts, USA: Association for Computing Machinery, 2016, 191–198. ISBN: 9781450340359. DOI: [10.1145/2959100.2959190](https://doi.org/10.1145/2959100.2959190). URL: <https://doi.org/10.1145/2959100.2959190> (see page 1).
- [Chu+24] Neo Christopher Chung, Hongkyou Chung, Hearim Lee, Lennart Brocki, Hongbeom Chung, and George Dyer. *False Sense of Security in Explainable Artificial Intelligence (XAI)*. 2024. arXiv: [2405.03820](https://arxiv.org/abs/2405.03820) [cs.CY]. URL: <https://arxiv.org/abs/2405.03820> (see page 1).
- [Cok+22] Beau Coker, Wessel P. Bruinsma, David R. Burt, Weiwei Pan, and Finale Doshi-Velez. *Wide Mean-Field Bayesian Neural Networks Ignore the Data*. 2022. arXiv: [2202.11670](https://arxiv.org/abs/2202.11670) [cs.LG]. URL: <https://arxiv.org/abs/2202.11670> (see pages 2, 12).



- [CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. **High Performance Convolutional Neural Networks for Document Processing**. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Guy Lorette. <http://www.suvisoft.com>. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. URL: <https://inria.hal.science/inria-00112631> (see page 45).
- [Dax+22] Erik Daxberger, Agustinus Kristiadi, Alexander Immer, Runa Eschenhagen, Matthias Bauer, and Philipp Hennig. *Laplace Redux – Effortless Bayesian Deep Learning*. 2022. arXiv: [2106.14806](https://arxiv.org/abs/2106.14806) [cs.LG]. URL: <https://arxiv.org/abs/2106.14806> (see pages 27, 29).
- [DB16] Guillaume Dehaene and Simon Barthelmé. *Expectation Propagation in the large-data limit*. 2016. arXiv: [1503.08060](https://arxiv.org/abs/1503.08060) [stat.CO]. URL: <https://arxiv.org/abs/1503.08060> (see page 12).
- [Dub+24] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: [2407.21783](https://arxiv.org/abs/2407.21783) [cs.AI]. URL: <https://arxiv.org/abs/2407.21783> (see page 1).
- [FM97] Brendan J. Frey and David J. C. MacKay. **A revolution: belief propagation in graphs with cycles**. In: *Proceedings of the 10th International Conference on Neural Information Processing Systems*. NIPS’97. Denver, CO: MIT Press, 1997, 479–485 (see page 12).
- [FT97] W.T. Freeman and J.B. Tenenbaum. **Learning bilinear models for two-factor problems in vision**. In: *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1997, 554–560. DOI: [10.1109/CVPR.1997.609380](https://doi.org/10.1109/CVPR.1997.609380) (see page 70).
- [GDY16] Soumya Ghosh, Francesco Delle Fave, and Jonathan Yedidia. **Assumed Density Filtering Methods for Learning Bayesian Neural Networks**. *Proceedings of the AAAI Conference on Artificial Intelligence* 30:1 (Feb. 2016). DOI: [10.1609/aaai.v30i1.10296](https://doi.org/10.1609/aaai.v30i1.10296). URL: <https://ojs.aaai.org/index.php/AAAI/article/view/10296> (see page 13).
- [Gra11] Alex Graves. **Practical Variational Inference for Neural Networks**. In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger. Vol. 24. Curran Associates, Inc., 2011. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2011/file/7eb3c8be3d411e8ebfab08eba5f49632-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2011/file/7eb3c8be3d411e8ebfab08eba5f49632-Paper.pdf) (see pages 14, 56).
- [Gre03] William H. Greene. **Econometric Analysis**. Fifth. Pearson Education, 2003. ISBN: 0-13-066189-9. URL: <http://pages.stern.nyu.edu/~wgreene/Text/econometricanalysis.htm> (see page 30).
- [GYD18] Soumya Ghosh, Jiayu Yao, and Finale Doshi-Velez. *Structured Variational Learning of Bayesian Neural Networks with Horseshoe Priors*. 2018. arXiv: [1806.05975](https://arxiv.org/abs/1806.05975) [stat.ML]. URL: <https://arxiv.org/abs/1806.05975> (see page 14).
- [HA15] José Miguel Hernández-Lobato and Ryan P. Adams. *Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks*. 2015. arXiv: [1502.05336](https://arxiv.org/abs/1502.05336) [stat.ML]. URL: <https://arxiv.org/abs/1502.05336> (see pages 12, 13, 68).
- [HW21] Eyke Hüllermeier and Willem Waegeman. **Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods**. *Machine Learning* 110:3 (Mar. 2021), 457–506. ISSN: 1573-0565. DOI: [10.1007/s10994-021-05946-3](https://doi.org/10.1007/s10994-021-05946-3). URL: <http://dx.doi.org/10.1007/s10994-021-05946-3> (see pages 1, 2).
- [HXP20] Wei Hu, Lechao Xiao, and Jeffrey Pennington. *Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks*. 2020. arXiv: [2001.05992](https://arxiv.org/abs/2001.05992) [cs.LG]. URL: <https://arxiv.org/abs/2001.05992> (see page 69).
- [Inn+18] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. **Fashionable Modelling with Flux**. *CoRR* abs/1811.01457 (2018). arXiv: [1811.01457](https://arxiv.org/abs/1811.01457). URL: <https://arxiv.org/abs/1811.01457> (see page 45).

- [Int24] Carnegie Endowment for International Peace. *AI and Product Safety Standards under the EU AI Act*. Accessed: 2024-09-18. 2024. URL: <https://carnegieendowment.org/research/2024/03/ai-and-product-safety-standards-under-the-eu-ai-act?lang=en> (see page 1).
- [JNV14] Pasi Jylänki, Aapo Nummenmaa, and Aki Vehtari. **Expectation Propagation for Neural Networks with Sparsity-Promoting Priors**. *Journal of Machine Learning Research* 15:54 (2014), 1849–1901. URL: <http://jmlr.org/papers/v15/jylanki14a.html> (see page 12).
- [KFL01] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. **Factor graphs and the sum-product algorithm**. *IEEE Transactions on Information Theory* 47:2 (2001), 498–519. DOI: 10.1109/18.910572 (see pages 3, 8, 12).
- [Kha+18] Mohammad Emtiyaz Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. *Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam*. 2018. arXiv: 1806.04854 [stat.ML]. URL: <https://arxiv.org/abs/1806.04854> (see page 14).
- [Kor+15] Anoop Korattikara, Vivek Rathod, Kevin Murphy, and Max Welling. *Bayesian Dark Knowledge*. 2015. arXiv: 1506.04416 [cs.LG]. URL: <https://arxiv.org/abs/1506.04416> (see pages 56, 70).
- [KR24] Mohammad Emtiyaz Khan and Håvard Rue. *The Bayesian Learning Rule*. 2024. arXiv: 2107.04562 [stat.ML]. URL: <https://arxiv.org/abs/2107.04562> (see page 2).
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. **ImageNet Classification with Deep Convolutional Neural Networks**. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012 (see page 6).
- [Kur+22] Richard Kurl, Ralf Herbrich, Tim Januschowski, Yuyang Wang, and Jan Gasthaus. *On the detrimental effect of invariances in the likelihood for variational inference*. 2022. arXiv: 2209.07157 [cs.LG]. URL: <https://arxiv.org/abs/2209.07157> (see page 2).
- [KW22] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML]. URL: <https://arxiv.org/abs/1312.6114> (see page 14).
- [Lec+98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. **Gradient-based learning applied to document recognition**. *Proceedings of the IEEE* 86:11 (1998), 2278–2324. DOI: 10.1109/5.726791 (see page 57).
- [Liu+22] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. *A ConvNet for the 2020s*. 2022. arXiv: 2201.03545 [cs.CV]. URL: <https://arxiv.org/abs/2201.03545> (see page 70).
- [LKF10] Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. **Convolutional networks and applications in vision**. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010, 253–256. DOI: 10.1109/ISCAS.2010.5537907 (see page 6).
- [LSK20] Wu Lin, Mark Schmidt, and Mohammad Emtiyaz Khan. *Handling the Positive-Definite Constraint in the Bayesian Learning Rule*. 2020. arXiv: 2002.10060 [stat.ML]. URL: <https://arxiv.org/abs/2002.10060> (see page 14).
- [Luc+22] Carlo Lucibello, Fabrizio Pittorino, Gabriele Perugini, and Riccardo Zecchina. **Deep learning via message passing algorithms based on belief propagation**. *Machine Learning: Science and Technology* 3:3 (July 2022), 035005. DOI: 10.1088/2632-2153/ac7d3b. URL: <https://dx.doi.org/10.1088/2632-2153/ac7d3b> (see pages 13, 62, 63, 68).
- [Min01] Thomas P. Minka. **Expectation propagation for approximate Bayesian inference**. In: *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*. UAI’01. Seattle, Washington: Morgan Kaufmann Publishers Inc., 2001, 362–369. ISBN: 1558608001 (see pages 3, 12).

- [Min05] Tom Minka. **Divergence Measures and Message Passing**. Tech. rep. MSR-TR-2005-173. Jan. 2005, 17. URL: <https://www.microsoft.com/en-us/research/publication/divergence-measures-and-message-passing/> (see pages 3, 9–12).
- [MWJ99] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. **Loopy belief propagation for approximate inference: an empirical study**. In: *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. UAI’99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., 1999, 467–475. ISBN: 1558606149 (see page 9).
- [ON15] Keiron O’Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE]. URL: <https://arxiv.org/abs/1511.08458> (see page 6).
- [Osa+19] Kazuki Osawa, Siddharth Swaroop, Anirudh Jain, Runa Eschenhagen, Richard E. Turner, Rio Yokota, and Mohammad Emtiyaz Khan. *Practical Deep Learning with Bayesian Principles*. 2019. arXiv: 1906.02506 [stat.ML]. URL: <https://arxiv.org/abs/1906.02506> (see pages 2, 14, 56, 63, 67–69).
- [Owe80] D. B. Owen. **A table of normal integrals**. *Communications in Statistics - Simulation and Computation* 9:4 (1980), 389–419. URL: <https://doi.org/10.1080/03610918008812164> (see page 31).
- [Pap+24] Theodore Papamarkou, Maria Skoularidou, Konstantina Palla, Laurence Aitchison, Julyan Arbel, David Dunson, Maurizio Filippone, Vincent Fortuin, Philipp Hennig, José Miguel Hernández-Lobato, Aliaksandr Hubin, Alexander Immer, Theofanis Karaletsos, Mohammad Emtiyaz Khan, Agustinus Kristiadi, Yingzhen Li, Stephan Mandt, Christopher Nemeth, Michael A. Osborne, Tim G. J. Rudner, David Rügamer, Yee Whye Teh, Max Welling, Andrew Gordon Wilson, and Ruqi Zhang. *Position: Bayesian Deep Learning is Needed in the Age of Large-Scale AI*. 2024. arXiv: 2402.00809 [cs.LG]. URL: <https://arxiv.org/abs/2402.00809> (see page 2).
- [Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: <https://arxiv.org/abs/1912.01703> (see page 57).
- [Pea88] Judea Pearl. **Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. ISBN: 1558604790 (see page 8).
- [Rad+22] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. *Robust Speech Recognition via Large-Scale Weak Supervision*. 2022. arXiv: 2212.04356 [eess.AS]. URL: <https://arxiv.org/abs/2212.04356> (see page 1).
- [Rad19] European Society of Radiology (ESR) communications@myesr.org Neri Emanuele de Souza Nandita Brady Adrian Bayarri Angel Alberich Becker Christoph D. Coppola Francesca Visser Jacob. **What the radiologist should know about artificial intelligence—an ESR white paper**. *Insights into imaging* 10:1 (2019), 44 (see page 1).
- [Rav+24] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. *SAM 2: Segment Anything in Images and Videos*. 2024. arXiv: 2408.00714 [cs.CV]. URL: <https://arxiv.org/abs/2408.00714> (see page 1).
- [She+24] Yuesong Shen, Nico Daheim, Bai Cong, Peter Nickl, Gian Maria Marconi, Clement Bazan, Rio Yokota, Iryna Gurevych, Daniel Cremers, Mohammad Emtiyaz Khan, and Thomas Möllenhoff. *Variational Learning is Effective for Large Deep Networks*. 2024. arXiv: 2402.17641 [cs.LG]. URL: <https://arxiv.org/abs/2402.17641> (see pages 2, 12–14, 67).



- [SHG09] David Stern, Ralf Herbrich, and Thore Graepel. **Matchbox: Large Scale Bayesian Recommendations**. In: *Proceedings of the 18th International World Wide Web Conference*. Jan. 2009. URL: <https://www.microsoft.com/en-us/research/publication/matchbox-large-scale-bayesian-recommendations/> (see page 17).
- [SHM14] Daniel Soudry, Itay Hubara, and Ron Meir. **Expectation Backpropagation: parameter-free training of multilayer neural networks with continuous or discrete weights**. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'14. Montreal, Canada: MIT Press, 2014, 963–971 (see pages 12, 68).
- [Tecnda] TechPowerUp. *NVIDIA A40 PCIe Specs*. <https://www.techpowerup.com/gpu-specs/a40-pcie.c3700>. Accessed: 2024-09-16. n.d. (see pages 66, 69).
- [Tecndb] TechPowerUp. *NVIDIA Tesla T4 Specs*. <https://www.techpowerup.com/gpu-specs/tesla-t4.c3316>. Accessed: 2024-09-16. n.d. (see pages 66, 69).
- [WB05] John Winn and Christopher M. Bishop. **Variational Message Passing**. *Journal of Machine Learning Research* 6:23 (2005), 661–694. URL: <http://jmlr.org/papers/v6/winn05a.html> (see page 12).
- [XJK24] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. *Hallucination is Inevitable: An Innate Limitation of Large Language Models*. 2024. arXiv: 2401.11817 [cs.CL]. URL: <https://arxiv.org/abs/2401.11817> (see page 1).
- [Zha+18] Cheng Zhang, Judith Butepage, Hedvig Kjellstrom, and Stephan Mandt. *Advances in Variational Inference*. 2018. arXiv: 1711.05597 [cs.LG]. URL: <https://arxiv.org/abs/1711.05597> (see pages 2, 14, 67).
- [Zha+24] Zelun Tony Zhang, Sebastian S Feger, Lucas Dullenkopf, Rulu Liao, Lukas Süßlin, Yuanling Liu, and Andreas Butz. **Beyond Recommendations: From Backward to Forward AI Support of Pilots' Decision-Making Process**. *arXiv preprint arXiv:2406.08959* (2024) (see page 1).
- [ZMG19] Guodong Zhang, James Martens, and Roger Grosse. *Fast Convergence of Natural Gradient Descent for Overparameterized Neural Networks*. 2019. arXiv: 1905.10961 [stat.ML]. URL: <https://arxiv.org/abs/1905.10961> (see page 14).



We now want to demonstrate how performance optimization could look like. We will compare nine slightly different versions of a function that computes the forward message for a matrix-matrix multiplication. In the process, we will explain some interesting details and code patterns that we use. The performance measurements for each version can be found in [Table A.1](#). After comparing the performance of these different versions, we also present general advice for performance optimization in Julia.

We begin with a naive implementation that runs a for-loop to compute each individual output element as a vector product. To keep things simple, we work on a function that only takes one parameter per operand. In [listing 3](#) we introduced a similar interface as "1: Prediction". As all variants of this code share the same function signature, we will omit it in the coming versions.

```
1  function forward_mult1(  
2      A::AbstractMatrix{Gaussian1d}, B::AbstractMatrix{Gaussian1d},  
3      c::AbstractVector{Gaussian1d}, out::AbstractMatrix{Gaussian1d}  
4  )  
5      # Compute separate vector product for each output  
6      for i in axes(out, 1)  
7          for k in axes(out, 2)  
8              out[i, k] = forward_mult(A[i, :], B[:, k], c[k])  
9          end  
10     end  
11     return  
12 end
```

**Listing 8:** Version 1.

This naive version already has reasonable performance. It is slightly faster to iterate `i` in the outer loop and `k` in the inner loop than the other way around, even though both ways require row-wise access from one of the two matrices. Besides the `axes` iterator, this version contains no interesting elements yet. However, we can achieve a striking performance improve with a small change:

```

1  # Compute separate vector product for each output
2  for i in axes(out, 1)
3      for k in axes(out, 2)
4          out[i, k] = forward_mult((@view A[i, :]), (@view B[:, k]), c[k])
5      end
6  end
7  return

```

**Listing 9:** Version 2.

Just by adding `@view`, we managed to improve the performance by more than double (i.e., less than half the runtime) and to cut memory usage down to a constant (which is more than 99% less than version 1 on the big input sample). By writing `(@view A[i, :])`, we create a pointer to the  $i$ -th row of  $A$ , whereas without the macro we would create a copy. In the overwhelming majority of cases, it is more performant to create a view. Instead of using the `@view` macro, it is also sometimes practical to use the equivalent

```
forward_mult(selectdim(A, 1, i), selectdim(B, 2, k), c[k]),
```

which also creates a view. We can improve the runtime by another factor of x3 by using Tullio instead of for-loops. Furthermore, the code is now short and expressive:

```

1  # Compute separate vector product for each output
2  @tullio out[i, k] = forward_mult((@view A[i, :]), (@view B[:, k]), c[k])
3  return

```

**Listing 10:** Version 3.

Similar to listing 6, we can write down the same operation using broadcast notation. However, writing this operation in broadcasting notation is not only more error-prone, it is also slower. Furthermore, it is more difficult to discover (or remember) the necessary interfaces. As a side note, using `reshape` also creates views, not copies.

```

1  # Compute separate vector product for each output
2  out .= forward_mult.(
3      reshape(eachrow(A), :, 1),
4      reshape(eachcol(B), 1, :),
5      reshape(c, 1, :)
6  )
7  return

```

**Listing 11:** Version 4.

So far, the Tullio version was by far the fastest. It internally leverages Julia's Threads library, yet we can build an even faster version by explicitly calling it ourselves. However, to apply

the `Threads.@threads` macro, we need to convert our code to a single outer loop again. For that, we can use the `CartesianIndices` iterator. If used as a direct replacement for the two for-loops in version 1 and 2, the performance becomes slightly worse. However, here it allows us to multi-thread the entire computation effectively:

```
1  # Compute separate vector product for each output
2  Threads.@threads for ind in CartesianIndices(out)
3      i, k = ind[1], ind[2]
4      out[i, k] = forward_mult((@view A[i, :]), (@view B[:, k])), c[k])
5  end
6  return
```

**Listing 12:** Version 5.

We have now reached the fastest CPU version that I have identified; version 5 is the actual implementation used in the linear algebra library. A similar macro as `Threads.@threads` is `@batch` from `Polyester.jl`. In this case, we found that they perform about the same. Since `Threads` is part of Julia base, we decided to use that by default. However, when each individual function call is cheap, it can be faster to use `@batch` instead. We still created a `Polyester` version to compare its performance:

```
1  # Compute separate vector product for each output
2  @batch for ind in CartesianIndices(out)
3      i, k = ind[1], ind[2]
4      out[i, k] = forward_mult((@view A[i, :]), (@view B[:, k])), c[k])
5  end
6  return
```

**Listing 13:** Version 6.

After optimizing for CPU, we now want to find the most performant GPU code. Version 3 is already good, but we can improve upon it for smaller input sizes. It should be noted that most real operations in our experiments are even smaller than the "Small" in our test. For example, vector-matrix operations are much more common and have smaller outputs than matrix-matrix operations.

All 6 versions so far have reduced the matrix-matrix multiplication into a series of independent vector-vector multiplications. This makes sense if the degree of parallelism is small - such as on a CPU. On GPU, we want to remove this additional level of abstraction and directly run all individual products with one dispatch. One way to achieve this is to use Tullio's dimension reduction feature. Because the index `j` is used in the right-hand expression but not in the left-hand array index, Tullio sums the `j`-dimension out.

```

1  # Compute product separately for mu and sigma
2  @tullio  $\mu[i, k] := \text{forward\_mult\_}\mu(A[i, j], B[j, k])$ 
3  @tullio  $\sigma2[i, k] := \text{forward\_mult\_}\sigma2(A[i, j], B[j, k])$ 
4
5  # Add c to the result and clean up
6  @tullio out[i, k] = Gaussian1d(
7       $\mu = \mu[i, k] + \text{mean}(c[k])$ ,
8       $\sigma2 = \sigma2[i, k] + \text{variance}(c[k])$ 
9  )
10 free_if_CUDA!(( $\mu$ ,  $\sigma2$ ))
11 return

```

**Listing 14:** Version 7.

The `:=` notation stands for creating a new array, which we then free again at the end. Another way to run all products in parallel is to first materialize all pairwise results, and to sum out the  $j$ -dimension in a separate step:

```

1  # First compute product as Gaussian1d
2  @tullio params_prod[i, j, k] := forward_mult(A[i, j], B[j, k])
3
4  # Then take parameters apart
5  @tullio temp[i, j, k] := mean(params_prod[i, j, k])
6   $\_mu = \text{sum}(temp, \text{dims}=2)$ 
7   $\mu = (@\text{view } \_mu[:, 1, :])$ 
8
9  @tullio temp[i, j, k] = variance(params_prod[i, j, k])
10  $\_sigma2 = \text{sum}(temp, \text{dims}=2)$ 
11  $\sigma2 = (@\text{view } \_sigma2[:, 1, :])$ 
12
13 # Add c to the result and clean up
14 @tullio out[i, k] = Gaussian1d(
15      $\mu = \mu[i, k] + \text{mean}(c[k])$ ,
16      $\sigma2 = \sigma2[i, k] + \text{variance}(c[k])$ 
17 )
18 free_if_CUDA!((params_prod, temp,  $\_mu$ ,  $\_sigma2$ ))
19 return

```

**Listing 15:** Version 8.

One interesting aspect of this implementation is that we first create  $\_mu$  and then a view  $\mu$ . Using `sum` (or `prod`) over some dimension does not actually remove that dimension from the array. Instead, it reduces the size of the dimension to 1. We could just access  $\_mu[i, 1, k]$  in the final Tullio call, but found it to be faster to remove the redundant dimension with a view. Because we still want to free the CUDA memory allocated by `sum` afterwards, we also have to keep a reference to  $\_mu$ .

There is only a small change from version 8 to our final version, but it still has a noticeable impact on performance. As described in [subsection 4.4.2](#), it is faster to avoid the creation of Gaussian1ds. Instead of first computing the products as Gaussian1ds and then extracting their moment parameters, we can directly compute the moment parameters.

```

1  # Compute product separately for mu and sigma
2  @tullio temp[i, j, k] := forward_mult_μ(A[i, j], B[j, k])
3  _μ = sum(temp, dims=2)
4  μ = (@view _μ[:, 1, :])
5
6  @tullio temp[i, j, k] = forward_mult_σ2(A[i, j], B[j, k])
7  _σ2 = sum(temp, dims=2)
8  σ2 = (@view _σ2[:, 1, :])
9
10 # Add c to the result and clean up
11 @tullio out[i, k] = Gaussian1d(;
12     μ=μ[i, k] + mean(c[k]),
13     σ2=σ2[i, k] + variance(c[k])
14 )
15 free_if_CUDA!((temp, _μ, _σ2))
16 return

```

**Listing 16:** Version 9.

We can now compare the performance of our 9 versions in [Table A.1](#). The CPU measurements were taken on my Apple MacBook with a 2 GHz Quad-Core Intel Core i5, whereas the GPU measurements were taken on an NVIDIA A40 GPU on the HPI cluster.

This exercise has already demonstrated the impact of small refactorings on runtime. Which version is most effective usually depends on the specifics of the problem, inputs, and devices. It is always a good idea to first identify performance-critical code sections and to then optimize those specific functions in a benchmark. There are two tools that we recommend in particular:

1. The `@profview` macro: When developing Julia in VS Code, running some function with `@profview` will open a profiler that allows to interactively explore the measured runtime. We found it highly useful for identifying performance-critical sections in single-threaded CPU code, but it becomes more complex when Julia has multiple threads available or when GPUs are used. It is a good idea to run everything at least once before starting `@profview` to reduce the amount of setup and compilation captured in the performance data.
2. BenchmarkTools: For profiling the overall performance of some function, we recommend the `@btime` macro from `BenchmarkTools.jl`. When running CUDA code, make sure to run `@btime CUDA.@sync ...` to measure the time until completing the CUDA job and not until the scheduling is finished.

Finally, here is a collection of insights that have not fit anywhere else:

Function	CPU				GPU			
	Small		Big		Small		Big	
	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.
Version 1	93.2ms	249MiB	594.5ms	1.52GiB	-	-	-	-
Version 2	41.2ms	8MiB	250.9ms	8MiB	-	-	-	-
Version 3	40.0ms	8MiB	84.9ms	8MiB	9.2ms	10KiB	<b>11.5ms</b>	9KiB
Version 4	39.5ms	8MiB	274.9ms	8MiB	-	-	-	-
Version 5	<b>10.6ms</b>	8MiB	<b>65.3ms</b>	8MiB	-	-	-	-
Version 6	10.9ms	8MiB	65.6ms	8MiB	-	-	-	-
Version 7	19.0ms	8MiB	107.2ms	9MiB	7.4ms	16KiB	17.3ms	16KiB
Version 8	56.5ms	187MiB	472.7ms	1.13GiB	3.3ms	23KiB	20.6ms	23KiB
Version 9	28.5ms	68MiB	210.2ms	391MiB	<b>2.7ms</b>	21KiB	16.8ms	21KiB

**Table A.1:** Performance comparison of different `forward_mult` functions on CPU and GPU. The fastest version for each comparison is highlighted in bold. Our library deploys version 5 for CPU and version 9 for GPU. The small input was  $(100, 783) \times (783, 100)$ , whereas the big input was  $(640, 783) \times (783, 640)$ . The tracked memory usage only includes CPU memory, which explains why we measured low memory usage on GPU for all versions.

1. When debugging CUDA code, begin by putting `CUDA.synchronize()` all around the suspected error. Because GPU operations run asynchronously by default, the error displayed in the console is often only the location when the CPU finds out about the error, not the location that caused it. Every `CUDA.synchronize()` will catch all errors that came before, which allows for fast localization of the error.
2. CUDA appears to be not thread-safe. In our experience, it is best to avoid (CPU-based) multi-threading completely in all code that handles CuArrays.
3. It is faster to define a constant outside of a Tullio statement than to use array-access within the statement:  

```
@tullio out[i] = input\_k / (c + samples[i]) (with input\_k=input[k])
@tullio out[i] = input[k] / (c + samples[i]) (25x slower)
```
4. For some reason, `rand` is faster than `rand!`, despite allocating. To get a non-allocating, fast version, we run a threaded for-loop over a pre-allocated array. This is one of the opportunities where `@batched` is fastest.
5. Calling `quadgk(...)` is faster than using `IntegralProblem` with `QuadGK` as a solver.



# B

## Choosing the FactorGraph's Priors

---

The strength of the prior determines the amount of data needed to obtain a useful posterior that fits the data. Our goal is to draw prior means and set prior variances so that the computed variances of all messages are on the order of  $\mathcal{O}(1)$  regardless of network width and depth. It is not entirely clear if this would be a desirable property; after all, adding more layers also makes the network more expressive and more easily able to model functions with very high or low values. However, if we let the predictive prior grow unrestricted, it will grow exponentially, leading to numerical issues. In the following, we analyze the predictive prior under simplifying assumptions to derive a prior initialization that avoids exponential variance explosion. While we fail to achieve this goal, our current prior variances are still informed by this analysis.

In the following, we assume that the network inputs are random variables. Then, the parameters of messages also become random variables, as they are derived from the inputs according to the message equations. Our goal is to keep the expected value of the variance parameter of the outgoing message at a constant size. We also assume that the means of the prior are sampled according to spectral initialization, as described in [Section 4.4.1](#).

### FirstGaussianLinearLayer - Input is a Constant

Each linear layer transforms some  $d_1$ -dimensional input  $\mathbf{x}$  to some  $d_2$ -dimensional output  $\mathbf{y}$  according to  $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ . In the first layer,  $\mathbf{x}$  is the input data. For this analysis, we assume each element  $x_i$  to be drawn independently from  $x_i \sim \mathcal{N}(0, 1)$ . Let  $\mathbf{x}$  be a  $d_1$ -dimensional input vector,  $\mathbf{m}_w$  be the prior messages from one column of  $\mathbf{W}$ , and  $\mathbf{z} = \mathbf{w}^\top \mathbf{x}$  be the vector product before adding the bias.

During initialization of the weight prior, we draw the prior means using spectral parametrization and set the prior variances to a constant:

$$\mathbf{m}_{w_i} = \mathcal{N}(\mu_{w_i}, \sigma_w^2) \text{ with } \mu_{w_i} \sim \mathcal{N}(0, l^2),$$
$$l = \frac{1}{\sqrt{k}} \cdot \min(1, \sqrt{\frac{d_2}{d_1}}).$$

By applying the message equations, we then approximate the forward message to the output with a normal distribution

$$\mathbf{m}_z = \mathcal{N}(\mu_z, \sigma_z^2).$$

Because  $\sigma_z^2$  depends on the random variables  $x_i$ , it is also a random variable that follows a scaled

chi-squared distribution

$$\sigma_z^2 = \sum_{i=1}^{d_1} x_i^2 \cdot \sigma_w^2$$

$$\sigma_z^2 \sim \chi_{d_1}^2 \cdot \sigma_w^2$$

and its expected value is

$$\mathbb{E}[\sigma_z^2] = d_1 \cdot \sigma_w^2.$$

We conclude that we can control the magnitude of the variance parameter by choosing  $\mathbb{E}[\sigma_z^2]$  and setting  $\sigma_w^2 = \frac{\mathbb{E}[\sigma_z^2]}{d_1}$ .

### GaussianLinearLayer - Input is a Variable

In subsequent linear layers, the input  $\mathbf{x}$  is not observed and we receive an approximate forward message that consists of independent normal distributions

$$m_{x_i} = \mathcal{N}(\mu_{x_i}, \sigma_{x_i}^2).$$

Following the message equations, the outgoing forward message to  $z$  then has a variance

$$\begin{aligned} \sigma_z^2 &= \sum_{i=1}^{d_1} (\sigma_{x_i}^2 + \mu_{x_i}^2) \cdot (\sigma_w^2 + \mu_{w_i}^2) - (\mu_{x_i}^2 * \mu_{w_i}^2) \\ &= \sum_{i=1}^{d_1} \underbrace{\sigma_{x_i}^2 \cdot \sigma_w^2}_{\text{I}} + \underbrace{\sigma_{x_i}^2 \cdot \mu_{w_i}^2}_{\text{II}} + \underbrace{\mu_{x_i}^2 \cdot \sigma_w^2}_{\text{III}} \end{aligned}$$

The layer's prior variance  $\sigma_w^2$  is a constant, whereas all other elements are random variables according to our assumptions. To make further analysis tractable, we also have to assume that the variances  $\sigma_{x_i}^2$  of the incoming forward messages are identical constants for all  $i$ , not random variables. We furthermore assume that the means are drawn i.i.d. from:

$$\mu_{w_i} \sim \mathcal{N}(0, l^2)$$

$$\mu_{x_i} \sim \mathcal{N}(\mu_{\mu_x}, \sigma_{\mu_x}^2).$$

The random variable  $\sigma_z^2$  then follows a generalized chi-squared distribution

$$\sigma_z^2 \sim \left( \sum_{i=1}^{d_1} \underbrace{\sigma_x^2 \cdot l^2 \cdot \chi^2(1, 0^2)}_{\text{II}} + \underbrace{\sigma_w^2 \cdot \sigma_{\mu_x}^2 \cdot \chi^2(1, \mu_{\mu_x}^2)}_{\text{III}} \right) + \underbrace{d_1 \cdot \sigma_w^2 \cdot \sigma_x^2}_{\text{I}}$$

and its expected value is

$$\begin{aligned}
\mathbb{E}[\sigma_z^2] &= \left( \sum_{i=1}^{d_1} \sigma_x^2 \cdot l^2 \cdot (1 + 0^2) + \sigma_w^2 \cdot \sigma_{\mu_x}^2 \cdot (1 + \mu_{\mu_x}^2) \right) + d_1 \cdot \sigma_w^2 \cdot \sigma_x^2 \\
&= d_1 \cdot \left( \sigma_x^2 \cdot l^2 + \sigma_w^2 \cdot \sigma_{\mu_x}^2 \cdot (1 + \mu_{\mu_x}^2) + \sigma_w^2 \cdot \sigma_x^2 \right) \\
&= \underbrace{d_1 \cdot \sigma_x^2 \cdot l^2}_{\text{II}} + \underbrace{d_1 \cdot (\sigma_{\mu_x}^2 \cdot (1 + \mu_{\mu_x}^2) + \sigma_x^2) \cdot \sigma_w^2}_{\text{I+III}}.
\end{aligned}$$

As  $\sigma_w^2$  has to be positive, we conclude that if we choose  $\mathbb{E}[\sigma_z^2] > d_1 \cdot \sigma_x^2 \cdot l^2$ , then we can set

$$\sigma_w^2 = \frac{\mathbb{E}[\sigma_z^2] - d_1 \cdot \sigma_x^2 \cdot l^2}{d_1 \cdot (\sigma_{\mu_x}^2 \cdot (1 + \mu_{\mu_x}^2) + \sigma_x^2)}.$$

We know (or choose)  $d_1$ ,  $l^2$ , and  $\mathbb{E}[\sigma_z^2]$ , but we require values for  $\sigma_x^2$ ,  $\mu_{\mu_x}^2$ , and  $\sigma_{\mu_x}^2$  to be able to choose  $\sigma_w^2$ . We will find empirical values for these parameters in the next section.

## Empirical parameters + LeakyReLU

To inform the choice of the prior variances of the inner linear layers, we also need to analyze LeakyReLU. We assume the network is an MLP that alternates between linear layers and LeakyReLU. As the message equations of LeakyReLU are too complicated for analysis, we instead use empirical approximation. Let  $m_a = \mathcal{N}(\mu_a, \sigma_a^2)$  be an incoming message (from the pre-activation variable to LeakyReLU). We assume that  $\sigma_a^2 = t$  is a constant and that  $\mu_a \sim \mathcal{N}(0, 1)$  is a random variable. By sampling multiple means and then computing the outgoing messages (after applying LeakyReLU), we can approximate the average variance of the outgoing messages, as well as the average and empirical variance over means of the outgoing messages.

We computed these statistics for 101 different leak settings with 100 million samples each, and found that the relationship between leak and  $\mu_{\mu_x}$  (average mean of the outgoing message) is approximately linear, while the relationships between leak and  $\sigma_{\mu_x}^2$  or  $\mu_{\sigma_x^2}$  are approximately quadratic. Using these samples, we fitted coefficients with an error margin below  $5 \cdot 10^{-5}$ . For our network, we chose a target variance of 1.5 and a leak of 0.1, resulting in

$$\begin{aligned}
\sigma_x^2 &= 0.8040586726631379 \\
\sigma_{\mu_x}^2 \cdot (1 + \mu_{\mu_x}^2) &= 0.44958619556324186.
\end{aligned}$$

These values are sufficient for now setting the prior variances of the inner linear layer according to the equations above. Finally, we set the prior variance of the biases to 0.5, so that the output of each linear layer achieves an overall target prior predictive variance of approximately  $t = 1.5 + 0.5 = 2.0$ .

## Results in practice

In practice, we found that the variance of the predictive posterior still goes up exponentially with the depth of the network despite our derived prior choices. However, if we lower the prior variance further to avoid this explosion, the network is overly restricted and unable to obtain a good fit during training. We therefore set the prior variances as outlined here, but acknowledge that choosing a good prior is still an unsolved problem.

**Synthetic Data - Depth Scaling:** We generated a dataset of 200 points by randomly sampling  $x$  values from the range  $[0, 2]$ . The true data-generating function was

$$f(x) = 0.5x + 0.2 \sin(2\pi \cdot x) + 0.3 \sin(4\pi \cdot x).$$

The corresponding  $y$  values were sampled by adding Gaussian noise:  $f(x) + \mathcal{N}(0, 0.05^2)$ . For the architecture, we used a three-layer neural network with the structure:

$$[\text{Linear}(1, 16), \text{LeakyReLU}(0.1), \text{Linear}(16, 16), \text{LeakyReLU}(0.1), \text{Linear}(16, 1)].$$

A four-layer network has one additional  $[\text{Linear}(16, 16), \text{LeakyReLU}(0.1)]$  block in the middle, and a five-layer network has two additional blocks. For the regression noise hyperparameter, we used the true noise  $\beta^2 = 0.05^2$ . The models were trained for 500 iterations over one batch (as all data was processed in a single active batch). The two results shown in [Chapter 5](#) use the random seeds 98 and 1293.

**Synthetic Data - Width Scaling:** For the width-scaling experiment, we used a similar dataset of 80 points with  $x$  values sampled in the range  $[0, 1]$ . The true data-generating function remained the same as in the depth-scaling experiment, but the architecture was simplified to a two-layer network with one hidden layer:

$$[\text{Linear}(1, w), \text{LeakyReLU}(0.1), \text{Linear}(w, 1)].$$

The width  $w$  of the hidden layer was varied between 4, 16, 32, 64, and 128. The regression noise hyperparameter was set to the true noise  $\beta^2 = 0.05^2$ , and the models were trained for 500 iterations, as in the depth-scaling experiment.

**Synthetic Data - Uncertainty Evaluation:** The same data-generation process was used as in the depth-scaling experiment, but this time,  $x$  values were drawn from the range  $[-0.5, 0.5]$ . The network architecture remained the same as the three-layer network, but the width of the layers was increased to 32. We trained 100 networks with different random seeds on the same dataset. We define a  $p$ -credible interval for  $0 \leq p \leq 1$  as:

$$[\text{cdf}^{-1}(0.5 - \frac{p}{2}), \text{cdf}^{-1}(0.5 + \frac{p}{2})].$$

For each credible interval mass  $p$  (ranging from 0 to 1 in steps of 0.01), we measured how many of the  $p$ -credible intervals (across the 100 posterior approximations) covered the true data-generating function. This evaluation was done at each possible  $x$  value (ranging from -20 to 20 in steps of 0.05), generating a coverage rate for each combination of  $p$  and  $x$ . For each  $p$ , we then computed

the median for  $x > 10$  and the median for  $x < -10$ . If we correlate the  $p$  values with the medians, we found that for the median obtained from positive  $x$  values the correlation was 0.96, for negative  $x$  it was 0.98, and for the combined set of medians it was 0.94. We also measured the coverage rate of posterior  $\sigma$ -intervals in a similar experiment, where the true data-generating function was drawn from the neural network’s prior.

**MNIST:** For our MNIST experiments, we used the default train-test split (60,000 training examples and 10,000 test examples). When training on restricted data, we randomly sampled subsets from the training set. We also normalize the data by subtracting the mean (0.1307) and dividing through the standard deviation (0.3081) of the training data. The test examples are also normalized with the training data’s statistics. The architectures used were LeNet-5 (with LeakyReLU(0.1) in place of ReLU) and three-layer MLPs with the structure:

[Linear(784, 256), LeakyReLU(0.1), Linear(256, 256), LeakyReLU(0.1), Linear(256, 1)].

We set the noise level to  $\beta^2 = 0.01$  for regression-based training. For argmax-based training, we added noise  $\beta^2 = 0.3$  to the forward message and applied regularization with  $\gamma^2 = 0.1$  in the backward message. To apply regression-based loss, we encoded MNIST labels as 10-dimensional one-hot vectors (with one position set to 1.0 and the rest to 0.0). We trained our message-passing models for 90 iterations over 18 epochs (distributed as follows: 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9). The Torch models were trained for 90 epochs using SGD with  $lr = 0.001$ , and we used the exact same architectures (including LeakyReLU). We also trained one three-layer MLP with width 2,000 and therefore  $(784 + 1) \cdot 2,000 + (2,000 + 1) \cdot 2,000 + (2,000 + 1) \cdot 10 = 5,592,010$  weights.

To measure calibration, we used 20 bins that were split to minimize within-bin variance. For relative calibration, we ordered all test examples by their max-class probability (the softmax probability of the most likely class). For each test sample  $s$  in this order, we then plot the accuracy among test samples that are at least as certain as  $s$ . The left-most measurement is the overall test accuracy, whereas the right-most measurement is simply the correctness of the most certain prediction. To compute the area under the relative calibration curve, we interpolated linearly between the measurements. For OOD recognition, we predicted the class of the test examples in MNIST (in-distribution) and FashionMNIST (OOD) and computed the entropy over softmax probabilities for each example. We then sort them by negative entropy and test the true positive and false positive rates for each possible (binary) decision threshold. The area under this ROC curve is computed in the same way as for relative calibration.

**Comparison to Related Work:** All results from related work are directly taken from the respective papers, and the only results we created ourselves were for MP. For MNIST, we trained the networks with the same number of iterations and training schedule as in the previous experiments, varying the width of the 3-layer MLP between 400, 800, and 1,200 units. For CIFAR-10, we used the same architectures and training setups as in the MNIST experiments (LeNet-5 and a 3-layer MLP, but now with width 501), although LeNet-5 does not need input padding for CIFAR-10 as the image size is bigger.

**Performance Measurements:** All experiments were conducted on a g4dn.xlarge AWS instance with 4 vCPUs, 16GB of memory, and an NVIDIA T4 GPU. Before running the actual experiments, we perform warm-up trials, and the final measurements are averaged over three runs.

The first experiment included three groups of layers: (1) a FirstGaussianLinearLayer and a GaussianLinearLayer (input sizes of 784, output sizes of 1,000) and LeakyReLU (input and output size of 784), (2) a FirstConv2d and Conv2d (input sizes of  $28 \times 28 \times 4$  and output sizes of  $24 \times 24 \times 8$  with a  $5 \times 5$  kernel) and  $2 \times 2$  MaxPool with stride 2 (input size of  $28 \times 28 \times 4$ , output size of  $14 \times 14 \times 4$ ), and (3) training layers (regression, softmax, and argmax) with input sizes of 100. The data was sampled from standard Gaussians (or for input messages, sampling precision-means and the absolute values of the precision from Gaussians), and the old version was based on Git commit "bc3432fcfefc855e62db0ac676703430b0ae488a". For each group, we measured the time required to compute forward and backward messages for 10 separate training examples.

In the second experiment, we compared all available versions on a small MNIST example. The architecture was [Linear(784, 100), LeakyReLU(0.1), Linear(100, 10)] with a batch size of 320. We ran a warm-up trial with one batch, followed by the main experiment with 10 batches in 1 epoch with 2 iterations (2 epochs in Torch). In the third experiment, we ran four episodes with a total of six iterations (1, 1, 2, 2) for our method, and six episodes for Torch. The training was performed on 3,200 examples, while the inference was conducted on 200,000 examples (by concatenating 20 copies of the original test set).