DeepDECS Quick Tutorial

March 6, 2023

Using DeepDECS is straightforward, and in this short tutorial we will take you through the basics in under 15 minutes. In doing so, we will illustrate the application of DeepDECS to a hypothetical toy example in which an autonomous car assesses the condition of the next segment of road in front of it at regular time intervals, and decides to drive over it "fast" or "slow" depending on whether a DNN classifier predicts it to be icy or not.

1 Prepare the DeepDECS inputs

Prior to using DeepDECS, you need to prepare its four inputs (see Fig. 1a from our DeepDECS research paper[1]): the DNN classifier that the autonomous system uses to perceive its environment, a representative test dataset for this DNN, a perfect-perception discrete-time Markov chain (DTMC) modelling the behaviour of the system, and the system requirements encoded in probabilistic computation tree logic (PCTL).

1.1 DNN classifier

Applying DeepDECS requires a DNN classifier that distinguishes between the different environment states relevant to the autonomous system. This perception DNN needs to be trained prior to using DeepDECS.

For the hypothetical autonomous car from our example, the perception DNN needs to be a two-class DNN classifier that takes sensor inputs about the state of the environment and predicts whether the road segment ahead is not covered in ice (class $k = 1$) or is icy (class $k = 2$).

1.2 Test dataset

To use DeepDECS, you will also need a large test dataset that is *representative* for the inputs that the perception DNN will encounter when the autonomous system is deployed in its operational design domain. This test dataset will be used to quantify the accuracy of your DNN classifier in the first DeepDECS stage.

For the hypothetical autonomous car and its two-class DNN classifier, this test dataset may comprise $10,000$ data samples from non-icy road segments and $6,000$ samples from icy road segments.

1.3 Perfect-perception DTMC model

DeepDECS also requires as input a DTMC that instantiates the DeepDECS *generic perfect-perception DTMC* from Fig. 2a in our paper[1] and models the behaviour of the autonomous systems under the assumption that it is capable of perceiving the state of its environment with 100% accuracy at all times.

Listing 1 shows the instance of the DeepDECS *generic perfect-perception DTMC* that models the operation of our hypothetical autonomous car. The DTMC comprises four modules: (i) $ManagedComponents$, (ii) $EnvironmentMonitor$, (iii) $Controller$ and (iv) $Turn$. The $ManagedComponents$ module represents the car actions of driving fast or slow. Those actions are autonomously performed by the $Controller$, which probabilistically decides ($x2$) if the car should slow down its speed ($slow' = true$), when an icy road ($k' = 2$) was perceived from the $EnvironmentMonitor$. Otherwise, when the $EnvironmentMonitor$ perceives there

[1]https://arxiv.org/abs/2202.03360

```
1  dtmc
2
3  module ManagedComponents // autonomous car
4    z : [0..1] init 0;
5
6    [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
7    [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
8  endmodule
9
10 module EnvironmentMonitor
11   k : [1..2]; // k=1 - no ice; k=2 - ice
12
13   [monitor] t=2 -> 0.6:(k'=1) + 0.4:(k'=2);
14 endmodule
15
16 const double x1; //probability of not slowing down on non-icy road (k=1)
17 const double x2; //probability of not slowing down on icy road (k=2)
18
19 module Controller
20   slow : bool init false;
21
22   [decide] t=3 & k=1 -> x1:(slow'=false) + (1-x1):(slow'=true);
23   [decide] t=3 & k=2 -> x2:(slow'=false) + (1-x2):(slow'=true);
24 endmodule
25
26 module Turn
27   t : [1..3] init 1;
28
29   [driveFast] true -> 1.0:(t'=2);
30   [driveSlow] true -> 1.0:(t'=2);
31   [monitor]   true -> 1.0:(t'=3);
32   [decide]    true -> 1.0:(t'=1);
33 endmodule
34
35 rewards "risk"
36   [driveFast] k=2 : 1;
37 endrewards
38
39 rewards "distance"
40   [driveFast] true : 3;
41   [driveSlow] true : 1;
42 endrewards
```

Listing 1: DTMC for the perfect perception model in PRISM language

is no ice on the road ($k' = 1$), the *Controller* decides to not slow down the speed of the car ($slow' = false$) following a probability distribution $x1$. Finally, the *Turn* module is responsible for assuring the communication between the *ManagedComponents*, the *EnvironmentMonitor* and the *Controller* modules follows a MAPE-K loop.

Considering our DTMC is reward-augmented, there are two reward structures in our model: *risk* and *distance*. Those structures allow us to compute, for example, *the cumulated risk reward along the driveFast action until 1000 time units have elapsed* (($\mathcal{R}^{\mathsf{risk}}$[C <= 1000])). The risk, in this case, indicates the car is driving fast on an icy road and therefore, some penalty (cost) should be computed in such a risky situation (cf. line 36). Analogously, we can compute the cumulated *distance* reward for the same 1000 time units (($\mathcal{R}^{\mathsf{distance}}$[C <= 1000])). We should note that for the *distance* reward structure, we assign a higher distance cost (cf. line 40) to the car if it takes a *driveFast* action and lower one for the *driveSlow* action (cf. line 41). Then the controller should be able to optimise its objectives as a tradeoff between minising *risk* and maximising *distance* rewards.

1.4 PCTL-encoded requirements

The last input that you will need in order to use DeepDECS is a PCTL encoding of the requirements (constraints and optimisation objectives) for your autonomous system. The PCTL syntax, semantics and the use of PCTL to express such requirements are explained in our paper.[1]

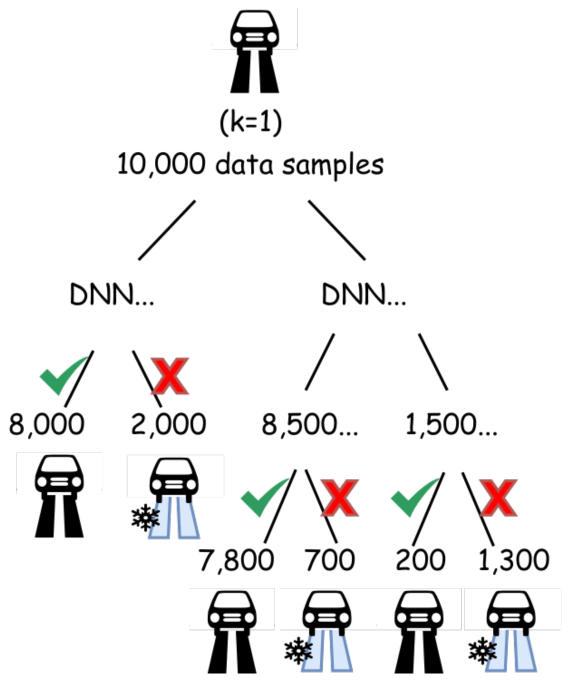
As an illustration, we will assume that the aim of the autonomous car from our example is to minimise the risk and to maximise the driven distance accrued over 1000 DTMC state transitions (corresponding to driving across 1000/3 road segments, since three state transitions are taken for each road segment as the turn variable t goes full cycle through its three values $1 \to 2 \to 3 \to 1$). These two optimisation criteria are encoded as:

$$\begin{array}{l}\text{minimise } \mathcal{R}^{\text{risk}}[C <= 1000] \\ \text{maximise } \mathcal{R}^{\text{distance}}[C <= 1000]\end{array} \tag{1}$$

2 DeepDECS Stage 1: DNN uncertainty quantification

In this DeepDECS stage, you need to quantify the uncertainty introduced by the use of your DNN to perceive the environment of the autonomous system. We explain below how you can do this both when you are using no DNN verification method, and when you are using a single verification method.

For the DNN uncertainty quantification, let us assume it follows the rates depicted in Figure 1.



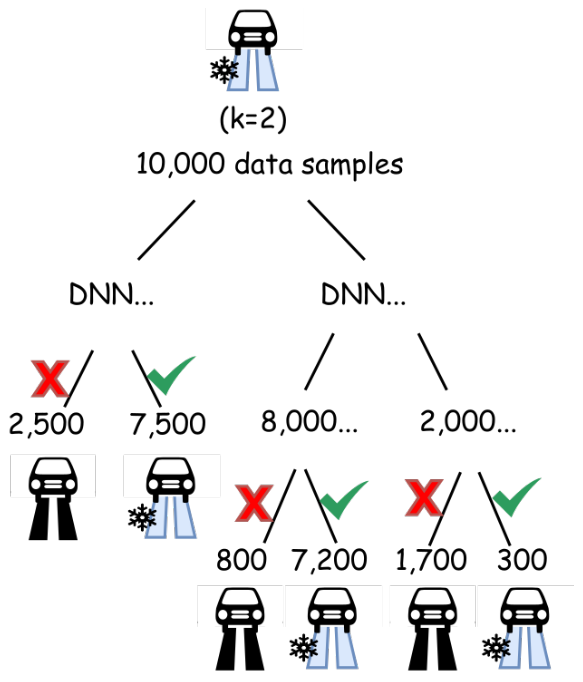


Figure 1: Assumed classification results for the road condition DNN classifier, without and with the verification method, assuming a test dataset that comprises 10,000 data samples for each of the two classes. Fix: The .svg figure is omitting the rest of the text and replacing with the suspension points....

So suppose you have 10,000 data samples for the non-icy (k=1) and icy road (k=2), respectively. In the first moment, you can use the DNN without the verification method in the first instance and the DNN in tandem with the verification method in the second instance. In the first instance, the DNN accurately classifies 8,000 samples for the non-icy road and 7,500 sample for the icy road. This gives the DNN an accuracy of 80% and 75% for the non-icy and icy road, respectively. In the second instance, we conjecture that the combined DNN classification with the verification method gives slightly better accuracy. In this scenario, the verification method is capable of further improving the initial 8,500 DNN classification (85% accuracy) into a $\approx$ 91,76% accuracy (7,800 over 8,500) for the non-icy road. An analogous reasoning goes for the icy road, as depicted in Figure 1.

2.1 No DNN verification method

For the pDTMC model of the autonomous car with only the DNN perception, the only extensions needed for the perfect perception model are in the *EnvironmentMonitor* and in the *Controller* modules. To do so, we introduce into the *EnvironmentMonitor* module the variable $k2$ (see line 12 in Listing 2 to represent the perception of the DNN for non-icy ($k2 = 1$) and icy ($k2 = 2$) roads, respectively. Additionally, the probability that the DNN accurately perceives a non-icy road is 90%, and 95% accuracy for the icy road. This probabilities may change if we align Figure 1 with the model.

In the *Controller* module, we need to make the Controller aware of the DNN perception (i.e. k2=1 or k=2) so that it decides on whether the car should slow down the car or not. Please check lines 24 and 25 of the *Controller* module where the guard condition of the controller considers $k2$ instead of k as the environment perception variable.

2.2 One DNN verification method

To conclude this first stage, we provide the pDTMC model for the autonomous car now provided with a single verification method. Similar to the pDTMC model, the only extensions needed are in the *EnvironmentMonitor* and in the *Controller* modules. On top of the variable $k2$ previously presented in Section 2.1, we introduce the variable $v1$ (see line 12 in Listing 3 to represent the outcome of the verification method, whether T or F.

In addition, to provide a more accurate decision, the *Controller* module needs to take into account whether to slow down the car or not based on the conjunction of the DNN perception ($k2$) and the newly introduced result of the verifier ($v1$). Lines 33 and 34 represent the cases that, with probability $x1false$, the DNN misclassifies non-icy road ($k2 = 1$) when v1 verification fails ($!v1$) and the case that, with probability $x1true$, the DNN correctly classifies non-icy road ($k2 = 1$) when v1 verification succeeds ($v1$). An analogous reasoning follows for the icy road on lines 36 and 37 where the probabilities for correct and incorrect DNN classification when v1 verification fails and succeeds are respectively represented in variables $x2false$ and $x2true$.

3 DeepDECS Stage 2: Model augmentation

3.1 No DNN verification method

3.2 One DNN verification method

4 DeepDECS Stage 3: Controller synthesis

In the final DeepDECS stage, you will need to use a probabilistic model checker such as PRISM or Storm to assemble a set of controllers that meet any constraints that the autonomous system requirements may include, and achieve Pareto-optimal trade-offs among the optimisation objectives from these requirements. Assuming you are synthesising probabilistic controllers, you have two options. We describe each of these options in turn.

The first option is to discretise the design space of the controller, e.g., by considering all values from the set $\{0.0, 0.1, \ldots, 0.9, 1.0\}$ for the probabilities with which the controller takes each of its actions. If you use this option, you have the opportunity to examine the outcome of using the controllers corresponding to all combinations of such probabilities, and to then identify those combinations that yield controllers which:

1. satisfy all constraints from the requirements of the autonomous system;
2. are Pareto-optimal with respect to the optimisation objectives from these requirements.

We will illustrate the use of this first option later in the section.

As a second option, you can use a search procedure to explore possible combinations of controller parameter values. This exploration can be performed using random search, or a more sophisticated search method such as genetic algorithms (e.g., using the EvoChecker tool for probabilistic model synthesis, which uses PRISM or Storm as a backend probabilistic analysis engine). Explaining this is beyond the scope of our tutorial, so we refer you to the EvoChecker website, etc.

```
dtmc

module ManagedComponents // autonomous car
  z : [0..1] init 0;

  [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
  [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
endmodule

module EnvironmentMonitor
  k : [1..2] init 1; // k=1 - no ice; k=2 - ice
  k2: [1..2] init 1; // DNN perception of the real world

  [monitor] t=2 -> 0.6*(8000/10000):(k'=1)&(k2'=1) + 0.6*(2000/10000):(k'=1)&(k2'=2) +
                   0.4*(2500/10000):(k'=2)&(k2'=1) + 0.4*(7500/10000):(k'=2)&(k2'=2);
endmodule

const double x1; //probability of not slowing down on non-icy road (k=1)
const double x2; //probability of not slowing down on icy road (k=2)

module Controller
  slow : bool init false;

  [decide] t=3 & k2=1 -> x1:(slow'=false) + (1-x1):(slow'=true);
  [decide] t=3 & k2=2 -> x2:(slow'=false) + (1-x2):(slow'=true);
endmodule

module Turn
  t : [1..3] init 1;

  [driveFast] true -> 1.0:(t'=2);
  [driveSlow] true -> 1.0:(t'=2);
  [monitor]   true -> 1.0:(t'=3);
  [decide]    true -> 1.0:(t'=1);
endmodule

rewards "risk"
  [driveFast] k=2 : 1;
endrewards

rewards "distance"
  [driveFast] true : 3;
  [driveSlow] true : 1;
endrewards
```

Listing 2: DTMC for the DNN model without a verification technique in PRISM language

4.1 Pareto-optimal controllers for the perfect-perception system

You may want to first obtain the Pareto front of controllers for the perfect-perception autonomous system. Although these controllers are not of practical value because a 100% accurate DNN cannot be achieved, they represent a good baseline to compare against in order to find out how far from the ideal your feasible controllers can operate.

We illustrate how this can be done for our hypothetical autonomous car ...

4.2 Pareto-optimal controllers with no DNN verification method

4.3 Pareto-optimal controllers with one DNN verification method

4.4 Combining the results

```
dtmc

module ManagedComponents // autonomous car
  z : [0..1] init 0;

  [driveFast] t=1 & z=0 & !slow -> 1.0:(z'=0);
  [driveSlow] t=1 & z=0 & slow -> 1.0:(z'=0);
endmodule

module EnvironmentMonitor
  k : [1..2] init 1; // k=1 - no ice; k=2 - ice
  k2: [1..2] init 1; // DNN perception of the real world
  v1: bool init false; //Result of the DNN single verification method

  [monitor] t=2 -> 0.6*(8500/10000)*(7800/8500):(k'=1)&(k2'=1)&(v1'=true) +
                   0.6*(8500/10000)*(1-7800/8500):(k'=1)&(k2'=2)&(v1'=true) +
                   0.6*(1-8500/10000)*(200/1500):(k'=1)&(k2'=1)&(v1'=false) +
                   0.6*(1-8500/10000)*(1300/1500):(k'=1)&(k2'=2)&(v1'=false) +
                   0.4*(8000/10000)*(800/8000):(k'=2)&(k2'=1)&(v1'=true) +
                   0.4*(8000/10000)*(7200/8000):(k'=2)&(k2'=2)&(v1'=true) +
                   0.4*(2000/10000)*(1700/2000):(k'=2)&(k2'=1)&(v1'=false) +
                   0.4*(2000/10000)*(300/2000):(k'=2)&(k2'=2)&(v1'=false);
endmodule

const double x1false; //prob. the DNN misclassifies non-icy road when v1 fails
const double x1true; //prob. the DNN classifies non-icy road when v1 succeeds
const double x2false; //prob. the DNN misclassifies icy road when v1 fails
const double x2true; //prob. the DNN classifies icy road when v1 succeeds

module Controller
  slow : bool init false;

  [decide] t=3 & k2=1 & !v1 -> x1false:(slow'=false) + (1-x1false):(slow'=true);
  [decide] t=3 & k2=1 & v1 -> x1true:(slow'=false) + (1-x1true):(slow'=true);

  [decide] t=3 & k2=2 & !v1 -> x2false:(slow'=false) + (1-x2false):(slow'=true);
  [decide] t=3 & k2=2 & v1 -> x2true:(slow'=false) + (1-x2true):(slow'=true);
endmodule

module Turn
  t : [1..3] init 1;

  [driveFast] true -> 1.0:(t'=2);
  [driveSlow] true -> 1.0:(t'=2);
  [monitor]   true -> 1.0:(t'=3);
  [decide]    true -> 1.0:(t'=1);
endmodule

rewards "risk"
  [driveFast] k=2 : 1;
endrewards

rewards "distance"
  [driveFast] true : 3;
  [driveSlow] true : 1;
endrewards
```

Listing 3: DTMC for the DNN model with a verification technique in PRISM language