Synthesizing Control for a System with Black Box Environment, Based on Deep Learning

Simon Iosti[1(✉)], Doron Peled[2(✉)], Khen Aharon[2(✉)], Saddek Bensalem[1(✉)], and Yoav Goldberg[2(✉)]

[1] University Grenoble Alpes VERIMAG, 38410 St. Martin d'Héres, France
iosti.simon@gmail.com, saddek.bensalem@gmail.com
[2] Department of Computer Science, Bar Ilan University, 52900 Ramat Gan, Israel
doron.peled@gmail.com, khen.aharon@gmail.com, yoav.goldberg@gmail.com

Abstract. We study the synthesis of control for a system that interacts with a black-box environment, based on deep learning. The goal is to minimize the number of interaction failures. The current state of the environment is unavailable to the controller, hence its operation depends on a limited view of the history. We suggest a reinforcement learning framework of training a Recurrent Neural Network (RNN) to control such a system. We experiment with various parameters: loss function, exploration/exploitation ratio, and size of lookahead. We designed examples that capture various potential control difficulties. We present experiments performed with the toolkit DyNet.

1 Introduction

Deep learning (DL) [8] led to a huge leap in the capabilities of computers. Notable examples include speech recognition, natural language processing, image recognition and calculating strategies for difficult games, like `Chess` [5] and `Go` [4].

We study the deep-learning based synthesis of control for finite state systems that interact with black-box environments. In the studied model, the internal structure of the environment is not provided and its current state is not observable during the execution. In each step, the system makes a choice for the next action, and the environment must follow that choice if the action is enabled. Otherwise, a failed interaction occurs and the system does not move while the environment makes some independent progress. The control enforces the next

S. Iosti and S. Bensalem—The research performed by these authors was partially funded by H2020-ECSEL grants CPS4EU 2018-IA call - Grant Agreement number 826276.

D. Peled and K. Aharon—The research performed by these authors was partially funded by ISF grants "Runtime Measuring and Checking of Cyber Physical Systems" (ISF award 2239/15) and "Efficient Runtime Verification for Systems with Lots of Data and its Applications" (ISF award 1464/18).

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12477, pp. 457–472, 2020.
https://doi.org/10.1007/978-3-030-61470-6_27

action of the system based on the available partial information, which involves the sequence of states and actions that occurred so far of the system and the indication of success/failure to interact at each point. The control goal is to minimize the number of times that the system will offer an action that will result in a failed interaction.

The motivation for this problem comes from the challenge to construct distributed schedulers for systems with concurrent processes that will lower the number of failed interactions between the participants. In this case, the environment of each concurrent thread is the collection of all other threads, interacting with it. An alternative approach for constructing schedulers that is based on distributed knowledge was presented in [2].

Algorithmic synthesis of control for enforcing temporal constraints on interacting with an environment was studied in [12,13]. It includes translating a temporal logic specification into an automaton [7], determinizing it [14], and computing a winning strategy for a game defined over that automaton [18]. Reinforcement learning (RL) [15] suggests algorithmic solutions for synthesizing control for systems where the goal is to maximize (minimize, respectively) some accumulated future reward (penalty, respectively) over the interaction with the environment. In RL, a complete model of the environment does not have to be given, and in this case control can be synthesized through experimenting with it. However, often the current state of the environment is fully observable; this is not the case in our studied problem.

One approach for constructing control of a system that interacts with a black-box environment is to first apply automata learning techniques [1] in order to obtain a model of the environment. However, a complete learning of a model for the environment is sometimes infeasible for various reasons, e.g., when one cannot interface directly with the environment in isolation for making the needed experiments, or due to high complexity.

We study the construction of control based on neural networks, where after training, the neural network is used to control the system. This control construction is *adaptive*, which is important in the cases where the environment can change its typical behavior or where the tradeoff between further training and improved performance is not clear in advance; we can resume the training after deploying the system, and collecting runtime statistics.

We employ Recurrent Neural Networks (RNN), which include a feedback loop that returns part of the internal state as additional input. This allows the output to depend on the history of inputs rather than only on the last input. An RNN is well suited to provide the control strategy due to the invisibility of the structure and the immediate state of the environment; the controller's choice for the next step of the system needs to be based on its limited view of the past execution history.

Neural networks have been used within reinforcement learning for finding strategies for difficult board games such as Chess and Go. They were trained to provide the *quality* function Q from the current state of the game to a numeric value. The obtained game strategy chooses the move that maximizes the current

quality function. Our model cannot calculate a Q function, since the current state of the environment, corresponding to the game board in those examples, is not visible. Instead, we exploit a summary of the visible part of the execution so far, as calculated by the trained neural network. Notable work that involve training an RNN includes playing Atari games such as *space invaders* or *breakout* [9]. The reason for using the RNN architecture for these games is that the strategy depends on a limited number of preceding states (specifically for these games, four states are used); this permits the strategy to be influenced by the speed and trajectory of the movement. By contrast, our study involves training an RNN to produce a strategy that depends on a long history of the interaction between the system and the environment.

In [11] we compared the potential use of automata learning, deep learning and genetic programming for control synthesis. In this paper we focus on deep learning and present a full study of a methodology for synthesizing control to a system that interacts with a black-box environment. We study the required architecture and training parameters and present experiments using the DyNet tool [6].[1]

2 Preliminaries

We study the construction of control for a finite state reactive system that interacts with black-box environment.

System and Environment. We study systems that are modeled as finite state automata. Let $\mathcal{A} = (G, \iota, T, \delta)$ be an automaton, where

- G is a finite set of *states* with $\iota \in G$ its *initial state*.
- T is a finite set of *actions* (often called the *alphabet*).
- $\delta : (G \times T) \to G \cup \{\perp\}$ is a partial *transition function*, where $\perp$ stands for *undefined*. We denote $en(g) = \{t | t \in T \wedge \delta(g,t) \neq \perp\}$, i.e., $en(g)$ is the set of actions *enabled* at the state g. We assume that for each $g \in G$, $en(g) \neq \emptyset$.
- An optional component is a probabilistic distribution on the selection of actions $d : G \times T \to [0,1]$ where $\Sigma_{\tau \in T} d(g,\tau) = 1$. Then, the model is called a *Markov Chain*. In this case, $\delta(g,t) = \perp$ iff $d(g,t) = 0$.

The *asymmetric combination* of a system and an environment $\mathcal{A}^s \lceil \mathcal{A}^e$ involves the *system automaton* $\mathcal{A}^s = (G^s, \iota^s, T^s, \delta^s)$, and the *environment* automaton $\mathcal{A}^e = (G^e, \iota^e, T^e, \delta^e)$, where $T^s \cap T^e \neq \emptyset$. The two components progress synchronously starting with their initial state. The system offers an action that is enabled from its current state. If this action is enabled also from the current state of the environment automaton, then the system and the environment change their respective states, making a *successful* interaction. If the action is

[1] DyNet is a python package for automatic differentiation and stochastic gradient training, similar to PyTorch, and TensorFlow but which is also optimized for strong CPU performance.

not currently enabled by the environment, the interaction *fails*; in this case, the system remains at its current state, and the environment chooses randomly some enabled action and moves accordingly. After a failed interaction, the system can offer the same or a different action.

Formally,

$$\mathcal{A}^s \lceil \mathcal{A}^e = (G^s \times G^e, (\iota^s, \iota^e), T^s \times T^e, \delta)$$

where

$$\delta((g^s, g^e), (t^s, t^e)) = \begin{cases} (\delta^s(g^s, t^s), \delta^e(g^e, t^s)), & \text{if } t^s \in en(g_e) \\ (g^s, \delta^e(g^e, t^e)), & \text{otherwise} \end{cases}$$

An environment with a probabilistic distribution on selecting actions makes a probabilistic choice only if the action offered by the system is currently disabled. In this case, $\delta((g^s, g^e), (t^s, t^e)) = (g^s, \delta^e(g^e, t^e))$ with probability $d(g^e, t^e)$. We restrict ourselves to non-probabilistic (and deterministic) systems.

An execution ξ of $\mathcal{A}^s \lceil \mathcal{A}^e$ is a finite or infinite alternating sequence

$$(g_0^s, g_0^e)\,(t_0^s, t_0^e)\,(g_1^s, g_1^e)\ldots$$

of pairs of states and pairs of actions, where $g_0^s = \iota^s$, $g^e = \iota^e$ and $(g_{i+1}^s, g_{i+1}^e) = \delta((g_i^s, g_i^e), (t_i^s, t_i^e))$. According to the definitions, if $t_i^s \neq t_i^e$, then $g_i^s = g_{i+1}^s$; this is the case where the interaction failed. We denote by $\xi|_i$ the prefix of ξ that ends with the states (g_i^s, g_i^e).

Consider the system in Fig. 2 (left) and its environment (middle). This system can always make a choice between the actions a, b and c, and the environment has to agree with that choice if it is enabled from its current state. If the system selects actions according to $(abc)^*$, then the environment can follow that selection with no failures. On the other hand, if the system selects actions according to $(baa)^*$, the system will never progress, while the environment keeps changing states.

Supervisory control studies the problem of constructing a controller that restricts the behavior of a system; the combination guarantees additional requirements [16]. Our work is related to two methods for supervisory control, *reinforcement learning* and *deep learning*.

Reinforcement Learning. Reinforcement learning includes methods for controlling the interaction with an environment [15]. The goal is to maximize the expected utility value that sums up the future rewards/penalties; these can be discounted by γ^n with respect to its future distance n from the current point, with $0 < \gamma \leq 1$, or be summed up with respect to a finite distance (horizon). Typically, the model for RL is a Markov Decision Process (MDP), where there is a probability distribution on the states that are reached by taking an action from a given state. When the current state of the environment is not directly known to the controller during the execution, the model is a Partially Observable MDP (POMDP).

A *value-based* control policy (strategy) can be calculated by maximizing either a state value function $V(s)$ or a state-action value $Q(s, a)$. When the structure of the environment is known, a procedure based on Bellman's equation [15] can be used. If the structure is unknown, a randomized-based (*Monte Carlo*) exploration method can be used to update the value function and convert towards the optimal policy.

Policy based RL methods avoid calculating the optimal utility value directly at each state, hence are more effective when the number of possible states is huge. The policy is parametric and its parameters are optimized based on gradient descent. Such parameters can be, in particular, the weights of a neural network.

The training data is either selected *a priori*, according to some predefined scheme, or using the output of the partially trained neural network. In *mixed* training mode, we perform with probability ϵ an *exploration step* based on a random choice with uniform probability and with probability $1-\epsilon$ an *exploitation* step, based on the selection of the partially trained network. The value of ϵ may diminish with the progress of the training.

Deep Learning. Deep learning is a collection of methods for training *neural networks*, which can be used to perform various tasks such as image and speech recognition or playing games at an expert level. A neural network consists of a collection of nodes, the *neurons*, arranged in several layers, each neuron connected to all the neurons in the previous and the next layer. The first layer is the *input layer* and the last layer is the *output layer*. The other layers are *hidden*.

The value x_i of the i^{th} neuron at layer $j+1$ is computed from the column vector $\mathbf{y} = (y_1, \ldots, y_m)$ of all the neurons at layer j. To compute x_i, we first apply a transformation $t_i = \mathbf{w}_i\mathbf{y}+b_i$ where $\mathbf{w}_i$ is a line vector of *weights*, and b_i is a number called *bias*. Then we apply to the vector $\mathbf{t} = (t_1, \ldots, t_n)$ an *activation function*, which is usually non-linear, making the value of each neuron a function of the values of neurons at the preceding layer. Typical activation functions include the sigmoid and tanh functions, as well as the `softmax`.

The `softmax` activation function takes a vector of values and normalizes it into a corresponding vector of probability distributions, i.e., with values between 0 and 1, summing up to 1.

$$\mathtt{softmax}(\mathtt{t_1}, \ldots, \mathtt{t_n}) = \left(\frac{\mathtt{e}^{\mathtt{t_1}}}{\Sigma_\mathtt{i}\mathtt{e}^{\mathtt{t_i}}}, \ldots, \ldots, \frac{\mathtt{e}^{\mathtt{t_n}}}{\Sigma_\mathtt{i}\mathtt{e}^{\mathtt{t_i}}} \right)$$

Given values for all neurons in the input layer, we can compute the values for all neurons in the network, and overall a neural network represents a function $\mathbb{R}^n \to \mathbb{R}^m$ where n is the size of the input layer, and m the size of the output layer.

The values of the weights w_i and the biases b_i are initially random, and modified through *training*. A *loss function* provides a measurement on the distance between the actual output of the neural net and the desired output. The goal of training is to minimize the loss function. Optimizing the parameters is performed from the output layer backwards based on gradient descent.

For applications where sequences of inputs are analyzed, as e.g. in language recognition, one often uses a form of network called *Recurrent Neural Network* (RNN). An RNN maintains a feedback loop, where values of some neurons are returned to the network as additional inputs in the next step. In this way an RNN has the capability of maintaining some long term memory that summarizes the input sequence so far. The backward propagation of the gradient descent is applied not only once to the RNN, but continues to propagate backwards according to the length of the input sequence, as the RNN has been activated as many times as the length of the input so far. This allows training the RNN with respect to the input sequence instead of the last input. However the long propagation increases the problem of vanishing/exploding gradient. A more specific type of RNN that intends to solve this problem is a *Long Short-Term Memory*, LSTM. It includes components that control what (and how much) is erased from the memory layer of the network and what is added.

3 Controlling a System Interfacing with a Black Box

We seek to construct a controller for the system that will reduce the number of failed interactions. This is based on the information visible to the system (hence also to the controller), which is the executed sequence of system states, actions offered and the success/failure status of these interactions. The controller maintains and uses a summary of visible part of the execution so far. In addition, it receives the information about the latest execution step: the state of the system, the action selected by the system and whether the interaction succeeded or not. It then updates the summary and selects an enabled action; see Fig. 1.

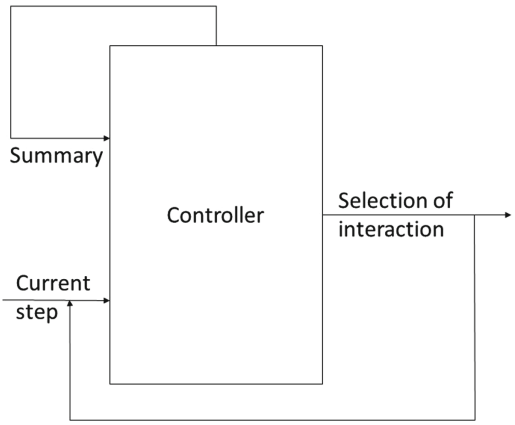

Fig. 1. A contoroller for the system

We now formally define the particular control problem. The *view* (visible part) $v(\xi)$ of an execution $\xi = (g_0^s, g_0^e)\,(t_0^s, t_0^e)\,(g_1^s, g_1^e)\ldots$ is the alternating

sequence $g_0^s(t_1^s, b_1)g_1^s(t_2^s, b_2)\ldots$, where $b_i = (t_i^s = t_i^e)$. $v(\xi|_i)$ is the prefix of $v(\xi)$ that ends with g_i^s. The control output is a function from the visible part of an execution to an action that is enabled in the current state of the system and will be offered to the environment in the next step.

Due to the invisibility of the environment structure and the state, there can be multiple histories (i.e., prefixes of an execution) that correspond to the same view. Future failing actions can cause the environment to make choices that are invisible to the controller; there can be multiple continuation from the current state. The goal of adding control is to minimize the sum of *penalties* due to failed interactions up to some fixed horizon.

The length of the current execution prefix, and subsequently the view, can grow arbitrarily. Instead, we maintain a finite *summary* of the view $m(v(\xi|_i))$ that we can use instead of $v(\xi|_i)$ itself. We seek a control function that is based on a summary of the view of the current prefix of the execution, and in addition the most recent information about the action selected by the system.

We assume a fixed probability on the selection of actions by the environment, in case of an interaction failure; for the experiments, we will assume a uniform distribution. We limit ourselves to *deterministic* policies, where at each point there is a unique selection of an enabled action. After training, our RNN will output some probability distribution on the selection of the next action, and the system will select the action with the highest probability.

A Suite of Examples

The following examples present for constructing a controller. They were used to select the loss function needed for training the neural networks used as controllers.

In Example `permitted` in Fig. 2, the system allows actions a, b and c. A controller that has the same structure as the environment would guarantee that the system never makes an attempt to interact that will fail, restricting the system to the single sequence $(abc)^*$. The action that appears in the figure next to a state of the controller is the action it enforces to select.

Note that the finite state controllers suggested in the figures of this section are *not* the actual ones that are implemented using neural networks trained through deep-learning.

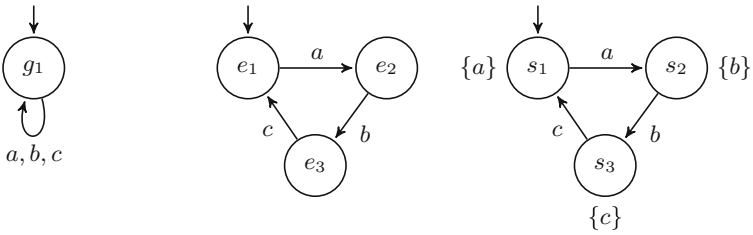

Fig. 2. `permitted`: System (left), Environment (middle) and Controller (right)

In Example `schedule` in Fig. 3, the controller must make sure that the system will never choose an a. Otherwise, after interacting on a, the environment will progress to e_3, and no successful interaction with b will be available further. A controller with two states that alternates between b and c, i.e., allows exactly the sequence of interactions $(bc)^*$ is sufficient to guarantee that no failure ever occurs.

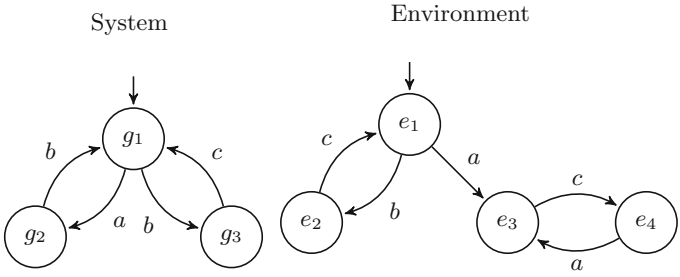

Fig. 3. `schedule`: The control needs to avoid the action a as its initial choice

In Example `cases` in Fig. 4 the system is obliged to offer an a from its initial state. The environment allows initially only b or c. Hence, the interaction will fail, the system will stay at the same state and the environment will progress to e_2 or to e_3, according to its choice, which is not visible to the system. After the first a, the system does not change state, hence it is again the only action that it offers. The controller that appears in Fig. 4 (left) moves after offering the first a, which is due to fail, from s_1 to s_2. It checks now whether offering a fails again; if not, it moves to s_3 and restricts the environment to offer $(ba)^*$. Otherwise, it moves to s_4 and will restrict the environment to offer $(ac)^*$.

In Example `strategy` in Fig. 5, the system offers initially only the interaction a, which necessarily fails, and consequently the environment makes a choice that is invisible to the system. After that, the system and the environment synchronize on an a. At this point, if the system chooses, by chance, an action that is enabled by the environment, making a successful interaction (b or c, respectively), it will

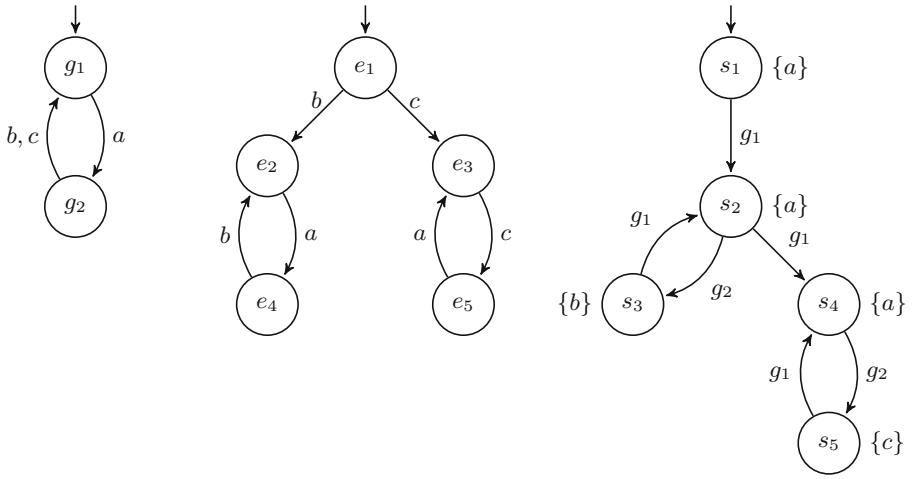

Fig. 4. `cases`: Needs to check if a succeeded

necessarily lead to entering self loops that are incompatible (at g_3 and e_7, or at g_4 and e_6, respectively), and no further successful interaction will happen. On the other hand, if the system chooses an interaction that immediately fails, it must repeat that same choice; This leads to compatible loops (g_3 and e_6, or g_4 and e_7), and there will be no further failures; flipping to the other choice after that will lead again to the incompatible loops.

Unfortunately, because of the invisible choice of the environment at the beginning of the execution, no controller can guarantee to restrict the number of failures in every execution. However, a weaker goal can be achieved, restricting the number of failures with some probability, which depends on the probability p of the environment to choose an b over c from its initial state. If we can learn the probability p, we may guarantee restricting the number of failures to two in at least $max(p, 1-p) \geq 0.5$ of the cases.

The Training Process

We will now show how an RNN of type LSTM can be trained to control a system in order to reduce the overall number of failures. The input layer is a vector of size $q \times |T|$, where q is the number of states of the system and $|T|$ is the number of transitions. The output of the last layer is passed through a `softmax` function to give the actual output of the network.

Let us consider the situation after a partial execution, where the RNN is fed the input and produces an output w. The system uses this output to choose an action a_j among its available actions. Let t be the number of such currently enabled actions. We distinguish two cases:

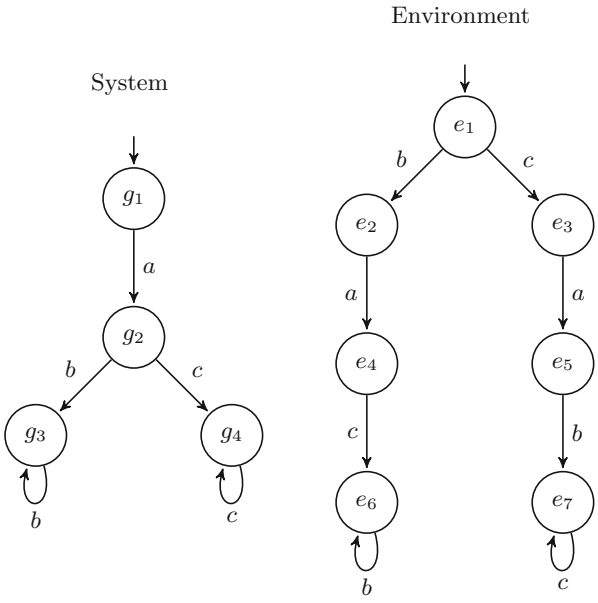

Fig. 5. `strategy`: Fail next, or succeed and fail forever

– If action a_j is *successful*, then the loss is defined to be

$$-log(\texttt{softmax(w)}_\texttt{j})$$

where $\texttt{softmax(w)}_\texttt{j}$ is the j^{th} coordinate of the softmax of the output w.
– If action a_j is *failed*, then the loss is defined to be

$$\Sigma_{1 \leq i \leq t, i \neq j} - \frac{1}{t-1} log(\texttt{softmax(w)}_\texttt{i})$$

Note that in the failed case, if there is only one available action ($t = 1$), then the loss is 0, which is not problematic since there is only one action to choose from. A backward propagation pass in then done through the RNN using this loss formula.

This loss function can initially seem counter-intuitive because of the second case, i.e., of a failed action. When offering action a_j fails, we chose to interpret this failure as meaning that *any other* choice would have been equally good. Consequently, we update their probabilities towards this. In other words, when a_j fails, instead of reducing its probability, we "even out" the probabilities of all other actions towards a uniform distribution among them. This effectively results in a reduction of the probability of a_j, but with more control over the probabilities of other actions.

We compare the above loss with another loss function (which we used in earlier experiments) that follows a more direct approach. It is computed in the

same way as the one described in the case of a successful action. In the case where action a_j is failed, the loss is computed as

$$log(\texttt{softmax(w)}_\texttt{j})$$

This results in effectively "punishing" the choice of selecting a_j, lowering its probability, instead of rewarding all other actions. We will call this loss the *naive* loss. Experiments show that this loss function has a tendency to learn less efficiently than the other one, and sometimes does not learn an optimal behavior at all.

Training Pattern with Fixed Lookahead. We fix a lookahead number $l \geq 0$ (where $l = 0$ corresponds to use the simple training pattern described above). We train the network at step n only when the execution reaches step $n + l$, and the loss is computed according to the same formulas as above, except that the l last steps of the execution are taken into account to compute the loss at step n. When the current partial execution of length $n + l$, we observe the output w that the RNN has generated at step n, with action a_j that was chosen by the system. Let $\mathsf{successes}$ be the number of successful actions taken by the system between step n and step $n + l$, and $\mathsf{failures}$ to be the number of failures between the same steps.

The loss is then computed as:

$$\mathsf{successes} \times -log(\texttt{softmax(w)}_\texttt{j}) + \mathsf{failures} \times -\Sigma_{1 \leq i \leq t} \texttt{log(softmax(w)}_\texttt{i})$$

The backward propagation is then applied to this loss at step n (and not at step $n + l$).

We also experimented with fixing a probability ϵ for exploring actions randomly, not based on the output of the partially trained RNN, both with a fixed ϵ and with diminishing the value of ϵ as training progressed. Another variant we experiment with is using a dual network, where, for diminishing instability of the updates, one is used for the exploitation of the partial training, and the other is updated. Switching between them each fixed number of steps.

The output provided by our RNN after training is, as a result of the `softmax` activation function, a probability distribution among the actions that can be taken from the current state. After training, we choose the action suggested by the controller with the highest probability. In many cases the training will converge towards a non probabilistic control policy, but will be limited due to the duration of the training. It is known from RL theory that for an environment that is deterministic or a Markov Chain, there exists an optimal deterministic policy that guarantees the highest expected discounted-sum utility from each state.

The fact that the environment is a black box does not change the situation, and a deterministic optimal control policy still exists. Yet, if we do not know how many states the environment has, we cannot even bound the size of such a policy. The number of states affects the number of required experiments and their

length. A further complication occurs because the state of the environment, and its selection in case of a failure, are unobservable. Based on a given finite view, the environment can be in one of several states according to some probability distribution (which we cannot calculate, since the environment is a black box). In fact, it is possible that there can be infinitely many such distributions for different views.

Even when there exists an optimal deterministic strategy, we have no guarantee that the deep learning training will converge towards a deterministic strategy. For consider again Example `strategy` in Fig. 5. In case that the environment makes its choice from e_1 with *uniform* probability, any probabilistic choice between b and c by the system will make as good (or bad) policy as making a fixed choice between these two actions.

4 Experiments

We describe several experiments with the examples from the training suite. The experiments show testing different training patterns on each individual example. All results have been obtained using a similar structure for the RNN associated to the network. The size of the LSTM layer is 10. Initializing the non-recurrent parameters of the network is according to the Glorot initialization. The training passes have been done in every case by generating training sequences of increasing size, from 1 to 50, and repeating this 50 times.

Comparison of the Actual and the Naive Loss. The aim of this section is to show the advantage of using our actual loss described in the previous section over using the naive loss.

We present in Table 1 the results of experiments on four examples from the training suite, with various values for the lookahead and ϵ. The rows correspond to the different environments, and the columns to the pair (l, ϵ), where l is the lookahead in the training, and ϵ is the probability of performing an exploration step. Shaded lines correspond to the results when using the naive loss, while unshaded lines show the results for our actual loss. The entries in the table show the average percentage of failures when generating 100 executions of size 200.

Table 1. Summary of Experiments; shaded results are obtained using the naive loss

%failures \ (l, ϵ)	$(0,0)$	$(0,0.2)$	$(0,0.5)$	$(3,0)$	$(3,0.2)$	$(3,0.5)$	$(20,0)$	$(20,0.2)$	$(20,0.5)$
`permitted`	0.0	0.0	0.0	0.0	0.0	7.6	15.9	22.2	27.0
	15.69	18.62	18.68	43.79	55.01	61.44	65.56	65.32	67.39
`schedule`	98.5	98.4	98.5	9.9	0.2	0.0	0.5	0.3	78.5
	98.39	96.47	97.08	0	0	0	0	0	94.75
`cases`	1.7	1.6	6.9	1.5	1.5	1.6	34.5	38.9	45.6
	3.04	1.78	2	1.56	1.54	1.94	56.06	42.98	47.04
`choice-scc`	46.7	49.6	44.9	80.1	85.0	85.0	28.7	22.3	4.5
	46.22	54.2	64.0	80.2	73.7	70.5	36.8	30.3	33.5

We first discuss the results concerning our actual loss function (unshaded lines in the table).

In example `permitted`, the basic no-lookahead learning without exploration works very well, and both lookahead and exploration tend to be counterproductive. This is not too surprising observing that a lookahead or exploration pattern here would only blur the fact that a good choice is immediately interpretable as such, depending on the failure or success of the action. In example `schedule`, the lookahead is crucial for learning, and the exploration is again either counterproductive, or at least not advantageous (depending on the lookahead). In example `cases`, a long lookahead is again not efficient, and the exploration is not necessary. In example `choice-scc`, a very long lookahead is beneficial, which is to be expected in this example since a good choice at the beginning of the execution can be identified as such only after the three first steps. Seeing several successful steps afterwards to counter the effect of the failures at the beginning. Even in the case of a long lookahead, exploration with the high probability of 0.5, improves dramatically the results, where a long lookahead is insufficient to reach an almost optimal behavior. This shows the importance of exploration in this kind of situations where better strongly connected components are only reachable through worse paths that the system tends to avoid in the case of pure exploitation.

Note that the training was performed on sequences of length at most 50, but the behavior of the controller is verified on sequences of length 200, showing that a training on short sequences allow the controller to generalize its behavior on longer sequences. This gives evidence that a finite training is effective to learn an optimal behavior on very (possibly arbitrary) long sequences. Of course, without having a good estimate on the number of states of the environment, a"combination locks" in it can hide behaviors that deviate from the learned control.

Results using the naive loss appear as shaded lines in Table 1, for comparison with our actual loss. We will use as point of comparison the values of the parameters where the use of one loss or the other raises results that are almost optimal. In examples `schedule` and `cases`, we can see that both losses perform similarly: the situations where the training is optimal are the same for both losses. In examples `permitted` and `choice-scc`, we see that the naive loss performs very badly in comparison with the actual loss. In both cases the naive loss never manages to reach an optimal behavior for the system, while the actual loss performs very well for good choices of parameters.

Additional Experiments. Several other experiments were made using direct variations of our training scheme. We tested two standard techniques from deep learning: diminishing the value of the exploration/exploitation ratio ϵ along the training, and using two "dual" networks. One of which is updated at every step as usual, and the other is trained only at the end of a training sequence but is used to generate these sequences. Depending on the examples we tested these variants on, the results were either slightly better or similar to our results without these.

Another kind of experiments that we did was using combinations of pairs of our examples from the training suite. We devised examples mixing the behaviors of `permitted` and `schedule`, and mixing the behavior of `permitted` and `choice-scc`. Both of these combined examples were built using examples for which the optimal values of the parameters were very different. Surprisingly, we found that we still were able to learn these examples, using the same training pattern but alternating variables l and ϵ from those that achieved the optimal values for the two original examples. On the other hand, the length and number of training sequences had to be chosen using some additional heuristics, because the original values for these were not efficient for training. We detail our results in the following table. P is the number of training passes; a training pass involves running the usual training scheme with the first values of (l, ϵ) with N sequences of length L, then again with the second values of (l, ϵ). Every experiment was repeated 20 times, and the lowest, highest, and average failure rates are in the shaded columns (Table 2).

Table 2. Parameters and results for combined examples

	Values of (l, ϵ)	L	N	P	lowest	highest	average
`permitted` and `schedule`	$(5, 0)$ and $(0, 0)$	8	1000	2	0.0	98.0	5.0
`permitted` and `choice-scc`	$(50, 0.5)$ and $(0, 0)$	50	10	8	1.5	66	33.5

5 Conclusions and Discussion

We presented a methodology for constructing control for a system that interacts with a black-box environment: it is not restricted to a specific objective, and it can be applied to various types of systems and environments. Instead of training the control for a specific system, we suggested the use of small, well designed, examples, which feature various potential challenges for training. We demonstrated our approach on training a set of examples using the DyNet tool.

Compared with the impressive use cases of deep learning, such as image recognition, translating natural languages, autonomous driving or playing games, the learning-based control synthesis problem that we considered here seems much simpler. However, it allows studying the principles and effect of different learning techniques.

Our use of recurrent deep learning, based on RNNs or LSTMs, has several benefits. First, the method is independent of knowing the states of the environment. The states of the constructed controller are kept implicitly in the hidden layer(s) of the neural network. We are oblivious of whether two internal representations are isomorphic w.r.t. the strategy, nor do we have to care.

Some works on deep reinforcement learning use recurrent deep learning, e.g., [9,10,17]. This was done for interactive Atari games, where the single current observed screen frame does not provide a complete current state. Since the control objective is the standard one, these methods apply a standard loss function that is based on the square of the difference between the current and the previous objective value.

Our long term goal is to expand this approach for constructing distributed schedulers for systems with concurrent processes that will lower the number of failed interactions between the participating processes. This can then be compared with an alternative approach for constructing schedulers that is based on distributed knowledge [2,3].

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Basu, A., Bensalem, S., Peled, D.A., Sifakis, J.: Priority scheduling of distributed systems based on model checking. Formal Methods Syst. Des. **39**(3), 229–245 (2011)
3. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge based controlling of distributed systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 52–66. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_6
4. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
5. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. CoRR, abs/1712.01815 (2017)
6. Neubig, G., et al.: DyNet: the dynamic neural network toolkit. CoRR, abs/1701.03980 (2017)
7. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M., (eds.) Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1995)
8. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep learning. In: Adaptive Computation and Machine Learning. MIT Press (2016)
9. Hausknecht, M.J., Stone, P.: Deep recurrent Q-learning for partially observable mdps. CoRR, abs/1507.06527 (2015)
10. Heess, N., Hunt, J.J., Lillicrap, T.P., Silver, D.: Memory-based control with recurrent neural networks. CoRR, abs/1512.04455 (2015)
11. Peled, D., Iosti, S., Bensalem, S.: Control synthesis through deep learning. In: Bartocci, E., Cleaveland, R., Grosu, R., Sokolsky, O. (eds.) From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday, pp. 242–255. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31514-6_14
12. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 179–190 (1989)

13. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, 22–24 October 1990, vol. II, pp. 746–757 (1990)
14. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24–26 October 1988, pp. 319–327. IEEE Computer Society (1988)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning - An Introduction. Adaptive Computation and Machine Learning, 2nd edn. MIT Press (2018)
16. Wonham, W.M., Ramadge, P.J.: Modular supervisory control of discrete-event systems. MCSS **1**(1), 13–30 (1988)
17. Zhu, P., Li, X., Poupart, P.: On improving deep reinforcement learning for pomdps. CoRR, abs/1704.07978 (2017)
18. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1–2), 135–183 (1998)