Neural Approach to the Discovery Problem in Process Mining

Timofey Shunin(✉), Natalia Zubkova, and Sergey Shershakov

Laboratory of Process-Aware Information Systems (PAIS Lab),
Faculty of Computer Science,
National Research University Higher School of Economics, Moscow 101000, Russia
{tmshunin,nszubkova}@edu.hse.ru, sshershakov@hse.ru

Abstract. Process mining deals with various types of formal models. Some of them are used at intermediate stages of synthesis and analysis, whereas others are the desired goals themselves. Transition systems (TS) are widely used in both scenarios. Process discovery, which is a special case of the synthesis problem, tries to find patterns in event logs. In this paper, we propose a new approach to the discovery problem based on recurrent neural networks (RNN). Here, an event log serves as a training sample for a neural network; the algorithm extracts RNN's internal state as the desired TS that describes the behavior present in the log. Models derived by the approach contain all behaviors from the event log (i.e. are perfectly fit) and vary in simplicity and precision, the key model quality metrics. One of the main advantages of the neural method is the natural ability to detect and merge common behavioral parts that are scattered across the log. The paper studies the proposed method, its properties and possible cases where the application of this approach is sensible as compared to other methods of TS synthesis.

Keywords: Process mining · Transition systems · Quality metrics
Recurrent neural networks · Process models synthesis · FSA/FSM

1 Introduction

Neural networks are statistical computing models that are applied today to many practical tasks, e.g. image processing, machine translation and pattern recognition. In *supervised learning*, a neural network trains on a sample of already known objects, i.e. for each input we have a predefined correct answer. The main idea behind training a neural network is to tune such a configuration, so that the model's answers are as close to the correct ones as possible. As for *recurrent neural networks* (RNN), they not only learn on input objects, but also provide the *context* for each following prediction. This helps a neural network preserve

This work is supported by the Basic Research Program of the National Research University Higher School of Economics and funded by RFBR according to the Research project No. 18-37-00438 "mol_a".

W. M. P. van der Aalst et al. (Eds.): AIST 2018, LNCS 11179, pp. 261–273, 2018.
https://doi.org/10.1007/978-3-030-11027-7_25

the state in which the decision was made. In this paper, we discuss applying RNN methods to process mining problem of *process discovery*.

The task of process discovery consists in deriving a model so that it captures the behavior present in the input data. As it is similar to the task of *pattern recognition* [1], in this paper we focus on solving the process discovery task using a recurrent neural network approach. Regarding an event log as an input training sample, for each event in the log we can train our neural network to predict the next event. Moreover, each state of the neural network can be interpreted as a state in a *finite state machine* (FSM), or, more generally, a *transition system* (TS), that is implicitly present behind the configuration of the neural network. Our ultimate goal is to extract this transition system that represents a model of a process present in the input log. We assume that one of the advantages of the neural method is its ability to detect common behavioral parts that are scattered across the log.

While transition systems themselves are appropriate process models, such models are commonly known as low-level. Thus, a derived TS that includes all behaviors present in the log, e.g. a prefix tree, is comparable in size to the log itself, which could lead to a model that is too complex to analyze. An *adequate* model should provide a certain level of abstraction allowing to study both simple and complicated processes. Another issue is that the nature of these models does not allow them to explicitly represent concurrency. This problem can be eliminated by using appropriate models allowing concurrency such as *BPMN* or *UML activity diagrams*, which are generally modeled based on Petri nets. There exists an approach to synthesize a Petri net from a TS, based on the theory of regions [2–4]. However, direct application of the region-based algorithm to intermediate models is exponentially dependent on the size of a TS, and, correspondingly, on the size of the input log [5]. In this paper we focus on synthesizing transition systems themselves and do not regard the following step of deriving models with concurrency.

Previously, a few works that cover methods of inferring transition systems using neural networks were published [6,7]. At that time, technologies based on neural networks were just starting to appear. Moreover, the computational powers of computers were significantly lower compared to the power of modern ones. Since the field of practical machine learning has experienced a rise in the recent decade, which resulted in rationality and great ease of implementing complex algorithms in practice, we believe that it is now that neural networks could solve practical tasks of process mining and be used to their full potential.

One of the main goals of our study is to analyze the efficiency of the proposed approach, i.e. see how different quality metrics depend on input data and user-specified parameters. There exists several approaches to evaluate the quality of models inferred by different process discovery techniques. In process mining, a number of *quality metrics* [8–11] are used to quantify an extent to which a model describes an input log. We follow the same way, prove that our approach produces *perfectly-fit* models, and compute metrics for the inferred models to assess their quality.

The main contributions are as follows: (1) describing and implementing a method of building transition systems using recurrent neural networks; (2) comparing the quality metrics of models constructed by the described method to quality metrics of models inferred using the *frequency-based reduction algorithm* (FQR) [10] and *prefix trees* [3]; (3) concluding properties of event logs where RNN method shows better results as compared to the FQR method.

The rest of the paper is organized as follows. Section 2 gives a brief overview of related works in the context of process mining and neural networks. In Sect. 3 we describe some relevant concepts needed for the explanation of the RNN approach which is itself described in Sect. 4. Experiments, metrics calculations and results are discussed in Sect. 5. Finally, Sect. 6 concludes the paper and gives some directions for future work.

2 Related Work

The idea of using neural networks to infer FSMs is not completely new. Das and Mozer in [6] described a clustering architecture for finite state machine induction. Their idea was to teach neural network on both positive and negative examples of strings, composed from a two-symbol alphabet, so the network could predict if the input string is a valid example or not. They clustered the inner states of a neural network to extract states for an FSM. This approach was evolved by Cook and Wolf in 1998 [7] in the context of process discovery. They were aiming at inferring grammars for the samples of valid strings. Their extension allowed more complex languages which were not limited by two symbols.

Buijs et al. in [8] formulate key quality metrics for assessing Petri net models in process mining, namely *replay fitness*, *simplicity*, *precision* and *generalization*. While their variant of calculating metrics is widely used by the community, it is not the only possible one. Other calculations of simplicity are rather similar to the one proposed in [8], whereas for precision there is no common baseline. In an attempt to bring such a baseline, recently Tax et al. [11] have proposed five axioms a good precision measure should satisfy in order to describe any log or any model consistently.

In [10], authors propose a new approach that allows to infer perfectly fit TSs whose simplicity and precision could be adjusted by configuring the algorithm parameters. The paper describes a method for inferring TS based on the frequency-based reduction algorithm. A new approach for measuring precision of perfectly fit TSs, based on models simulation, is also present there. In this paper, we use that approach to calculate simplicity and precision for comparing models.

3 Preliminaries

3.1 Process Mining Related Concepts

Let A be a set of activities generated by a process. An *event* e is an occurence of an activity in a log. A *token* is an element of a log that is passed to a neural

network. We assume an event and a token to be synonyms which are used in different contexts. A set Λ of all unique events of the log is called an input alphabet. By $|\Lambda|$ we denote the cardinality of the input alphabet.

Definition 1 (Trace, Event log). *A* trace σ *is a finite sequence of events* $\sigma = \langle e_1, ..., e_n \rangle \in A^+$*, where* A^+ *is the set of all non-empty finite sequences over* A*. An event log* L *is a multiset of traces.*

$L = [\langle a, b, c \rangle, \langle a, b, d \rangle]$ is an example of an event log of two traces.

Definition 2 (Transition System). *A* transition system *TS is a tuple* $TS = (S, E, T, s_0, AS)$*, where* S *is finite space of states,* E *is a finite set of labels,* $T \subseteq S \times E \times S$ *is a set of transitions,* $s_0 \in S$ *is an initial state, and* $AS \subseteq S$ *is a set of final (accepting) states. By* $s_i \in S$ *we denote a* i^{th} *state in a TS.*

3.2 Neural Networks Related Concepts

A *neural network* [12] is a computing model that usually consists of three main parts: an *input layer*, a *hidden layer* and an *output layer*. Each layer is a set of neurons, and each neuron is a computational unit that applies certain *activation functions* to data flowing through it. The layers are linked by weighed connections that transmit signals from a neuron in one layer to one or more neurons in the next layer. The resulting transformation of data passed to the input layer should meet the conditions implied by a problem solved by the network. Such conditions in a neural network are represented by a *loss function*. In order to get this transformation, the weights of the tying connections are configured. The process of configuring weights is called a *learning, or training process*. It consists in optimizing the loss function, or error, using *back propagation* [12]. On each training iteration, the current error is propagated to the previous layer, where it is used to modify the weights in such a way that the error is minimized. In [1], Bishop provides a more complete insight on the meaning and purpose of computations present in a neural network.

The computations could also be influenced by some parameters of a neural network that could not be optimized during the learning process. Such parameters are called *hyperparameters* and they should be defined before the learning process begins. The number of neurons in the hidden layer is an example of such a parameter.

The approach described in this paper is based on *recurrent neural networks* (RNN). Such networks have one or more recurrent connections linking together output and input layers, which allows a network to use its previous computations for the following predictions. The training of an RNN is usually similar to the training of an ordinary neural network and often involves *stochastic gradient descent (SGD)* [12].

4 Description of the Neural Approach

For clarification of the approach we consider the event log $L = [\langle a, b, c, d, e \rangle, \langle a, b, d \rangle]$ as a running example.

The inner computations in a neural network require preprocessing our traces as follows:

1. Adding two new reserved tokens "$" and "#" to each trace indicating the beginning and the end of the trace respectively.
2. Padding all traces with "#" symbol so all traces have the same length.

This way, L transforms into $\tilde{L} = [\langle \$, a, b, c, d, e, \# \rangle, \langle \$, a, b, d, \#, \#, \# \rangle]$, where all traces have the same length of 7. We also encode present tokens with integers from 0 to $|\Lambda| + 1$.

In order to build the desired TS we create a neural network that is capable of generating traces from the input log. On each step the network predicts the probabilities of next possible tokens and chooses the next token from the most probable ones. Interpreting the problem of predicting tokens as a task of *classification* allows us to use specific methods and architectures [13] that have proven their efficiency [14]. For each *token* (event) α_i we want to determine a *class of tokens* of the following token α_{i+1}, i.e. an activity represented by it. Moreover, we should take into consideration the history of the trace; otherwise the prediction would only depend on the previous token, which would lead to learning a graph of relations between individual events. That is why a recurrent connection is present, composing a more complex input vector. If the network is capable of generating plausible traces, then the inner state, from which the prediction is made, is credible and represents both the history of the trace and the current token, thus could be used as a state in the TS. To solve this task, we developed a neural network architecture presented in Fig. 1.

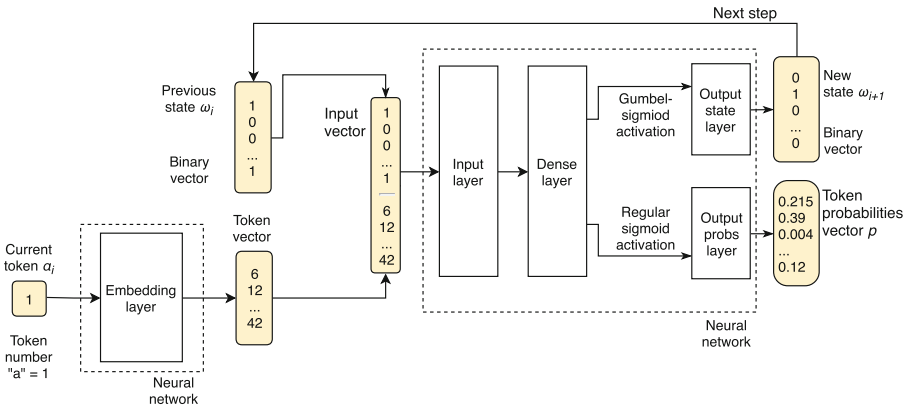


Fig. 1. Neural network architecture with example values

In the *embedding layer*, an integer, that encodes the next token from the trace is transformed into a *token vector*. The *input vector* for the neural network is a concatenation of two vectors: a *binary vector*, which represents a *previous state*

Table 1. Token probabilities for the first trace of $\tilde{L}$

$	0.215
a	0.39
...	...
#	0.12

Table 2. Cross entropy matrix for $\tilde{L}$

$	a	b	c	d	e	#	
2.57	1.45	2.65	1.77	3.72	4.62	0	16.78
$	a	b	d	#	#	#	
2.57	1.45	2.65	2.51	0	0	0	9.18
Average							12.98

of the neural network, and a *token vector*. Then the *input vector* is sent to the *dense* (*hidden*) *layer*, which applies a linear transformation to the vector. After that, different activation functions are applied to the computed vector in order to extract two different entities from the output layer. The first one is a new *discrete binary vector*, interpreted as the *new state of the RNN*, and returned as the new parameter for the next iteration. To compute it we use the *Gumbel Sigmoid* as it has proven its efficiency in approximating discrete values [15]. The other is a vector of token probabilities p (see Table 1). Here, we apply regular sigmoid function as its output appertains to the segment of $[0; 1]$. We use the vector of token probabilities to calculate our loss function as follows.

We build a $m \times r$ loss (cross entropy) matrix, where m is the number of traces in the input log and r is the maximum length of a preprocessed trace from the input log. Each row of the matrix relates to a trace in the log, each element relates to a token (event) in a trace. On every step, an element is calculated as cross entropy $H(\hat{y_i}, y_i)$, where $\hat{y_i}$ is the predicted probability of a token and y_i is the real one.

Since after preprocessing we have some excess padding symbols which we do not want to teach our model on, we need to discard some of the cells by multiplying the calculated loss matrix by a mask matrix. A mask matrix is a matrix that has ones in cells of real and "$" tokens, and zeros in cells of "#" tokens. Thus, the multiplication leaves only the necessary cross entropy values. The loss function is then computed as the mean of the sums of every row (see Table 2).

Training takes form of presenting our event log to the neural network trace by trace and minimizing the loss function described above via *stochastic gradient descent*, thus updating the weights of connections between layers.

As for our goal of inferring FSM from this neural network approach, first, we illustrate how states of the neural network correspond to the states of the synthesized transition system (Fig. 2).

Let Ω be a set of states of a neural network represented by the states of neurons in the output layers. By $\omega_i \in \Omega$ we denote a state of a recurrent neural network on step i of our algorithm. Then, we define three bijective functions $\xi : \Omega \to \mathbb{N}$, $\psi : \mathbb{N} \to S$ and $\phi : \Omega \to S$ as follows. As ω_i is a binary vector, ξ interprets it as a binary number and converts it to the equivalent decimal

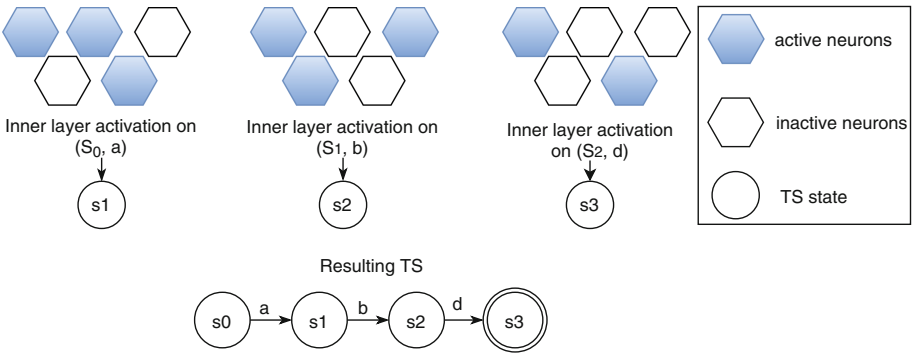
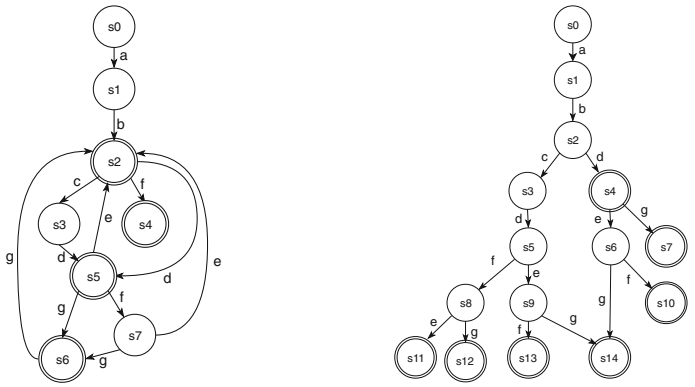


Fig. 2. Inner layer activation patterns on different inputs

number; ψ converts a number j from $\mathbb{N}$ into a state s_j; ϕ is a composition of ψ and ξ; Ω is isomorphic to S through $\mathbb{N}$.

The construction of the $TS = (S, E, T, s_0, AS)$ proceeds according to the following steps. We start by adding state s_0 to the originally empty TS. The initial state ω_0 of the neural network is a vector of zeros. On i-th step the current state of the RNN is ω_i and a new input token is α_i obtained from the *original* input trace. Both compose the input vector of the RNN, and output vectors are ω_{i+1} and p_{i+1}. The current state of TS is $s_i = \phi(\omega_i)$; the following state $s_{i+1} = \phi(\omega_{i+1})$ is added in the TS if it is not present or reused otherwise. These states are connected by the transition $t_i = (s_i, \alpha_i, s_{i+1})$. The described step is repeated for all non-"#" tokens of each trace.



(a) 10 neurons **(b)** 64 neurons

Fig. 3. TS inferred for *Log1*

An extracted TS has a unique transition $t_0 = (s_0, \$, s_1)$ due to the fact that we marked the beginning of all traces with "$". Both the transition and the

token have a purely technical nature and do not bear any meaningful information relevant to the input log behavior. Therefore, we can simply remove the transition t_0 and consider the state s_1 as the initial starting point of the TS. The state s_0 is still present in the TS if other valid transitions connect it to other states. Examples of extracted TSs for *Log1*[1] with two different hyperparameter values are present in Fig. 3.

This method synthesizes FSMs that are perfectly fit (i.e. they can always replay the whole log) *by construction*. It can be easily proved by induction following the same steps as in Theorem 1 in [10]. The described algorithm also gives an opportunity to control the states and transitions created. This allows us to make not perfectly fit models, though give them other features, e.g. being robust to noise. This gives a solid ground for further research, as there are plenty of ways of analysing predictions of token probabilities mentioned above.

The neural network presented above has one *hyperparameter*: the number of neurons in the dense layer. Each additional neuron not only has direct influence on the learning quality, but also doubles the output state space. Technically, the model has more hyperparameters. The number of neurons in the output vector and the embedding method mentioned earlier could also be regarded as such. We believe that they hold less importance and thus excluded them from consideration during our experiments. However, in further experiments they might be considered.

5 Experiments and Discussion

Experiments were done using mixed software and languages. We used Python libraries, namely `lasagne`, `theano` and `agentnet`, for building neural networks. We also used `LDOPA`[2] library to infer TSs with FQR algorithm and calculate metrics.

5.1 Plan of Experiments

All the experiments were conducted according to the following steps:

1. Logs were extracted from the .sq3 database format and processed as described in Sect. 4.
2. Hyperparameters were configured (number of neurons in the dense layer was chosen), the neural network was trained on a log, and the TS was extracted and serialized in the .json format.
3. `LDOPA` was used to convert .json files into internal `LDOPA` object model to calculate the metrics of inferred TSs.

In our experiments we use *Log1, Log2, Log3*. Their key features are presented in Table 3. In conducting the experiments, the "HP Pavillion dv6" PC with Intel Core i7-4510U CPU was used.

[1] See Table 3 for the log's details.

[2] The software is avaliable at the website https://pais.hse.ru/research/projects.

Table 3. Log information

Log	Number of traces	Number of unique tokens	Total number of tokens
Log1	8	7	41
Log2	243	143	2444
Log3	774	46	6604

Table 4. Untrained NNs and prefix trees

Log	Model type	Simplicity	Precision
Log2	Untrained net	0.067	0.943
	Prefix tree	0.053	1
Log3	Untrained net 1	0.022	0.731
	Untrained net 2	0.012	0.809
	Prefix tree	0.007	1

5.2 Experiments

Tuning and configuring model parameters influences metrics of inferred TSs. In our experiments we try to determine the nature of the influence. To evaluate the quality of the synthesized TSs, we focus on simplicity and precision metrics and how the model balances between them. In these experiments we do not evaluate the generalization results and consider parameters' influence on it a future work.

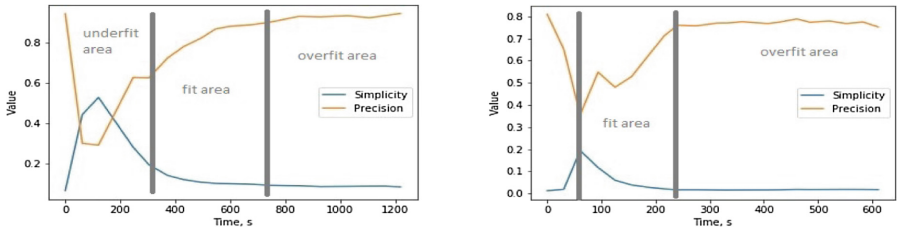
In the conducted experiments we compare the described approach to two different algorithms of constructing TSs. The first one is the *k-tail* algorithm of constructing prefix trees [16]. This approach allows to generate models with the maximum precision value of 1. However, such models tend to be very complex and, therefore, have low simplicity. The second one is the *frequency-based reduction* algorithm, which is thoroughly described in [10]. This method allows balancing between simplicity and precision by varying algorithm parameters. We compared the RNN inferred TSs with those inferred by FQR with various parameters in order to have a group of models for comparison.

Our general assumption is that as a neural network trains, it learns to recognize similar behavior fragments in the log even if they are sparsely distributed. An *untrained* neural network possesses a random configuration after initializing, so its reaction to input data is undetermined. As initial weights are assigned random values, random neurons are activated. This results in generating sequences of random states which are still connected by the correctly labeled transitions. Such TSs would be less precise than the prefix tree ones, while being similar in complexity. Table 4 confirms this fact. During the learining period we expect specific neurons to start activating when specific token patterns appear. This leads to the increase of the simplicity metric as same output states are obtained from ϕ.

We still expect very high precision from TSs inferred by this approach, as for each state the chosen loss function forces the model to minimize the chances of predicting wrong (unprobable) tokens. This results in the increase in the number of possible states and reduction of the number of outgoing transitions, which consequently makes the inferred TS more precise.

Comparing RNN Models Depending on Their Parameters. We start by comparing inferred models with varying training times. Figure 4 describes

the change in simplicity and precision values over the training period. Models in both figures consist of 64 neurons in the dense layer. In the first few training iterations we observe the expected increase in simplicity, because the network is able to detect the general features of the log. However, as the number of training iterations grows, the model starts to distinguish even the slightest differences in logs, thus creating different states for each pattern and increasing precision of the TS. Due to the fact that the described architecture is fairly simple, we regard overfitting as the most probable explanation. This means that the trained model finds a combination of weights that allows it to be more optimal on the training sample. Such a combination of weights is not always what we are looking for, as it harms the ability of a neural network to generalize behaviors. There exist several methods to overcome this problem, e.g. using *drop-out* techniques or *k-fold cross validation*. The impact of these methods on the results of the RNN approach needs further research. Figure 4 illustrates three stages of a model depending on the training time: underfit, fit[3] and overfit. Both underfit and overfit models are similar in structure to prefix tree models, that is why metric values return to the initial figures over time.



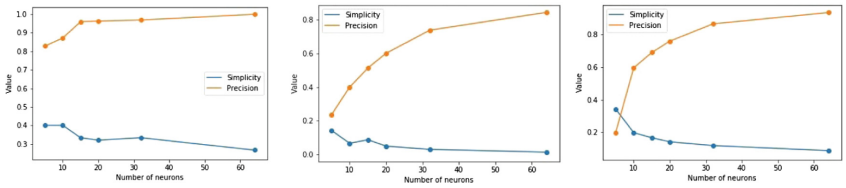
(a) *Log2* **(b)** *Log3*

Fig. 4. Simplicity and precision change over time

Next parameter to experiment on is the number of neurons in the dense layer in neural networks after the training period. The rise of computational power and classification capability, combined with the exponential growth of the number of output states and possible overfitting, results in behaviour illustrated on Fig. 5. As we can see in Fig. 3b, neural networks are capable of learning the prefix tree models, which also illustrates our assumption about overfitting.

Comparing the RNN Approach to the Frequency-Based Reduction Algorithm. It is necessary to compare models inferred by the neural network approach with those inferred by the FQR algorithm on close precision and simplicity values. By close values, figures whose residual is less than 0.02 have been accounted for.

[3] Here fit is not used in the context of quality metrics.



(a) *Log1* **(b)** *Log2* **(c)** *Log3*

Fig. 5. Simplicity and precision change over the number of neurons

Figures 6 and 7 show the figures for *Log2* comparing RNN approach (*red*) and FQR with different parameters [10] (*blue*). For this particular log, RNN proves to be better in both precision and simplicity.

Fig. 6. Simplicity values for close precision, *Log2* (Color figure online)

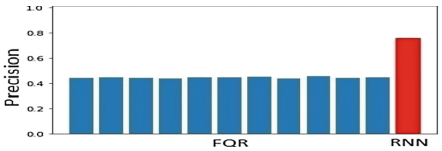


Fig. 7. Precision values for close simplicity, *Log2* (Color figure online)

However, for *Log3*, the RNN figures are less than those of FQR (Figs. 8 and 9). That is why we need to analyse the structure of both logs to interpret our results and conduct further research.

Fig. 8. Simplicity values for close precision, *Log3*

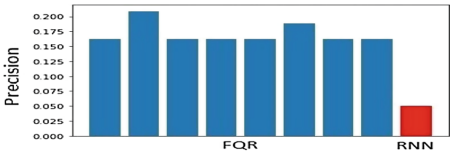


Fig. 9. Precision values for close simplicity, *Log3*

In regard to the most popular sequences of events that occur in the logs, *Log3* has a small amount of very popular sequences of actions. This log also has a relatively small number of unique events. Combined with the a big number of traces and events overall, it is assumed that the neural network is forced to distinguish even relatively similar sets of actions and thus generate new states for these sets as it is the only way of maximizing predictions of tokens appearing.

Behavior present in *Log2* contrasts with the one we have just described. For fewer traces and events, traces are more diverse as more unique events are present. Also strictly subsequent events are present: they appear only as a part of a certain subsequence and do not appear alone in other parts of the log. We assume that such event groups tend to increase simplicity of the model as the model tends to reuse already inferred states for them as only those states give best results from the standpoint of the loss function.

6 Conclusion

In this work a modern neural approach of synthesizing TSs from event logs is developed and implemented. Experiments showed that for some log types, namely for those where similar behaviors appear across the log, this approach gives better results than FQR algorithm. This determines the area of applicability of this method. If one is faced with a log of a structure similar to the structure of *Log2*, it is sensible to use the described approach.

Neural networks are known for their ability to tackle data with noise. As in this work we only built perfectly-fit TSs for the logs, we might have not discovered the full potential of this method yet. Further research in the field of tasks not requiring perfectly fit models is needed in order to find a task where such a model could be of use.

If we regard this approach, different architectures of the inner layers of the net might be considered, as now only a simple neural network with a single hidden dense layer is being experimented on. Moreover, it is concluded that dealing with overfitting using techniques mentioned in 5.2 can potentially provide better results.

References

1. Bishop, C.: Pattern Recognition and Machine Learning. Springer, New York (2006)
2. Lavagno, L., Kishinevsky, M., Yakovlev, A., Cortadella, J.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers **47**, 859–882 (1998)
3. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Softw. Syst. Model. **9**(1), 87 (2008)
4. Solé, M., Carmona, J.: Region-based foldings in process discovery. IEEE Trans. Knowl. Data Eng. **25**(1), 192–205 (2013)
5. Badouel, E., Bernardinello, L., Darondeau, P.: The synthesis problem for elementary net systems is NP-complete. Theor. Comput. Sci. **186**(1), 107–134 (1997)
6. Das, S., Mozer, M.: A unified gradient-descent/clustering architecture for finite state machine induction. In: Morgan Kaufmann Advances in Neural Information Processing Systems, vol. 6, pp. 19–26 (1994)
7. Cook, J., Wolf, A.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**, 215–249 (1998)

8. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 305–322. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_19
9. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Measuring precision of modeled behavior. Inf. Syst. e-Bus. Manag. **13**(1), 37–67 (2015)
10. Shershakov, S.A., Kalenkova, A.A., Lomazova, I.A.: Transition systems reduction: balancing between precision and simplicity. In: Koutny, M., Kleijn, J., Penczek, W. (eds.) Transactions on Petri Nets and Other Models of Concurrency XII. LNCS, vol. 10470, pp. 119–139. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-55862-1_6
11. Tax, N., Lu, X., Sidorova, N., Fahland, D., van der Aalst, W.M.P.: The imprecisions of precision measures in process mining. CoRR, abs/1705.03303 (2017)
12. Alpaydin, E.: Introduction to Machine Learning. MIT Press, Cambridge (2010)
13. Weiss, S., Kulikowski, C.: Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems. Morgan Kaufmann Publishers Inc., Burlington (1991)
14. Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Proceedings of ICSA, pp. 1045–1048 (2010)
15. Jang, E., Gu, S., Poole, B.: Categorical reparameterization with Gumbel-softmax. In: 5th International Conference on Learning Representations (2017)
16. Biermann, A., Feldman, J.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **C-21**(6), 592–597 (1972)