PySigma: Towards Enhanced Grand Unification for the Sigma Cognitive Architecture

Jincheng Zhou[1,2(✉)] and Volkan Ustun[2]

[1] Department of Computer Science, University of Southern California, Los Angeles, USA
jinchenz@usc.edu
[2] Institute for Creative Technologies, University of Southern California, Los Angeles, USA
{jzhou,ustun}@ict.usc.edu

Abstract. The Sigma cognitive architecture is the beginning of an integrated computational model of intelligent behavior aimed at the grand goal of artificial general intelligence (AGI). However, whereas it has been proven to be capable of modeling a wide range of intelligent behaviors, the existing implementation of Sigma has suffered from several significant limitations. The most prominent one is the inadequate support for inference and learning on continuous variables. In this article, we propose solutions for this limitation that should together enhance Sigma's level of *grand unification*; that is, its ability to span both traditional cognitive capabilities and key non-cognitive capabilities central to general intelligence, bridging the gap between symbolic, probabilistic, and neural processing. The resulting design changes converge on a more capable version of the architecture called PySigma. We demonstrate such capabilities of PySigma in neural probabilistic processing via deep generative models, specifically variational autoencoders, as a concrete example.

Keywords: Sigma · Cognitive architecture · Probabilistic graphical model · Message passing algorithm · Approximate inference · Deep generative model

1 Introduction

The Sigma cognitive architecture is the beginning of an integrated computational model of intelligent behaviors, with an end goal of becoming a working implementation of a complete cognitive system [13]. In service of this goal, four design desiderata have been in place to guide the research and development: (1) *grand unification*, that the architecture should span both traditional cognitive capabilities and key non-cognitive aspects – such as the symbolic, probabilistic, and neural processings – central to an intelligent agent, (2) *generic cognition*, that it should span both natural and artificial cognition at an appropriate level

B. Goertzel et al. (Eds.): AGI 2021, LNAI 13154, pp. 355–366, 2022.
https://doi.org/10.1007/978-3-030-93758-4_36

of abstraction, (3) *functional elegance*, that it should yield human-level intelligent behaviors from a simple and theoretically elegant base, and (4) *sufficient efficiency*, that it should execute quickly enough for real-time applications.

Sigma is implemented as a two-level architecture: a cognitive language level on top of a graphical architecture level. The cognitive language level provides a user-friendly interface for programming Sigma models by offering two essential constructs: `Predicates` that store and represent first-order relational knowledge and `Conditionals` that organize predicates into graphs and rules that enable deriving new knowledge and update existing memories. The graphical architecture level compiles the model written in the cognitive language into an *augmented factor graph*. This augmented factor graph inherits the essential semantics of the conventional factor graphs but augments them in several key ways to allow broader model capacity, such as allowing a variable node to represent first-order variables and introducing unidirectional edges. The actual computations then occur over a sequence of cognitive cycles by, within each, passing messages around this factor graph until quiescence (the elaboration phase) and then updating message contents in factor nodes as necessary (the modification phase).

Sigma's graphical architecture has successfully assisted in modeling a wide range of cognitive and non-cognitive abilities [13]. However, throughout the years of research on it, it has been apparent that the current implementation suffers from at least two fundamental limitations: difficulty in dealing with continuous variables; and lack of a critical cognitive capability for structure learning from experience; known as "chunking." In this article, we focus primarily on tackling the first limitation. We first propose a fundamental change to the message structure to efficiently represent continuous variables, accompanied by an enhanced design of the graphical architecture, message propagation algorithm, and inference/learning mechanisms. These changes ultimately converge on a new version of Sigma called PySigma that, by design, inherits all of Sigma's inferential capabilities and also extends them in significant ways. We then illustrate one of such extension regarding the support for neural probabilistic processing by demonstrating the modeling of a canonical deep generative model – the Variational Autoencoder (VAE) [8]. Finally, we briefly touch upon a potential future research direction inspired by the experience of modeling VAE that may lead to solutions to the second limitation.

2 Message Representation with Continuous Variables

The cornerstone of Sigma's graphical architecture is the *message*: a data structure that captures the first-order relational knowledge about certain facts. For example, the location of an object in the blocks world environment can be represented by a predicate `Location(O:object, X:value, Y:value)`, where variable `O` of type `object` refers to the specific object, and variables `X, Y` of type `value` describes the object's X-Y coordinate. The specific value of this predicate at any given time is then stored as a message with the same signature: `m(O:object, X:value, Y:value)`.

In Sigma, the message data structure leverages Piecewise-Linear Maps [13]. This implementation can approximate functions over continuous variables; however, it can get quite messy when learning the probability density function of a continuous random variable. For example, after multiple cycles of updating the piecewise-linear map, the messages tend to become too fractured and intractable to compute efficiently unless proper smoothing is applied.

To remedy this issue in PySigma, we recognize that a message is essentially encoding a *batch of independently distributed distributions*. Drawing inspiration from Tensorflow's distribution package design [4], we make the distinction between a message's variables that index the batch of distributions versus those variables that are their event variables. The former are called *Relational Variables* in PySigma, and the latter *Random Variables*. Then, to tractably encode the distributions, the Piecewise-Linear Map representation is replaced with two alternatives: the parameter and particle-lists format. A message assumes that the distributions come from a parametric distribution family and uses the parameter vectors to encode its distributions with the parameter format. With the particle-lists format, particles (or samples from the distributions) are stored rather than the parameters, allowing estimation of the statistics of the underlying distributions [3]. Under such formulation, any Piecewise-Linear Map message in Sigma can be effectively represented by a particle-lists format message in PySigma, enabling PySigma to be fully reduced to Sigma.

Because tensors can best represent components of both parameters and particle lists, we build PySigma's message structure on top of PyTorch tensors and implement PySigma's graphical architecture essentially as pipelines of tensor manipulations. Since PyTorch is a commonly used deep learning library, such design choices not only ensure the speed and accuracy of any tensor operation but also prepares PySigma for further integration with standard deep learning modules, such as deep neural networks [11].

3 Generalized Factor Graph, Approximate Inference, and Gradient-Based Learning

Apart from the new message representation and implementation, PySigma fundamentally changes several other aspects of Sigma's graphical architecture. First, PySigma cognitive models are compiled to an augmented factor graph that is further generalized, specifically in the factor node function formulation. In Sigma, a factor node function is formulated similarly to the tabular factor in a conventional factor graph and implemented as a Piecewise-Linear Map, which records every function value corresponding to each tuple of the function variable values. However, such a formulation needs exponential space when the number of function variables increases. PySigma relaxes this restricted formulation and instead relies on the following one, which defines two types of factor functions:

Definition 1. *An n-ary* ***generative factor function*** F *is a mapping* $\prod_{i=1}^{n} R_{V_i} \rightarrow [0, 1]$ *such that*

$$p = F(V_1, V_2, \ldots, V_n)$$

An n-ary ***discriminative factor function*** *G is a mapping* $\prod_{i=1}^{n-1} R_{V_i} \rightarrow R_{V_n}$ *such that*

$$V_n = G(V_1, V_2, \ldots, V_{n-1})$$

where $V_1, V_2, \ldots, V_n$ *are the random variables, and* R_{V_i} *is the support set of the random variable* V_i*.*

By doing so, PySigma admits arbitrary function implementations that conform to the above definition as factor node functions. Thus, for example, a deep neural network module can be used as a factor function approximator to resolve the issue of space explosion.

It should be noted that how PySigma integrates neural networks is very different from the previous approach in Sigma [12,14]. In Sigma, neural networks are reimplemented leveraging the existing components of the Sigma graphical architecture. What is done here for PySigma, on the other hand, extends the architecture to admit neural modules as the primary, inseparable components. Thus, compared to Sigma, PySigma is more deeply integrated with neural processing and hence, is prepared to take one step forward to bridge the gap between neural, probabilistic, and symbolic reasoning.

All of Sigma's message passing, inference, and learning mechanisms have been overhauled in PySigma to work with the new message structure and the generalized factor graph. PySigma still inherits the same two-phased cognitive cycles as in Sigma, but the implementations of both phases are fundamentally changed. PySigma also clearly distinguish between inference and learning: inference performs local updates on the predicates' working memory based on incoming messages to find the posterior distribution given the observations, whereas learning updates the model parameters to optimize a pre-defined objective function.

For message passing, the existing Sigma uses the Sum-Product algorithm, a form of exact inference algorithm, which suffers from tractability issues when facing continuous variables and generalized factor functions [13]. For PySigma, we have developed a set of generalized message passing rules to perform approximate inference. It is a combination of Particle Belief Propagation (PBP) [3,5] and Variational Message Passing (VMP) [2,15] on factor graphs. In the most general case, a conditional subgraph (the group of variable and factor nodes that a conditional is compiled to) expects all incoming messages from the incident predicate subgraph (the group of nodes that a predicate is compiled to) to be of particle-lists format. It then computes outgoing messages using the PBP message update rule on factor nodes [5]. However, if all of the incoming messages are also of parameter format and their assumed distribution classes are not only the exponential class but also conjugate to the factor function, then the VMP message update rule on factor nodes is used, which directly manipulates the exponential family distributions' natural parameters [15]. Such a hybrid message passing algorithm mostly produces the generalized particle-based approximate messages. However, the algorithm efficiently generates a parameter-based exact message when the incoming messages and the factor function share particular structures. Moreover, message passing in PySigma would be identical to the

Sum-Product rules if messages are discrete and the factor functions are in tabular format. In this way, PySigma can be easily reduced to Sigma, thus supporting all of the cognitive capabilities that Sigma's inference can achieve in principle.

For the inference update, we follow the PBP formulation so that particles are only sampled at the predicate subgraph, and a conditional subgraph does not alter incoming particles or sample new particles. This restriction is to prevent incoming messages to a predicate subgraph from having contradicting lists of particles [5]. We thus categorize PySigma's predicates into three types: the *non-memorial* type, the *variational memorial*, and the *Markov Chain Monte Carlo (MCMC) memorial*. The non-memorial type predicate is reserved for relaying messages from one conditional to another. The last two memorial types, however, actively maintain an internal message as their working memory and updates this message using the incoming messages. A variational memorial predicate explicitly assumes that a variational posterior distribution describes its memory and updates this memory by directly summing the incoming message's parameters if all of the incoming messages are of parameter format [2]. Otherwise, if any of the incoming messages are of particle-lists format, it resorts to the stochastic gradient update, similar to the update method proposed by the Reparameterization Gradient Message Passing [1]. Unlike the variational one, the MCMC memorial predicate does not make any assumption about the posterior distribution but relies on the MCMC method to iteratively update its memory particles [7].

Finally, learning is carried out in PySigma by gradient backpropagation, the same technique used for training deep neural networks. A PyTorch computational graph [11] that is automatically built when PySigma propagates messages in the elaboration phase of each cognitive cycle enables the gradient backpropagation. Although PyTorch automatically does backpropagation, PySigma actively controls the scope of the gradient flow to prevent one part of the graph from affecting the learning of another irrelevant part. Such a control mechanism also enables PySigma to choose different optimizers for each scope of the gradient and apply different fine-tuning techniques for each optimizer, such as early-stopping and learning rate schedules.

4 Deep Generative Modeling in PySigma

The expressive power of the new message representation scheme, combined with the generality of the generalized message propagation, inference, and learning algorithm, enables PySigma to capture an even more comprehensive range of models than Sigma, hence taking a step forward toward the ultimate goal of grand unification. Such increased capability can be best demonstrated by modeling deep generative models, a class of probabilistic models that takes strength from deep neural networks, which Sigma has difficulties modeling. To illustrate the deep generative modeling in PySigma, we will discuss one specific such model, the variational autoencoder (VAE), in detail [8], demonstrate how to model it in PySigma, and analyze the correctness of the resulting PySigma model.

4.1 Preliminaries

Variational Autoencoders are canonical deep generative models widely used for learning a smooth representation space for continuous data such as images and audio. It is a successful fusion between probabilistic modeling and deep neural networks: the model can be mathematically analyzed as a probabilistic graphical model at a conceptual level but relies on deep neural networks for the implementation and backpropagation-based stochastic gradient descent for optimization.

From a probabilistic inference perspective, VAE is a class of methods for working with the Latent Variable Model, a class of models where a group of latent variables is interlinked to explain the behaviors of the observed variables. Figure 1a shows the simplest possible latent variable model as a Bayesian network that consists of only a latent variable $\mathbf{z}$ and an observed variable $\mathbf{x}$. Figure 1b presents the factor graph representation of this model. The $p_\theta(\mathbf{z})$ factor node encodes the prior distribution over z, and $p_\theta(\mathbf{x} \mid \mathbf{z})$ factor node encodes the conditional distribution that determines the underlying dynamics between z and x. Throughout the following sections, we will concentrate on this simple model for illustration. We will assume a known and fixed prior distribution and call the unknown conditional distribution $p_\theta(\mathbf{x} \mid \mathbf{z})$ the *reconstruction model*. We will also assume the reconstruction model comes from a very general model family $\mathcal{P}$ for which exact inference algorithms such as the Sum-Product algorithm cannot solve in polynomial time. However, despite the generality, the reconstruction models can be parameterized by a parameter θ, and the probability density function of the model is differentiable with respect to θ.

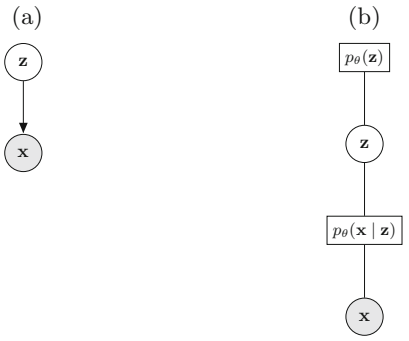

Fig. 1. Left (a): simple latent variable model as a directed Bayesian network. Right (b): the same model expressed as a factor graph.

There are two objectives of the above latent variable model that are worth pursuing:

1. **Inference:** with the reconstruction model $p_\theta(\mathbf{x} \mid \mathbf{z})$ fixed, find the posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$ for the latent variable, given data $\mathbf{x} \in \mathcal{D}$.
2. **Learning:** to select a reconstruction model $p_\theta(\mathbf{x} \mid \mathbf{z})$ from the model family $\mathcal{P}$ such that the entire graphical model (prior + reconstruction) best fit the data distribution $\mathcal{D}$.

VAE approaches the inference task by introducing an *amortized* variational posterior $q_\phi(\mathbf{z} \mid \bar{\mathbf{x}})$ that is implemented by a neural network $f_\phi : \mathcal{X} \to \Lambda$ mapping a mini-batch of observed data points $\bar{\mathbf{x}}$ to the posterior distribution parameter λ that is then used to instantiate the variational posterior. This neural network is also known as the *recognition model*. Therefore, rather than iteratively finding a posterior $q(\mathbf{z})$ for multiple steps each time the reconstruction model $p_\theta(\mathbf{x} \mid \mathbf{z})$ is updated, VAE more efficiently obtains an approximated posterior through a single forward propagation of the neural network f_ϕ. Moreover, the inference task is converted into yet another learning task that is to update f_ϕ, and both the inference and learning tasks can be solved simultaneously by jointly optimizing the Evidence Lower Bound Objective (ELBO) for parameters θ and ϕ:

$$\mathcal{L}(\theta, \phi; \bar{\mathbf{x}}) = \mathbb{E}_{q_\phi(\mathbf{z}|\bar{\mathbf{x}})}\left[\log \frac{p_\theta(\bar{\mathbf{x}} \mid \mathbf{z}) p_\theta(\mathbf{z})}{q_\phi(\mathbf{z} \mid \bar{\mathbf{x}})}\right] \tag{1}$$

where $\bar{\mathbf{x}}$ is a mini-batch of observed data points. Accordingly, the updates are:

$$\phi \leftarrow \phi + \nabla_\phi \tilde{\mathcal{L}}(\theta, \phi; \bar{\mathbf{x}}) \tag{2}$$

$$\theta \leftarrow \theta + \nabla_\theta \tilde{\mathcal{L}}(\theta, \phi; \bar{\mathbf{x}}) \tag{3}$$

where $\tilde{\mathcal{L}}$ is the Monte Carlo estimate to the lower bound $\mathcal{L}$. Updating ϕ effectively solves the inference task, and updating θ effective solves the learning task.

4.2 VAE as a Message-Passing Factor Graph

With the VAE objective and optimization mechanics established, we now consider modeling the VAE as a factor graph to prepare for a PySigma implementation. VAEs, although inherently are directed latent variable models, are often optimized with a black-box optimization procedure (or end-to-end training in the deep learning context) in practice. In such a procedure, the model itself (including both the "reconstruction" model p_θ and the "recognition" model q_ϕ) is treated as a black box and optimized by first taking a global gradient back-propagation to the parameters of both models p_θ and q_ϕ with a subsequent joint parameter update to both θ and ϕ. This black-box optimization procedure, however, is very different from the design principles of a factor graph. The latter emphasizes breaking a global model into local pieces and relies on locally correct message update procedures to achieve the global optimization objective.

Fortunately, although modeling VAEs while completely conforming to conventional factor graph semantics is very hard, the task is simpler if we consider the augmented factor graph PySigma leverages. Figure 2a shows a modified version of the model in Fig. 1b, where a unidirectional message passing gadget is added alongside the original model, encapsulating the recognition model $q_\phi(\mathbf{z} \mid \mathbf{x})$. Figure 2b shows the actual compiled PySigma model, which will be analyzed in the next section.

To start, we notice that Eq. (1) can be expressed in the following factorized format:

$$\mathcal{L}(\theta, \phi; \bar{\mathbf{x}}) = \mathbb{E}_{q_\phi(\mathbf{z}|\bar{\mathbf{x}})}\Big[\log p_\theta(\bar{\mathbf{x}} \mid \mathbf{z}) + \log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z} \mid \bar{\mathbf{x}})\Big] \tag{4}$$

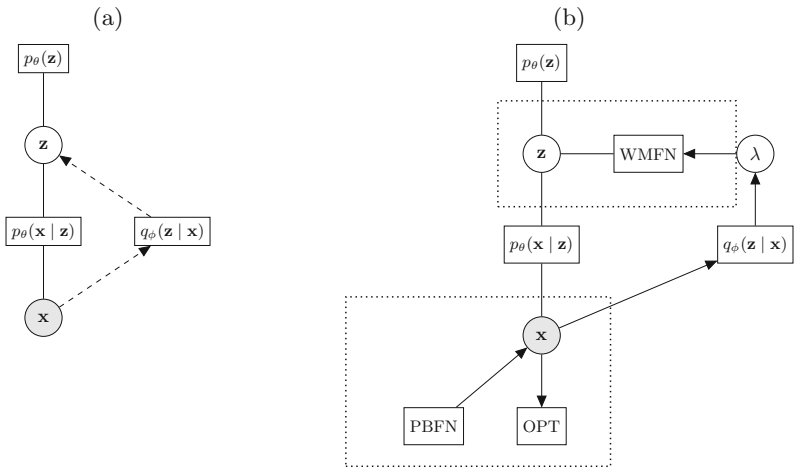

Fig. 2. Left (a): Latent variable model with conditional recognition factor. Note that the two unidirectional dashed arrows indicate that this model uses PySigma's augmented factor graph semantics. Right (b): Compiled PySigma model of the Variational Autoencoder. The nodes grouped by a dashed rectangle together construct a predicate subgraph.

At first glance, the three terms to be taken expectation seem to exactly map onto the three factor nodes in Fig. 2a, yet the subtraction sign before the third term indicates otherwise. Indeed, messages from factor nodes in a conventional factor graph will be taken product with at the variables nodes, leading to logarithmic summations in the overall probability density. However, this challenge can be overcome if we view $q_\phi(\mathbf{z} \mid \mathbf{x})$ not as a conventional factor, but as a *sampling factor*, from which particles of $\mathbf{z}$ are sampled. In this formulation, the term $p_\theta(\mathbf{z})/q_\phi(\mathbf{z} \mid \mathbf{x})$ as well as the term $p_\theta(\mathbf{x} \mid \mathbf{z})/q_\phi(\mathbf{z} \mid \mathbf{x})$ can be interpreted as weights of the particles drawn from $q_\phi(\mathbf{z} \mid \mathbf{x})$ that are *important weighted* against the distributions $p_\theta(\mathbf{z})$ and $p_\theta(\mathbf{x} \mid \mathbf{z})$ respectively.

To further illustrate the model's correctness under PBP, we break down the particle messages when the factor graph reaches quiescence, which is shown in Table 1. The "Particle Source" indicates where the particles of each message were originally drawn from, the "Log Sampling Densities" indicate the particles' relative frequencies, and the "Importance Weight" is the importance weight of the particles derived by dividing the target values by the particles' log sampling densities.

Table 1. Messages of the model in Fig. 2a when in quiescence state

Direction	Particle source	Importance weight	Log sampling densities
$\mathbf{x} \to q_\phi(\mathbf{z} \mid \mathbf{x})$	Data points $\bar{x} \in \mathcal{D}$	Uniform	Uniform
$q_\phi(\mathbf{z} \mid \mathbf{x}) \to \mathbf{z}$	Particles $z \sim q_\phi(\mathbf{z} \mid \mathbf{x})$	Uniform	$\log q_\phi(z \mid \bar{x})$
$\mathbf{x} \to p_\theta(\mathbf{x} \mid \mathbf{z})$	Data points $\bar{x} \in \mathcal{D}$	Uniform	Uniform
$p_\theta(\mathbf{x} \mid \mathbf{z}) \to \mathbf{z}$	Particles $z \sim q_\phi(\mathbf{z} \mid \mathbf{x})$	$\sum_{\bar{x}} p_\theta(\bar{x} \mid z)/q_\phi(z \mid \bar{x})$	$\log q_\phi(z \mid \bar{x})$
$\mathbf{z} \to p_\theta(\mathbf{z})$	Particles $z \sim q_\phi(\mathbf{z} \mid \mathbf{x})$	$\sum_{\bar{x}} p_\theta(\bar{x} \mid z)/q_\phi(z \mid \bar{x})$	$\log q_\phi(z \mid \bar{x})$
$p_\theta(\mathbf{z}) \to \mathbf{z}$	Particles $z \sim q_\phi(\mathbf{z} \mid \mathbf{x})$	$p_\theta(z)$	$\log q_\phi(z \mid \bar{x})$
$\mathbf{z} \to p_\theta(\mathbf{x} \mid \mathbf{z})$	Particles $z \sim q_\phi(\mathbf{z} \mid \mathbf{x})$	$p_\theta(z)$	$\log q_\phi(z \mid \bar{x})$
$p_\theta(\mathbf{x} \mid \mathbf{z}) \to \mathbf{x}$	Data points $\bar{x} \in \mathcal{D}$	$\sum_z p_\theta(\bar{x} \mid z)p_\theta(z)/q_\phi(z \mid \bar{x})$	Uniform

We can thus derive the marginal posterior for both node $\mathbf{x}$ and node $\mathbf{z}$. The former is simply the message $p_\theta(\mathbf{x} \mid \mathbf{z}) \to \mathbf{x}$, whereas the latter is the product of two messages: $p_\theta(\mathbf{z}) \to \mathbf{z}$ and $p_\theta(\mathbf{x} \mid \mathbf{z}) \to \mathbf{z}$.

$$post(\mathbf{x}) = \sum_z \frac{p_\theta(\bar{x} \mid z)p_\theta(z)}{q_\phi(z \mid \bar{x})} \tag{5}$$

$$post(\mathbf{z}) = \sum_{\bar{x}} \frac{p_\theta(\bar{x} \mid z)p_\theta(z)}{q_\phi(z \mid \bar{x})} \tag{6}$$

where $\bar{x} \in \mathcal{D}$ and $z \sim q_\phi(z \mid \bar{x})$. Here the notation *post* denotes that the values estimate the marginal posterior.

Looking back to the global ELBO Eq. (1), we notice that expressions (5) and (6) are both a Monte Carlo estimator to the ELBO, only with the former summarizing over variable $\mathbf{z}$ and the latter summarizing over variable $\mathbf{x}$. Therefore, if we optimize $post(\mathbf{x})$ with respect to the model parameter θ and optimize $post(\mathbf{z})$ with respect to the variational parameter ϕ separately, we recover the optimization procedures (2) and (3). Moreover, since both optimization on θ and ϕ are done separately and only using local incoming messages, we have achieved the goal of global optimization via local updates.

4.3 Model Implementation in PySigma

Figure 2b presents the schema of a compiled PySigma model that implements the variational autoencoder. It is an augmented version of the factor graph in

Fig. 2a with several PySigma's special purpose factor nodes added to complete the predicate subgraph.

At the bottom, the Perception Buffer Factor Node (PBFN) perceives a batch of data points at the start of each cognitive cycle and encapsulates the data points as message particles with uniform log sampling densities. The Optimization Node (OPT) calculates the loss value (5) by taking the product of the incoming message's particle weights. This loss value will then be used to back-propagate gradients to the reconstruction factor $p_\theta(\mathbf{x} \mid \mathbf{z})$ during the modification phase. Both PBFN and OPT, together with the observed variable node $\mathbf{z}$, construct the observed predicate structure.

To the right, the recognition factor node $q_\phi(\mathbf{z} \mid \mathbf{x})$, implemented by a neural network module, takes in the batch of data points and produces a batch of variational distribution parameters λ, which is then relayed through the variable node λ. The Working Memory Factor Node (WMFN) receives λ and instantiates a batch of independent variational distributions, from which particles are sampled with non-uniform log sampling densities. During the modification phase, it computes the loss value (6) and propagates gradients back to the recognition factor $q_\phi(\mathbf{z} \mid \mathbf{x})$. Both the WMFN and the variable node $\mathbf{z}$ together construct the latent predicate subgraph.

5 Future Work: Chunking

Chunking was implemented years ago in the Soar cognitive architecture, a predecessor to Sigma, where trees of rule firings could be summarized and replaced by single rules that are efficient to compute [10]. Unfortunately, due to the probabilistic nature of Sigma, summarizing Sigma's probabilistic logical rules is much harder, and there is yet to be a satisfactory solution. However, the modeling of VAE in PySigma provides an exciting lead. The recognition subgraph, in particular, might be generalizable in that it can be attached to any PySigma models to efficiently approximate the posterior of an arbitrary predicate given observations at some other predicates.

The implication of the recognition subgraph being an architectural pattern is far-reaching. For example, the parallel existence of a complex reasoning pathway with a simple neural approximator within a cognitive model can be related to the conception of fast and slow thinking in cognitive science [6]. Moreover, The ability of the architecture to automatically decide whether to attach such a recognition subgraph or an active control over the usage of the slow and fast message pathways are canonical metacognitive capabilities that are significant to the general intelligence [9]. Thus, it is worthwhile to investigate how well VAE can be further generalized in PySigma.

6 Conclusion

Sigma cognitive architecture and system results from decades of research on the integrated computational model of intelligent behaviors. Sigma has successfully

modeled a wide range of capabilities yet faces a severe challenge regarding inference and learning with continuous variables. In this article, we presented several fundamental changes to Sigma's implementation that converge on a new version of Sigma called PySigma. We also provided a glimpse of the new capability these changes have unveiled, particularly the support for deep generative models. Overall, we demonstrated that PySigma is more capable than Sigma to unite neural and probabilistic processing, hence taking a step toward the ultimate desiderata of grand unification.

Acknowledgements. Part of the effort depicted is sponsored by the U.S. Army Research Laboratory (ARL) under contract number W911NF-14-D-0005, and that the content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. We also would like to thank Dr. Paul Rosenbloom for his comments and suggestions, which helped improve the quality of this paper. More importantly, we appreciate Dr. Rosenbloom's continuous and invaluable guidance in enhancing our understanding of cognitive architectures and the design choices for Sigma.

References

1. Akbayrak, S., De Vries, B.: Reparameterization gradient message passing. In: European Signal Processing Conference, EUSIPCO, September 2019 (September 2019). https://doi.org/10.23919/EUSIPCO.2019.8902930
2. Dauwels, J.: On variational message passing on factor graphs. In: Proceedings of the IEEE International Symposium on Information Theory, pp. 2546–2550 (2007). https://doi.org/10.1109/ISIT.2007.4557602
3. Dauwels, J., Korl, S., Loeliger, H.A.: Particle methods as message passing. In: Proceedings of the IEEE International Symposium on Information Theory, pp. 2052–2056 (2006). https://doi.org/10.1109/ISIT.2006.261910
4. Dillon, J.V., et al.: TensorFlow distributions (2017). http://arxiv.org/abs/1711.10604
5. Ihler, A., McAllester, D.: Particle belief propagation. In: van Dyk, D., Welling, M. (eds.) Proceedings of the 12th International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 5, pp. 256–263, 16–18 April 2009. PMLR, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA (2009). http://proceedings.mlr.press/v5/ihler09a.html
6. Kahneman, D.: Thinking, Fast and Slow. Farrar, Straus and Giroux, New York (2011). https://doi.org/10.1037/h0099210
7. Kim, H., Robert, C.P., Casella, G.: Monte Carlo statistical methods. Technometrics **42**(4), 430 (2000). https://doi.org/10.2307/1270959
8. Kingma, D.P., Welling, M.: Auto-encoding variational Bayes. In: 2nd International Conference on Learning Representations, Conference Track Proceedings, ICLR 2014, Banff, AB, Canada, 14–16 April 2014 (2014)
9. Kralik, J.D., et al.: Metacognition for a common model of cognition. Procedia Comput. Sci. **145**, 730–739 (2018). https://doi.org/10.1016/j.procs.2018.11.046, https://www.sciencedirect.com/science/article/pii/S1877050918323329. Postproceedings of the 9th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2018 (Ninth Annual Meeting of the BICA Society), held 22–24 August 2018 in Prague, Czech Republic

10. Laird, J.E.: The Soar Cognitive Architecture (2018). https://doi.org/10.7551/mitpress/7688.001.0001
11. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019). https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf
12. Rosenbloom, P.S., Demski, A., Ustun, V.: Rethinking sigma's graphical architecture: an extension to neural networks. In: Steunebrink, B., Wang, P., Goertzel, B. (eds.) AGI-2016. LNCS (LNAI), vol. 9782, pp. 84–94. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41649-6_9
13. Rosenbloom, P.S., Demski, A., Ustun, V.: The sigma cognitive architecture and system: towards functionally elegant grand unification. J. Artif. Gen. Intell. **7**(1), 1–103 (2016). https://doi.org/10.1515/jagi-2016-0001. https://www.degruyter.com/downloadpdf/j/jagi.2016.7.issue-1/jagi-2016-0001/jagi-2016-0001.pdf
14. Rosenbloom, P.S., Demski, A., Ustun, V.: Toward a neural-symbolic sigma: introducing neural network learning. In: Proceedings of the 15th International Conference on Cognitive Modeling, ICCM 2017, pp. 73–78 (2017). http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture13.html
15. Winn, J.: Variational message passing and its applications. Ph.D. thesis (2003). http://johnwinn.org/Publications/thesis/Winn03_thesis.pdf