# Design and Simulation of a Multilayer Chemical Neural Network That Learns via Backpropagation

Matthew R. Lakin
University of New Mexico
Department of Computer Science
Department of Chemical and Biological Engineering
Center for Biomedical Engineering
mlakin@cs.unm.edu

**Abstract** The design and implementation of adaptive chemical reaction networks, capable of adjusting their behavior over time in response to experience, is a key goal for the fields of molecular computing and DNA nanotechnology. Mainstream machine learning research offers powerful tools for implementing learning behavior that could one day be realized in a wet chemistry system. Here we develop an abstract chemical reaction network model that implements the backpropagation learning algorithm for a feedforward neural network whose nodes employ the nonlinear "leaky rectified linear unit" transfer function. Our network directly implements the mathematics behind this well-studied learning algorithm, and we demonstrate its capabilities by training the system to learn a linearly inseparable decision surface, specifically, the XOR logic function. We show that this simulation quantitatively follows the definition of the underlying algorithm. To implement this system, we also report ProBioSim, a simulator that enables arbitrary training protocols for simulated chemical reaction networks to be straightforwardly defined using constructs from the host programming language. This work thus provides new insight into the capabilities of learning chemical reaction networks and also develops new computational tools to simulate their behavior, which could be applied in the design and implementations of adaptive artificial life.

**Keywords**
Neural networks, chemical reaction networks, backpropagation, leaky rectified linear unit, simulations



## 1 Introduction

The field of molecular computing has made great strides in the implementation of sophisticated information processing systems at the nanoscale, including logic circuits (Qian & Winfree, 2011a; Seelig et al., 2006), distributed algorithms (Chen et al., 2013), and artificial neural networks (Cherry & Qian, 2018; Qian et al., 2011). Such systems often use DNA molecules to implement their computational processes because the sequence-specific nature of DNA chemistry makes it possible to encode structures and interactions based solely on the sequences of the constituent molecules that make up these systems. The ability to process information at the nanoscale has a range of enticing applications in biomedical diagnostics and therapeutics (Chatterjee et al., 2018; Chen et al., 2015; Groves et al., 2016; C. Zhang et al., 2020) and in the control of nanoscale robotic devices (Thubagere, Li, et al., 2017).

 

The fact that molecular computing devices can mimic the computational behavior of artificial neural networks is of particular interest because it hints at the possibility of implementing adaptive behavior in engineered biochemical systems, if those molecular systems could be made to learn. To date, this has not been accomplished in a molecular implementation of an artificial neural network. However, living systems exhibit a range of adaptive behaviors, and even single-celled organisms have been shown to adapt to changes in their environment (Dexter et al., 2019), with claims made in the literature that even these simple organisms are capable of learning (Hennessey et al., 1979; Wood, 1988). This implies that there may be a molecular basis for learning behaviors in biological systems and, furthermore, that engineered molecular computing systems might be able to replicate some of that behavioral complexity. The design and implementation of learning algorithms for molecular computing systems is of great interest and would enable the development of engineered nanoscale control systems capable of intelligently responding to changes in their environment. For example, smart biomedical devices endowed with molecular learning circuits could be programmed to learn and adapt to the specifics of a particular patient's physiology.

In mainstream machine learning research, artificial neural networks have become established as a powerful mechanism for training models. Given that previous work has demonstrated the implementation of neural networks using DNA strand displacement reactions (Cherry & Qian, 2018; Qian et al., 2011), neural networks are a promising framework for implementing machine learning algorithms in a molecular computing system. Furthermore, mathematically well-defined learning algorithms exist for training, such as the well-established backpropagation algorithm (Rumelhart et al., 1986). Therefore, in this article, we show that a simple neural network learning algorithm based on backpropagation can be realized as an abstract chemical reacton network (CRN). CRNs are the underlying "programming language" of molecular computing circuits (Cook et al., 2009), and it has been shown that any such network can be compiled into a corresponding network in real wet chemistry using DNA strand displacement reactions (Soloveichik et al., 2010). This provides a potential route to a future laboratory implementation of our circuit designs. Furthermore, simulating CRNs is of great interest for molecular programming.

Here we outline the design of an abstract CRN that can implement backpropagation learning in a multilayer artificial neural network architecture. The neurons in our system utilize the leaky rectified linear unit (leaky ReLU) nonlinearity, which is widely used in modern machine learning models. We demonstrate the use of our system to train a simple neural network to learn a logic function that is not linearly separable, by simulating an deterministic ordinary differential equation (ODE) model of the CRN. We illustrate the correctness of this CRN design by comparing its output to that from a reference implementation of the underlying backpropagation algorithm. We develop a flexible, general-purpose simulator to enable these simulations to be carried out straightforwardly. Our work therefore demonstrates the first implementation of backpropagation learning in a multilayer nonlinear chemical neural network design, thereby opening the door to future development of chemical reaction network designs with advanced learning capabilities.

## 2 Related Work

Previous work by ourselves and others has developed and simulated designs for neural network–like chemical reaction systems capable of learning, from which we draw inspiration for the current work. In early work, we developed designs for systems capable of supervised learning in networks of DNAzyme-based reactions (Lakin et al., 2014) and toehold-mediated DNA strand displacement reactions (Lakin & Stefanovic, 2016). Blount et al. (2017) reported a feedforward chemical network that uses a backpropagation-style algorithm for learning; however, the system design does not implement a rigorous mathematical specification of the algorithm, and thus its behavior is somewhat hard to predict a priori. Other approaches to learning in chemical systems have also been investigated, including training via reinforcement learning (Banda et al., 2014) and learning via weight

perturbation learning in a multilayer network of neurons using the tanh nonlinearity (Arredondo & Lakin, 2022). The crucial goal in all of this work has been to establish systems in which the learning process is carried out solely within the simulated chemistry and not via external computational mechanisms. Autonomous chemical learning systems could, at least in principle, be developed to function autonomously and be trained by an experimenter to implement specific computational functions. However, there remains a lack of circuit designs that can implement the standard and well-known algorithm of backpropagation for supervised learning in a well-understood and clearly specified manner; the current article aims to address this.

Theroretical work on chemical neural networks that can compute outputs but cannot learn has yielded promising approaches to use deep learning to program molecular circuits (Vasić et al., 2020). Previous work has discovered rate-independent chemical reaction systems for computing the nonleaky version of the rectified linear unit (ReLU) nonlinearity (Vasić, Soloveichik, & Khurshid, 2020b) and has also explored the potential for DNA-based implementations of binarized neural networks (Linder et al., 2021). However, while using neurons with binarized weights greatly simplifies the forward computation, training such neurons via gradient descent is challenging because gradient descent works by making many small changes to the weight values, which cannot be done if the weights are binarized. Thus an additional continuous representation of the weights must be maintained for training purposes, which is then binarized for carrying out the forward computations (Simons & Lee, 2019).

Much of the work discussed has focused on high-level abstract specifications of CRNs, in which the system design is expressed in terms of transformation rules between abstract chemical reactants and products. While abstract, such specifications can nevertheless be related to experimental science via translation into networks of toehold-mediated DNA strand displacement reactions (D. Y. Zhang & Seelig, 2011). This correspondence was established by Soloveichik et al. (2010) for arbitrary abstract CRNs, and previous work has studied a range of distinct translation schemes (Cardelli, 2010, 2013). Thus abstract CRNs are conveniently high level while also linked to low-level molecular implementations; therefore a range of computational tools to design and test these translations has been developed. Such tools specialized for DNA nanotechnology include the Nuskell and Peppercorn tools developed in the Winfree group (Badelt et al., 2017, 2020), which are also embedded within Python. The Visual DSD system for DNA strand displacement modeling (Lakin et al., 2011) has limited facilities for perturbing the system (Yordanov et al., 2014). With regard to tools more suited for simulations of adaptive and trainable chemical systems, other, more general simulators, such as StochKit2 (Sanft et al., 2011) and GillesPy (Abel et al., 2016), use both time-based and state-based event triggers for system perturbations that can be used to model external interference from an experimenter or other feedback from the chemical environment. The Kappa system includes perturbation events expressed in the Kappa DSL itself (Boutillier et al., 2018), along with alarms that check these periodically. The COEL cloud-based simulation framework (Banda & Teuscher, 2016) uses perturbation definitions based on the Java Math Expression Parser. To our knowledge, these related systems do not permit the use of arbitrary computation in a general-purpose host language to specify arbitrarily complex interventions in the system when those triggers are activated, as is possible with the simulator on which we report herein.

Finally, some experimental work on computational DNA strand displacement circuits has yielded exciting results in terms of neural network implementations. Most notably, Qian et al. (2011) implemented a recurrent Hopfield neural network using catalytic "seesaw" DNA strand displacement reactions (Qian & Winfree, 2011b). That work was subsequently scaled up to much larger networks that could carry out pattern recognition tasks on a version of the MNIST data set of handwritten digits (Cherry & Qian, 2018). These results hint at exciting future developments in molecular learning circuits. However, these systems are still very small and very simple by the standards of modern-day silicon-based computing hardware, and the amount of additional circuitry required to go from a one-shot neural network implementation to a version with built-in learning capabilities is significant. Thus computational work still has a part to play in laying out future paths for this work.

## 3 Chemical Reaction Networks

In this article, we will use abstract chemical reaction networks under a deterministic mass-action semantics to define and simulate our trainable chemical neural network implementation. Briefly, an abstract CRN consists of a finite set of *species* $\{X_1, \ldots, X_n\}$ and a finite set of *reactions* $\{r_1, \ldots, r_q\}$ over those species. Each reaction $r_i$ has the form

$$\sum_{j=1}^{n} \ell_{ij} X_j \xrightarrow{k_i} \sum_{j=1}^{n} m_{ij} X_j$$

where $\ell_{ij}$ and $m_{ij}$ are nonnegative integer-valued stoichiometric coefficients. (Equivalently, the left-hand and right-hand sides of the reaction are both multisets over the set of species.) The left-hand species are said to be the *reactants*, and the right-hand species are said to be the *products*. Note that these may be zero, meaning that the given species does not appear in that reaction as a reactant or product or both, as appropriate.

The *stoichiometry* of species $X_j$ in reaction $r_i$ is then defined as

$$\text{stoich}(X_j, r_i) = m_{ij} - \ell_{ij}$$

This quantity represents the net change in the quantity of species $X_j$ caused by a single execution of reaction $r_i$. As per the law of mass action, the flux through reaction $r_i$ is proportional to the products of the concentrations of all of the reactants, with the constant of proportionality being the nonnegative real-valued rate constant, $k_i$; we write this as follows:

$$\text{flux}(r_i) = k_i \times \sum_{j \in \{i, \ldots, n\}} [X_j]^{\ell_{ij}}$$

Here the concentration $[X_j]$ is raised to the power of $\ell_{ij}$ to account for the fact that multiple copies of $X_j$ could in theory be required to be present as reactants.

Therefore the differential equation for $\frac{\mathrm{d}[X_j]}{\mathrm{dt}}$, the rate of change of the concentration of species $X_j$ over time, can be expressed as follows:

$$\frac{\mathrm{d}[X_j]}{\mathrm{dt}} = \sum_{i \in \{1, \ldots, q\}} \sum_{j \in \{1, \ldots, n\}} \text{stoich}(X_j, r_i) \cdot \text{flux}(r_i)$$

where $\text{stoich}(X_j, r_i)$ and $\text{flux}(r_i)$ are as defined earlier. For any given abstract CRN, this set of coupled ODEs can be formed via a mechanical translation, with one differential equation per species. Given an initial set of species concentrations, the CRN can be simulated by numerically integrating these ODEs with respect to time using any suitable ODE solver, thereby solving the corresponding initial value problem. (As a practical matter, one can implement such simulations computationally by forming the corresponding *stoichiometry matrix*, a matrix that records all the stoichiometric coefficients of all species in all reactions in the CRN. In conjunction with a function that calculates the flux through each reaction at each time point based on the current species concentrations, the time derivatives can be calculated via simple matrix multiplication operations.)

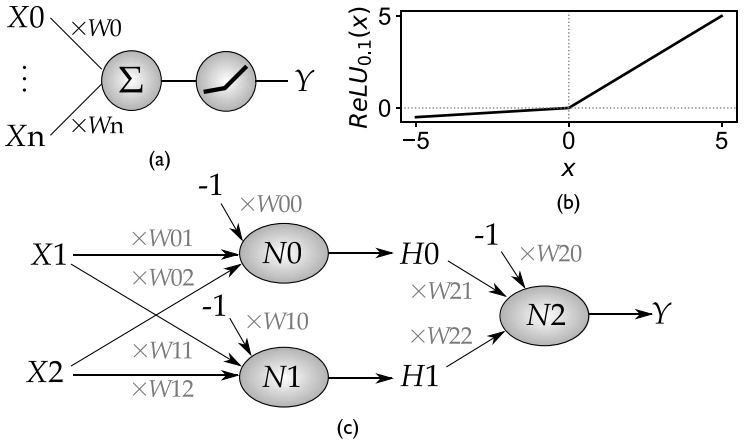


Figure 1. Neural network structure used in this work. (a) Schematic of a single neuron. (b) Plot of leaky ReLU transfer function when $\alpha = 0.1$ (the gradient for input $x < 0$) and $\beta = 1$ (the gradient for input $x \geq 0$). (c) Schematic of the whole network simulated in this work.

## 4 Results

### 4.1 Neural Network Structure and Definition

In this work, we use abstract CRNs to input trainable neural networks composed of leaky ReLU neurons. This nonlinearity is in widespread use in mainstream machine learning research because of its relative simplificity and good learning performance. Figure 1 outlines the schematics of single neurons and the network structure under study. In a single neuron, the input signals are multipled by weights before being passed through the leaky ReLU nonlinearity. The whole network under study accepts two input signals, $X1$ and $X2$, that are passed into two hidden layer neurons, $N_0$ and $N_1$. These produce output signals $H0$ and $H1$, respectively. These signals are fed into a single output layer neuron, $N_2$, which produces the overall output signal, $Y$. Each neuron also has a third input, a bias signal that is clamped to $-1$. Importantly, this is a feedforward network, which means that we can straightforwardly apply the backpropagation algorithm to calculate the weight updates, as outlined in what follows.

The most general definition of the leaky ReLU transfer function is as follows:

$$\mathrm{ReLU}_{\alpha,\beta}(x) = \begin{cases} \alpha \cdot x & \text{if } x < 0 \\ \beta \cdot x & \text{if } x \geq 0 \end{cases}$$

In this work, we set $\beta = 1$ and assume that the values of $\alpha$ and $\beta$ are identical for all neurons in the network. Given this definition, the output $Y$ of our network for a given $\alpha$ and $\beta$ can then be defined as follows:

$$\begin{aligned} \mathrm{NET0} &= (W00 \times (-1)) + (W01 \times X1) + (W02 \times X2) \\ H0 &= \mathrm{ReLU}_{\alpha,\beta}(\mathrm{NET0}) \\ \mathrm{NET1} &= (W10 \times (-1)) + (W11 \times X1) + (W12 \times X2) \\ H1 &= \mathrm{ReLU}_{\alpha,\beta}(\mathrm{NET1}) \\ \mathrm{NET2} &= (W20 \times (-1)) + (W21 \times H0) + (W22 \times H1) \\ Y &= \mathrm{ReLU}_{\alpha,\beta}(\mathrm{NET2}) \end{aligned}$$

where we write NET$i$ for the result of the linear weighting calculation carried out within neuron $N_i$, before that value is passed through the leaky ReLU nonlinearity.

### 4.2 Derivations of Loss Derivatives

The goal of this work is to design and simulate a CRN that implements a neural network capable of being trained via backpropagation, a well-defined and well-understood training algorithm from mainstream machine learning research. For this, we use the basic gradient descent weight update rule,

$$Wji := Wji - \alpha \times \frac{\partial L}{\partial Wji}$$

which requires us to derive the partial derivative of the loss function with respect to each weight $Wji$ in the network. The backpropagation algorithm for supervised learning in a feedforward neural network (such as ours) works by propagating input signals forward through the network to produce an output signal, using this and the target value to compute the loss function. This loss value is then backpropagated through the network to compute the partial derivative of the loss with respect to each weight via repeated application of the chain rule. Here and henceforth, we follow the notational convention of Mitchell (1997).

See section S1 of the Supplemental Material for a full derivation. Briefly, the derivative of the leaky ReLU transfer function with respect to its input is

$$\frac{\partial}{\partial x}\mathrm{ReLU}_{\alpha,\beta}(x) = \begin{cases} \alpha & \text{if } x < 0 \\ \beta & \text{if } x \geq 0 \end{cases}$$

Furthermore, in general, by the chain rule, we have the following:

$$\frac{\partial L}{\partial Wji} = \frac{\partial L}{\partial \mathrm{NET}j} \cdot \frac{\partial \mathrm{NET}j}{\partial Wji} = \frac{\partial L}{\partial \mathrm{NET}j} \cdot Xji$$

where $Xji$ is the $i$th input to the $j$th neuron. Thus the main issue will be to calculate $\frac{\partial L}{\partial \mathrm{NET}j}$ for each neuron in the system. There are two cases to consider: an output neuron ($N_2$) and a hidden layer neuron ($N_0$ and $N_1$).

We define the loss function as $L = \frac{1}{2} \cdot (Y - \mathrm{TARGET})^2$, and for the output neuron $N_2$, we calculate $\frac{\partial L}{\partial Y}$ and $\frac{\partial Y}{\partial \mathrm{NET2}}$ and then use the chain rule to give an expression for $\frac{\partial L}{\partial \mathrm{NET2}}$. Thus we obtain the following set of loss derivatives with respect to the weights associated with neuron $N_2$:

$$\frac{\partial L}{\partial W20} = -1 \cdot Q_2 \qquad \frac{\partial L}{\partial W21} = H0 \cdot Q_2 \qquad \frac{\partial L}{\partial W22} = H1 \cdot Q_2$$

where

$$Q_2 = \begin{cases} \alpha \cdot (Y - \mathrm{TARGET}) & \text{if NET2} < 0 \\ \beta \cdot (Y - \mathrm{TARGET}) & \text{if NET2} \geq 0 \end{cases}$$

For a hidden layer neuron, say, $N_1$ in general, we would need to sum over the contributions from all downstream neurons, thus

$$\frac{\partial L}{\partial \mathrm{NET1}} = \sum_{k \in \mathrm{downstream}(N_1)} \left( \frac{\partial L}{\partial \mathrm{NET}k} \cdot \frac{\partial \mathrm{NET}k}{\partial \mathrm{NET1}} \right)$$

However, in our network, there is only one downstream neuron ($N_2$), so we can simplify the preceding equation to

$$\frac{\partial L}{\partial \text{NET1}} = \frac{\partial L}{\partial \text{NET2}} \cdot \frac{\partial \text{NET2}}{\partial \text{NET1}}$$

We have already computed $\frac{\partial L}{\partial \text{NET2}}$; thus we need only to compute $\frac{\partial \text{NET2}}{\partial \text{NET1}}$. This we can obtain by using the chain rule on $\frac{\partial \text{NET2}}{\partial H1}$ and $\frac{\partial H1}{\partial \text{NET1}}$. Thus we obtain the following set of loss derivatives with respect to the weights associated with neuron $N_1$:

$$\frac{\partial L}{\partial W10} = -1 \cdot Q_1 \qquad \frac{\partial L}{\partial W11} = X1 \cdot Q_1 \qquad \frac{\partial L}{\partial W12} = X2 \cdot Q_1$$

where

$$Q_1 = \begin{cases} \alpha \cdot W22 & \text{if NET1} < 0 \\ \beta \cdot W22 & \text{if NET1} \geq 0 \end{cases}$$

The loss derivatives for the weights associated with $N_0$ can be calculated similarly. We now have expressions for all of the values that our neural network CRN must compute, not just for forward propagation of input signals to produce an output signal and the corresponding loss value but also for the backpropagation of the loss signal through the network to produce loss derivatives and thus the desired weight update values.

### 4.3 Designing a CRN to Directly Compute the Loss Derivatives

Given the mathematical definition of our network's desired operation, both for output generation and for learning, we can now start to build an abstract CRN implementation of this behavior. This is a key advantage of our approach; by grounding our system in a well-studied and well-understood network implementation and learning algorithm, we have a concrete target against which to test our CRN implementation so as to gauge its performance. In this section, we describe the various features of our CRN neural network structure and implementation. Although the various components of the CRN design are presented separately, our system is a single well-mixed solution in which all reactions are assumed to be occurring simultaneously within a one-pot reaction vessel. As outlined in what follows, the timing of reactions is mediated by the use of a chemical clock signal, but the reactions are all assumed to be occurring in parallel and competing for reactants; indeed, this competition is a key aspect of our system design.

#### 4.3.1 Regulation by a Molecular Clock Signal

Drawing on previous work (Vasić, Soloveichik, & Khurshid, 2020a) and our own previous work on learning via weight perturbation in CRNs (Arredondo & Lakin, 2022), here we use a molecular oscillator to organize the reactions in our neural network system into discrete stages. To this end, we use a molecular oscillator consisting of reactions of the form

$$C1 + C2 \xrightarrow{\text{kClock}} C2 + C2 \qquad C2 + C3 \xrightarrow{\text{kClock}} C3 + C3 \qquad \ldots$$
$$C25 + C25 \xrightarrow{\text{kClock}} C26 + C26 \qquad C26 + C1 \xrightarrow{\text{kClock}} C1 + C1$$

where kClock = 0.1 is the corresponding bimolecular rate constant. By initializing the clock species $C25$ and $C26$ with concentration 1.0 (written $[C25]) = [C26] = 1.0$ and with $[Ci] = 10^{-6}$ for all other clock species $Ci$, we obtain a robust oscillation that cycles through precisely one of the clock species going high (with a maximum concentration of 2.0) at any given time, in order. By using these

Table 1. Division ofoperations between clock phases.

| Clock phase | Operations carried out |
|---|---|
| $C1$ | Input fanout |
| $C3$ | Input copying and weight calculations for hidden layer neurons ($N_0$) and $N_1$ |
| $C5$ | Preparing leaky ReLU calculations for $N_0$ and $N_1$ |
| $C7$ | Executing leaky ReLU calculations for $N_0$ and $N_1$ |
| $C9$ | Input copying and weight calculations for output layer neuron ($N_2$) |
| $C11$ | Preparing leaky ReLU calculation for $N_2$ |
| $C13$ | Executing leaky ReLU calculation for $N_2$ |
| $C15$ | Calculation of network loss |
| $C17$ | Calculation of backpropagation signal into $N_2$ |
| $C19$ | Calculation of weight updates for $N_2$ and backpropagation signals into $N_1$ and $N_0$ |
| $C21$ | Calculation of weight updates for $N_1$ and $N_0$ |
| $C23$ | Application of weight updates |
| $C25$ | Cleanup (degradation of various signals) in preparation for next training round |

clock species as catalysts to drive other reactions, we can ensure that those reactions experience only a nonnegligible flux through them when the particular clock species in question is the high one. As in previous work (Arredondo & Lakin, 2022; Vasić, Soloveichik, & Khurshid, 2020a), we use only odd-numbered clock species as catalysts, to prevent any overlap between the phases. Our design uses a relatively large number of distinct clock phases, though it is possible that this could be optimized in future work. The division of different parts of the feedforward network computation and backpropgation learning process between the various clock phases is summarized in Table 1; the details of these operations and their implementations are described in the following pages.

### 4.3.2 Dual-Rail Signal Representation

The basic representation for signals in our system is as the concentration of an abstract chemical species corresponding to that variable. However, although the signals involved in our network may take on negative values (for example, the bias inputs to the neurons), chemical concentrations cannot fall below zero. Therefore, as in previous work (Arredondo & Lakin, 2022), here we use a *dual-rail* representation of a signal $X$ as a *pair* of chemical species: $X_\mathrm{p}$ to represent the positive component and $X_\mathrm{m}$ to represent the negative component. We interpret the true value of the signal $X$ and $[X_\mathrm{p}] - [X_\mathrm{m}]$. However, for each such species pair, we include a dual-rail annihilation reaction of the form

$$X_\mathrm{p} + X_\mathrm{m} \xrightarrow{k_\mathrm{Annih}} \emptyset$$

with a rate constant value of $k$Annih $= 1.0$. These reactions do not require a clock species as catalyst and can therefore fire at any time. This ensures that only one of the two dual-rail species will be present at any given time with a nonnegligible concentration, which simplifies the interpretation of the system state.

### 4.3.3 Input Fan-out

In each training round, the inputs to the network are provided via two dual-rail input signals ($X1$ and $X2$) and the bias input (BIAS). The latter always takes the value $-1$, however, we include both dual-rail species for consistency with the other, similar signals. In the first ($C1$) clock cycle, these inputs are transferred into multiple locations as required for the calculation of multiple neuron signals that operate on the same inputs. This is achieved by simple reactions, catalyzed by the $C1$ clock signal, that consume the inputs entirely and produce the same concentration of several output species as products. For example, the dual-rail positive input species $X1\text{p}$ is fanned out into corresponding input species $X01\text{p}$ and $X11\text{p}$ for neurons $N_0$ and $N_1$, respectively, via the single reaction

$$C1 + X1\text{p} \xrightarrow{k} C1 + X01\text{p} + X11\text{p}$$

This and the vast majority of subsequent reactions described use rate constant $k = 1.0$. Similar reactions are used for the fan-out of $X1\text{m}$ and both dual-rail components of the $X2$ signal. The fan-out reactions for the bias signals are similar but include an additional product species because the bias signal must be duplicated for all three neurons in the network, not just for the two input neurons, for example, for BIASm

$$C1 + \text{BIASm} \xrightarrow{k} C1 + \text{BIAS0m} + \text{BIAS1m} + \text{BIAS2m}$$

Note that, in theory, we could eliminate these fan-out reactions and require the input to be provided separately to each of the neurons in equal concentrations; however, for simplicity of the input interface, we have the first phase of the network calculation do these reactions internally.

### 4.3.4 Input Weighting

All input signals to our neurons must be scaled by multiplying them by the corresponding weight value. In order that the weights can be adjusted autonomously over time as the network learns, these weights must also be represented in the system as the concentrations of (dual-rail) chemical species. We used the same approach to this issue as in our previous work on training chemical neural networks (Arredondo & Lakin, 2022). Briefly, as inspired by previous work (Buisman et al., 2009), the concentration of an "input" species $X$ can be transferred into the concentration of an "output" species $Y$, scaled by the concentration of a "weight" species $W$, via the pair of competitive reactions

$$W + X \xrightarrow{k} W + X + Y \qquad\qquad X \xrightarrow{k} \emptyset.$$

Forming the differential equations for this small CRN, we observe that the flux per unit time through the first reaction (that produces the output $Y$) is $k \cdot [W] \cdot [X]$, while the flux through the second reaction is just $k \cdot [X]$. In the absence of other reactions simultaneously producing or consuming these species (which is effectively the case in our clocked CRN implementation), each input $X$ will thus catalyze the first reaction $[W]$ times before it is consumed in the second reaction, leading to the steady state concentration $[Y]$ being the product of the initial values of $[W]$ and $[X]$. The input $X$ is consumed by this process, which is slightly different to the approach from Buisman et al. (2009), in which the inputs act catalytically to produce the outputs with a scaled concentration. This feature fits well with some aspects of our network design, for example, the fact that the calculation of the leaky ReLU nonlinearity depends on one of the positive or negative dual-rail signals annihilating the

other. However, we do need the original input values for the backpropagation calculations, because the partial loss derivatives with respect to the weights do depend on the input values. Therefore we must also copy the input values so they are available during backpropagation, as outlined in the Input Copying section.

We used the preceding template to generate sets of similar reactions for scaling the input species to all of our neurons (both in the hidden layer and in the output layer), as well as the bias inputs, by the corresponding weight values. These reactions must be provided for all combinations of the positive and negative dual-rail species, with the correct combination of signs in each case. The assignment of weight names to neuron inputs in our model reactions follows the naming scheme set out in Figure 1(c). These reactions must also be catalyzed by the correct clock species, as per Table 1. As a concrete example, the complete set of reactions for weighting the bias input to neuron $N_0$ is as follows:

$$C3 + W00\text{p} + \text{BIAS0p} \xrightarrow{k} C3 + W00\text{p} + \text{BIAS0p} + \text{NET0p}$$
$$C3 + W00\text{m} + \text{BIAS0m} \xrightarrow{k} C3 + W00\text{m} + \text{BIAS0m} + \text{NET0p}$$
$$C3 + W00\text{p} + \text{BIAS0m} \xrightarrow{k} C3 + W00\text{p} + \text{BIAS0m} + \text{NET0m}$$
$$C3 + W00\text{m} + \text{BIAS0p} \xrightarrow{k} C3 + W00\text{m} + \text{BIAS0p} + \text{NET0m}$$
$$C3 + \text{BIAS0p} \xrightarrow{k} C3 \qquad C3 + \text{BIAS0m} \xrightarrow{k} C3$$

For a given neuron, all reactions produce the same output species (NET0, for instance), which means that the resulting concentration of that species is the *sum* of the weight inputs, as required.

### 4.3.5 Input Copying

As mentioned, we must copy the input signals for each neuron into a separate species before they are consumed during the weight process, so that they may be used in the subsequent backpropagation calculations to calculate the weight update values. In the case of hidden layer neuron $N_0$, this can be achieved via the reactions

$$C3 + \text{BIAS0p} \xrightarrow{k} C3 + \text{BIAS0p} + K00\text{p} \qquad C3 + \text{BIAS0m} \xrightarrow{k} C3 + \text{BIAS0m} + K00\text{m}$$
$$C3 + X01\text{p} \xrightarrow{k} C3 + X01\text{p} + K01\text{p} \qquad C3 + X01\text{m} \xrightarrow{k} C3 + X01\text{m} + K01\text{m}$$
$$C3 + X02\text{p} \xrightarrow{k} C3 + X02\text{p} + K02\text{p} \qquad C3 + X02\text{m} \xrightarrow{k} C3 + X02\text{m} + K02\text{m}$$

Each input serves as a catalyst for the generation of the corresponding "copy" ($K$) species. Because these reactions are occurring in parallel with the input weighting reactions outlined previously, the combination of this catalysis reaction and the input degradation reactions shown as part of the input weight reactions means that the final concentration of each copy species matches the initial concentration of the corresponding input species. Thus the input copying reactions record the input values for use in the subsequent weight update calculations specified in the backpropagation algorithm, and because the inputs serve as catalysts in these reactions, the operation of the input weighting reactions is unaffected. Similar reactions take place in clock phase $C3$ for the $N_1$ inputs, and similar reactions also take place in clock phase $C9$ for the inputs to the output neuron $N_2$ (which are the outputs from the hidden layer neurons, as shown in Figure 1(c)).

### 4.3.6 Calculation of Leaky ReLU Nonlinearity

Having calculated the weighted sums of the inputs, and copied the input values as appropriate, the next step in the feedforward network computation is to pass the weighted inputs through the leaky ReLU nonlinearity. Previous work (Linder et al., 2021; Vasić, Soloveichik, & Khurshid, 2020b) has uncovered rate-independent CRNs for calculating the nonleaky variant of the ReLU nonlinearity; however, here we wish to use the leaky variant and thus must design our own, new CRN.

Our approach is driven by the observation that computing the leaky ReLU is another simple linear weighting process, once it is known whether the input is positive or negative. Therefore we first select the required gradient for the leaky ReLU unit in a given neuron, say, $N_0$ based on the value of the corresponding weighted sum of the inputs, which we call NET0 in the case of $N_0$. We assume the existence of a pair of species that store the values of the two different gradients for the leaky ReLU unit, corresponding to the $\alpha$ and $\beta$ constants from the definition of the leaky ReLU transfer function. These do not need to be dual-rail species, as the leaky ReLU gradients must be positive. There is a separate pair of such species for each neuron, so the gradients could be varied on a per-neuron basis if desired, though here we do not do so. For neuron $i$, these species are DPOS$i$ and DNEG$i$ for the gradients for positive and negative $x$ input values, respectively. One of these will then be copied into the $Di$ species, which represents the correct gradient to be applied to the waiting input value. This is achieved for neuron $N_0$ via the following reactions:

$$C5 + \text{NET0p} + \text{DPOS0} \xrightarrow{\text{kCopy}} C5 + \text{NET0p} + \text{DPOS0} + D0\text{p} \qquad C5 + \text{NET0p} + D0\text{p} \xrightarrow{\text{kCopy}} C5 + \text{NET0p}$$

$$C5 + \text{NET0m} + \text{DNEG0} \xrightarrow{\text{kCopy}} C5 + \text{NET0m} + \text{DNEG0} + D0\text{p} \qquad C5 + \text{NET0m} + D0\text{p} \xrightarrow{\text{kCopy}} C5 + \text{NET0m}$$

These reactions use a new rate constant, kCopy, which we set to 1.0 by default but which could enables us to tune the behavior of the leaky ReLU calculations around the discontinuity when the input $x = 0$, if required (see Discussion). These are simple copying reactions using the scheme of Buisman et al. (2009). They work as expected because the dual-rail annihilation reactions will have already ensured that only one of NET0p and NET0m is present at the start of the $C5$ clock phase. Then, the value of either DPOS0 or DNEG0 will be copied into the $D0$ species. (For the sake of simplifying the model setup code, we use a dual-rail species for $Di$, even though it will only ever be a positive value.) We assume that the correct concentrations of the DPOS$i$ and DNEG$i$ species for each neuron have been supplied in the initial system setup, and our network does not modify these values during its execution. We also assume that the values of the $Di$ species are neglible at the start of each training round; this is true in our initial setup, and the cleanup phase at the end of each training round (outlined later) ensures that this invariant holds for subsequent training rounds. Similar reactions take place in the $C5$ clock phase for the second hidden layer neuron, $N_1$, and in the $C11$ clock phase for the output neuron, $N_2$.

Having chosen the correct gradient to apply based on the sign of the input value, actually calculating the output from the leaky ReLU unit is just a simple weighting calculation in which the input value is multiplied by the chosen gradient. The reaction scheme for accomplishing this is identical to that for the linear input weighting outlined earlier. We run these in the clock phase after the gradient selection reactions to ensure that the results from the calculation are accurate. For example, the corresponding reactions for neuron $N_0$ are as follows:

$$C7 + D0\text{p} + \text{NET0p} \xrightarrow{k} C7 + D0\text{p} + \text{NET0p} + H0p$$
$$C7 + D0\text{m} + \text{NET0m} \xrightarrow{k} C7 + D0\text{m} + \text{NET0m} + H0p$$
$$C7 + D0\text{p} + \text{NET0m} \xrightarrow{k} C7 + D0\text{p} + \text{NET0m} + H0m$$
$$C7 + D0\text{m} + \text{NET0p} \xrightarrow{k} C7 + D0\text{m} + \text{NET0p} + H0m$$
$$C7 + \text{NET0p} \xrightarrow{k} C7 \qquad C7 + \text{NET0m} \xrightarrow{k} C7$$

The resulting final output from $N_0$ is stored in the $H0$ dual-rail species. Similar reactions take place in the $C7$ clock phase for neuron $N_1$, producing the $H1$ output signal. Finally, similar reactionstake place in the $C13$ clock phase for the output neuron $N_2$, producing the $Y$ signal. The concentration of this dual-rail signal represents the overall output from the feedforward network computation. Importantly, the chosen gradient values remain stored in the $Di$ species after this phase has

completed, which means they are available for subsequent use in the backpropagation algorithm that calculates the weight updates, as outlined in what follows.

### 4.3.7 Calculation of Network Error

Having completed the feedforward network computation, before we can start the backpropagation learning stage, we must first calculate the network error (also known as loss) between the output signal $Y$ produced by the output neuron and the expected output value TARGET, which we assume is supplied by the experimenter along with the input signals at the start of each training round.

In fact, what we actually calculate to serve as the input to the backpropagation algorithm is the partial derivative of the loss with respect to the network output, $\frac{\partial L}{\partial Y}$. Given that we define the loss as $L = \frac{1}{2} \cdot (Y - \text{TARGET})^2$, this partial derivative can be computed straightforwardly as $\frac{\partial L}{\partial Y} = Y - \text{TARGET}$. This obviates the need to calculate the square of the difference between the output $Y$ and the target output TARGET, although this could in principle be computed by a CRN using previously reported techniques (Buisman et al., 2009).

We compute $\frac{\partial L}{\partial Y}$ and store it in the dual-rail CRN signal $E$ using the following four simple reactions, which occur in the $C15$ clock phase:

$$C15 + Y\text{p} \xrightarrow{k} C15 + E\text{p} \qquad C15 + Y\text{m} \xrightarrow{k} C15 + E\text{m}$$
$$C15 + \text{TARGETp} \xrightarrow{k} C15 + E\text{m} \qquad C15 + \text{TARGETp} \xrightarrow{k} C15 + E\text{p}$$

These reactions simply transfer the values from the $Y$ and TARGET species into the $E$ dual-rail signal, with the signs matched in the case of $Y$ and inverted in the case of TARGET, so as to correcly realize the subtraction operation.

### 4.3.8 Calculation of Loss Derivatives via Backpropagation

The loss derivative $\frac{\partial L}{\partial Y}$ having been calculated as the dual-rail species $E$, this can serve as the input to the backpropagation calculations required to compute the weight updates for gradient descent learning. This is the crux of our CRN learning algorithm. The outline structure of the backpropagation calculations is summarized in Figure 2. Once the input signals and chosen leaky ReLU gradients have been copied as outlined earlier, the backpropagation phase is just a series of straightforward multiplications, following the derivations presented in section S1 of the Supplemental Material.

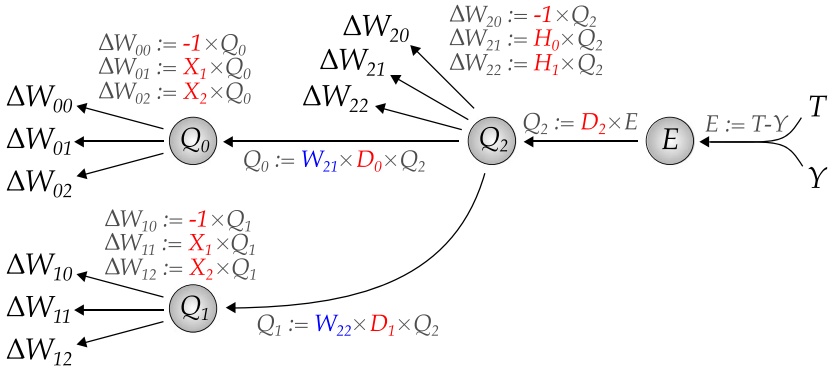


Figure 2. Overview of backpropagation computations in learning CRN design. The dual-rail $E$ signal stores the partial derivative of the network loss with respect to output, and this value is fed back through this network, which parallels the structure of the neural network itself from Figure 1(c), to compute the Delta$Wij$ values that represent the updates to be applied to each weight $Wij$ from the current training round. The terms highlighted in red are constants that are copied into a temporary species during the feedforward calculation phase for use in the backpropagation phase. The weight terms highlighted in blue can be used without the need for copying into a temporary species.

For each neuron, we divide the backpropagation calculation into two parts, the computing of the backpropagation signal "into" that neuron, which is the same for all weight updates associated with that neuron, and the computing of the individual weight updates, which differ between the various weights associated with a given neuron. To simplify the analysis of the CRN, these occur in distinct clock phases, although they could be combined if reducing clock phases were a concern. We use species $Qi$ to represent the shared incoming value for neuron $N_i$ in the backpropagation calculation.

In the case of the output neuron $N_2$, the value stored in the $Q2$ signal represents the value of $\frac{\partial L}{\partial \mathrm{NET2}}$, which is calculated (see section S1 of the Supplemental Material) by multiplying the $E$ signal generated earlier by the leaky ReLU gradient for $N_2$ in this training round, which is stored in the $D2$ species, as outlined previously. This is a straightforward dual-rail multiplication process, implemented by the following reactions which occur in the $C17$ clock phase:

$$C17 + E\mathrm{p} + D2\mathrm{p} \xrightarrow{k} C17 + E\mathrm{p} + D2\mathrm{p} + Q2\mathrm{p} \qquad C17 + E\mathrm{m} + D2\mathrm{m} \xrightarrow{k} C17 + E\mathrm{m} + D2\mathrm{m} + Q2\mathrm{p}$$
$$C17 + E\mathrm{p} + D2\mathrm{m} \xrightarrow{k} C17 + E\mathrm{p} + D2\mathrm{m} + Q2\mathrm{m} \qquad C17 + E\mathrm{m} + D2\mathrm{p} \xrightarrow{k} C17 + E\mathrm{m} + D2\mathrm{p} + Q2\mathrm{m}$$
$$C17 + D2\mathrm{p} \xrightarrow{k} C17 \qquad C17 + D2\mathrm{m} \xrightarrow{k} C17$$

The $Q2$ signal having been calculated, this can be used to compute the partial loss derivatives for the $N_2$ weights. It must also be propagated backward through the network to serve as one of the inputs for the equivalent calculations for the hidden layer neurons, $N_1$ and $N_0$. Because these reactions will consume the $Q2$ species, as earlier, these processes need to happen in parallel, during the $C19$ clock phase. With regard to the $N_2$ weight updates, these are calculated by multiplying the value stored in $Q2$ by the copied input value associated with each weight, producing a dual-rail signal Delta$Wij$ representing the partial derivative of the network loss with respect to weight $Wij$. (These will be scaled by the learning rate parameter and actually applied to the weights in a subsequent phase, outlined later.) Given that the input scaled by weight $Wij$ has already been copied into the species $Kij$, it follows that we must multiply $Q2$ by $K2j$ to produce the corresponding weight update Delta$W2j$. This is implemented via the following reactions:

$$C19 + K20\mathrm{p} + Q2\mathrm{p} \xrightarrow{k} C19 + K20\mathrm{p} + Q2\mathrm{p} + \mathrm{Delta}W20\mathrm{p}$$
$$C19 + K20\mathrm{m} + Q2\mathrm{m} \xrightarrow{k} C19 + K20\mathrm{m} + Q2\mathrm{m} + \mathrm{Delta}W20\mathrm{p}$$
$$C19 + K20\mathrm{p} + Q2\mathrm{m} \xrightarrow{k} C19 + K20\mathrm{p} + Q2\mathrm{m} + \mathrm{Delta}W20\mathrm{m}$$
$$C19 + K20\mathrm{m} + Q2\mathrm{p} \xrightarrow{k} C19 + K20\mathrm{m} + Q2\mathrm{p} + \mathrm{Delta}W20\mathrm{m}$$
$$C19 + Q2\mathrm{p} \xrightarrow{k} C19 \qquad C19 + Q2\mathrm{m} \xrightarrow{k} C19$$

Similar reactions are executed in the $C19$ clock phase to calculate Delta$W21$ and Delta$W22$. Note, however, that the degradation reactions shown for $Q2$p and $Q2$m should *not* be duplicated; otherwise, this will be the equivalent of multiplying the rate constant for those reactions, and the calculations will be incorrect because the competition will be biased in favor of the degradation reactions.

In parallel, the $Q2$ species must be used to calculate $Q1$ and $Q0$, as outlined in Figure 2. Considering neuron $N_1$ as an example, the value of $Q1$ represents $\frac{\partial \mathrm{NET2}}{\partial \mathrm{NET1}} \cdot \frac{\partial L}{\partial \mathrm{NET2}}$; thus we must multiply the value of $Q2$ by $W22$ and also by the leaky ReLU gradient from neuron $N_1$ in this cycle, which is stored as $D1$. This double multiplication can be achieved via the following reactions, which follow the same approach as for multipication, except with an additional catalyst representing the extra multiplier; note that there are now eight reactions to cover all combinations of signs for the three

dual-rail inputs to this multiplication process:

$$C19 + W22\text{p} + D1\text{p} + Q2\text{p} \xrightarrow{k} C19 + W22\text{p} + D1\text{p} + Q2\text{p} + Q1\text{p}$$
$$C19 + W22\text{m} + D1\text{p} + Q2\text{p} \xrightarrow{k} C19 + W22\text{m} + D1\text{p} + Q2\text{p} + Q1\text{m}$$
$$C19 + W22\text{p} + D1\text{m} + Q2\text{p} \xrightarrow{k} C19 + W22\text{p} + D1\text{m} + Q2\text{p} + Q1\text{m}$$
$$C19 + W22\text{m} + D1\text{m} + Q2\text{p} \xrightarrow{k} C19 + W22\text{m} + D1\text{m} + Q2\text{p} + Q1\text{p}$$
$$C19 + W22\text{p} + D1\text{p} + Q2\text{m} \xrightarrow{k} C19 + W22\text{p} + D1\text{p} + Q2\text{m} + Q1\text{m}$$
$$C19 + W22\text{m} + D1\text{p} + Q2\text{m} \xrightarrow{k} C19 + W22\text{m} + D1\text{p} + Q2\text{m} + Q1\text{p}$$
$$C19 + W22\text{p} + D1\text{m} + Q2\text{m} \xrightarrow{k} C19 + W22\text{p} + D1\text{m} + Q2\text{m} + Q1\text{p}$$
$$C19 + W22\text{m} + D1\text{m} + Q2\text{m} \xrightarrow{k} C19 + W22\text{m} + D1\text{m} + Q2\text{m} + Q1\text{m}$$

Note that we again reuse the same degradation reactions for the $Q2\text{p}$ and $Q2\text{m}$ already specified to run in the $C19$ clock phase to provide the competition for these multiplication reactions. Similar reactions are run simultaneously in the $C19$ phase to compute the corresponding $Q0$ signal for $N_0$. These are followed by reactions during the $C21$ clock phase to calculate the weight updates for the weights associated with $N_1$ and $N_0$; these follow the same pattern as described for computing the weight updates for the output neuron $N_2$. Thus, once the $C21$ clock phase finishes, the backpropagation phase of our learning CRN will have computed all of the partial loss derivatives $\frac{\partial L}{\partial Wij}$ and stored them in corresponding Delta$Wij$ dual-rail chemical species.

### 4.3.9 Application of Weight Updates

Having calculated the partial loss derivatives, we can now apply the updates to the weights. Note that each weight cannot be updated immediately as the partial loss derivatives are calculated because the original weight values used in the feedforward network computation phase of each training cycle are required to compute the backpropagation signals; these computations are going on at the same time, as outlined previously. Thus we compute the partial loss derivatives as earlier for all weights, then apply the weight updates in the next clock phase ($C23$) after the backpropagation process has concluded.

Recall the gradient descent weight update rule, which is as follows:

$$Wji := Wji - \alpha \times \frac{\partial L}{\partial Wji}$$

Having calculated the partial loss derivatives, we must thus simply multiply each of these by the learning rate $\alpha$ (a small positive constant that scales the size of the step taken at each training round) and subtract these products from the current value of the corresponding weight to produce the new weight value. Given that the learning rate must be positive, we will represent this as the concentration of a single species Alpha and note it as a dual-rail species, which will simplify these reactions. Furthermore, the subtraction reaction can be combined with the multiplication by inverting the signs of the products with respect to the dual-rail inputs. For example, the reactions associated with updating weight $W00$ are as follows:

$$C23 + \text{Alpha} + \text{Delta}W00\text{p} \xrightarrow{k} C23 + \text{Alpha} + \text{Delta}W00\text{p} + W00\text{m}$$
$$C23 + \text{Alpha} + \text{Delta}W00\text{m} \xrightarrow{k} C23 + \text{Alpha} + \text{Delta}W00\text{m} + W00\text{p}$$
$$C23 + \text{Delta}W00\text{p} \xrightarrow{k} C23 \qquad C23 + \text{Delta}W00\text{m} \xrightarrow{k} C23$$

Similar reactions occur in clock phase *C*23 for all of the other weights, meaning that all weights are updated together after the backpropagation phase has concluded, in preparation for the next round of training.

### 4.3.10 Cleanup at the End of Training Cycle

Finally, having updated the stored weights for all neurons in the network, the last task that remains is to clean up those species still remaining with nonnegligible concentrations that must be removed so that the next training round can proceed with a clean state. Given that many of the reactions outlined earlier actually consume their input signals as they compute, not every signal involved in the learning network must be reset in this way. Specifically, we must clean up the *E* dual-rail species that stored the partial derivative of network loss with respect to network output, the *Di* dual-rail species that stored the leaky ReLU gradient used by each neuron in the last cycle, and the *Kij* dual-rail signals that stored the duplicated values of each input value presented to the network in the preceding training round. These are removed via simple degradation reactions catalyzed by the *C*25 clock signal, which is the last one actually used to drive reactions in our circuit design. For example, the *E* dual-rail signal is reset to approximately zero by the following two reactions:

$$C25 + E\mathrm{p} \xrightarrow{k} C25 \qquad C25 + E\mathrm{m} \xrightarrow{k} C25$$

Thus, at the end of the cleanup phase, all species that require it will have been degraded to approximately zero concentration (the concentrations will asymptote toward zero under the deterministic ODE semantics). The system will therefore be in a state equivalent to that at the start of the training round, except with modified weights due to the backpropagation learning algorithm. Thus it is primed and ready for the cycle to begin again with a new set of inputs and a new target output value, so that training of the network can proceed through multiple cycles.

### 4.3.11 Chemical Reaction Network Size

The numbers of species and reactions required are a common metric for CRN design complexity; the CRN design outlined herein includes a total of 135 species and 315 reactions. This is similar to the simpler of the two similar networks from our previous work that we trained via weight perturbation (Arredondo & Lakin, 2022); that CRN involved 144 species and 286 reactions. However, only one weight could be adjusted per training round in that version, while our backpropgation network adjusts every weight in each training cycle. The similar network that could adjust every weight in each training cycle via weight perturbation was significantly larger, involving 583 species and 1,213 reactions. Thus the design outlined herein improves on previous work to enable the entire network to be trained using a more compact design than previously available. A full listing of the backpropagation CRN species and reactions is presented in section S3 of the Supplemental Material.

### 4.3.12 Initial Network State

In the initial state of the CRN, the concentrations of the clock species *C*25 and *C*26 are set to 1.0, with all other clock species concentrations set to $10^{-6}$. Thus the system starts in the "last" cycle of the oscillator, and the first full training round will begin shortly after the simulation begins. The dual-rail weight species *Wij* are initialized with values corresponding to the initial values of all the corresponding weights in the untrained network. The DPOS*i* and DNEG*i* species for each neuron $N_i$ are initialized with the values for the leaky ReLU gradient parameters $\alpha$ and $\beta$ for each neuron. (These could, in principle, be different for each neuron.) Finally, the Alpha species is assigned a concentration corresponding to the value of the learning rate parameter for the network. The initial concentrations of all other CRN species are set to zero.

### 4.3.13 Training Implementation

The CRN is trained by increasing the concentrations of the species representing the dual-rail input signals ($X1$, $X2$, and BIAS) at the start of each $C1$ oscillator phase, which corresponds to the start of a new training cycle. The amount of the increase represents the value of that signal for this training cycle. An additional concentration of the TARGET species, representing the expected output of the network given these inputs, is also supplied. These additions represent the actions of an experimenter external to the system; thus, the system will undergo *supervised learning*. We assume that the volume increase associated with these additions is neglible so that we do not need to model dilution effects. This is repeated at the start of the next training cycle, and so on. These additions are modeled via a system of *model perturbations* implemented in our simulation engine, which we describe later.

## 4.4 Simulation Methodology

Having established a design for abstract CRNs capable of backpropagation learning, we need a methodology to easily simulate their behavior. To this end, we have developed ProBioSim, a simulator that incorporates features specifically to simulate such systems, in which the CRN component interacts with an environment that is specified externally to the CRN itself but that can modify the CRN in an arbitrarily complex, state- and time-dependent manner. Given that existing designs for molecular systems are currently beyond our experimental capabilities, many of these potential applications of DNA nanotechnology have been studied *in silico*. This motivates the development of software tools targeted specifically for the study of the interactions of engineered molecular systems with externally specified environments.

### 4.4.1 ProBioSim Simulator Outline

As outlined in Figure 3(a), ProBioSim is a Python library that incorporates domain-specific language (DSL) for molecular circuits (Lakin & Phillips, 2020). The DSL included within ProBioSim is a simple syntax for specifying the reactions, rate constants, and simulation settings required to define and simulate the behavior of abstract CRNs. This language also includes the capability to specify simple external interventions to add or remove quantities of certain chemical species at specified points in simulation time. This syntax can encode unconditional, deterministic perturbations of the system like those used in previous work on supervised learning in chemical systems (Lakin & Stefanovic, 2016) and allows for rapid prototyping and testing of candidate CRN designs. The fact that ProBioSim is based on Python means that it can be trivially integrated with systems for rapid prototyping and visual feedback, such as Jupyter Notebooks (Kluyver et al., 2016), which also enhances reproducibility of modeling and simulation workflows.

There are two types of perturbation in ProBioSim: perturbation *actions* and perturbation *functions*. In each case, the perturbation is scheduled to occur at a specific time point in the simulation, at

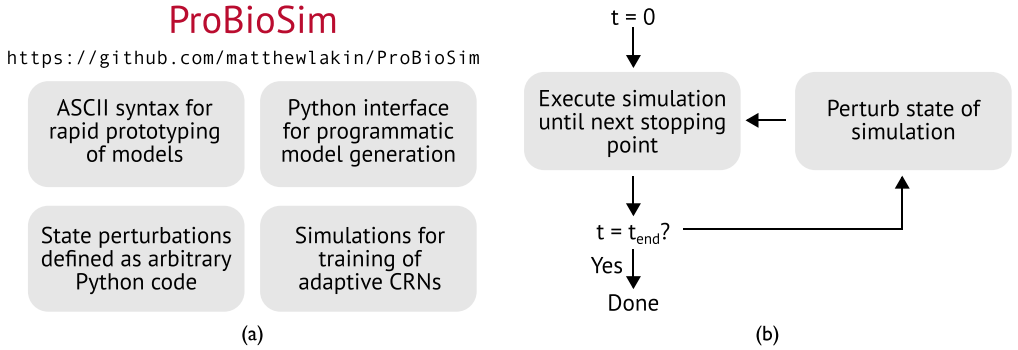


Figure 3. ProBioSim, a simulation tool for training adaptive chemical reaction networks. (a) Summary of the key features of ProBioSim. (b) Flowchart of basic ProBioSim simulation process.

which point the simulation (either stochastic or deterministic) is paused, the specified changes to the simulation state are applied, and the simulation is restarted, as outlined in Figure 3(b). Multiple perturbation actions and functions can be specified to execute at the same time point.

### 4.4.2 Perturbation Actions

*Perturbation actions* are explicitly specified modifications to the state of the simulation. They can be specified within the text-based CRN specification language included within ProBioSim and can be exported to and imported from text files that specify the model. The set of perturbation actions that can be specified in this way is limited to the following:

- *increment* the concentration or count of a species by a specified amount,
- *decrement* the concentration or count of a species by a specified amount, or
- *set* the concentration or count of a species to a specified absolute amount.

As an example, a perturbation that adds 10 units of species `X` at simulation time 5.0 would be specified by the following line in the model definition input text:

```
perturbation X += 10 @ 5.0
```

### 4.4.3 Arbitrary Perturbation Functions Expressed as Python Code

The real power of the ProBioSim system comes from its ability to define perturbations to the CRN system in terms of arbitrary code. This is possible because, in contrast to some other molecular design tools (Lakin et al., 2011), ProBioSim is embedded within a host language: the Python general-purpose programming language. This provides a ready-made language for expressing arbitrary responses to CRN behaviors, which can be used to simulate the interactions of the CRN with an experimenter or a responsive external environment. Such *perturbation functions* are modifications to the simulation state that are specified in terms of functions written in Python. As such, perturbation functions are significantly more powerful than perturbation actions, as they allow arbitrary computations to be carried out to determine how the state of the system should be updated. ProBioSim itself does not check the content of user-supplied perturbation functions. However, those perturbation functions must be written in a particular way to enable them to be called by the simulator as required. Specifically, the form of a perturbation function must be as follows:

```
def pfun(t, x0, get, set, adjust):
    # ... perturbation code here ...
```

where the meanings of the function arguments `t`, `x0`, `get`, `set`, `adjust` are as follows:

- `t` is the simulation time at which the function is called;
- `x0` is the simulation state passed into the function;
- `get` is a function that, when called as `get(x0, sp)`, returns the current value associated with species `sp` in the current state `x0`;
- `set` is a function that, when called as `set(x0, sp, n)`, updates the state `x0` by assigning the value `n` to the species `sp`; and
- `adjust` is a function that, when called as `adjust (x0, sp, n)`, updates the state `x0` by adding `n` (which could be negative) to the value currently assigned to the species `sp`.

Note that the exact names used for these arguments in the function are unimportant, but the number of arguments does need to be correct. These function arguments abstract away from the internal

implementation details of the simulation engine. Thus the perturbation function can access the current state of the CRN and also modify it. This approach allows arbitrary Python code, including any Python library, to be used to compute the additions or removals of species to or from the system. In principle, the `adjust` function can be implemented in terms of the `get` and `set` functions but is included for the sake of convenience. These functions can be used as outlined to get state values, perform arbitrary computations on them, and update the simulation state as a side effect. Because the current simulation time is passed to the perturbation function as an argument, the perturbation functions can be time-dependent, thereby modeling dynamically changing environments. For example, the perturbation action example from earlier can be implemented (with some terminal output) as the following function:

```
def pfun(t, x0, get, set, adjust):
    print('>>> Value of X __before__ perturbation at time ' +
          str(t) + ' = ' + str(get(x0, 'X')))
    adjust(x0, 'X', 10.0)
    print('>>> Value of X __after__ perturbation at time ' +
          str(t) + ' = ' + str(get(x0, 'X')))
```

Once defined, perturbation functions must be attached to the model object programmatically at the desired simulation times. For simple examples consisting of a single function call, the Python `lambda` keyword can be used to implement simple perturbation functions.

This approach to defining perturbations obviates the need to define a full DSL capable of implementing a desired set of environmental responses to the behavior of the simulated CRN. In addition, free variables in the perturbation function can be used to pass in additional information to the perturbation function. For example, random number generator objects can be straightforwardly defined in the simulation setup code and referenced as free variables in the perturbation function. In this way, randomized responses can be trivially implemented, as outlined in section S4 in the Supplemental Material. Other uses of this technique could include passing a file handle into the simulator to enable facile logging of the simulation state at each perturbation.

### 4.4.4 Applying Perturbations

Perturbations can be defined in any order in the input and are organized into time order by the simulator. If multiple perturbation actions or functions are specified at the same time point (or at time points that are indistinguishable up to the tolerance of floating-point arithmetic), then all of the perturbation *actions* will be applied to the simulation first, followed by all of the perturbation *functions*. Within these two categories, the actions and functions will be applied in the order in which they were added to the model. (Note that changing the order of addition could therefore change the overall result of the perturbations applied at a given time point.)

After the perturbation actions and functions are applied, the resulting state must be checked to ensure the integrity of the simulation. First, the resulting state of the system is checked to ensure that there are no negative values; if there are, these are set to zero (negative concentrations or species counts being unrealistic). Then, if the simulation is stochastic, any noninteger values are converted to integers, by *rounding down* if necessary. This prevents fractional species counts being recorded. The simulator is then restarted, beginning from the new state that resulted from the application of the perturbations.

### 4.4.5 Model Import and Export

ProBioSim models can be saved to a human-readable text file and subsequently reparsed back from that file (with the exception of perturbation functions, which must be added to the model programmatically). Models can also be typeset via generation of corresponding LaTeX code. However, all model construction and modification functions can also be carried out programmatically from a Python script, enabling the automated, modular construction of large-scale CRNs. See section S4

in the Supplemental Material for example model definitions specified partially as text and defined programmatically using Python code.

### 4.4.6 Simulating ProBioSim Models

Time courses of ProBioSim models can be simulated under mass-action kinetics either with a deterministic semantics or with a stochastic semantics. Deterministic simulations are implemented by forming an ODE representation of the mass-action system kinetics. Briefly, for each species $X_i$, the corresponding ODE is given by

$$\frac{d[X_i]}{dt} = \sum_j (\text{stoich}(X_i, r_j) \times \text{rate}(r_j))$$

where $r_j$ are the reactions included in the model, the stoich$(X_i, r_j)$ function returns the stoichiometry (net change) in $X_i$ from one occurrence of reaction $r_j$, and the rate$(r_j)$ function returns the rate constant value associated with reaction $r_j$. The system of such mass-action ODEs for all species $X_i$ in the model is formed and then integrated over time using the `solve_ivp` function from the `scipy.integrate` module (Virtanen et al., 2020). The user can choose between using a stiff solver, which uses the `LSODA` method, or a nonstiff solver, which uses the `RK45` Runge–Kutta method.

Stochastic simulations are implemented using Gillespie's (1977) direct method, which provides an exact implementation of stochastic mass-action kinetics. Briefly, for each reaction $r_j$, the counts of all reactant species are multiplied together and by the reaction's rate constant rate$(r_j)$ to calculate the propensity $\rho_j$ of that reaction. The next reaction is chosen at random, with the probability of choosing $r_j$ given by $\rho_j/(\sum_j \rho_j)$, that is, with probability proportional to the corresponding reaction propensity. The *time* until the next reaction fires is drawn from an exponential distribution with mean $1/(\sum_j \rho_j)$. The reaction is applied to update the state, the propensities are recalculated, and the simulation loop moves on to the next iteration. (Note, however, that our backpropagation learning simulations will only use the deterministic ODE solving simulation method.)

In the following section, we present results from learning simulations on our backpropagation-based chemical neural network design, carried out using the ProBioSim simulator. We refer the reader to section S4 of the Supplemental Material for further examples of ProBioSim model code and simulation outputs, including some examples involving state-dependent and probabilistic perturbations that show the full power of the ProBioSim approach, including some examples of associative learning in simple abstract chemical reaction networks.

## 4.5 Learning Simulation Results

To demonstrate the successful operation of our CRN, we simulated it learning the XOR logic function. The truth table for this function can be summarized as follows:

F XOR F = F  F XOR T = T  T XOR F = T  T XOR T = F

Importantly, this function is not linearly separable and thus provably requires a multilayer network to implement. Unlike previous work that used the tanh nonlinearity (Arredondo & Lakin, 2022), which squashes its output into the interval $(-1, +1)$, our use of the leaky ReLU nonlinearity here means that the outputs from the system are not constrained to a finite output domain. In addition, our use of an ODE solver to carry out the learning process in the simulated chemistry means that very long training runs, such as those common in mainstream machine learning, can be challenging. Therefore, we ran some initial screens to identify systems that can be trained to implement XOR in a tractable number of training rounds.

The specific CRN instance that we simulated starts with the following initial weight values: $W00 = -0.8$, $W01 = -0.8$, $W02 = -0.8$, $W10 = -2.0$, $W11 = -0.8$, $W12 = -1.0$, $W20 = -1.2$, $W21 = 1.6$, and $W22 = -1.0$. For each neuron $i$, the values of DPOS$i$ and DNEG$i$,

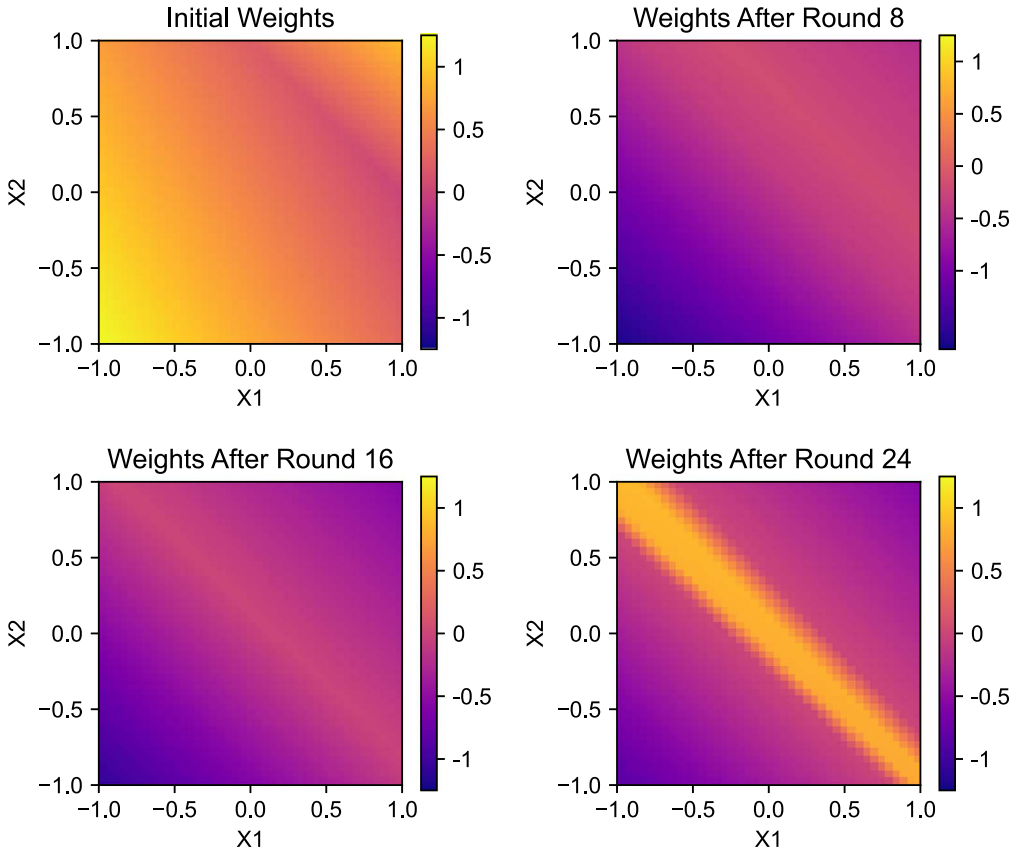


Figure 4. Heat maps of decision surfaces illustrating training of the CRN to learn the XOR logic function. The training schedule is as outlined in the main text; we present heat maps showing the outputs of the system with the initial weights, the weights after the final training round (round 24), and several intermediate points. Colors indicate the outputs from the network for each pair of input values; these were plotted by extracting the weights from the CRN and calculating the expected model output for those weights. These example plots show the system learning a band of high output from the top left to bottom right, corresponding to the expected output from the XOR function when precisely one input signal is high. Thus, our CRN design can successfully learn XOR, which is not linearly separable and thus can be implemented only by a multilayer network, via the backpropagation algorithm.

representing the gradients for the positive and negative arms of the leaky ReLU transfer function, are 1.0 and 0.1, respectively. The value of the learning rate parameter was $\alpha = 0.5$. To train the system, the following sequence of eight training instances was repeated three times, for a total of 24 training rounds:

- $X1 = -1, X2 = -1$, TARGET $= -1$;
- $X1 = -1, X2 = 1$, TARGET $= 1$;
- $X1 = 1, X2 = -1$, TARGET $= 1$;
- $X1 = 1, X2 = 1$, TARGET $= -1$;
- $X1 = -0.5, X2 = -0.5$, TARGET $= -1$;
- $X1 = -0.5, X2 = 0.5$, TARGET $= 1$;
- $X1 = 0.5, X2 = -0.5$, TARGET $= 1$; and
- $X1 = 0.5, X2 = 0.5$, TARGET $= -1$.

Table 2. Table of errors in weights learned by the CRN example illustrated in Figure 4 after each of the 24 training rounds.

| Round | W00 | W01 | W02 | W10 | W11 | W12 | W20 | W21 | W22 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | −0.0029 | −0.0029 | −0.0029 | 0.0018 | 0.0018 | 0.0018 | −0.0010 | 0.0028 | 0.0043 |
| 2 | −0.0030 | −0.0030 | −0.0028 | −0.0013 | −0.0013 | 0.0049 | −0.0007 | 0.0028 | 0.0032 |
| 3 | −0.0031 | −0.0029 | −0.0029 | −0.0048 | 0.0022 | 0.0014 | −0.0003 | 0.0029 | 0.0007 |
| 4 | −0.0005 | −0.0055 | −0.0056 | −0.0045 | 0.0018 | 0.0011 | −0.0000 | 0.0045 | 0.0002 |
| 5 | −0.0005 | −0.0055 | −0.0056 | −0.0046 | 0.0018 | 0.0010 | −0.0000 | 0.0045 | 0.0003 |
| 6 | −0.0006 | −0.0056 | −0.0055 | −0.0054 | 0.0014 | 0.0014 | 0.0000 | 0.0045 | 0.0006 |
| 7 | −0.0006 | −0.0055 | −0.0056 | −0.0060 | 0.0017 | 0.0011 | 0.0000 | 0.0045 | 0.0010 |
| 8 | −0.0004 | −0.0057 | −0.0057 | −0.0060 | 0.0017 | 0.0010 | 0.0000 | 0.0047 | 0.0010 |
| 9 | −0.0004 | −0.0057 | −0.0057 | −0.0063 | 0.0014 | 0.0008 | 0.0001 | 0.0048 | 0.0010 |
| 10 | −0.0005 | −0.0057 | −0.0057 | −0.0067 | 0.0010 | 0.0012 | 0.0001 | 0.0048 | 0.0014 |
| 11 | −0.0005 | −0.0057 | −0.0057 | −0.0072 | 0.0015 | 0.0007 | 0.0001 | 0.0048 | 0.0018 |
| 12 | −0.0006 | −0.0056 | −0.0057 | −0.0072 | 0.0015 | 0.0007 | 0.0002 | 0.0047 | 0.0018 |
| 13 | −0.0005 | −0.0056 | −0.0057 | −0.0069 | 0.0017 | 0.0009 | 0.0001 | 0.0046 | 0.0020 |
| 14 | −0.0006 | −0.0056 | −0.0057 | −0.0072 | 0.0016 | 0.0011 | 0.0001 | 0.0046 | 0.0024 |
| 15 | −0.0006 | −0.0056 | −0.0057 | −0.0075 | 0.0017 | 0.0009 | 0.0000 | 0.0046 | 0.0028 |
| 16 | −0.0003 | −0.0058 | −0.0058 | −0.0075 | 0.0017 | 0.0009 | 0.0000 | 0.0048 | 0.0028 |
| 17 | −0.0003 | −0.0057 | −0.0058 | −0.0074 | 0.0018 | 0.0010 | 0.0000 | 0.0048 | 0.0029 |
| 18 | −0.0004 | −0.0058 | −0.0058 | −0.0077 | 0.0015 | 0.0013 | −0.0000 | 0.0048 | 0.0033 |
| 19 | −0.0004 | −0.0057 | −0.0058 | −0.0080 | 0.0018 | 0.0010 | −0.0001 | 0.0048 | 0.0037 |
| 20 | −0.0005 | −0.0056 | −0.0057 | −0.0081 | 0.0019 | 0.0010 | 0.0001 | 0.0046 | 0.0037 |
| 21 | −0.0005 | −0.0056 | −0.0057 | −0.0074 | 0.0022 | 0.0013 | −0.0000 | 0.0045 | 0.0037 |
| 22 | −0.0006 | −0.0057 | −0.0056 | −0.0067 | 0.0026 | 0.0010 | −0.0007 | 0.0045 | 0.0040 |
| 23 | −0.0006 | −0.0057 | −0.0056 | −0.0066 | 0.0025 | 0.0010 | −0.0015 | 0.0045 | 0.0042 |
| 24 | −0.0003 | −0.0058 | −0.0057 | −0.0065 | 0.0025 | 0.0010 | −0.0015 | 0.0047 | 0.0041 |

*Note.* Error values were calculated as CRN output minus expected output and are shown to four decimal places. These values demonstrate that the learning calculations carried out by our CRN design produce results very close to those expected based on a direct mathematical calculation of the expected system behavior.

Here we use values of 1 and −1 to represent "true" and "false" logic values, respectively. Thus the first four training rounds specify the desired behavior at the extrema of the intended input ranges. The final four training rounds specify certain intermediate values, so as to help drive the system toward the desired form of decision surface.

The learning CRN corresponding to this system was compiled into reactions and perturbations, and the resulting ProBioSim model is presented in section S3 of the Supplemental Material. This CRN was simulated by integrating a deterministic ODE model, and the resulting weights were used to calculate the decision surface after certain training rounds; these are plotted as heat maps in Figure 4. The heat maps show the decision surface induced by the weights shifting toward the desired form over the course of the training rounds; eventually, it ends up in the expected shape, with a band of high output running diagonally between the $(-1,+1)$ and $(+1,-1)$ corners of the space, with low output in the vicinity of the $(-1,-1)$ and $(+1,+1)$ corners. This qualitatively indicates that our CRN system is indeed able to learn the challenging XOR example using the backpropagation learning algorithm. To illustrate the sequence of key operations that takes place during training, section S2 of the Supplemental Material presents example time course figures from the first training round of this simulation, covering both the feedforward and backpropagation phases.

To assess the quantitative performance of our learning CRN, we compared the sequence of weights obtained in this simulation with those that would be expected to arise from a direct reference implementation of the backpropagation algorithm. We calculated the difference between the weight values observed at the end of each training round in our simulated learning CRN and the corresponding weight values obtained from a reference implementation written in Python. These values are presented in Table 2. We observe accuracy to at least two decimal places in all cases and to three or more in many cases. This indicates that our system is quantitatively correct in that its behavior closely matches that of the underlying learning algorithm that it aims to implement. We briefly outline some potential sources of error in the following pages.

## 5 Discussion

### 5.1 CRN Learning Performance

The results presented in Table 2 demonstrate that our learning CRN design qualitatively implements the backpropagation learning algorithm for our leaky ReLU neural network with relatively small errors. However, although the errors are small, they are nonzero, which means there is some potential room for improvement in our designs. One posssible source of error is that the various reactions in our CRN design are scheduled via a molecular clock; though the off-cycle clock phases are low in the times between their correct cycles, they are not zero, which means that the reactions catalyzed by them are still happening in our model, albeit with very low fluxes through them. This introduces small errors into the computation that cannot be avoided if we wish to use an autonomous clock of this form to organize the reactions, as in previous work (Arredondo & Lakin, 2022; Vasić, Soloveichik, & Khurshid, 2020a). An additional potential source of inaccuracy is the fact that our leaky ReLU calculations have some inherent error around the discontinuity in the derivative when the input equals zero. This is because the species representing the input signal catalyzes copying of the species representing the gradient values into the corresponding species required for computing the partial loss derivatives during the backpropagation phase; this process might fail to complete before the associated clock cycle ends if the input value is very close to zero. This could be alleviated by increasing the value of the kCopy rate constant.

### 5.2 Learning CRN Design Considerations

The design of our CRN and neural network reflects certain limitations of the CRN formalism and the ability to train within that formalism. In particular, the basic arithmetic operations that our system must perform (addition, subtraction, and multiplication) can all be implemented straightforwardly in an abstract CRN using established techniques (Buisman et al., 2009). We use the molecular clock methodology to sequence reactions into discrete phases in large part because this makes it simpler to calculate the discontinuous derivatives of the leaky ReLU units by simply copying the appropriate value into a specific output species, before using that value for computation

in a subsequent step. Other approaches, such as rate-independent calculations of the nonleaky ReLU function (Vasić, Soloveichik, & Khurshid, 2020b), are valid only for integer-valued signal representations in stochastic models; furthermore, as has been established, that nonlinearity suffers from the "dying ReLU" problem in that neurons can no longer learn if their input goes negative and the derivative is zero. Note also that in previous work (Arredondo & Lakin, 2022), we used a tanh nonlinearity and trained the system via weight perturbation rather than backpropagation; that was done because our approach to calculating the $\tanh(x)$ function could not be straightforwardly applied to compute its derivative, which is $1 - \tanh^2(x)$, because the second derivative at $x = 0$ is already zero. By not using a sigmoidal nonlinearity here, we avoid that problem, but there is the alternative possibility of unbounded output signals being produced by our network, which can cause training to fail. Thus the approach taken here is shaped by the desire to produce a system that can be trained within the chemistry using relatively well-known approaches, while still using a transfer function that is of practical relevance.



### 5.3 Future Directions for Work on Learning CRNs

Future work on learning CRNs such as ours could take a number of directions. For example, the approach taken here of using sequential phases to execute the various parts of the calculation could be adapted to these other approaches with discontinuities in their derivatives. Previous work (Linder et al., 2021) also used the "hardtanh" transfer function (a piecewise linear approximation to tanh); this could be amenable to our approach. However, that system suffers from some of the same issues as the nonleaky ReLU function due to having gradients of zero in both the positive and negative directions. In addition, the use of complex CRN designs such as those presented here to implement these learning systems would make them extremely challenging to realize in an actual wet chemistry experiment. Some of these challenges could be ameliorated in future iterations of our learning CRN design, for example, by trying to reduce the number of distinct clock phases required to drive the system, or perhaps by doing away with the autonomous molecular clock altogether and using manually controlled signals to regulate the operation of the circuit.

Ideally, chemical learning systems would leverage the massive parallelism inherent in chemistry to advantage, and possible approaches might eventually move beyond feedforward networks to recurrent networks. Related work on chemical Boltzmann machines (Poole et al., 2017) and message-passing inference algorithms (Napp & Adams, 2013) could offer inspiration here, as could work on chemical implementations of reservoir computing systems (Goudarzi et al., 2013), which use a highly recurrent fixed network attached to a linear readout layer that could be trained relatively straightforwardly.

Given the practical difficulties with building molecular computing networks in the laboratory, the development of adaptive and trainable molecular systems is an important goal for the field of molecular programming, not least because it would enable a "build now, train later" approach that could greatly simplify the development and deployment of molecular circuits with specific autonomous behaviors for practical tasks, such as the diagnosis of disease (Lopez et al., 2018; C. Zhang et al., 2020). Finally, molecular learning systems could find applications in Artificial Life, as they could serve as adaptive control programs for engineered lifelike systems and synthetic cells.

Thus a major challenge in the field of chemical learning networks is to find a formalism that both fits naturally with the CRN framework and is also tractable to simulate and, ultimately, build in the laboratory. The latter criterion would imply that the system would need to be very simple; it could well be that learning systems that go beyond supervised learning to embrace alternative frameworks, like reinforcement learning (Banda et al., 2014) or associative learning (Fernando et al., 2009), could hold the answer.

With regard to our particular design, the presence of trimolecular reactions is a possible sticking point for future experimental implementations; however, we note that CRN implementation frameworks such as DNA strand displacement reactions (Soloveichik et al., 2010; D. Y. Zhang & Seelig, 2011) are theoretically capable of realizing such systems via their multistep encodings. However, it

is worth noting that the third reactant in the trimolecular reactions in our scheme is always a clock catalyst; this could make it easier to realize in an experimental system via alternative means, such as the presence or absence of some nonchemical signal that controls whether the reactions may proceed, such as an optical signal (Prokup et al., 2012). Furthermore, given that our design relies on competition for input species to drive the computation, rate constants do need to be well balanced to ensure that the results are accurate; this is a fundamental difficulty in analog computation. Recent work on rate-independent chemical implementations of ReLU networks (Vasić et al., 2022) offers a possible way forward here; alternatively, for an experimental implementation using DNA strand displacement, models now exist to predict strand displacement kinetics based on sequence (J. X. Zhang et al., 2021). Experimentalists are also able to tune the concentrations of other components as necessary to recover the desired kinetic balance (Thubagere, Thachuk, et al., 2017).

### 5.4 Future Directions for Work on the ProBioSim Simulator

Future work to further develop the ProBioSim simulator may include performance optimizations as well as the implementation of alternative simulation algorithms, such as tau-leaping (Gillespie, 2001), or the implementation of additional rate laws, such as Michaelis–Menten kinetics. To enhance the expressiveness of the system, the format of the perturbation functions outlined herein could be generalized to modify other simulation parameters. In particular, the ability to perturb rate constant values would allowProBioSim to straightforwardly model physical changes in the system, such as changes in temperature or pH (Idili et al., 2014) or the activation of optical control signals (Prokup et al., 2012). These are of interest because they could be used to modulate the behavior of an experimental implementation of the learning CRNs outlined earlier. The simulator could also be modified to explicitly model dilution effects when species are added to perturb the system, as these are *not* currently modeled explicitly. Our current assumption is reasonable if one makes the assumption that the volume of new sample added during the perturbation is negligible compared to the overall reaction volume. However, it is worth noting that the flexibility of our perturbation function approach could already be used to model these effects; any changes in this regard would be purely for convenience. One could also imagine implementing triggers for events based on state-based predicates, such as the counts or concentrations of various species. These could be implemented (to some extent) within the perturbation function framework, although such conditional perturbations could only possibly be applied at the prespecified times at which the perturbation function is applied. Finally, the ProBioSim simulator could in principle be integrated with techniques for parameter estimation to infer rate constants and other model parameters based on experimental data, although this would only be of real use if such data were available: this is not yet the case for experimental implementations of learning in chemical reaction networks.

## 6 Conclusions

In conclusion, we have reported a design for novel abstract CRNs that can implement supervised learning via the standard backpropagation training algorithm. This CRN design directly implements the mathematics of backpropagation for a class of nonlinear neurons using the leaky ReLU nonlinear transfer function. We have demonstrated the capabilities of this system to carry out learning tasks by training it to learn the two-input XOR logic function, a function that is not linearly separable and thus provably requires the use of a multilayer network. In addition, we have reported ProBioSim, a simulator for CRNs that incorporates the novel capability to define perturbations to the concentrations (or counts) of species as the results of arbitrary pieces of code expressed in the host programming language, Python. This enables the straightforward modeling of nontrivial environmental responses to the behavior of the CRN component of the system, such as our training protocols. These would otherwise require an ad hoc infrastructure for calculating and implementing these perturbations to be constructed around the underlying simulator. Our approach allows such calculations to be integrated directly into the simulation engine via a straightforward mechanism,

enabling facile programming of complex interdependencies between an abstract CRN and the simulated training environment. ProBioSim is released under an open source license, and the source code is available online at https://github.com/matthewlakin/ProBioSim/.

## Acknowledgments

This material is based on work supported by the National Science Foundation under grants 1518861 and 1935087.

## References

Abel, J. H., Drawert, B., Hellander, A., & Petzold, L. R. (2016). GillesPy: A Python package for stochastic model building and simulation. *IEEE Life Sciences Letter*, *2*(3), 35–38. https://doi.org/10.1109/LLS.2017.2652448, PubMed: 28630888

Arredondo, D., & Lakin, M. R. (2022). Supervised learning in a multilayer, nonlinear chemical neural network. *IEEE Transactions on Neural Networks and Learning Systems*. https://doi.org/10.1109/TNNLS.2022.3146057, PubMed: 35133970

Badelt, S., Grun, C., Sarma, K. V., Wolfe, B., Shin, S. W., & Winfree, E. (2020). A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. *Journal of the Royal Society Interface*, *17*(167), 20190866. https://doi.org/10.1098/rsif.2019.0866, PubMed: 32486951

Badelt, S., Shin, S. W., Johnson, R. F., Dong, Q., Thachuk, C., & Winfree, E. (2017). A general-purpose CRN-to-DSD compiler with formal verification, optimization, and simulation capabilities. In R. Brijder & L. Qian (Eds.), *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming* (Lecture Notes in Computer Science, Vol. 10467, pp. 232–248). Springer. https://doi.org/10.1007/978-3-319-66799-7_15

Banda, P., & Teuscher, C. (2016). COEL: A cloud-based reaction network simulator. *Frontiers in Robotics and AI*, *3*, 13. https://doi.org/10.3389/frobt.2016.00013

Banda, P., Teuscher, C., & Stefanovic, D. (2014). Training an asymmetric signal perceptron through reinforcement in an artificial chemistry. *Journal of the Royal Society Interface*, *11*(93), 20131100. https://doi.org/10.1098/rsif.2013.1100, PubMed: 24478284

Blount, D., Banda, P., Teuscher, C., & Stefanovic, D. (2017). Feedforward chemical neural network: An in silico chemical system that learns XOR. *Artificial Life*, *23*(3), 295–317. https://doi.org/10.1162/ARTL_a_00233, PubMed: 28786723

Boutillier, P., Maasha, M., Li, X., Medina-Abarca, H. F., Krivine, J., Feret, J., Cristescu, I., Forbes, A. G., & Fontana, W. (2018). The Kappa platform for rule-based modeling. *Bioinformatics*, *34*(13), i583–i592. https://doi.org/10.1093/bioinformatics/bty272, PubMed: 29950016

Buisman, H. J., ten Eikelder, H. M. M., Hilbers, P. A. J., & Liekens, A. M. L. (2009). Computing algebraic functions with biochemical reaction networks. *Artificial Life*, *15*(1), 5–19. https://doi.org/10.1162/artl.2009.15.1.15101, PubMed: 18855568

Cardelli, L. (2010). Strand algebras for DNA computing. *Natural Computing*, *10*(1), 407–428. https://doi.org/10.1007/s11047-010-9236-7

Cardelli, L. (2013). Two-domain DNA strand displacement. *Mathematical Structures in Computer Science*, *23*, 247–271. https://doi.org/10.1017/S0960129512000102

Chatterjee, G., Chen, Y. J., & Seelig, G. (2018). Nucleic acid strand displacement with synthetic mRNA inputs in living mammalian cells. *ACS Synthetic Biology*, *7*(12), 2737–2741. https://doi.org/10.1021/acssynbio.8b00288, PubMed: 30441897

Chen, Y.-J., Dalchau, N., Srinivas, N., Phillips, A., Cardelli, L., Soloveichik, D., & Seelig, G. (2013). Programmable chemical controllers made from DNA. *Nature Nanotechnology*, *8*(10), 755–762. https://doi.org/10.1038/nnano.2013.189, PubMed: 24077029

Chen, Y.-J., Groves, B., Muscat, R. A., & Seelig, G. (2015). DNA nanotechnology from the test tube to the cell. *Nature Nanotechnology*, *10*(9), 748–760. https://doi.org/10.1038/nnano.2015.195, PubMed: 26329111

Cherry, K. M., & Qian, L. (2018). Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, *559*(7714), 370–376. https://doi.org/10.1038/s41586-018-0289-6, PubMed: 29973727

Cook, M., Soloveichik, D., Winfree, E., & Bruck, J. (2009). Programmability of chemical reaction networks. In A. Condon, D. Harel, J. N. Kok, A. Salomaa, & E. Winfree (Eds.), *Algorithmic bioprocesses* (pp. 543–584). Springer. https://doi.org/10.1007/978-3-540-88869-7_27

Dexter, J. P., Prabakaran, S., & Gunawardena, J. (2019). A complex hierarchy of avoidance behaviors in a single-cell eukaryote. *Current Biology*, *29*(24), 4323–4329. https://doi.org/10.1016/j.cub.2019.10.059, PubMed: 31813604

Fernando, C. T., Liekens, A. M. L., Bingle, L. E. H., Beck, C., Lenser, T., Stekel, D. J., & Rowe, J. E. (2009). Molecular circuits for associative learning in single-celled organisms. *Journal of the Royal Society Interface*, *6*(34), 463–469. https://doi.org/10.1098/rsif.2008.0344, PubMed: 18835803

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, *81*(25), 2340–2361. https://doi.org/10.1021/j100540a008

Gillespie, D. T. (2001). Approximate accelerated stochastic simulation of chemically reacting systems. *Journal of Chemical Physics*, *115*(4), 1716–1733. https://doi.org/10.1063/1.1378322

Goudarzi, A., Lakin, M. R., & Stefanovic, D. (2013). DNA reservoir computing: A novel molecular computing approach. In D. Soloveichik & B. Yurke (Eds.), *Proceedings of the 19th International Conference on DNA Computing and Molecular Programming* (Lecture Notes in Computer Science, Vol. 8141, pp. 76–89). Springer. https://doi.org/10.1007/978-3-319-01928-4_6

Groves, B., Chen, Y. J., Zurla, C., Pochekailov, S., Kirschman, J. L., Santangelo, P. J., & Seelig, G. (2016). Computing in mammalian cells with nucleic acid strand exchange. *Nature Nanotechnology*, *11*(3), 287–294. https://doi.org/10.1038/nnano.2015.278, PubMed: 26689378

Hennessey, T. M., Rucker, W. B., & McDiarmid, C. G. (1979). Classical conditioning in paramecia. *Animal Learning and Behavior*, *7*(4), 417–423. https://doi.org/10.3758/BF03209695

Idili, A., Vallée-Bélisle, Alexis., & Ricci, F. (2014). Programmable pH-triggered DNA nanoswitches. *Journal of the American Chemical Society*, *136*(16), 5836–5839. https://doi.org/10.1021/ja500619w, PubMed: 24716858

Kluyver, T., Ragan-Kelley, B., Pérez, Fernando., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). Jupyter Notebooks—A publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (Eds.), *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (pp. 87–90). IOS Press. https://doi.org/10.3233/978-1-61499-649-1-87

Lakin, M. R., Minnich, A., Lane, T., & Stefanovic, D. (2014). Design of a biochemical circuit motif for learning linear functions. *Journal of the Royal Society Interface*, *11*(101), 20140902. https://doi.org/10.1098/rsif.2014.0902, PubMed: 25401175

Lakin, M. R., & Phillips, A. (2020). Domain-specific programming languages for computational nucleic acid systems. *ACS Synthetic Biology*, *9*(7), 1499–1513. https://doi.org/10.1021/acssynbio.0c00050, PubMed: 32589838

Lakin, M. R., & Stefanovic, D. (2016). Supervised learning in adaptive DNA strand displacement networks. *ACS Synthetic Biology*, *5*(8), 885–897. https://doi.org/10.1021/acssynbio.6b00009, PubMed: 27111037

Lakin, M. R., Youssef, S., Polo, F., Emmott, S., & Phillips, A. (2011). Visual DSD: A design and analysis tool for DNA strand displacement systems. *Bioinformatics*, *27*(22), 3211–3213. https://doi.org/10.1093/bioinformatics/btr543, PubMed: 21984756

Linder, J., Chen, Y. J., Wong, D., Seelig, G., Ceze, L., & Strauss, K. (2021). Robust digital molecular design of binarized neural networks. In M. R. Lakin & P. Šulc (Eds.), *27th International Conference on DNA Computing and Molecular Programming (DNA 27)* (Vol. 205, pages 1:1–1:20). Schloss Dagstuhl/Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.DNA.27.1

Lopez, R., Wang, R., & Seelig, G. (2018). A molecular multi-gene classifier for disease diagnostics. *Nature Chemistry*, *10*, 746–754. https://doi.org/10.1038/s41557-018-0056-1, PubMed: 29713032

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.

Napp, N. E., & Adams, R. P. (2013). Message passing inference with chemical reaction networks. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* (Vol. 26, pp. 2247–2255). Curran Associates.

Poole, W., Ortiz-Muñoz, A., Behera, A., Jones, N. S., Ouldridge, T. E., Winfree, E., & Gopalkrishnan, M. (2017). Chemical Boltzmann machines. In R. Brijder & L. Qian (Eds.), *Proceedings of the 23rd International*

*Conference on DNA Computing and Molecular Programming* (Lecture Notes in Computer Science, Vol. 10467, pp. 210–231). Springer. https://doi.org/10.1007/978-3-319-66799-7_14

Prokup, A., Hemphill, J., & Deiters, A. (2012). DNA computation: A photochemically controlled AND gate. *Journal of the American Chemical Society*, *134*(8), 3810–3815. https://doi.org/10.1021/ja210050s, PubMed: 22239155

Qian, L., & Winfree, E. (2011a). Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, *332*(6034), 1196–1201. https://doi.org/10.1126/science.1200520, PubMed: 21636773

Qian, L., & Winfree, E. (2011b). A simple DNA gate motif for synthesizing large-scale circuits. *Journal of the Royal Society Interface*, *8*(62), 1281–1297. https://doi.org/10.1098/rsif.2010.0729, PubMed: 21296792

Qian, L., Winfree, E., & Bruck, J. (2011). Neural network computation with DNA strand displacement cascades. *Nature*, *475*(7356), 368–372. https://doi.org/10.1038/nature10262, PubMed: 21776082

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536. https://doi.org/10.1038/323533a0

Sanft, K. R., Wu, S., Roh, M., Fu, J., Lim, R. K., & Petzold, L. R. (2011). StochKit2: Software for discrete stochastic simulation of biochemical systems with events. *Bioinformatics*, *27*(17), 2457–2458. https://doi.org/10.1093/bioinformatics/btr401, PubMed: 21727139

Seelig, G., Soloveichik, D., Zhang, D. Y., & Winfree, E. (2006). Enzyme-free nucleic acid logic circuits. *Science*, *314*(5805), 1585–1588. https://doi.org/10.1126/science.1132493, PubMed: 17158324

Simons, T., & Lee, D. J. (2019). A review of binarized neural networks. *Electronics*, *8*(6), 661. https://doi.org/10.3390/electronics8060661

Soloveichik, D., Seelig, G., & Winfree, E. (2010). DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences of the United States of America*, *107*(12), 5393–5398. https://doi.org/10.1073/pnas.0909380107, PubMed: 20203007

Thubagere, A. J., Li, W., Johnson, R. F., Chen, Z., Doroudi, S., Lee, Y. L., Izatt, G., Wittman, S., Srinivas, N., Woods, D., Winfree, E., & Qian, L. (2017). A cargo-sorting DNA robot. *Science*, *357*(6356), eaan6558. https://doi.org/10.1126/science.aan6558, PubMed: 28912216

Thubagere, A. J., Thachuk, C., Berleant, J., Johnson, R. F., Ardelean, D. A., Cherry, K. M., & Qian, L. (2017). Compiler-aided systematic construction of large-scale DNA strand displacement circuits using unpurified components. *Nature Communications*, *8*, 14373. https://doi.org/10.1038/ncomms14373, PubMed: 28230154

Vasić, M., Chalk, C., Khurshid, S., & Soloveichik, D. (2020). Deep molecular programming: A natural implementation of binary-weight ReLU neural networks. In *Proceedings of ICML 2020* (pp. 9701–9711). PMLR.

Vasić, Marko., Chalk, C., Luchsinger, A., Khurshid, S., & Soloveichik, D. (2022). Programming and training rate-independent chemical reaction networks. *Proceedings of the National Academy of Sciences of the United States of America*, *119*(24), e2111552119. https://doi.org/10.1073/pnas.2111552119, PubMed: 35679345

Vasić, M., Soloveichik, D., & Khurshid, S. (2020a). CRN++: Molecular programming language. *Natural Computing*, *19*, 391–407. https://doi.org/10.1007/s11047-019-09775-1

Vasić, M., Soloveichik, D., & Khurshid, S. (2020b). CRNs exposed: A method for the systematic exploration of chemical reaction networks. In C. Geary & M. J. Patitz (Eds.), *26th International Conference on DNA Computing and Molecular Programming (DNA 26)* (Vol. 174, pp. 4:1–4:25). Schloss Dagstuhl/Leibniz-Zentrum für Informatik.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2, PubMed: 32015543

Wood, D. C. (1988). Habituation in *Stentor*: A response-dependent process. *Journal of Neuroscience*, *8*(7), 2248–2253. https://doi.org/10.1523/JNEUROSCI.08-07-02248.1988, PubMed: 3249222

Yordanov, B., Kim, J., Petersen, R. L., Shudy, A., Kulkarni, V. V., & Phillips, A. (2014). Computational design of nucleic acid feedback control circuits. *ACS Synthetic Biology*, *3*(8), 600–616. https://doi.org/10.1021/sb400169s, PubMed: 25061797

Zhang, C., Zhao, Y., Xu, X., Xu, R., Li, H., Teng, X., Du, Y., Miao, Y., Lin, H.-C., & Han, D. (2020). Cancer diagnosis with DNA molecular computation. *Nature Nanotechnology*, *15*(8), 709–715. https://doi.org/10.1038/s41565-020-0699-0, PubMed: 32451504

Zhang, D. Y., & Seelig, G. (2011). Dynamic DNA nanotechnology using strand-displacement reactions. *Nature Chemistry*, *3*(2), 103–113. https://doi.org/10.1038/nchem.957, PubMed: 21258382

Zhang, J. X., Yordanov, B., Gaunt, A., Wang, M. X., Dai, P., Chen, Y.-J., Zhang, K., Fang, J. Z., Dalchau, N., Li, J., Phillips, A., & Zhang, D. Y. (2021). A deep learning model for predicting next-generation sequencing depth from DNA sequence. *Nature Communications*, *12*(1), 4387. https://doi.org/10.1038/s41467-021-24497-8, PubMed: 34282137