To Appear in the *Proceedings of the 15th Annual Meeting of the International Conference on Cognitive Modeling*, 2017.

Toward a Neural-Symbolic Sigma: Introducing Neural Network Learning

Paul S. Rosenbloom[a,b] (Rosenbloom@USC.Edu), Abram Demski[a,b] (ADemski@ICT.USC.Edu), Volkan Ustun[a] (Ustun@ICT.USC.Edu)

[a]Institute for Creative Technologies & [b]Department of Computer Science, University of Southern California
12015 Waterfront Dr., Playa Vista, CA 90094 USA

Abstract

Building on earlier work extending Sigma's mixed (symbols + probabilities) graphical band to inference in feedforward neural networks, two forms of neural network learning – target propagation and backpropagation – are introduced, bringing Sigma closer to a full neural-symbolic architecture. Adapting Sigma's reinforcement learning (RL) capability to use neural networks in policy learning then yields a hybrid form of neural RL with probabilistic action modeling.

Keywords: cognitive architecture; neural-symbolic; neural networks; learning; reinforcement learning

Introduction

One of the greatest overall challenges in cognitive modeling is developing cognitive architectures that bridge the biological and cognitive bands – spanning, respectively, 100 μs - 10 ms and 100 ms - 10 s – from Newell's (1990) analysis of the time scales of human action. The boundary between these bands sits somewhere in the region of 10-100 ms and conventionally divides symbolic from subsymbolic behavior, although the relationship between them may be in reality both subtler and more complex.

One approach to this challenge provides distinct mechanisms for the two bands that can cooperate in prescribed ways (Sun, 2016); a second seeks the emergence of cognitive mechanisms from biological ones (Eliasmith, 2013); and a third replaces components of existing cognitive architectures with neural models that yield similar results (Cho, Rosenbloom & Dolan, 1991; Jilk et al., 2008).

The approach taken in Sigma (Rosenbloom, Demski, & Ustun, 2016a) has been to generalize the notion of a biological band to that of a *graphical band* – which in Sigma is based on factor graphs, a general form of *graphical model*, plus the summary product message-passing algorithm (Kschischang, Frey & Loeliger, 2001) – that then implements the cognitive band. Recently it was discovered, however, that with one simple enhancement to this graphical band it was possible to include *feedforward neural networks*, without yet learning, among the graphs supported (Rosenbloom, Demski & Ustun, 2016b). This inspired a rethinking of Sigma's graphical band to a broader graphical notion within which factor graphs became just one particularly useful specialization and neural networks another. It also raised the possibility of a broader variation on the third approach mentioned above.

This preliminary work is extended here to weight learning in feedforward neural networks. A general form of parameter learning, via gradient descent on factor functions, was first implemented in Sigma for probability distributions (Rosenbloom et al., 2013) and then later extended to distributed vectors (Ustun et al., 2014). These are both forms of *generative learning* that learn patterns of coactivation across variables, much as in *Hebbian learning*.

Starting with this approach for distributed vectors, a variant of *target propagation* (Lee et al., 2015) has been implemented in Sigma via normal undirected (bidirectional) factor graphs, by backward propagating target values for the units' outputs, and discriminatively learning weights from differences between target and actual outputs. However, issues with this approach led us also to implement *backpropagation*, the standard discriminative approach to neural learning (Rumelhart, Hinton & Williams, 1986), that is based instead on a unidirectional forward-backward arc.

Both of these approaches reuse Sigma's message passing for backward propagation and its gradient descent for parameter learning. Backpropagation also leverages a variant of affective *appraisal* (Rosenbloom, Gratch & Ustun, 2015) to compute the error needed to initiate the backward pass. The net result is *functionally elegant* neural learning that is largely based on new combinations of existing mechanisms rather than on new modules cut from whole cloth. By extending Sigma's graphical band in this way, neural networks are potentially usable wherever factor graphs already were used, including in long-term memory, perception and learning. When combined with the earlier work on distributed vectors, a general *neural-symbolic architecture* begins to emerge that may, among other things, provide principled architectural guidance in how to combine deep learning (Goodfellow, Bengio, & Courville, 2016) with other critical cognitive capabilities.

The core result in this article thus concerns the relatively abstract yet fundamental problem of building a functionally elegant bridge from a cognitive architecture to the biological band rather than specific matches to human data. In service of this, after a review of Sigma and its earlier extension to feedforward neural networks, we will introduce neural-network learning in Sigma, followed by experiments with *classification* and *regression* networks, and the leveraging of such networks in *neural reinforcement learning*.

Sigma and Feedforward Neural Networks

Sigma is composed of two distinct architectures, one for the cognitive band and one for the graphical band. In the *cognitive architecture*, knowledge is based on *predicates* for specifying relations over typed – numeric (discrete or continuous) or symbolic – arguments; and *conditionals* for specifying patterns over combinations of predicates. *Functions* may be included in predicates to provide

distributions over their arguments, and in conditionals to provide distributions over combinations of their variables.

A segment of working memory exists for each predicate, as does also a segment of long-term memory if there is a predicate function. An additional segment of long-term memory is also created for each conditional. A pattern in a conditional may be a *condition*, which acts like a rule condition by matching to working memory; an *action*, which acts like a rule action by changing working memory; or a *condact*, which combines the effects of a condition and an action to yield bidirectional constraints on the contents of working memory. Procedural memory is largely based on conditions and actions – i.e., rules – and declarative memory on condacts. Decisions are made by selecting values from predicate arguments based on distributions over them.

Figure 1, for example, displays two conditionals – each effectively a (non-symbolic) rule with an associated weight function – that together implement the two-layer neural network in Figure 2. All argument types here are discrete numeric, but with three elements for `Input` and two each for `Hidden` and `Output`. The single argument (`arg`) in each pattern is specified here by variables – *`i`, `h`*, and *`o`* – with the function in each conditional being defined over its pair of variables. The **`s`** in the conditionals' actions denotes that a sigmoid/logistic function is to be applied before working memory is changed (other possibilities include `r` for RELU, `t` for tanh, `e` for exponential, and `x` for softmax). The one modification required to make this work in Sigma was extending to these functions its existing ability to include non-linear transformations in conditional patterns.

This particular way of encoding a neural network in Sigma involves one conditional per layer, with the structure of the layers implicit in the argument types and conditional functions. Although it is also possible to encode such networks via one conditional per link, with one element per type and a single weight per function, here the focus is on the more concise representation illustrated in Figure 1.

```
CONDITIONAL C-Layer1
  Conditions: (Input arg:i)
  Actions: (Hidden s arg:h)
  Function<i,h>: .2:<0,0>, .7:<0,1>, …

CONDITIONAL C-Layer2
  Conditions: (Hidden arg:h)
  Actions: (Output s arg:o)
  Function<h,o>: 1.1:<0,0>, 3.1:<0,1>, …
```

Figure 1: Conditionals for the network in Figure 2.

Sigma's compiler converts knowledge specified in its cognitive architecture into undirected bipartite graphs of variable and factor nodes – essentially factor graphs – in the *graphical architecture*. Functions are stored in factor nodes. Processing occurs via message passing – essentially the summary product algorithm – with each message encoding a distribution over the variables in the variable node on the link. Incoming messages are pointwise multiplied together at nodes, along with the node function at factor nodes, and then variables not needed in outgoing messages are summarized out, typically via either integral or maximum. For conditions and actions, messages are passed in only one direction, from working memory for conditions and towards working memory for actions, whereas condact message passing is bidirectional. Learning occurs by gradient descent at factor nodes, with gradients based on messages arriving from adjacent variable nodes.

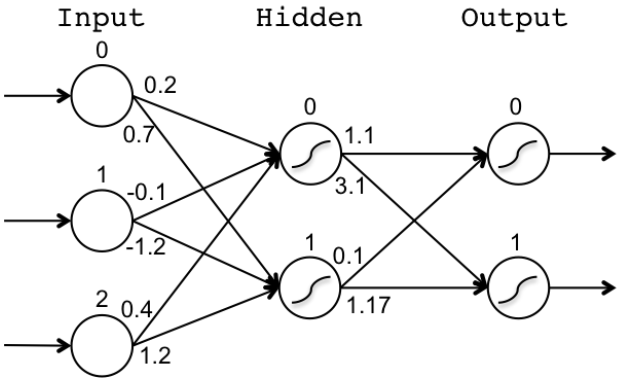

Figure 2: Two-layer neural network (adapted from http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture13.html).

Target Propagation

With target propagation, *targets* – that is, desired values – rather than errors are propagated backward over the network, with errors then computed locally at factor nodes based on subtracting computed outputs from desired outputs. To support this, the unidirectional rules in Figure 1 are converted to bidirectional constraints, with conditions and actions becoming condacts, as shown in Figure 3.

```
CONDITIONAL C-Layer1-TP
  Condacts: (Input arg:i)
            (Hidden s arg:h)
  Neural:h
  Function<i,h>: <Random in [-.1,.1]>

CONDITIONAL C-Layer2-TP
  Condacts: (Hidden arg:h)
            (Output s arg:o)
  Neural:o
  Function<h,o>: <Random in [-.1,.1]>
```

Figure 3: Target propagation conditionals for two-layer weight learning.

The weights in the functions are initialized randomly, and then learned online from training examples. The `Neural` attribute in the conditionals specifies that local discriminative learning is to be used here, with the gradient based on subtracting the output message for the specified variable (i.e., its computed value) from its input message (i.e., its desired/target value). Learning from this error-based gradient then follows the simplified additive form earlier developed for distributed vectors rather than the more complex form originally developed for distributions.

Starting with the targets for the network's output units, computing the targets and gradients for interior units

leverages the bidirectionality of condacts to send messages backward in the graph. However, in contrast to backward messages in normal factor graphs, proper processing of these messages requires that the functions be inverted at factor nodes. This is straightforward for the logistic function, as its inverse is simply the *logit* function: $log(x/[1-x])$. However, this does raise a deeper problem, in that the domain of this function is (0,1), whereas there is no guarantee that a target – particularly one generated inside the network – will fall in this range. To work around this, backward messages at these nodes are truncated to $[\varepsilon,1-\varepsilon]$.

A second problem arises at the factor nodes where the learned weight functions must be inverted. Rather than attempting to do this analytically, inversion is approximated empirically by gradient descent over the node's backward output. In particular, the product of the output error and the weight function is multiplied by a pseudo-learning rate (.05) and then added to the forward input message to yield the backward output message.

Aside from the nonstandard approach to computing backward messages, the result is a form of target propagation that otherwise fits cleanly into normal factor graphs, including respecting the constraint that all messages over a link are distributions over the link's variables.

Backpropagation

With backpropagation, a difference is computed only once, for the network's output units, and propagated backward successively from there. Sigma already supports an architectural *desirability* appraisal that calculates differences between distributions over goals and their associated states, and which is used in both guiding problem solving and directing attention. What is needed for backpropagation is an analogous *correctness* appraisal that operates over point values rather than distributions. The error is then simply the difference between the `output` predicate's specified target/goal and its computed value/state.

Unlike with target propagation, however, the error cannot just be propagated backward over a bidirectional network, as that would violate the constraint that all of the messages on a link should be distributions over the values of the link's variables. In the forward direction the messages are (unnormalized) distributions – effectively *activations* – over variables, each of which corresponds to the set of units at one layer of the network. Sending errors backward over these same links would be invalidly inhomogeneous.

Instead, what has been done is to complement each unidirectional forward network with a unidirectional backward network over which errors are sent, with the appraisal at the end of the forward network serving as the nexus connecting it to the backward network. Figure 4 shows an abstract graph for how this all works.

The left (green) path is the forward one, stretching from the perceptual buffer for the `Input` predicate up through two layers of weights to the `Output` predicate. The squares are factor nodes, where the weight functions are stored, whereas the circles are variable nodes. The two sigmoid transformations occur at additional factor nodes that are abstracted away in this figure. The output of the forward path joins with the target values for the outputs at the appraisal of correctness.

Figure 5 shows the forward conditionals for this. They are like those in Figure 1 in having conditions and actions, and like those in Figure 3 in using random initial weights, but they replace the `Neural` attribute with the `Vector` attribute to signal that distributed-vector gradients should be used in learning without target propagation's gradient-based approach to backward message passing.

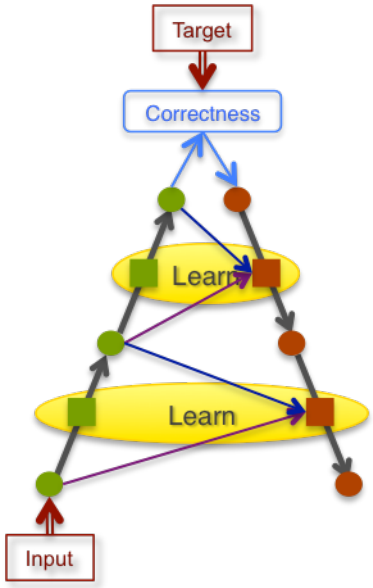

Figure 4: Structure of backpropagation for a two-layer network.

The right (red) path in Figure 4 is the backward one. It includes its own factor and variable nodes, but with crucial linkages added to the forward path. The sigmoid nodes are also abstracted away here, but in the network compute the derivative of the logistic – $x(1-x)$ – rather than its inverse.

```
CONDITIONAL C-Layer1
  Conditions: (Input arg:i)
  Actions: (Hidden s arg:h)
  Function<i,h>: <Random in [-.1,.1]>
  Vector: T

CONDITIONAL C-Layer2
  Conditions: (Hidden arg:h)
  Actions: (Output s arg:o)
  Function<h,o>: <Random in [-.1,.1]>
  Vector: T
```

Figure 5: Forward conditionals for two-layer backpropagation network.

Figure 6 shows the corresponding backward conditionals. In general, backward predicates – such as `Hidden*Back` – are introduced that correspond to the forward ones, and conditions are swapped with actions. However, there are slight variations for the first and last layers. In the first layer, the backward propagation of information can stop at the weight function, rather than going all of the way back to the input units, so there is no action in `C-Layer1-B`. In the last layer, backward propagation starts with the appraisal for the `Output` predicate – `Output*Error` – so the error is used directly rather than a new backward predicate.

Learning occurs based on messages arriving at the factor nodes in the backward path, but the functions in these nodes are *tied* to those in the forward path – shown by the yellow

ellipses in Figure 4 – so that any changes made to the former are directly reflected in the latter. This is enabled by the `Forward-Conditional` attributes in Figure 6, which specify the corresponding forward conditionals.

```
CONDITIONAL C-Layer1-B
  Conditions: (Hidden*Back arg:h)
              (Hidden s arg:h)
  Forward-Conditional: C-Layer1
  Exclude-Forward-Backward T
  Vector: T

CONDITIONAL C-Layer2-B
  Conditions: (Output*Error arg:o)
              (Output s arg:o)
  Actions: (Hidden*Back arg:h)
  Forward-Conditional: C-Layer2
  Exclude-Forward-Backward T
  Vector: T
```

Figure 6: Backward conditionals for two-layer backpropagation network.

In its simplest form, the gradient in backpropagation is the product of: (1) the learning rate; (2) the forward message at the weight function; (3) the output difference; and (4) the derivative of the sigmoid function. The forward message, as shown by the upward slanting (purple) unidirectional links from the forward path, is added automatically to the graph by the conditional compiler given the `Forward-Conditional` attribute. The output difference comes from above in the figure, as derived from the first condition in a backward conditional. The computation of the derivative of the sigmoid, although abstracted away in the graph, arrives at the backward factor node via the downward slanting (blue) links from the forward path, based on the second condition in a backward conditional. As with target propagation, the resulting gradient is handled in the simple additive manner developed earlier for distributed vectors.

In contrast to target propagation, here the backward message out of the factor node – that is, the propagated output difference – is computed simply by message/function multiplication and summarization, as is standard in factor graphs. There is one important caveat though. As specified by the `Exclude-Forward-Backward` attribute, the purple message from the forward path is not included in this product, so the backward output is just the product of the output difference, the derivative of the sigmoid function and the weight function in the node. This exception to the normal rule is motivated by backpropagation, but justified independently in factor graph terms by the fact that a message coming into a node on a bidirectional link should not be used in computing the reverse message on the same link. Here there are two unidirectional links, but they effectively comprise a single logical bidirectional path.

Basic Experiments

Regression and classification problems provide two forms of standard benchmarks for learning with neural networks. The network in Figure 2, for example, defines a regression problem, where two functions are to be learned from the inputs, one for each output. Small experiments with this network, starting with uniform weights, do show that both forms of propagation can learn weights in Sigma that yield outputs like those generated by the network in the figure. But what is really needed for verification is an investigation into how Sigma compares with standard packages.

For this, we have compared Sigma with PyBrain, a Python machine learning library (Schaul et al., 2010), via three standard machine learning datasets: (1) *Iris* – https://archive.ics.uci.edu/ml/datasets/Iris – a classification problem with 3 classes; (2) *Robot Arm* – http://mldata.org/repository/data/viewslug/uci-20070111-kin8nm/ – a regression problem that learns to predict the end effector position for an 8 link robot arm; and (3) *MNIST* – http://yann.lecun.com/exdb/mnist/ – a classification problem over the digits 0-9, based on 28x28 pixel images. Table 1 shows the static information for these datasets.

Table 1: Input, hidden and output units; training and test instances; learning rate; and training epochs.

	I	H	O	Train	Test	λ	Ep.
Iris	4	10	3	138	12	.1	100
Robot	8	100	1	6530	838	.01	100
MNIST	784	30	10	10K[1]	10K	.01	50

Our experiments so far with target propagation have not yet yielded reasonable results on these datasets, most likely because of the truncation required for the logit. So Table 2 only shows backpropagation results. The first and most critical result is that Sigma's accuracy is indistinguishable from that produced by PyBrain with the same settings. Second, Sigma is slower, by up to a factor of ~100. Although a slowdown with a general architecture is not surprising, it should actually be possible to close this gap with a more efficient message representation plus SIMD (as in PyBrain) and GPU hardware. It is also worth note though that these results are at most a factor of 2 slower than the human cognitive cycle time of ~50 ms, a factor that can be relevant when concerned with real-time cognitive models.

Table 2: Accuracy (% correct for Iris and MNIST, RMSE for Robot Arm); seconds per epoch; and ms per decision.

	Py A.	Σ A.	Py s/Ep.	Σ s/Ep	Σ ms/D
Iris	.917	.917	.082	.215	2
Robot	.173	.173	3.51	54.33	8
MNIST	.867	.867	9.8	1029.2	103

Neural Reinforcement Learning

Figure 7 shows a simple 1D grid in which reinforcement learning (RL) was initially explored in Sigma (Rosenbloom, 2012). The agent can move left or right in locations 1 through 6, with locations 0 and 7 being forbidden boundary

Figure 7: 1D grid with agent, goal location and rewards.

[1] Only the first 10K training examples are used for MNIST.

regions. When the goal (location 4, with a reward of 9) is reached the trial halts. This has since been extended to larger 2D grids and to other tasks, but its simplicity provides a good starting point for exploring neural RL.

Like neural learning, Sigma's RL capability is not a distinct architectural module. Instead it is deconstructed in terms of a set of conditionals plus learning of distributions over rewards, state utilities and action policies. Neural RL in Sigma is much like this – Figure 8 – with similar conditionals and learning of the same quantities. For example, the rightward arrow at the bottom of the figure indicates a conditional with a transition function (i.e., an action model) that predicts the location resulting from applying an operator, while the leftward arrow(s) at the top of the figure show the discounted backward propagation of the sum of the projected future utility and the reward for the predicted next state to the projected future utility of the current state and the policy for the current state and reward.

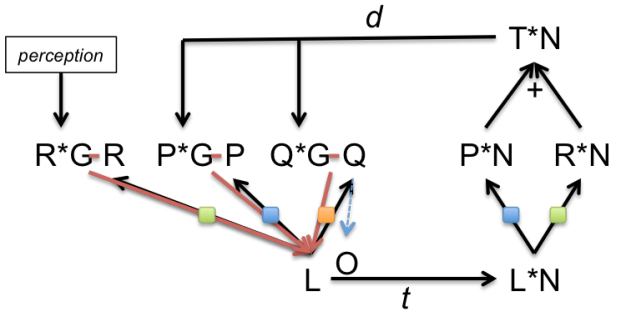

Figure 8: Diagram of neural RL in Sigma.

Still, there are several key differences implied by the top-level shift from distributional to neural learning that go beyond simply which form of learning is used. First, because backpropagation is used, there is a forward-backward arc of unidirectional conditionals (with functions) – as in Figure 4 – for each quantity to be learned, rather than a functional predicate or bidirectional conditional. In Figure 8, the forward paths are the upward black arrows from the location (`L`) to the reward (`R`), the projected future utility (`P`) and the policy (`Q`), whereas the backward paths are the downward red arrows back to `L` from correctness calculations (such as `R*G - R`). The tied functions are shown as path-spanning squares. A single-layer network – i.e., logistic regression – is used here due to the simplicity of the problem, but this can easily be extended to multiple layers.

Second, because neural learning structurally distinguishes input from output in the network, implying an asymmetry that need not exist in distributional learning, the arguments for these quantities must appear in different predicates. Semantically, distributions may be symmetric, as when they are joint, or they may be asymmetric, as when conditional; but both can appear identically in a graphical model. In distributional RL, conditional distributions are learned, but single symmetric predicates – such as `Reward(x:`*`x`*`, value:`*`r`*`)` – are used. For neural RL this must be split in two, to yield `Location(x:`*`x`*`)` and `Reward(value:`*`r`*`)`, as shown by `L` and `R` in the figure.

Third, instead of learning a distribution over all possible output values, with sums (of rewards and projected future utilities) and products (by discount factors) computed by *affine transforms*, in neural learning a single value is learned, with sums resulting from adding the effects of multiple actions (top-right of Figure 8) and products from multiplying the effects of multiple conditions (*d* at top of the figure). For example, in the distributional case the domain of the `value` argument for `Reward` includes all possible rewards, and the function over this and `x` is the conditional distribution over the value given the location. Summing two such values occurs by *translating* the distribution, and discounting by *scaling* it. In the neural case, there is instead only one domain element in the `value` argument, with the function over this element simply the learned reward, and computations on this occurring during pattern combination. Thus, not only is just a point value learned in the neural case, that value is implicit in the range of the learned function rather than explicit in the domain of the function, and computation with it occurs in a rather different manner.

Fourth, because with distributional learning the arguments all exist within one predicate, potentially providing a full cross-product among their elements, a table is effectively acquired from which multiple answers can be extracted simultaneously via conditions with appropriate constants and variables. With neural learning, extracting each answer requires either running the network once for each input, or including a distinct forward network for each possible input (but with tied functions across them). This latter approach has been used in neural RL to access in parallel the rewards and projected utilities of the current state and the predicted next state. In Figure 8, the separate paths for the next state are shown to the right, with function coloring indicating tying to the corresponding functions in the current path.

Despite these differences, Figure 9 shows that the point values learned for the neural policy are still appropriate, with rightward movement preferred when to the left of the goal location and leftward movement when to its right. This policy is averaged over ten runs of 500 trials each, with each trial starting at location 1 or 6, and all ending at location 4.

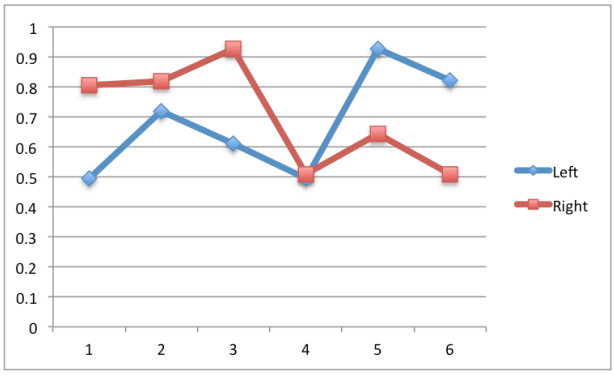

Figure 9: Policy (Q) function learned via neural RL.

Two other things are also worth noting from this simple neural RL experiment, which included an equivalent trial

sequence for the distributional version. First, The neural version was approximately five times faster than the distributional one in terms of time per decision – 12 versus 62 ms/D – largely due to the smaller functions and messages possible when not using full distributions. Second, both versions learn models of their actions distributionally – in terms of a conditional probability distribution over the next location given the current location and action – while engaged in RL. The neural case thus illustrates the ease with which neural and distributional learning can combine in Sigma across subproblems in the same overall problem.

Conclusion

Building upon Sigma's feedforward neural-network inference capability and its distributed vector learning capability, two forms of neural network learning – target propagation and backpropagation – have been implemented via a combination of extensions to existing architectural mechanisms and knowledge expressed as predicates and conditionals. In both variations, the backward propagation of information occurs through message passing in Sigma's graphical architecture rather than via special purpose mechanisms; and in backpropagation, the initial error computation occurs via a form of appraisal.

For backpropagation we get results of comparable quality to, but slower than, a standard package; and we see the possibility of combining it with other capabilities, such as reinforcement learning and probabilistic action modeling. Neural network inference and learning are thus now becoming pervasively available within Sigma's central cognitive cycle, a major step toward a full neural-symbolic architecture that is based on a functionally elegant bridge to the biological band. In addition, although somewhat of a side point here, these extensions enable Sigma to perform discriminative learning over point values in general, whether for use in neural learning or not, to complement the existing ability of generative learning over full distributions.

Future work includes extension to the full power of deep learning and the handling of temporal sequences via techniques such as LSTMs (Hochreiter & Schmidhuber, 1997). Also planned is further optimization and integrations of neural networks with other critical cognitive capabilities.

Acknowledgments

The work described in this article was sponsored by the U.S. Army. Statements and opinions expressed may not reflect the position or policy of the United States Government, and no official endorsement should be inferred. We would also like to thank Sruthi Madapoosi Ravi for significant help with the regression and classification experiments.

References

Cho, B., Rosenbloom, P. S. & Dolan, C. P. (1991). Neuro-Soar: A neural-network architecture for goal-oriented behavior. *Proceedings of the 13th Annual Conference of the Cognitive Science Society* (pp. 673-677).

Eliasmith, C. (2013). *How to Build a Brain.* Oxford: Oxford University Press.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. Cambridge, MA: MIT Press.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*, 1735–1780.

Jilk, D. J., Lebiere, C., O'Reilly, R. C. and Anderson, J. R. (2008). SAL: an explicitly pluralistic cognitive architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, *20*, 197-218

Kschischang, F. R., Frey, B. J. & Loeliger, H.-A. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*. *47*, 498-519.

Lee, D.-H., Zhang, S., Fischer, A., & Bengio, Y. (2015). Difference target propagation. In A. Appice, P. P. Rodrigues, V. S. Costa, C. Soares, J. Gama & A. Jorge (Eds.), *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015*. Switzerland: Springer International Publishing.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Rosenbloom, P. S. (2012). Deconstructing reinforcement learning in Sigma. *Proceedings of the 5th Conference on Artificial General Intelligence* (pp. 262-271).

Rosenbloom, P. S., Demski, A., Han, T., & Ustun, V. (2013). Learning via gradient descent in Sigma. *Proceedings of the 12th International Conference on Cognitive Modeling* (pp. 35-40).

Rosenbloom, P. S., Demski, A. & Ustun, V. (2016a). The Sigma cognitive architecture and system: Towards functionally elegant grand unification. *Journal of Artificial General Intelligence*, *7*, 1-103.

Rosenbloom, P. S., Demski, A. & Ustun, V. (2016b). Rethinking Sigma's graphical architecture: An extension to neural networks. *Proceedings of the 9th Conference on Artificial General Intelligence* (pp. 84-94).

Rosenbloom, P. S., Gratch, J. & Ustun, V. (2015). Towards emotion in Sigma: From appraisal to attention. *Proceedings of the 8th Conference on Artificial General Intelligence* (pp. 142-151).

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533-536.

Schaul, T., Bayer, J., Wierstra, D., Yi, S., Felder, M., Sehnke, F., Rückstieß, T. & Schmidhuber, J. (2010). PyBrain. Journal of Machine Learning Research, *11*, 743-746.

Sun, R. (2016). *Anatomy of the Mind: Exploring Psychological Mechanisms and Processes with the Clarion Cognitive Architecture*. New York, NY: Oxford University Press.

Ustun, V., Rosenbloom, P. S., Sagae, K., & Demski, A. (2014). Distributed vector representations of words in the Sigma cognitive architecture. *Proceedings of the 7th Annual Conference on Artificial General Intelligence* (pp. 196-207).