Information and Computation ••• (••••) ••••••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Automated deep abstractions for stochastic chemical reaction networks

Denis Repin [a,b], Tatjana Petrov [a,b,*]

[a] Department of Computer and Information Science, University of Konstanz, Konstanz, Germany
[b] Centre for the Advanced Study of Collective Behaviour, University of Konstanz, Konstanz, Germany

ARTICLE INFO

Article history:
Received 31 July 2020
Received in revised form 27 May 2021
Accepted 18 July 2021
Available online xxxx

Keywords:
Model abstraction
Stochastic simulation
Chemical reaction networks
Deep learning
Neural architecture search

ABSTRACT

Predicting stochastic cellular dynamics emerging from chemical reaction networks (CRNs) is a long-standing challenge in systems biology. Deep learning was recently used to abstract the CRN dynamics by a mixture density neural network, trained with traces of the original process. Such abstraction is dramatically cheaper to execute, yet it preserves the statistical features of the training data. However, in practice, the modeller has to take care of finding the suitable neural network architecture manually, for each given CRN, through a trial-and-error cycle. In this paper, we propose to further automatise deep abstractions for stochastic CRNs, through learning the neural network architecture along with learning the transition kernel of the stochastic process. The method is applicable to any given CRN, time-saving for deep learning experts and crucial for non-specialists. We demonstrate performance over a number of CRNs with multi-modal phenotypes and a multi-scale scenario where CRNs interact across a spatial grid.

1. Introduction

Reaction networks are a simple but powerful formalism for modeling dynamical systems. For example, the following set of reactions constitutes a reaction network with three species S, I, R and three reactions: $S \to I$, $S + I \to 2I$, $I \to R$. This simple model is the well-known epidemiological SIR model describing the spread of an infectious disease in well-mixed population: susceptible (S) may become infected (I), and an infectious species can spread the disease by infecting a susceptible, or recovering (R). Each reaction fires with a corresponding rate. A reaction network induces a stochastic dynamical system - continuous-time Markov chain (CTMC), describing how the state - a vector enumerating multiplicities of each of the species - changes over time due to reaction events. Computationally predicting the distribution of species over time from such CTMC is generally challenging. The challenge is especially prominent in systems and synthetic biology research: when hypothesising low-level mechanisms of molecular interactions with (chemical) reaction networks, one deals with a large number of species, stochasticity and events happening at multiple time-scales. Two major approaches are used to analyse such CTMCs. The first approach focuses on computing the transient evolution of the probability related to each state of the CTMC numerically. The transient distribution evolves according to the Kolmogorov forward equation (chemical master equation in the chemistry literature), and it is typically very difficult to solve the forward equations except for the simplest systems. The second approach is based on a statistical estimation of trace distribution and event probabilities of

* Corresponding author.
E-mail addresses: den.ne.repin@gmail.com (D. Repin), tatjana.petrov@gmail.com (T. Petrov).

https://doi.org/10.1016/j.ic.2021.104788

the CTMC by generating many sample traces [1], often referred to as stochastic simulation or Gillespie algorithm (SSA). While this method generally allows to trade-off computational tractability with precision, even simulating a single trace can still take considerable processor time, especially when some reactions fire at very fast time-scales relative to the global time horizon of interest. At the same time, we are often not interested in predictions at such small time-scales or transient distributions for each of the species. For all these reasons, it is desirable to develop abstraction techniques for stochastic reaction networks, which allow for efficient simulation, yet faithfully abstract the context-relevant emerging features of the hypothesised mechanism.

Example 1 *(Motivating and running example).* The following set of reactions constitutes a reaction network with three species G_1, G_2, P_1, and six reactions:

$$G_1 \underset{\alpha_{12}}{\overset{\alpha_{11}}{\rightleftharpoons}} G_1 + P_1,\ G_2 \xrightarrow{\alpha_{21}} G_2 + P_1,\ P_1 \xrightarrow{\beta_1} \emptyset,\ G_1 \underset{\epsilon\gamma_{21}}{\overset{\epsilon\gamma_{12}}{\rightleftharpoons}} G_2, \tag{1}$$

where $0 < \epsilon \ll 1$. This fast-and-slow network ([2], Eq. 16) models a gene slowly switching between two different expression modes. In Fig. 1, we show a sample trajectory for Ex. (1): notice the different magnitudes of species' abundances and times between reaction events. Assume that we are only interested in reproducing the distribution of protein P_1 at several interesting time points, e.g. at the four time points shown in Fig. 2 (bottom row). Can these distributions be efficiently predicted, by abstracting away all reaction steps within a predefined fixed time interval?

Deep abstractions, introduced in [3], propose to use available simulation algorithms to generate a suitable number of simulated system traces, and then learn an abstract model from such data. The task of learning the transition kernel for this Markov process is modelled as a probability mixture with parameters depending on the system state, and a deep neural network is trained on simulated data to parametrise this probability mixture. Such abstract model preserves the statistics of the original network dynamics, but runs on a discrete time-scale representing equally distributed time intervals, abstracting away all intermediate transitions, which can lead to significant computational savings.

1.1. Contributions

The performance of any deep learning application largely depends on the choice of the neural network architecture, usually constructed by the user (neural network engineer) through a trial-and-error process. When applying deep abstractions proposed in [3], the modeller has to take care of finding the suitable architecture manually, for each given CRN. The main contributions of this paper are the following:

- We define a framework for deep abstractions which can be automatically applied to any given CRN model. Beyond the framework shown in [3], the neural network architecture search is automated: in parallel to learning the kernel of the stochastic process, we learn a neural network architecture, by employing the recent advances on this topic in the deep learning community [4,5]. Moreover, in addition to the initial state, the neural network can be parametrised with the kinetic rates (as a part of the input).
- We analyse the quality of model reduction on an extended suite of case studies relevant for systems and synthetic biology: on large-scale, detailed mechanistic model of signal transduction (Epidermal Growth Factor Receptor - EGFR pathway), and a number of small-scale auto-regulatory genetic networks with multi-scale dynamics and multi-modal phenotypes.
- We demonstrate how the proposed model abstraction with deep learning can be used for efficient multi-scale modelling of complex collective dynamics. More specifically, we describe a simulation of a number of individual agents, each occupying one cell in a spatial grid. Each agent is a chemical reaction network (CRN) itself, and agents communicate among themselves via shared species that diffuse locally across space. In this setup, we illustrate that our reduction enables efficient simulation of a collective motion emerging from the mechanistic description of individual dynamics, which would otherwise be impossible. Finally, we point to the limitations and challenges for future works.

It is worth mentioning that the main goal of the automated architecture search proposed in this manuscript is not to out-perform any possible man-made architecture used for deep abstractions, but rather to automatise the search of neural network architecture for any given input CRN, and thus save time and effort of manual design and trials. The accompanying tool is implemented is available as an open source Python library StochNetV2, available on GitHub and described in [6].

1.2. Related works

Different techniques on reducing stochastic CRNs have been proposed in literature and practice over the years. Classical limit approximations of deterministic limit, moments or mean-field approximation [7,8] can provide significant computational savings, but they do not apply to general stochastic CRNs, especially when species distributions emerging over time are non-Gaussian, as for example is the case shown in Ex. (1). Moreover, principled model reduction techniques have been

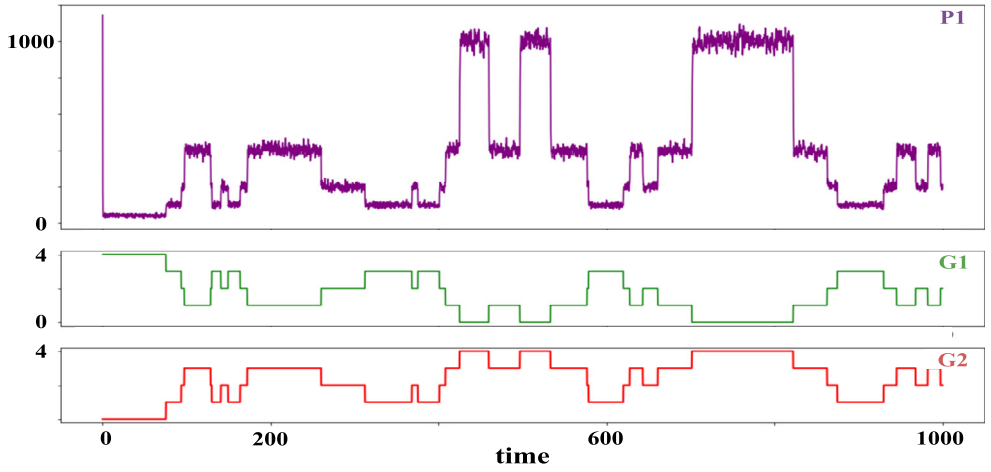


Fig. 1. Sample trajectory of Ex. (1) network.

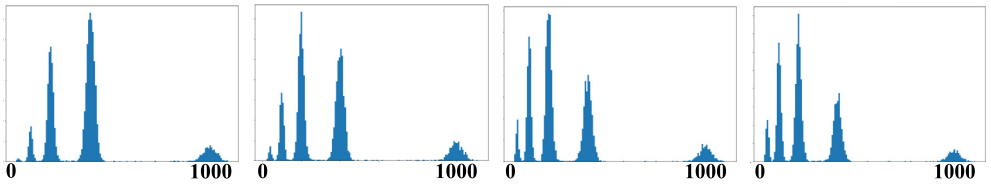


Fig. 2. Distribution (histogram) of the protein P_1 at times 20, 50, 100, and 200 for Ex. (1) network.

proposed in several aggregation [9–15] and time-scale separation frameworks [16–19]. These techniques are generally based on detecting species, reactions or states which are behaviourally indistinguishable or similar. In these methods, the space of considered abstractions is typically discrete and as a consequence, it does not allow smooth tuning of abstracted entities, or control of abstraction accuracy. In other words, the achieved abstraction may or may not be significant, and once the method is applied, it is difficult to further improve it, both in terms of abstraction size, and accuracy. This is different to deep abstractions, where the abstraction accuracy can be improved by increasing the model size and/or adding more training data, and increasing time discretisation interval improves abstraction efficiency.

The idea of deep abstractions, first introduced in [3], has been recently employed for emulating epidemiological processes with Mixture Density Networks (MDNs) [20]; However, this work has focused on a single, uni-modal model of epidemics and only stationary regime, while our method is generic - applicable to any given CRN. Other abstractions based on statistical analysis of traces, close to the idea of deep abstractions, include [21], who proposed to construct abstractions using information theory to discretise the state space and select a subset of all original variables and their mutual dependencies (a Dynamic Bayesian Network is constructed to produce state transitions). Another statistical approach to approximate dynamics of fast-and-slow models was developed in [22], where Gaussian Processes are used to predict the state of the fast equilibrating internal process as a function of the environmental state. It is worth noting that all the mentioned reduction techniques, except from the exact frameworks based on syntactic criteria, such as in [12,9], do not guarantee error bounds a priori.

2. Background and preliminaries

In this Section, we briefly recall the notions used throughout this manuscript: chemical reaction networks equipped with stochastic semantics (CRNs), neural networks (NNs), and mixture density networks (MDNs). Set of natural numbers will be denoted by $\mathbb{N}$ and reals by $\mathbb{R}$. For presentation clarity, some notation will not be uniquely used throughout (e.g. we will use x to represent the species multiplicity vector that is a state of a CRN and at the same time we will use x to represent the input of a NN); However, each variable will be uniquely and clearly referred to in given context.

2.1. Stochastic chemical kinetics

Chemical reaction networks (CRN) are a simple yet powerful and versatile formalism for modelling population dynamics, used across different application domains, ranging from cancer biology to theoretical ecology, including modelling spreading processes such as epidemics.

Definition 1 *(CRN).* A Chemical Reaction Network is a pair (S, R), such that $\mathsf{S} = \{S_1, \ldots, S_m\}$ is a finite set of species, and $\mathsf{R} = \{\mathsf{r}_1, \ldots, \mathsf{r}_r\}$ is a finite set of reactions. The state of a system can be represented as a multi-set of species, denoted by $\mathbf{x} = (x_1, ..., x_m) \in \mathbb{N}^m$. Each reaction is a triple $\mathsf{r}_j \equiv (\mathbf{a}_j, \boldsymbol{\nu}_j, c_j) \in \mathbb{N}^m \times \mathbb{N}^m \times \mathbb{R}_{\geq 0}$, written down in the following form:

$$a_{1j}S_1, \ldots, a_{mj}S_m \xrightarrow{c_j} a'_{1j}S_1, \ldots, a'_{mj}S_m, \text{ such that } \forall i.a'_{ij} = a_{ij} + \nu_{ij},$$

where the vectors $\mathbf{a}_j$ and $\mathbf{a}'_j$ are respectively the *consumption* and *production* vectors due to jth reaction, vector $\boldsymbol{\nu}_j$ specifies the net-change upon the reaction r_j (if the jth reaction occurs, after being in state $\mathbf{x}$, the next state will be $\mathbf{x}' = \mathbf{x} + \boldsymbol{\nu}_j$, and it can occur only if sufficiently many reactants are available, i.e. $x_i \geq a_{ij}$ for $i = 1, \ldots, m$). Finally, c_j is the respective *kinetic rate*. In stochastic chemical kinetics used for modelling molecular interactions, the existence of reaction rates is justified by the laws of physical chemistry, and reaction lists are equipped with the stochastic semantics that is valid under mild physical assumptions [1].

Stochastic semantics. The species multiplicities follow a continuous-time Markov chain (CTMC) $\{\eta_t\}_{t\geq 0}$, defined over the state space $S = \{\mathbf{x} \mid \mathbf{x} \text{ is reachable from } \mathbf{x}_0 \text{ by a finite sequence of reactions from } \{\mathsf{r}_1, \ldots, \mathsf{r}_r\}\}$. In other words, the probability of moving to the state $\mathbf{x} + \boldsymbol{\nu}_j$ from $\mathbf{x}$ after time Δ is

$$\mathsf{P}(\eta_{t+\Delta} = \mathbf{x} + \nu_j \mid \eta_t = \mathbf{x}) = \lambda_j(\mathbf{x})\Delta + o(\Delta),$$

with λ_j the propensity of jth reaction, assumed to follow the principle of mass-action: $\lambda_j(\mathbf{x}) = c_j \prod_{i=1}^{m} \binom{x_i}{a_{ij}}$, and $\frac{o(\Delta)}{\Delta}$ negligible for infinitesimally small Δ. The binomial coefficient $\binom{x_i}{a_{ij}}$ reflects the probability of choosing a_{ij} molecules of species S_i out of x_i available ones.

Computing the transient. Using the vector notation $\boldsymbol{\eta}_t \in \mathbb{N}^n$ for the marginal of process $\{\eta_t\}_{t\geq 0}$ at time t, we can compute this transient distribution by integrating the *chemical master equation* (CME). Denoting by $p^{(t)}(\mathbf{x}) = \mathsf{P}(\boldsymbol{\eta}_t = \mathbf{x})$, the CME for state $\mathbf{x} \in \mathbb{N}^m$ reads

$$\frac{\mathrm{d}}{\mathrm{d}t}p^{(t)}(\mathbf{x}) = \sum_{j=1, \mathbf{x}-\boldsymbol{\nu}_j \in \mathsf{S}}^{r} \lambda_j(\mathbf{x} - \boldsymbol{\nu}_j)p^{(t)}(\mathbf{x} - \boldsymbol{\nu}_j) - \sum_{j=1}^{r} \lambda_j(\mathbf{x})p^{(t)}(\mathbf{x}). \tag{2}$$

The solution may be obtained by solving the system of differential equations, but, due to its high (possibly infinite) dimensionality, it is often statistically estimated by simulating the traces of $\{\eta_t\}$, known as the stochastic simulation (SSA) or Gillespie algorithm in chemical literature [1]. As the statistical estimation often remains computationally expensive for desired accuracy, for the case when the deterministic model is unsatisfactory due to the low multiplicities of many molecular species [23], different further approximation methods have been proposed, major challenge to which remains the quantification of approximation accuracy (see [24] and references therein for a thorough review on the subject).

2.2. Neural networks

In the last decade, deep neural networks (DNNs, NNs) gathered a lot of attention from researchers as well as from industry, bringing breakthroughs in various application areas, such as computer vision, time-series analysis, speech recognition, machine translation, etc. Neural networks are well known as a powerful and versatile framework for high-dimensional learning tasks. The key feature of neural networks is that they can represent an arbitrary function, mapping a set of input variables $(x_1, \ldots, x_n)$ to a set of output variables $(y_1, \ldots, y_m)$. Further we denote them as x and y for simplicity.

Neural networks are typically formed by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions f_1, f_2, f_3 connected in a chain to form $f(x) = f_3(f_2(f_1(x)))$. In this case, f_1 is called the first layer of the network, f_2 is called the second layer, and so on. The outputs of each layer are called *features*, or *latent representation* of the data. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity [25].

In particular, each layer usually computes a linear transformation $Wx + b$ where W is a weight matrix and b is bias vector. Additional nonlinearities are inserted in between the layers which allow the network to represent arbitrary nonlinear functions (see the illustration in Fig. 3).

NNs are trained on a set of training examples $\{(x, y)\}$ with the aim not to memorize the data, but rather to learn the underlying dependencies within the variables, so that the output y can be predicted for unseen values of x. During training, the weights in a network are adjusted to minimize the learning objective - loss function - on training data. The quality of a trained model is usually measured on unseen (test) data, which is addressed as the model's generalization ability.

2.3. Mixture density networks

However, conventional neural networks perform poorly on a class of problems involving the prediction of continuous variables, for which the target data is multi-valued. Minimising a sum-of-squares error encourages a network to approximate the conditional average of the target data, which does not capture the information on the distribution of output values.

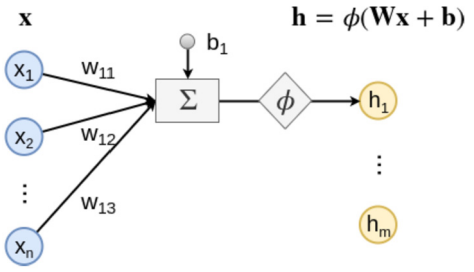


Fig. 3. A single layer of a neural network. Outputs are computed as a linear transformation $Wx + b$ followed by a non-linear activation function ϕ.

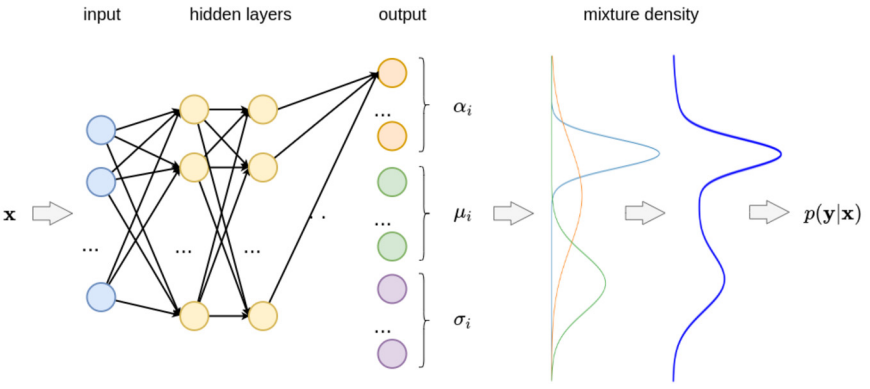


Fig. 4. Mixture Density Network structure. Given $\mathbf{x}$, neural network outputs values μ_i and $\sigma_i, i = 1, ..., m$ that define m Gaussian distributions. Weighted by mixing coefficients α_i, they form a mixture density - conditional probability density $p(\mathbf{y}|\mathbf{x})$.

Mixture Density Networks (MDN), proposed in [26], are a class of models which overcome these limitations by combining a conventional neural network with a mixture density model, illustrated in Fig. 4. This neural network provides the parameters for multiple distributions, which are then mixed by the weights (also provided by the network). Therefore Mixture Density Networks can in principle represent arbitrary conditional distributions, in the same way that a conventional neural network can represent arbitrary non-linear functions.

To construct an abstract model defined by a Markov process, we need to learn its transition kernel. In other words, given a vector η_t describing the system state at time t, we wish to predict the state $\eta_{t+\Delta t}$, which follows a distribution, conditioned on η_t. Therefore we can use Mixture Density Networks to learn the conditional distribution $p(\eta_{t+\Delta t}|\eta_t)$.

3. Deep abstractions

Here we present the main steps of the abstraction technique originally proposed in [3].

3.1. Abstract model

Let $\{\eta_t\}_{t\geq 0}$ be the CTMC describing a CRN with the state space $S = \mathbb{N}^m$. As mentioned earlier, with the abstract model we wish to reproduce the dynamics of the original process at a fixed temporal resolution. Let $\{\tilde{\eta}_i\}_{i\in\mathbb{N}}$ be a discrete-time stochastic process such that

$$\tilde{\eta}_i := \eta_{t_0+i\Delta t} \quad \forall i \in \mathbb{N} \tag{3}$$

with fixed time interval Δt and initial time t_0. The resulting process $\tilde{\eta}_i$ is a time-homogeneous discrete-time Markov chain (DTMC), with a transition kernel

$$K_d(s, s_0) = \mathbb{P}(\eta_{\Delta t} = s \mid \eta_0 = s_0) \quad \forall s_0, s \in S. \tag{4}$$

Further, two following approximations take place:

1. The state space $S = \mathbb{N}^m$ is embedded into the continuous space $\tilde{X} = \mathbb{R}^m_{\geq 0}$. The abstract model takes values in $\tilde{X}$.
2. The kernel K_d is approximated by a new kernel $K(x|x_0)$ operating in the continuous space $\tilde{X}$. The kernel $K(x|x_0)$ is modelled by a MDN.

To evaluate the abstract model, we introduce a time-bounded *reward function* r that monitors the properties we wish to preserve in the reduced model. This function, therefore, maps from the space of discrete-time trajectories S^M to an arbitrary space T (here M is an upper bound on the length of discrete-time trajectories, and $\tilde{\eta}_{[0,M]}$ denotes time-bounded trajectories). For example, it can be a projection, counting the populations of a subset of species, or it can take Boolean values corresponding to some linear temporal logic properties. Note that $r(\tilde{\eta}_{[0,M]})$ is a probability distribution on T.

As an error metric we use the distance d between the abstract distribution and $r(\tilde{\eta}_{[0,M]})$, which is evaluated statistically, as the distance among histograms, h [27]. In our experiments as a distance d we use L_1 metric:

$$d(X, Y) = \sum_z |h_X(z) - h_Y(z)|, \tag{5}$$

or Intersection over Union (IoU) distance:

$$d(X, Y) = \frac{\sum_z \min(h_X(z), h_Y(z))}{\sum_z \max(h_X(z), h_Y(z))}. \tag{6}$$

Here is the formal definition of model abstraction [3]:

Definition 2. Let $\{\eta_i\}_{i=0}^M$ be a discrete time stochastic process over an arbitrary state space S, with $M \in \mathbb{N}_+$ a time horizon, and let $r : S^M \to T$ be the associated reward function. An abstraction of (η, r) is a tuple $(\bar{S}, p, \bar{r}, \bar{\eta} = \{\bar{\eta}_i\}_{i=0}^M)$ where:

- $\bar{S}$ is the abstract state space;
- $p : S \to \bar{S}$ is the abstraction function;
- $\bar{r} : \bar{S}^M \to T$ is the abstract reward;
- $\bar{\eta} = \{\bar{\eta}_i\}_{i=0}^M$ is the abstract discrete time stochastic process over $\bar{S}$.

Let $\epsilon > 0$. $\bar{\eta}$ is said to be ϵ-close to η with respect to d if, for almost any $s_0 \in S$,

$$d(r(\tilde{\eta}_{[0,M]}), \bar{r}(\bar{\eta}_{[0,M]})) < \epsilon \quad \text{conditioned on } \eta_0 = s_0,\ \bar{\eta}_0 = p(s_0) \tag{7}$$

The simplest choice for the abstraction function could be an identity mapping. Alternatively, one can follow [21] to identify a subset of species having the most influence on the reward function. Inequality (7) is typically experimentally verified simulating a sufficiently high number of trajectories from both the original system $\tilde{\eta}$ and the abstraction $\bar{\eta}$ starting from a common initial setting. As there is no way to ensure that the inequality holds for almost any s_0 in S, we evaluate it for many different initial settings that the model did not see during training. Evaluation examples are presented in supplementary material.

Example 2 *(Running example cont'd).* The abstract model for our example shown in (1) as follows:

- The abstract state space $\bar{S}$ is $\mathbb{R}^3_{\geq 0}$, i.e. the continuous approximation of $S = \mathbb{N}^3$;
- The abstraction function p is the identity function that maps each point of S into its continuous embedding in $\bar{S}$;
- The reward function r is the projection on the protein P_1;
- The discrete time stochastic process $\bar{\eta} = \{\bar{\eta}\}_{i=0}^M$ is a DTMC with transition kernel represented by an MDN trained on simulation data.

3.2. Dataset generation

We build our datasets as sets of pairs $\mathcal{D} := \{(x, y)\}$ where each y is a sample from the distribution $\mathbb{P}(\eta_{t_0+\Delta t} \mid \eta_{t_0} = x)$, i.e. each pair corresponds to a single transition in discrete-time trajectories $\tilde{\eta}$. For this, we simulate trajectories starting from random initial settings from t_0 to $t_0 + M\Delta t$, and take the consecutive states $(\eta_{t_0+i\Delta t}, \eta_{t_0+(i+1)\Delta t}),\ i \in \{0, ..., M-1\}$ as (x, y) pairs.

Example 3 *(Running example: dataset).* For the example CRN network shown in Eq. (1), we simulate 100 trajectories for each of 100 random initial settings. We run simulations up to 200 time units, and fix the time step Δt to 20 time units for both training and evaluation (histogram) dataset. Therefore the time horizon M for evaluation is 10 steps.

3.3. Model training

Let $\mathcal{M}$ be a parameterized family of mixture distributions and g_θ be an MDN, where θ are network weights. Then, for every input vector x, $g_\theta(x) \in \mathcal{M}$. During training, weights θ are optimized to maximize the likelihood of samples y w.r.t. the parameterized distribution $g_\theta(x)$, so that the kernel K_d is approximated by K:

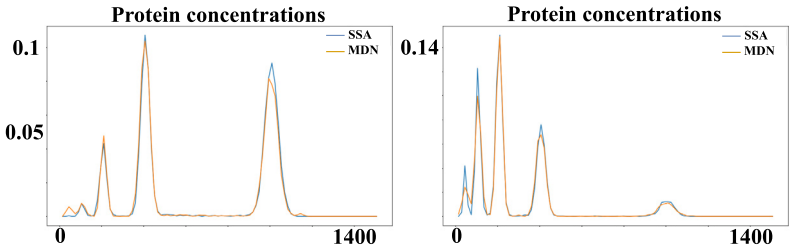


Fig. 5. Ex. (1): Sample histograms of protein P_1 concentration after 1 time step (left) and 9 time steps (right). (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

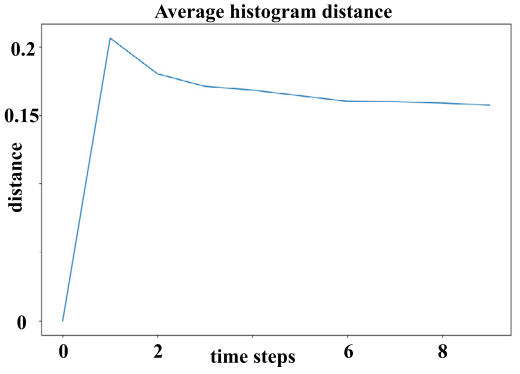


Fig. 6. Ex. (1): Mean histogram distance (IoU) of protein P_1 concentration for different time-lags.

$$K_d(s \mid s_0) = \mathbb{P}(\eta_{\Delta t} = s \mid \eta_0 = s_0) \approx \mathbb{P}(g_\theta(s_0) \in B_s) := K(B_s \mid s_0) \tag{8}$$

where $B_s := \{x \in \tilde{X} \mid \|x - s\| < \frac{1}{2}\}$ is the infinity norm ball with radius $\frac{1}{2}$ centered in s, needed to properly compare approximating continuous distribution with the original discrete distribution. Though model training is a relatively time-consuming task, once we have a trained model, sampling states are extremely fast, especially with the use of highly parallelized GPU computations.

3.4. Abstract model simulation and evaluation

With a trained MDN g_θ, for an initial state η_{t_0}, the distribution of the system state at the next time instant $t_0 + \Delta t$ is given by $g_\theta(\eta_{t_0})$, and the values of the next state $\bar{\eta}_1$ can be sampled from this distribution. These values then can be used to produce the next state, and so on:

$$\bar{\eta}_{i+1} \sim g_\theta(\bar{\eta}_i)$$

Every iteration in this procedure has a fixed computational cost, and therefore choosing Δt equal to the timescale of interest, we can simulate arbitrarily long trajectories at the needed level of time resolution, without wasting computational resources.

To evaluate the abstract model, we chose a number of random initial settings and for every setting we simulate (sufficiently many) trajectories from both the original and the abstract model up to the time horizon $M\Delta t$, evaluating the distance (7). Note that we train the MDN to approximate the kernel for a given Δt, i.e. it approximates model dynamics step-wise. However, we can evaluate the abstract model using the reward function on a time span longer than Δt. As noted in [3], the idea is that a good local approximation of the kernel should result in a good global approximation on longer time scales.

Example 4 *(Running example: evaluation).* For the histogram dataset, we simulate 10000 trajectories with the stochastic simulation Gillespie algorithm (SSA) up to time 200 for 25 random initial settings, and extract state values of the discrete-time process $\tilde{\eta}_{[0,M]}$ with Δt fixed to 20 time units.

With the trained MDN, we simulate 10000 traces starting from the same initial settings for 10 consecutive steps and therefore obtain the values $\bar{\eta}_{[0,M]}$ (Fig. 5).

Finally, we can evaluate the inequality (7), and estimate the average histogram distance. For every time-step i in range from 1 to $M = 10$, we average the distance $d(r(\tilde{\eta}_i), \bar{r}(\bar{\eta}_i))$ over 25 initial settings. This displays the performance of the

abstract model in predicting for many time steps in future, see Fig. 6. Note that, to draw the next state $\bar{\eta}_{i+1}$, the model uses its own prediction from the previous step.

4. Automated architecture search

The performance of machine learning algorithms depends heavily on the latent representation, i.e. a set of features. In contrast to simple machine learning algorithms, neural networks learn not only a mapping from representation to output but also the representation itself. As mentioned above, neural networks form the desired complex function from many small transformations represented by different layers. Each layer produces a new representation of the input features, which then can be used as an input to the following layers. The final representation, therefore, depends on the layer types used across the network, as well as on the graph describing connections between these layers.

Usually, neural networks are manually engineered by the experts via the trial-and-error procedure, which is a very time-consuming and error-prone process. Complex tasks require models of large size, which makes model design even more challenging.

Convolutional neural networks are a good example of a gain that comes from introducing incremental improvements in the network architecture. Step by step, in a series of publications, better design patterns were developed, improving the quality of models and reducing the computational demands. Even though a new model outperforms previous approaches, one could never argue that it is optimal. This raises an interest in the Neural Architecture Search (NAS) algorithms that help to find the near-optimal model configuration given a task.

One of the first successes in this field was achieved in [28] where reinforcement learning was applied to discover novel architectures that outperformed human-invented models on a set of tasks such as image classification, object detection, and semantic segmentation. It is worth to mention that it took 2000 GPU days of training to achieve this result. Later publications [4,5] introduced a gradient-based approach which significantly reduced required computational powers and allowed to achieve compatible results within one to two days using a single GPU.

In this work, we propose the algorithm inspired by [4,5] for the automated architecture design of MDN. Given a dataset and only a few hyper-parameters, it learns the architecture that best fits the data.

4.1. Our framework for automated neural network search

Broadly speaking, all NAS methods vary within three main aspects: *search space*, *search policy*, and *evaluation policy*.

Search space defines which architectures can be represented in principle. Therefore, to define a search space we fix a set of possible operation/layer candidates, a set of rules to connect them, and the architecture size (number of connections/layers).

Search policy describes a strategy of exploring the search space, e.g. random search, Bayesian optimization, evolutionary methods, reinforcement learning (RL), or gradient-based methods.

Evaluation policy includes the set of metrics of interest, such as accuracy on test data, number of parameters, latency, etc.

4.1.1. Search space

Similarly to DARTS architecture search method, proposed in [4], we consider a network that consists of several computational blocks or cells. A cell is a directed acyclic graph consisting of an ordered sequence of C_s nodes. Each node $x^{(i)}$ is a hidden state (latent representation) and each directed edge (i, j) is associated with some operation $o^{(i,j)}$ that transforms $x^{(i)}$.

Each cell has two input nodes and a single output node. The input nodes are the outputs of the previous two cells (or the model inputs if there are no previous cells). The output node is obtained by applying an aggregating operation (e.g. *sum* or *mean*) on the intermediate nodes (see Fig. 9). Each intermediate node is computed based on all the predecessors:

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{9}$$

A special zero operation is also included to indicate a lack of connection between two nodes.

To allow expanding the feature space within a network, we define two kinds of cells: *normal cell* preserving the number of neurons (features) received at inputs, and *expanding cell* that produces d times more activations, where d is an *expansion multiplier* parameter, see Fig. 8 for illustration. To serve this purpose, additional expanding operations (e.g. Dense layer) are applied to the inputs of a cell to produce the first two (input) nodes of the desired size. The very first cell is usually an expanding cell, and the rest are normal.

Therefore, our model is defined by the number of cells C_n, cell size C_s, expansion multiplier d, and the set of operations on the edges. We consider C_n, C_s and d to be hyper-parameters defining the model backbone, and fix the set of operation candidates. The task of architecture search thus reduces to learning the operations $o^{(i,j)}$ on the edges of each cell.

The set of all operation candidates in our setting includes *Dense layer* (with activation or not), *Element-wise layer*, *Gated Linear Unit*, *ReLU activation,* Swish activation, Identity *(skip-connection),* Zero-operation *(no connection)* (see Fig. 7). Users can

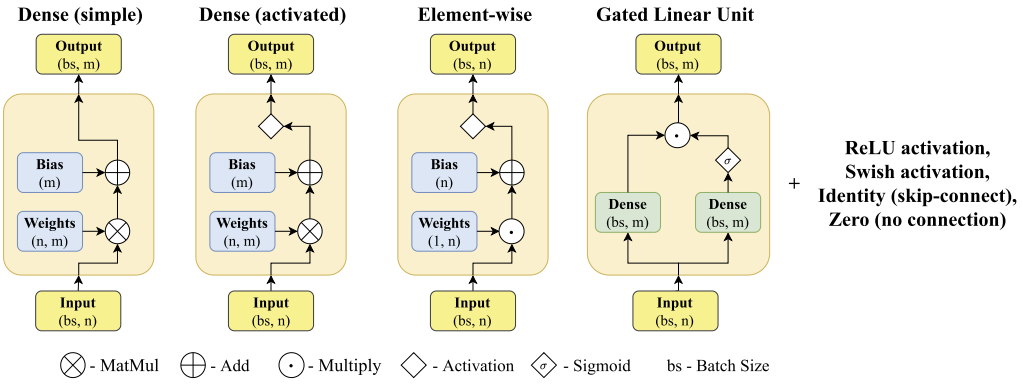


Fig. 7. Operation candidates.

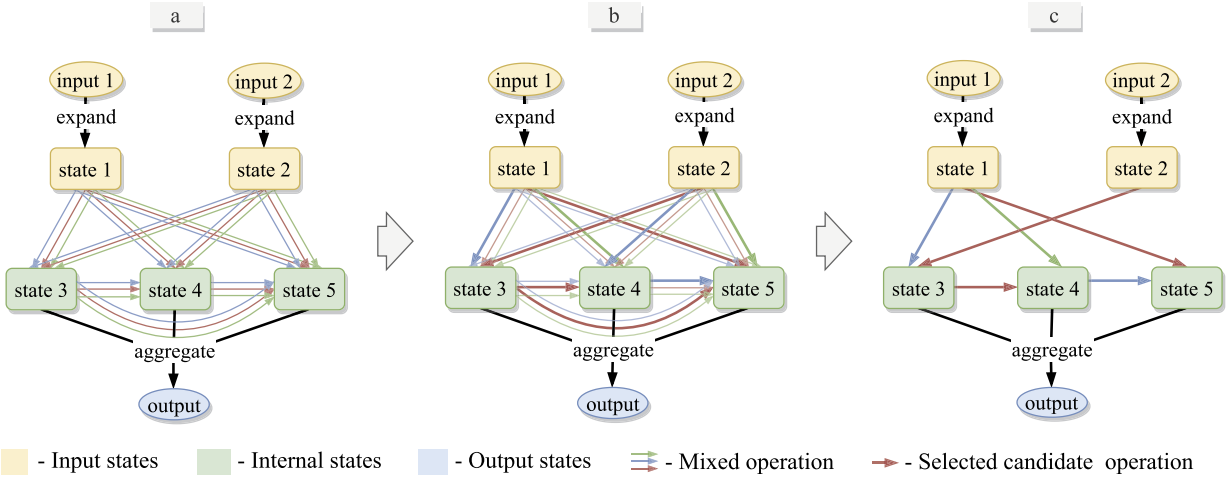


Fig. 8. Learning a computational cell. *a*): Operations connecting internal states are unknown and set to a mixture of candidate operations (coloured edges). Every state is connected to all its predecessors. *b*): During training, the weights for candidates are adjusted to prioritize the most important operations. *c*): Operations with the highest weights are selected for every edge. Further, only two edges with the highest scores are selected to be inputs for each state.

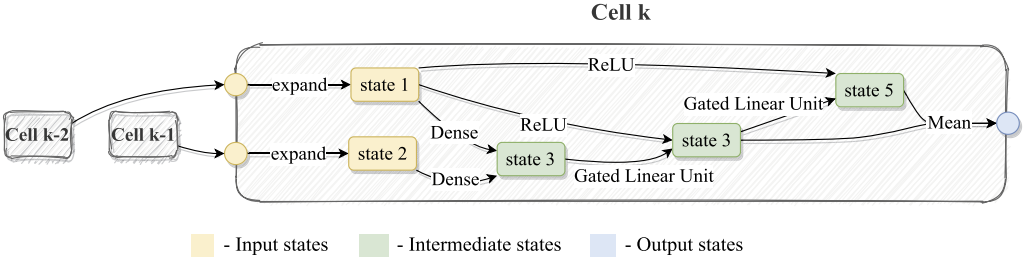


Fig. 9. Sample cell structure.

easily select a subset of these operations black-listing out others, as well as implement new ones. For example, we can exclude activation functions and use non-activated (Dense) layers and vice-versa.

Unlike the original works on automated architecture search ([4], [5]) targeting convolution architectures for two-dimensional image data, the choice of available candidates operating on one-dimensional data such as ours is only a dense layer architecture. For this reason, we have created our own Element-wise layer, and we added GatedLinearUnit layer which is widely used in attention-like mechanisms. Some NAS algorithms do not include activation functions to candidate operations and use activated (and sometimes non-activated) layers instead. To achieve more search flexibility, we include two activation functions: ReLU (most common, simple yet powerful) and Swish (reported to work better than ReLU [29]). Other (Zero and Identity) were inspired from the original DARTS algorithm.

4.1.2. Search strategy

The discrete search space we constructed leads to a challenging combinatorial optimisation problem, especially if we search for a model that is deep enough. As a neural network performs a chain of operations adjusted to each other, replacing even a single one requires a complete re-training. Therefore, each configuration in exponentially large search space should be trained separately. Gradient-based methods tackle this issue by introducing a continuous relaxation for the search space so that we can leverage gradients for effective optimization.

Let $\mathcal{O} = \{o_1, \ldots, o_N\}$ be the set of N candidate operations (e.g. dense, identity, zero, etc.). To represent any architecture in the search space, we build an over-parameterized network, where each unknown edge is set to be a mixed operation $m_{\mathcal{O}}$ with N parallel paths.

First, we define weights for the edges as a softmax over N real-valued architecture parameters α_i (note that outputs of softmax operation are positive and sum up to one, therefore we can treat weights p_i as probabilities):

$$p_i = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)}. \tag{10}$$

For each $m_{\mathcal{O}}$, only one operation (path) is sampled according to the probabilities p_i to produce the output. Path binarization process defined in [5] is described by:

$$m_{\mathcal{O}}^{Binary}(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1, \\ \ldots & \\ o_N(x) & \text{with probability } p_N \end{cases} \tag{11}$$

where g_i are binary gates:

$$g = \text{binarize}(p_1, ..., p_N) = \begin{cases} [1, 0, \ldots, 0] & \text{with probability } p_1, \\ \ldots & \\ [0, 0, \ldots, 1] & \text{with probability } p_N. \end{cases} \tag{12}$$

In this way, the task of learning the architecture reduces to learning a set of parameters α_i within every cell. The final network is obtained by replacing each mixed operation $m_{\mathcal{O}}$ by the operation o_{i^*} having the largest weight: $i^* = \arg\max_i \alpha_i$.

4.1.3. Optimization

After building an over-parameterised network, our goal is to jointly optimise the architecture parameters α and the weights w within all mixed operations. As discussed in [4], the best model generalisation is achieved by reformulating our objective as a bi-level optimisation problem. We minimise the validation loss $\mathcal{L}_{val}(w^*, \alpha^*)$ w.r.t. α^*, where the weights w^* are obtained by minimising the training loss $\mathcal{L}_{train}(w, \alpha^*)$. In other words, training is performed by alternating two separate stages for several epochs each, see Fig. 10.

When training weight parameters, we first freeze the architecture parameters α. Then for every example in the training dataset, we sample binary gates according to (12) and update the weight parameters of active paths via standard gradient descent.

When training architecture parameters, we freeze the weight parameters and update the architecture parameters on validation data. For every batch, binary gates are sampled w.r.t. updated architecture parameters.

However, due to the nature of the binarization procedure, the paths probabilities p_i are not directly involved in the computational graph, which means that we can not directly compute gradients

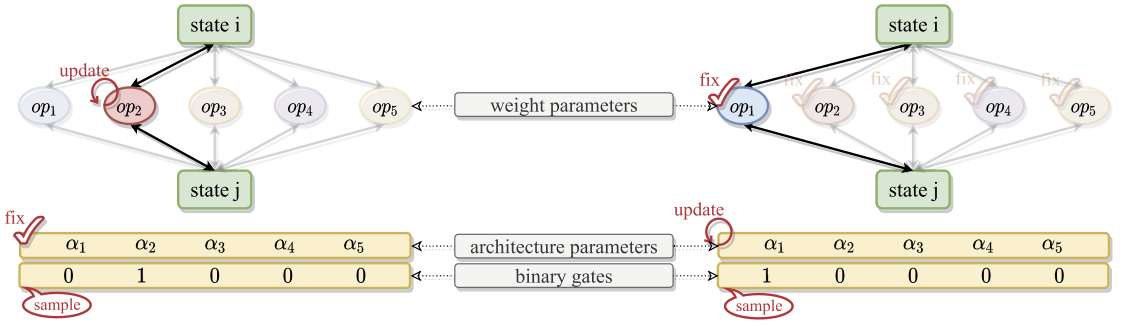
$$\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial p_j} \frac{\partial p_j}{\partial \alpha_i} \tag{13}$$

to update α_i using the gradient descent. As it was proposed in [5,30], we update the architecture parameters using the gradient w.r.t. its corresponding binary gate g_i, i.e. using $\partial L/\partial g_i$ instead of $\partial L/\partial p_i$ in (13).

Example 5 *(Running example: architecture search).* We search for the network consisting of 2 cells each of size 2, the first cell is an expanding cell. We train for 100 epochs in total: first 20 epochs only networks weights are updated, and the following 80 epochs training is performed as on Fig. 10. See Fig. 11 for the example of learned architecture.

5. Implementation

We implemented the proposed abstraction framework as python library StochNetV2 available openly at GitHub and described in detail in [6]. The library has implementations for all parts of the workflow:



(a) Update weight parameters (b) Update architecture parameters

Fig. 10. Optimization stages. *Left*: Weight parameters w are updated on training data while α parameters are frozen. *Right*: Architecture parameters α are updated on validation data while w parameters are frozen.

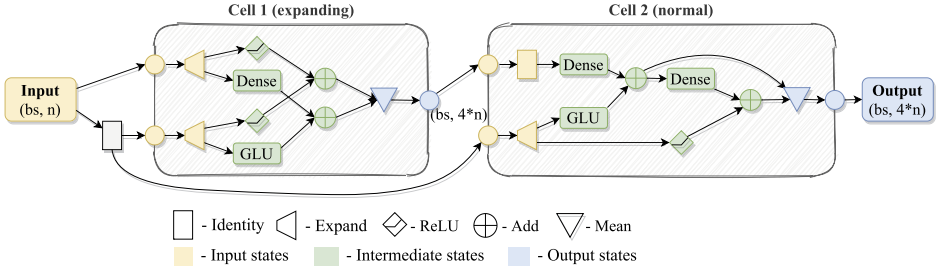


Fig. 11. Ex. (1): Learned network structure.

- defining a custom CRN model or importing SBML/Kappa file,
- producing CRN trajectories and creating datasets for training and evaluation,
- training custom models and automated architecture search,
- model evaluation,
- generating traces with trained model,
- various visualisations (traces, histograms, mixture parameters, architecture design, model benchmarking etc.).

To simulate traces more effectively, we provide scripts that run many simulations in parallel using multi-threading routines.

We use *luigi* [31] package as a workflow manager, which allows creating complex pipelines and managing complex dependencies. Neural networks, random variables, and optimisation algorithms are implemented in *TensorFlow* [32] and deep learning framework.

5.1. CRN models and simulation data

CRN models are handled by *gillespy* python library [33], which allows to define a custom class for model or import a model in SBML format. Note that not all models can be imported correctly, due to high variability of the SBML format. In those cases one can use a custom class with some pre-processing of the imported model.

Simulated trajectories are split into training examples $(x, y) := (\tilde{\eta}_i, \tilde{\eta}_{i+1})$. As neural networks show better convergence when the training data is standardised so that it varies in a reasonable range (e.g. [-1, 1] or [0, 1]) or has zero mean and variance, we apply a preprocess step to the input data such that it is scaled to a desired range. Then it is split into training (80%) and validation (20%) parts. The training dataset is used to optimise network weights, and validation dataset is used to optimise architecture parameters.

To increase generalisation capabilities of the model, it is important to build the dataset that covers the most variability of the original process. Although having more training data is always beneficial for a model, it increases training time. Therefore, depending on the variation of trajectories starting from the same initial conditions, we might prefer to run a few simulations for many initial conditions or more simulations for fewer initial conditions. When generating the dataset for evaluation, to make histograms more consistent, we usually simulate much more trajectories (from 1000 to 10000) for several initial settings.

Table 1
Execution time required to complete each step of the abstraction pipeline for different models. The last two rows display the difference in simulation times between the Gillespie SSA algorithm and the MDN abstract model. All simulations here performed in similar conditions: for every model we simulate the same number of trajectories up to the same time horizon (typically 10-20 time steps Δt), using the same number of CPU cores.

Task	Time		
	EGFR	Gene	X16
Generate training data	820.8 s	999.5 s	198.7 s
Format dataset	1.32 s	0.72 s	0.66 s
Train model	213 min	25 min	49 min
Generate hist. data (SSA)	46.8 s	9825.4 s	1274.5 s
Generate hist. data (MDN)	41.3 s	19.4 s	37.7 s

5.2. Network structure and computational cells

In our experiments, we learn the network typically constructed from two to three computational cells each of size 2 to 4. The first cell is expanding with a multiplier in a range from 10 to 20, and other cells are normal cells. Having multiple cells is not necessary, so it may consist of only one (larger) cell.

A computational cell described in the previous sections may also be altered. First, the *expanding operations* in the beginning of a cell can be represented either by Dense layers or by identity operations with tiling. For instance, for a multiplier 2 it transforms a vector $(x_1, x_2, ..., x_n)$ into a vector $(x_1, x_1, x_2, x_2, ..., x_n, x_n)$. Second, we can vary the number of intermediate nodes of a cell being aggregated to produce the output (e.g. all intermediate nodes or the last one, two, etc.), as well as the aggregating operation itself (e.g. *sum* or *mean*). Smaller number of aggregated nodes may lead to smaller cells after removing redundant connections at the final stage of architecture selection.

5.3. Random variables and MDN training

Our implementation has various random variables from Gaussian family: *Normal Diagonal*, *Normal Triangular*, *Log-Normal Diagonal*. Different combinations of these variables can be selected as the components for the mixture distribution, as long as their samples have the same dimensionality. In our experiments, we usually use from 5 to 8 components of Normal Diagonal variables. Replacing 2-3 Normal Diagonal components with the same number of Normal Triangular components sometimes improves model quality, though slows down both training and simulation times.

When training the architecture search, we have three main stages:

- *heat-up stage*, when weight parameters are trained for 10-20 epochs without updating the architecture parameters,
- *architecture search stage*, when weight parameters and architecture parameters are updated as described in NAS section (see Fig. 10), for 50 to 100 epochs in total, each turn of updating weight/architecture parameters typically lasts 3 to 5 epochs,
- *fine-tuning stage*, when the final architecture is fine-tuned for 10 to 20 epochs.

6. Experimental results

In this Section, we report the performance of the proposed method over three case studies: one reaction network modelling a crosstalk between two signalling pathways, and two reaction networks modelling gene regulatory dynamics with multimodal phenotypes. See Table 3 for an overview of the number of species, number of reactions, and number of modes in the steady-state, as well as the chosen time unit, for each of the three case studies.

We report the execution time required to complete each step of the abstraction pipeline in Table 1, the execution times required to simulate trajectories in Table 2, and the histogram distances in Table 3.

For all models, we set the hyper-parameters to the following values: cell size $C_s = 2$, number of cells $C_n = 2$, expansion multiplier $d = 20$. The user can opt to use these default values for any given CRN. However, there is a flexibility to change them. For each given example, our choice of hyper-parameters – C_s and C_n in ranges from two to five, the expansion multiplier up to $d = 30$ – was guided by the following heuristic exploration. We first choose small model size, hence with small parameters C_s, C_n and d that determine the depth and width of the neural network architecture, respectively. Then, if the abstract model performs poorly on both test and training data, we increase model size. In case the performance is good on train data and unsatisfactory on test data, the model is too large (or data-set too small), so we decrease model size again.

Table 2
Execution time (seconds) required to simulate trajectories of 25 time steps starting from the same initial conditions for EGFR model ($\Delta t = 0.5$), Gene model ($\Delta t = 400$). All simulations performed on CPU using the same number of parallel threads. *(1)*: 100 initial settings, 100 trajectories per setting, *(2)*: 10 initial settings, 1000 trajectories per setting, *(3)*: 1000 initial settings, 10 trajectories per setting, *(4)*: 1000 initial settings, 1000 trajectories per setting.

	EGFR		Gene	
	SSA	MDN	SSA	MDN
10^4 traces (1)	83.6	8.9	1952.8	1.5
10^4 traces (2)	13.1	8.9	2745.9	1.5
10^4 traces (3)	796.7	8.9	2076.1	1.5
10^6 traces (4)	962.3	900.9	TO($>$ 8 h)	148.1

Table 3
Histogram distances between trajectories of SSA and MDN models. Distances are averaged over 10000 sampled trajectories for each of the 25 random initial settings. Self-distance for the SSA simulations is presented as a reference value, reflecting the error margin arising due to distributions being obtained by statistical sampling. For all models, we set the following hyperparameters: cell size $C_s = 2$, number of cells $C_n = 2$, expansion multiplier $d = 20$.

Time horizon	EGFR (#sp.,#r.)=(23, 25) $\Delta t = 0.5$		Gene (#sp.,#r.)=(4, 6) $\Delta t = 400$		X16 (#sp.,#r.)=(4, 6) $\Delta t = 20$	
	3 steps	10 steps	3 steps	10 steps	3 steps	10 steps
SSA (self)	0.043	0.118	0.152	0.168	0.129	0.137
MDN	0.088	0.286	0.197	0.206	0.174	0.153

6.1. EGFR

Epidermal growth-factor receptor (EGFR) reaction model of cellular signal transduction, with 25 reactions and 23 different molecular species (Figs. 12–15):

$$
\begin{aligned}
&R + EGF \underset{k_{1b}}{\overset{k_{1f}}{\rightleftharpoons}} Ra\\
&2Ra \underset{k_{2b}}{\overset{k_{2f}}{\rightleftharpoons}} R2\\
&R2 \underset{k_{3b}}{\overset{k_{3f}}{\rightleftharpoons}} RP\\
&RP \xrightarrow{v_4 * RP/(k_4+RP)} R2\\
&RP + PLCg \underset{k_{5b}}{\overset{k_{5f}}{\rightleftharpoons}} PRLCg\\
&RPLCg \underset{k_{6b}}{\overset{k_{6f}}{\rightleftharpoons}} RPLCgP\\
&RPLCgP \underset{k_{7b}}{\overset{k_{7f}}{\rightleftharpoons}} PLCgP + RP\\
&PLCgP \xrightarrow{v_8 * PLCgP/(k_8+PLCgP)} PLCg\\
&Grb + RP \underset{k_{9b}}{\overset{k_{9f}}{\rightleftharpoons}} RG\\
&RG + SOS \underset{k_{10b}}{\overset{k_{10f}}{\rightleftharpoons}} RGS\\
&RGS \underset{k_{11b}}{\overset{k_{11f}}{\rightleftharpoons}} GS + RP\\
&GS \underset{k_{12b}}{\overset{k_{12f}}{\rightleftharpoons}} Grb + SOS\\
&Shc + RP \underset{k_{13b}}{\overset{k_{13f}}{\rightleftharpoons}} RSh\\
&RSh \underset{k_{14b}}{\overset{k_{14f}}{\rightleftharpoons}} RShP
\end{aligned}
\tag{14}
$$

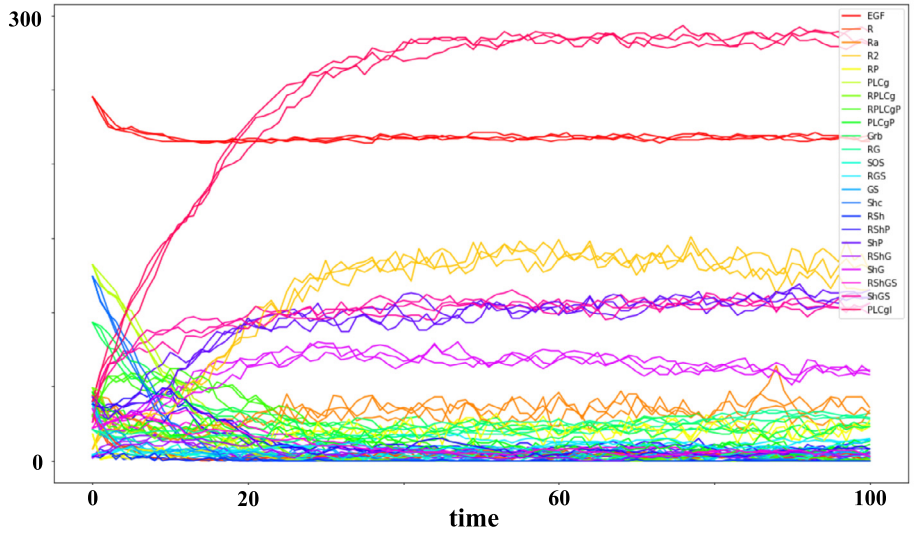


Fig. 12. Three sample trajectories of EGFR network starting from same initial state for 100 time steps, each of duration $\Delta t = 0.5$ time units.

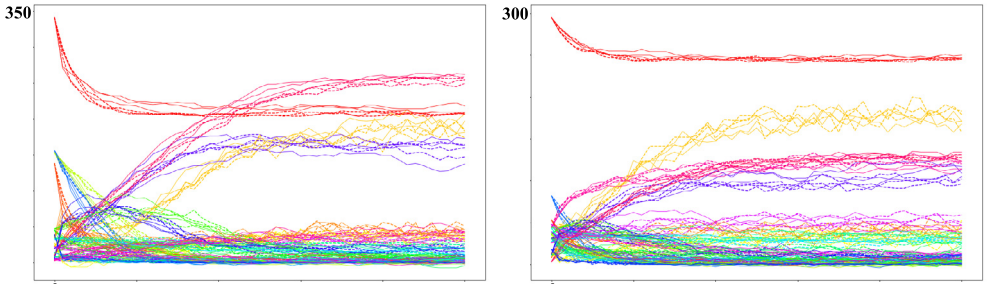


Fig. 13. EGFR: trajectories simulated by Gillespie algorithm (dashed lines) and MDN (full lines) for 50 consecutive time steps, $\Delta t = 0.5$.

$$RShP \underset{k_{15b}}{\overset{k_{15f}}{\rightleftharpoons}} RP + ShP$$

$$ShP \xrightarrow{v_{16}*ShP/(k_{16}+ShP)} Shc$$

$$RShP + Grb \underset{k_{17b}}{\overset{k_{17f}}{\rightleftharpoons}} RShG$$

$$RShG \underset{k_{18b}}{\overset{k_{18f}}{\rightleftharpoons}} ShG + RP$$

$$SOS + RShG \underset{k_{19b}}{\overset{k_{19f}}{\rightleftharpoons}} RShGS$$

$$RShGS \underset{k_{20b}}{\overset{k_{20f}}{\rightleftharpoons}} ShGS + RP$$

$$Grb + ShP \underset{k_{21b}}{\overset{k_{21f}}{\rightleftharpoons}} ShG$$

$$ShG + SOS \underset{k_{22b}}{\overset{k_{22f}}{\rightleftharpoons}} ShGS$$

$$ShGS \underset{k_{23b}}{\overset{k_{23f}}{\rightleftharpoons}} GS + ShP$$

$$RShP + GS \underset{k_{24b}}{\overset{k_{24f}}{\rightleftharpoons}} RShGS$$

$$PLCgP \underset{k_{25b}}{\overset{k_{25f}}{\rightleftharpoons}} PLCgI$$

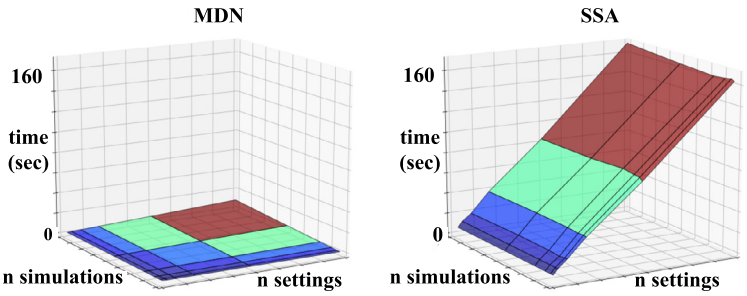


Fig. 14. Simulation times for EGFR model. Left: MDN model, right: Gillespie simulation. Times are measured to simulate traces of length 5 for different combinations of number of initial settings and number of traces for each setting.

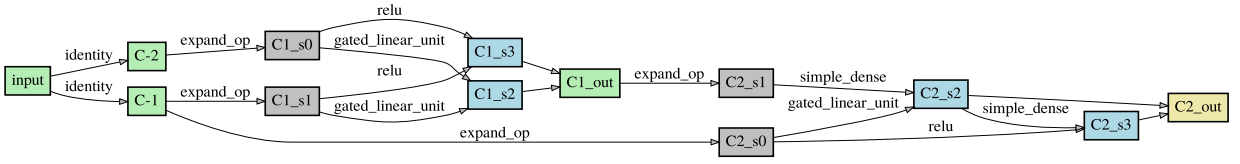


Fig. 15. EGFR: learned network structure. Gray rectangles represent input nodes, blue - intermediate nodes, green and yellow - output nodes (or cell inputs). Intermediate nodes compute the sum of values on incoming edges, output nodes - the average.

6.2. Gene

Self-regulated gene network [34,3]: a single gene G is transcribed to produce copies of a mRNA signal molecule M, which are in turn translated into copies of a protein P; P acts as a repressor with respect to G - it binds to a DNA-silencer region, inhibiting gene transcription (Figs. 16–20).

$$\begin{aligned} G &\xrightarrow{k_{prodM}} G + M \\ M &\xrightarrow{k_{prodP}} M + P \\ M &\xrightarrow{k_{degM}} \emptyset \\ P &\xrightarrow{k_{degP}} \emptyset \\ G + P &\xrightarrow{k_{bindP}} Gb \\ Gb &\xrightarrow{k_{unbindP}} G + P \end{aligned} \tag{15}$$

6.3. X16

The following fast-slow network ([2]) displays interesting dynamics with multimodal species distribution changing through time, as well as for different initial settings:

$$\begin{aligned} G_1 &\underset{\alpha_{12}}{\overset{\alpha_{11}}{\rightleftharpoons}} G_1 + P_1 \\ G_2 &\xrightarrow{\alpha_{21}} G_2 + P_1 \\ P_1 &\xrightarrow{\beta_1} \emptyset \\ G_1 &\underset{\epsilon\gamma_{21}}{\overset{\epsilon\gamma_{12}}{\rightleftharpoons}} G_2, \quad 0 < \epsilon \ll 1 \end{aligned} \tag{16}$$

Network (16) may be interpreted as describing a gene slowly switching between two expressions G_1 and G_2. When in state G_1, the gene produces and degrades protein P_1, while when in state G_1, it only produces P_1, but generally at a different rate than when it is in state G_1. Furthermore, P_1 may also spontaneously degrade (Figs. 21–25).

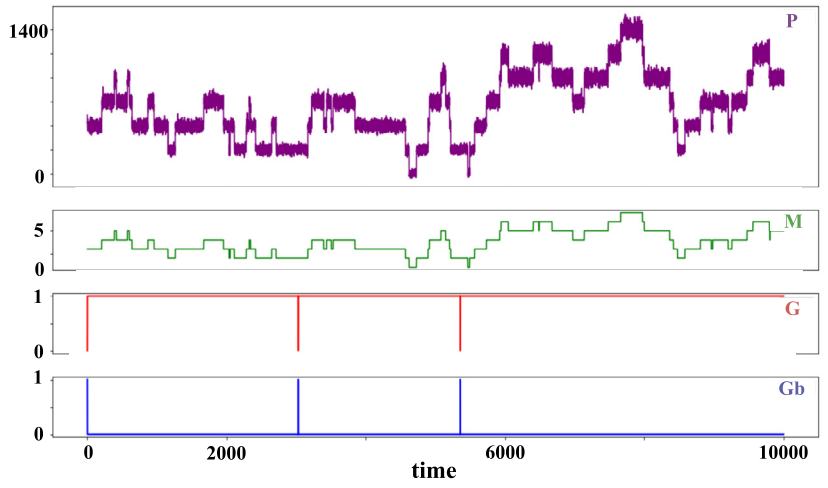


Fig. 16. Gene: sample trajectory of gene regulatory network Eq. (15).

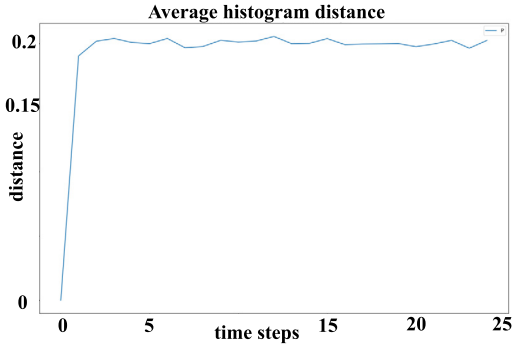


Fig. 17. Gene: mean histogram distance (intersection over union) averaged over 25 different initial settings.

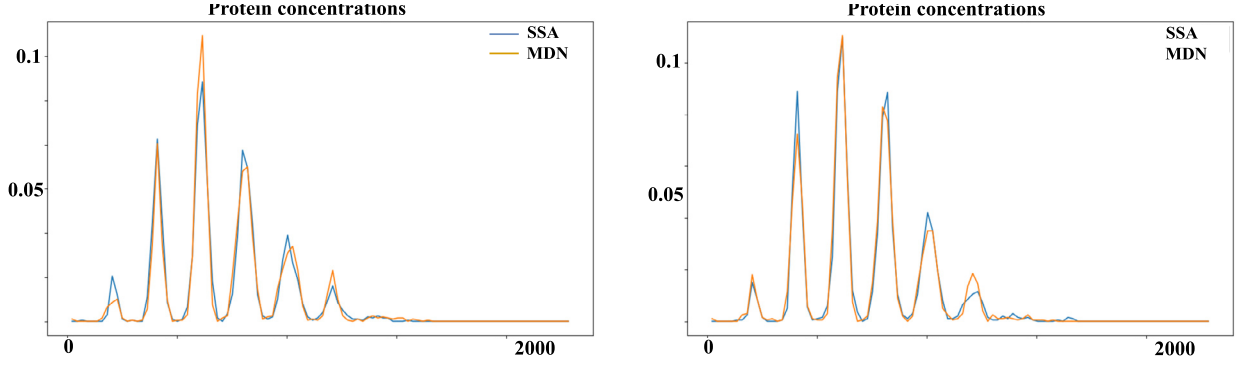


Fig. 18. Gene: histograms of protein P concentration after 5 time steps (left) and 25 time steps (right).

7. Multi-scale simulation with deep abstractions

Bridging from microscopic to macroscopic behaviours is a prominent concept across all of science, and especially biology, with major challenges being scalability, error propagation across scales and model validation [35]. We ask if deep abstractions can facilitate 'in silico' modelling of a population of reaction networks. For this purpose, we propose a simulation of multiple CRN instances over a (spatial) grid with communication via shared species diffusing across neighbouring grid nodes. We implement a concrete scenario of honeybee colony defense, in which a trade-off between worker loss and effi-

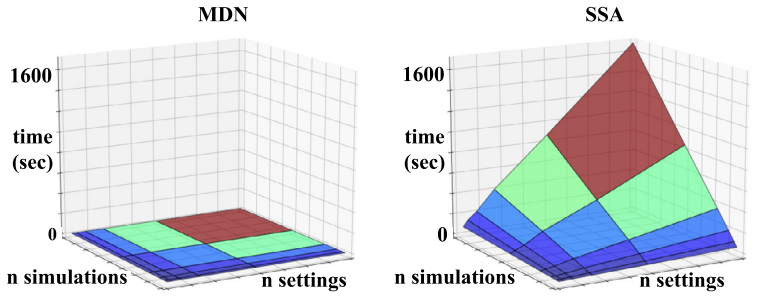


Fig. 19. Gene: simulation times for MDN model (left) and Gillespie simulation (right). Times are measured to simulate traces of length 5 for different combinations of number of initial settings and number of traces for each setting.

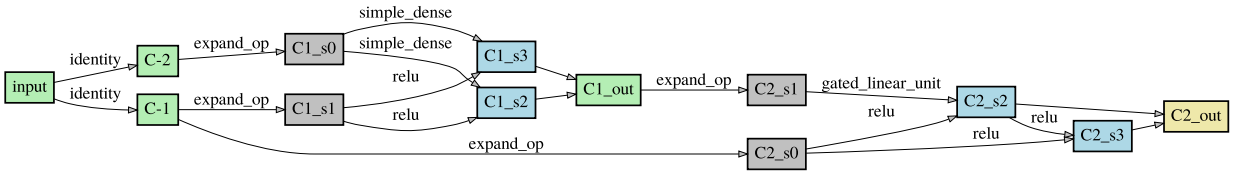


Fig. 20. Gene: learned network structure. Gray rectangles represent input nodes, blue - intermediate nodes, green and yellow - output nodes (or cell inputs). Intermediate nodes compute the sum of values on incoming edges, output nodes - the average.

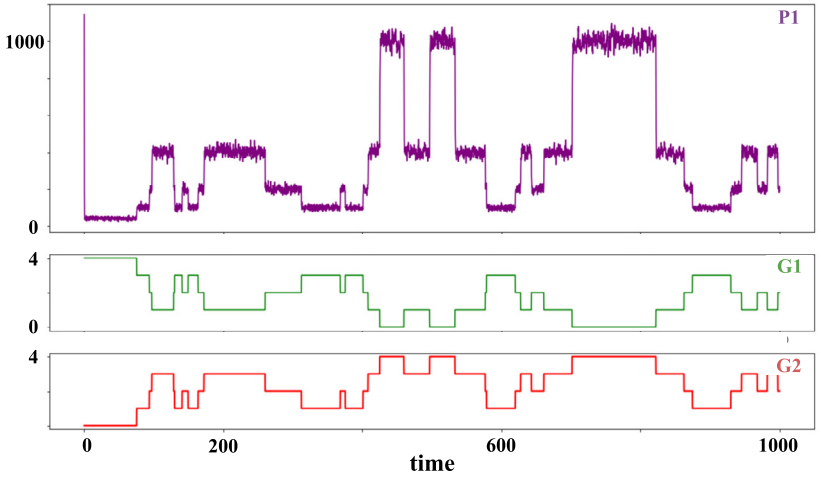


Fig. 21. Sample trajectory of X16 network.

cient defense is not yet fully understood [36].[1] We model the mechanism of each bee's defense as a reaction network, and we simulate the propagation of pheromone over a spatial grid: we first spread groups of bees across a grid, and set initial concentrations of pheromone in several places (or alternatively making some of them initially aggressive). Then, by alternating model steps and pheromone propagation steps, we obtain a discretised approximation of the colony behaviour. As the composition of CRN models amounts to simply merging reaction lists together, the mentioned scenario is implemented as a composition of copies of single-agent CRNs. Position within a spatial grid is encoded through identifiers unique to grid position (e.g. species $P_{i,j}$ denotes the pheromone amount at position (i, j)). For further detail, we refer the reader to the implementation described in [6].

A 10x10 grid with initially 10 bees in each grid position, and several pheromone units initially seeded, simulated for 100 time units, show a dramatic speed-up in favour of the abstract model: while the original, concrete model required more than 10 h, the abstract model took less than 2 s.

Although the observed speed-up is encouraging, the quality of abstraction was unsatisfactory, especially when cells were executed over initial settings which were underrepresented in or distant from the ones seen in the training data set. The explanation behind this observation is that our model is never trained on 'empty cells' - during training, there is always

[1] Honeybees defend their colonies by stinging and releasing an alarm pheromone; They thus alert others to defend but sacrifice own lives.

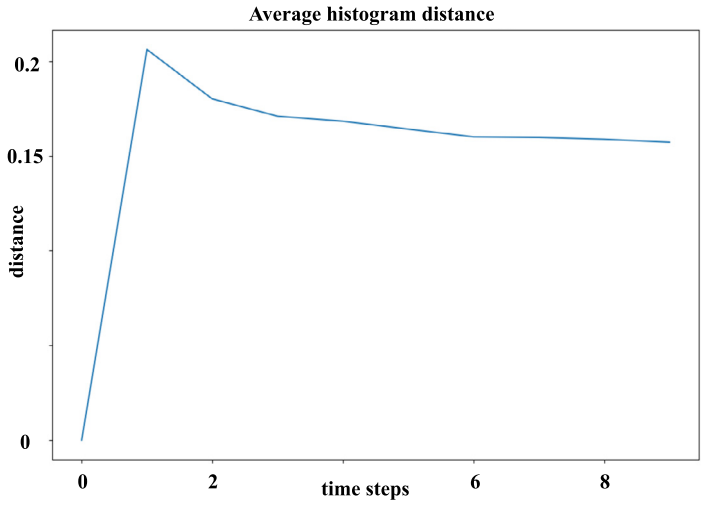


Fig. 22. X16: mean histogram distance (intersection over union) averaged over 25 different initial settings.

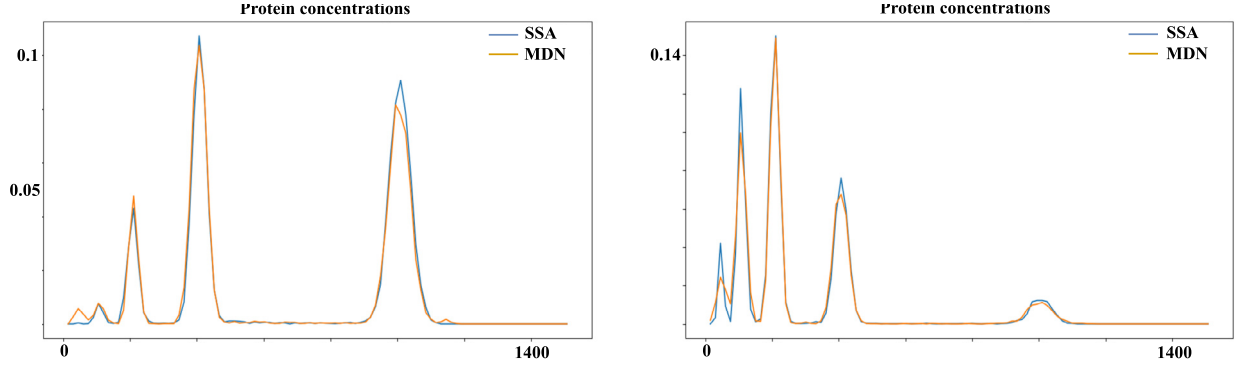


Fig. 23. X16: histograms of protein P_1 concentration after 1 time step (left) and 9 time steps (right).

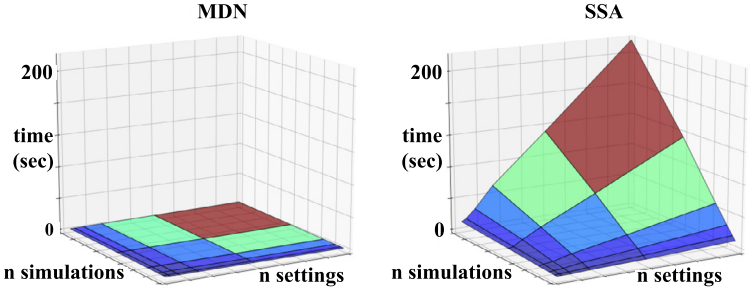


Fig. 24. X16: simulation times for MDN model (left) and Gillespie simulation (right). Times are measured to simulate traces of length 5 for different combinations of number of initial settings and number of traces for each setting.

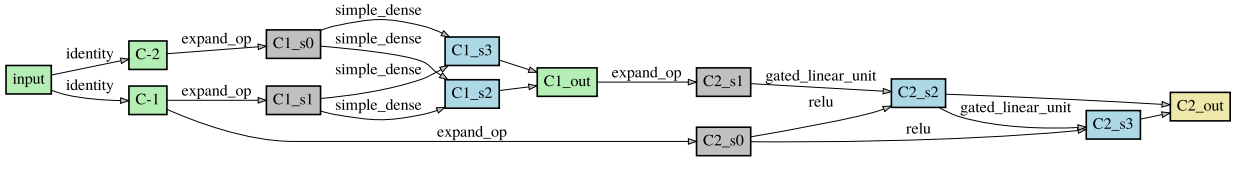


Fig. 25. X16: learned network structure. Gray rectangles represent input nodes, blue - intermediate nodes, green and yellow - output nodes (or cell inputs). Intermediate nodes compute the sum of values on incoming edges, output nodes - the average.

some initial reaction mixture. Consequently, the learnt MDN does not capture well degenerate distributions (e.g. with near-to-zero mean and zero deviation). Once an error occurs in a multi-scale simulation, it propagates further, rendering the overall abstraction not satisfactory. It is hence worth exploring how to improve the quality of abstraction, given its sensitivity to the amount of training data and the initial settings. Systematically examining the robustness of deep abstractions in multiscale setting - error propagation across scales and model validation - is beyond the scope of this manuscript and represents a compelling research topic for future works.

8. Conclusions and discussion

In this paper, we proposed how to automatise deep abstractions for stochastic CRNs, through learning the neural network architecture along with learning the transition kernel of the abstract process. Automated search of the architecture makes the method applicable directly to any given CRN, which is time-saving for deep learning experts and crucial for non-specialists. Contrary to the manual approach where the user has to create a neural network by hand, test it for his/her use-case, and adopt it accordingly, our method allows to find a solution with minimal efforts, within a reasonable amount of time. To this end, the main goal of our contribution is not to out-perform any possible man-made architecture used for deep abstractions, but to automatise the search for any given input CRN, and thus save time and effort of manual design and trials. The procedure we propose involves four hyper-parameters: three of them allowing to tune the model size, and one providing a set of operation candidates. Yet, the user can opt to set the four hyper-parameters to their default values, and run the abstraction fully automatically. Moreover, the set of operations on the edges (the forth hyper-parameter of our proposed search procedure), can be left as they are, if the modeller considers this set of operations to be general enough.

We implemented the method and demonstrated its performance on three representative CRNs, two of which exhibit multi-modal emergent phenotypes. Compared to the plain stochastic simulation, our method is significantly faster in almost all use-cases (see Table 2 and Table 1).

The proposed methodology, especially automated architecture search, enables fast simulation of computationally expensive Markov processes. As such, it opens up possibilities for efficiently simulating interactions between many individual entities, each described by a complex reaction network. We demonstrate this possibility over a scenario of multiple CRN instances communicating via shared species across a spatial grid, inspired by a pheromone-mediated communication in honeybee colonies [36]. Our implementation reveals limitations of the method, related to its sensitivity to the choice of initial reaction mixtures in the training set. In future works, we plan to relax this limitation and develop a strategy that is agnostic to the initial mixture of the CRN being modelled.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

TP's research is supported by the Ministry of Science, Research and the Arts of the state of Baden-Württemberg and DR's research was supported by Young Scholar Fund (YSF), project no. *P*83943018*F P*430_/18. Both authors were supported by the DFG Centre of Excellence 2117 'Centre for the Advanced Study of Collective Behaviour' (ID: 422037984) and Max Planck Institute of Animal Behaviour in Radolfzell. The authors would like to thank to Luca Bortolussi for inspiring discussions on the topic.

References

[1] D. Gillespie, Exact stochastic simulation of coupled chemical reactions, J. Phys. Chem. 81 (1977) 2340–2361.
[2] T. Plesa, R. Erban, H.G. Othmer, Noise-induced mixing and multimodality in reaction networks, Eur. J. Appl. Math. 30 (5) (2019) 887–911.
[3] L. Bortolussi, L. Palmieri, Deep abstractions of chemical reaction networks, in: M. Češka, D. Šafránek (Eds.), Computational Methods in Systems Biology, Springer International Publishing, Cham, 2018, pp. 21–38.
[4] H. Liu, K. Simonyan, Y. Yang, DARTS: differentiable architecture search, in: International Conference on Learning Representations, 2019, https://openreview.net/forum?id=S1eYHoC5FX.
[5] H. Cai, L. Zhu, S. Han, ProxylessNAS: direct neural architecture search on target task and hardware, CoRR, arXiv:1812.00332 [abs], 2018, arXiv:1812.00332, http://arxiv.org/abs/1812.00332.
[6] D. Repin, N.-H. Phung, T. Petrov, StochNetV2: a tool for automated deep abstractions for stochastic reaction networks, in: International Conference on Quantitative Evaluation of Systems, Springer, 2020, pp. 27–32.
[7] D. Anderson, T.G. Kurtz, Continuous-time Markov chain models for chemical reaction networks, Tech. Rep., University of Wisconsin - Madison, Jul. 2010.
[8] L. Cardelli, M. Kwiatkowska, L. Laurenti, Stochastic analysis of chemical reaction networks using linear noise approximation, Biosystems 149 (2016) 26–33.
[9] L. Cardelli, M. Tribastone, M. Tschaikowski, A. Vandin, Syntactic Markovian bisimulation for chemical reaction networks, in: Models, Algorithms, Logics and Tools, Springer, 2017, pp. 466–483.
[10] T. Petrov, H. Koeppl, Approximate reductions of rule-based models, in: European Control Conference (ECC), 2013, pp. 4172–4177.
[11] J. Feret, H. Koeppl, T. Petrov, Stochastic fragments: a framework for the exact reduction of the stochastic semantics of rule-based models, Int. J. Softw. Inform. (2013), in press.

[12] A. Ganguly, T. Petrov, H. Koeppl, Markov chain aggregation and its applications to combinatorial reaction networks, J. Math. Biol. (2013) 1–31.
[13] J. Feret, T. Henzinger, H. Koeppl, T. Petrov, Lumpability abstractions of rule-based systems, Theor. Comput. Sci. 431 (2012) 137–164.
[14] M. Tribastone, A. Vandin, Speeding up stochastic and deterministic simulation by aggregation: an advanced tutorial, in: Proceedings of the 2018 Winter Simulation Conference, IEEE Press, 2018, pp. 336–350.
[15] H. Conzelmann, J. Saez-Rodriguez, T. Sauter, B.N. Kholodenko, E.D. Gilles, A domain-oriented approach to the reduction of combinatorial complexity in signal transduction networks, BMC Bioinform. 7 (2006) 34.
[16] T.A. Henzinger, M. Mateescu, L. Mikeev, V. Wolf, Hybrid numerical solution of the chemical master equation, CoRR (2010).
[17] A. Beica, C. Guet, T. Petrov, Efficient reduction of kappa models by static inspection of the rule-set, in: Lecture Notes in Computer Science, 2015, pp. 173–191.
[18] J. Gunawardena, A linear framework for time-scale separation in nonlinear biochemical systems, PLoS ONE 7 (5) (2012) e36321.
[19] C.D. Pahlajani, P.J. Atzberger, M. Khammash, Stochastic reduction method for biological chemical kinetics using time-scale separation, J. Theor. Biol. 272 (1) (2011) 96–112.
[20] C.N. Davis, T.D. Hollingsworth, Q. Caudron, M.A. Irvine, The use of mixture density networks in the emulation of complex epidemiological individual-based models, PLoS Comput. Biol. 16 (3) (2020) 1–16, https://doi.org/10.1371/journal.pcbi.1006869.
[21] S.K. Palaniappan, F. Bertaux, M. Pichené, E. Fabre, G. Batt, B. Genest, Abstracting the dynamics of biological pathways using information theory: a case study of apoptosis pathway, Bioinformatics 33 (13) (2017) 1980–1986, https://doi.org/10.1093/bioinformatics/btx095, arXiv:http://oup.prod.sis.lan/bioinformatics/article-pdf/33/13/1980/25155844/btx095.pdf.
[22] M. Michaelides, J. Hillston, G. Sanguinetti, Statistical abstraction for multi-scale spatio-temporal systems, in: Lecture Notes in Computer Science, 2017, pp. 243–258.
[23] T.G. Kurtz, Limit theorems for sequences of jump Markov processes approximating ordinary differential processes, J. Appl. Probab. 8 (2) (1971) 344–356.
[24] D. Schnoerr, G. Sanguinetti, R. Grima, Approximation and inference methods for stochastic biochemical kinetics—a tutorial review, J. Phys. A, Math. Theor. 50 (9) (2017) 093001.
[25] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.
[26] C.M. Bishop, Mixture density networks, http://publications.aston.ac.uk/id/eprint/373/, 1994.
[27] Y. Cao, L. Petzold, Accuracy limitations and the measurement of errors in the stochastic simulation of chemically reacting systems, J. Comput. Phys. 212 (1) (2006) 6–24, https://doi.org/10.1016/j.jcp.2005.06.012, http://www.sciencedirect.com/science/article/pii/S0021999105003074.
[28] B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning, CoRR, arXiv:1611.01578 [abs], 2016, arXiv:1611.01578, http://arxiv.org/abs/1611.01578.
[29] P. Ramachandran, B. Zoph, Q.V. Le, Searching for activation functions, CoRR, arXiv:1710.05941 [abs], 2017, arXiv:1710.05941, http://arxiv.org/abs/1710.05941.
[30] M. Courbariaux, Y. Bengio, J. David, Binaryconnect: training deep neural networks with binary weights during propagations, CoRR, arXiv:1511.00363 [abs], 2015, arXiv:1511.00363, http://arxiv.org/abs/1511.00363.
[31] [link], https://github.com/spotify/luigi.
[32] [link], https://www.tensorflow.org/.
[33] [link], https://github.com/johnabel/gillespy.
[34] C. Bodei, L. Bortolussi, D. Chiarugi, M.L. Guerriero, A. Policriti, A. Romanel, On the impact of discreteness and abstractions on modelling noise in gene regulatory networks, Comput. Biol. Chem. 56 (2015) 98–108, https://doi.org/10.1016/j.compbiolchem.2015.04.004, http://www.sciencedirect.com/science/article/pii/S1476927115000547.
[35] A. Hoekstra, B. Chopard, P. Coveney, Multiscale modelling and simulation: a position paper, Philos. Trans. R. Soc. A, Math. Phys. Eng. Sci. 372 (2021) (2014) 20130377.
[36] M. Hajnal, M. Nouvian, D. Šafránek, T. Petrov, Data-informed parameter synthesis for population Markov chains, in: International Workshop on Hybrid Systems Biology, Springer, 2019, pp. 147–164.