ASSURANCE OF SELF-DRIVING CARS: A REINFORCEMENT LEARNING APPROACH

KE QUAN

COMP8755 INDIVIDUAL COMPUTING PROJECT
SUPERVISED BY: DR. HANNA KURNIAWATI
AUSTRALIAN NATIONAL UNIVERSITY

JUNE 2020
## Contents

1 Acknowledgements  
2 Abstract  
3 Introduction  
   3.1 Motivation  
   3.2 Related Work  
4 Background  
   4.1 Reinforcement Learning  
   4.2 Gaussian Process  
   4.3 Monte Carlo Tree Search  
5 Method  
   5.1 Top-Level Algorithm  
   5.2 Formulation of Problem  
   5.3 Offline Model Learning  
      5.3.1 PILCO Gaussian Processes  
      5.3.2 Sample from Latent Function  
      5.3.3 Data Gathering Policy  
      5.3.4 Offline Model Learning Algorithm  
   5.4 Online Planning  
      5.4.1 \( \rho \)UCT  
      5.4.2 Model Error and Heuristics  
   5.5 Software Architecture  
      5.5.1 Agent Space  
   5.6 CARLA Space  
   5.7 Data Processing  
6 Results  
   6.1 Experiment Setup  
   6.2 Model Learning  
   6.3 Planning  
   6.4 Discussion  
      6.4.1 Assurance Discoveries  
7 Conclusion  
   7.1 Conclusion  
   7.2 Future Work  
   7.2.1 Prediction  
   7.2.2 Scenario Simplicity  
8 Appendix 1: Study Contract
9 Appendix 2: Software Details 33
10 Appendix 3: Software Manual 34
List of Figures

1. Overview of our work .................................................. 8
5. Inside the Agent ............................................................. 15
7. Software Architecture ..................................................... 20
8. Multiple-step prediction results using PILCO-GPs with 50 training sets ... 25
9. Decision-time Planning Results ......................................... 26
10. Failure happens when sun altitude is low ........................... 27

List of Tables

1. Prediction Accuracy Results ............................................. 24
2. Model Learning Time ..................................................... 24
3. Hyper Parameters for MCTS ............................................ 26

List of Algorithms

1. Top-level Algorithm ..................................................... 14
2. Batch Optimisation of PILCO GPs ..................................... 17
3. ρUCT Planning .......................................................... 19
4. ρUCT Sample ............................................................ 19
5. CARLA Client Algorithm ................................................ 22
1 Acknowledgements

First and foremost, I would like to thank my supervisor Hanna Kurniawati for giving me the great opportunity to take on this exciting and interesting project. She has been very encouraging and supportive since the beginning of my journey on this work. The guidance and advises provided by her are not only greatly valuable to this project but also to my own learning. I would also like to express my gratitude to Jimy Cai, without whose help on the work of CARLA this project would not have gone this far. Also thanks the whole team of RDL for giving me all kinds of helps and those interesting discussions. In addition, I would like to thank my examiner Alwen Tiu for his time in helping review this work. Finally, I would like to thank my wife for giving me all the encouragement and support that carried me through this journey.
2 Abstract

This paper aims to introduce a principled approach to learn the dynamics of an autonomous vehicle and further utilise models learned to explore potential vulnerabilities of the underlying autonomous controller. Recent crashing incidents of autonomous vehicles have put the stability and consistency of autonomous controllers under question. Environments that lead to controller failures are hard to reproduce in physical settings, which makes identifying and addressing of potential weakness in controllers difficult. Our work serves as a first step in constructing an assurance framework for autonomous controllers by devising a reinforcement learning agent in a simulation environment and search for optimal changes in the environment that drives autonomous controller to yield hazardous controls.

Keywords: Reinforcement Learning, Autonomous Driving, CARLA Simulator, Gaussian Process, Monte-Carlo Tree Search
3 Introduction

3.1 Motivation

Autonomous vehicles (or self-driving vehicles) have seen fast growth in recent years, where collaborative efforts from multiple research fields including computer vision, robotics control and AI planning [5, 10, 14] have greatly pushed the boundary closer towards human-level capability. The impact of autonomous vehicles on the society could be far-reaching due to its broad potential in improving efficiencies for both individuals and organisations. For example, autonomous driving could turn commuting time into working time, reduce traffic load through better planning, decrease the city space needed for parking etc. Public road experiments by autonomous vehicles providers such as Waymo and Tesla have encouraged broad social aspiration on the coming of full autonomous driving.

However, current self-driving vehicles are still distant from human-level calibre as evidenced by recent crashing incidents from multiple manufacturers of autonomous vehicles. A critical perspective of these incidents is the environment configuration. Slippery road surface, extreme lighting conditions and other factors interfering with sensor inputs such as fog or rain, all pose challenges to the stability of autonomous vehicle controllers. An assurance framework is thus in need to identify potential configurations of environments that have substantial probability of leading to a failure of autonomous vehicle controllers. Nevertheless, conducting physical tests in these less usual environments to assure the functionality of the controller is greatly expensive and certain traits of the environment are naturally hard to control.

On the other hand, driven by the strong need for better graphics from the gaming entertainment industry, new technologies in both hardware and software have enabled modern games and simulators to deliver next-to-real graphics and lightings as well as complex interactive environments. Increasing interest from academia to utilise these games or simulators for research purposes in computer vision and related fields have been observed. The highly customisable environments offered by games or simulators seem to be a viable testbed for searching the vulnerability of autonomous controllers.

Therefore, we decided to explore the possibility of constructing an assurance framework in simulators for discovering the vulnerability of autonomous controllers. From a set of games and simulators, we choose CARLA for the purpose of this work. CARLA is a relatively new open-source driving simulator that provides a good level of graphics and a comprehensive set of APIs to control the dynamics of actors and the environment in the simulation. We will devise an reinforcement learning agent to learn the dynamics of the target vehicle, under the control of a test AI controller, and explore the most effective changes in environments that would lead to a crashing of the controlled vehicle.
Readers familiar to the field of reinforcement learning will notice that the interesting bit of our work comes from that in classic reinforcement learning problems, an agent is trying to learn the dynamics of an environment, whereas in our case we are learning the dynamics of an agent (the test AI controller) and exerts updates to the environment posing the agent, in searching for an environment configuration that minimises the agent’s reward. In loose words, our reinforcement learning agent could be seen as impersonating the environment, which is interacting with a test AI controller.

3.2 Related Work

Related works aiming to predict the uncertainty in vision system could be found in [7] where a Bayesian SegNet is devised to quantify the uncertainty of segmentation outcomes of input images. The model in [7] could be potentially used to categorise the risk in the environment facing the autonomous vehicle.

In light of predicting the failure of autonomous driving, [6] devised a neural network solution to predict a scene drivability score for every snapshot of the environment captured by the vehicle’s sensors. The solution [6] considers the temporal relations among controls produced by the AI controller within a short period and compares each series of AI controls to the ground-true i.e. controls produced by human drivers to train a set of RNNs.

The scene drivability model in [6] is more of a system that performs supervised learning tasks, that a set of training targets is known to the system and the goal of the system is to produce a classification result.
Also from the field of computer vision, [16] proposed an anticipation framework for traffic accidents by training an RNN with a large set of video clips of traffic accidents that are of various configurations of vehicle size, speed and magnitude of collision. Similar to our work, the algorithm in [16] also involves learning of vehicle dynamics and benchmark the status of vehicle and intention of the driver to average time to collision and produce a classification result.
4 Background

4.1 Reinforcement Learning

In a decision problem concerns an agent and the environment, a reinforcement learning system approaches the problem by defining four major elements: reward function, value function, policy and model of the environment.

A reward function returns an immediate reward to the agent on each time step, or at each state if state-space planning is adopted. The agent’s objective is to choose optimal actions (or find optimal policy) that maximise the overall rewards received over the long-term. Though a reward function could be non-deterministic, we define the reward function to be deterministic in the scope of our problem. A value function, in contrast to reward function, is an estimate of how good or bad a state is in relation to the agent’s objective. Value of a state could be measured by the expected total reward received from all possible trajectories of interactions with the environment, starting from the current state.

A policy defines an agent’s behaviour, i.e. determines which action to play given a time step (or a state). Policy could be seen as a mapping from state to action, though policy often takes the form of stochastic function. Reinforcement learning’s goal is to find the optimal policy that maximises agent’s value at every state.

Last but not least, the model of the environment (or transition model) specifies the dynamics of the environment, i.e. given the current state and the action agent played, what is the next state or what’s the possibility of another state being the next state. In model-based reinforcement learning, the agent learns an approximated model of the environment and performs planning utilising this model. Markov decision processes (MDP) are often used in reinforcement learning to formulate the problem. MDP defines a tuple $<S,A,T,R>$ each corresponds to state space, action space, transition model and reward function. Bellman’s equation provides a recursive solution to the value function of an MDP given a policy:

$$V_\pi(s) = \sum_a (a|s) \sum_{s'} p(s'|s,a)[r(s') + \gamma V_\pi(s')]$$ (1)

And thus the one-step policy improvement by :

$$\pi'(s) = \arg \max_a \sum_{s'} p(s'|s,a)[r(s') + \gamma V_\pi(s')]$$ (2)

Estimation of value function and policy improvement could be conducted in cycles until a convergence is reached, this technique is known as policy iteration [15].
However, as policy iterations operate in sweeps of the entire state-action space, direct application of the method is intractable in infinite (or large) problem space. In our problem, we use Monte Carlo Tree Search (MCTS) for the purpose of planning.

### 4.2 Gaussian Process

Gaussian Processes (GP) is a generalisation of the Gaussian probability distribution, that instead of describing a distribution over random variables, it describes the distribution over functions [12]. Formally, a GP is a collection of random variables that are jointly Gaussian [4].

As a nonparametric method, a GP makes no assumption over the latent transition function that maps a state-action pair to the next state. A GP is defined by a mean function $\mu(\cdot)$ and a covariance function $\Sigma(\cdot, \cdot)$. In practice, a GP is often given a prior mean as zero and a squared exponential (SE) covariance function [4].

$$\Sigma(x, x') = \sigma^2 \exp \left( -\frac{||x - x'||^2}{2\lambda^2} \right)$$

(3)

For simplicity, we denote $x = (s, a)$. The covariance function, also widely referred to as kernel function, incorporates a set of hyper parameters. In the equation of SE kernel function above, $\sigma^2$ denotes the variance of the latent function and $\lambda$ denotes a characteristic length-scale, which controls the distance that the algorithm extrapolates away from the training input [11]. The hyper parameters of the kernel functions could be selected through a number of methods, maximising likelihood is one of the methods that are widely adopted.

The posterior GP $p(s' | x^*)$ gives one-step prediction of transition in regards to an arbitrary test input $x^*$ and the target state $s'_*$. The mean and variance of the this posterior distribution is given by:

$$\mu(s'_*) = \Sigma(x, x^*)\left(\Sigma(x, x) + \sigma_n^2 I\right)^{-1}s$$

(4)

$$\text{cov}(s'_*) = \Sigma(x^*, x^*) - \Sigma(x^*, x)\left(\Sigma(x, x) + \sigma_n^2 I\right)^{-1}\Sigma(x^*, x)$$

(5)

GP is widely considered as the state-of-the-art method in learning stochastic transition models [11].

### 4.3 Monte Carlo Tree Search

Monte Carlo method (MC) has a long history of application in games and Artificial Intelligence, but is known to provide no guarantee on the optimality of action selection despite being executed in many iterations [8].
Coulom [2] proposed a novel that combined MC with tree search, in which random rollouts are executed from the current state to reveal the structural information of the trajectories. The UCT algorithm proposed by [9] was a major breakthrough for MCTS, that the action selection mechanism guided by UCB1 balances exploitation and exploration by account for the estimated value of an action while at the same time penalise actions that have been visited more times than others.

Figure 3: MCTS Process, Source: adopted from [1]

The basic step iterative algorithm of MCTS consists of four steps: selection, expansion, simulation and backpropagation. Selection involves the traversal of the tree from the root to a deepest expandable node following a tree policy that evaluates how good a node is, one prominent example being the UCB1 strategy adopted in UCT which we will cover in detail in Section 5.4. Expansion adds a new node to the new while simulation conducts rollout using a default policy to estimate the value of the new node, the estimate value is then used to update the new node’s parent and starts a chain of update all the way to the root, which is knowns as backpropagation. The MCST algorithm comes with great flexibility that different choices of tree policy, default policy and order of accessing unvisited actions during expansion will construct varied features in the resulting algorithm. Out of the many derivations of MCTS algorithms, UCT is particularly important as it provides theoretical guarantee on the convergence to the optimal given sufficient time [8]. Our experiments also leverage on a modified version of UCT as discussed in 5.4.
5 Method

5.1 Top-Level Algorithm

Our reinforcement learning system is realised through a combination of offline model learning via PILCO Gaussian Processes (PILCO GPs) as outlined by [3] and online planning with ρUCT, proposed by [17]. At a high level, we conduct model learning activities by gathering a large set of episodes from CARLA Simulator using an arbitrary policy, PILCO GPs are optimised offline using these data. Once we obtained a good confidence in prediction accuracy of PILCO GPs, we conduct online planning by starting a new episode and building up a ρUCT at each state. Building up of ρUCT is as per described in Section 5.4 through interaction with the approximated PILCO GPs transition model. Note that, the UCB selection policy employed in ρUCT is still possible to recommend an explorative action depending on the outcome of playouts [17].

Figure 4: Algorithm overview, Source: ρUCT adopted from [17]

The top level algorithm works as illustrated in above. It is worth pointing out that we do not follow the Dyna model [15] that are often followed in reinforcement learning solutions, where new experience obtained from the planning phase is fed back to the model learning process to further improve the approximate transition model. The major reason is that optimisation of PILCO GPs is significantly time-consuming with a large set of experience. Thus, interleaving online planning in decision time with model learning would make the solution time intractable under the computation resource we have.
**Algorithm 1: Top-level Algorithm**

**Result:**
initialise CARLA environment;

**while** experience is less than target training set size + target test set size **do**

- gather experience data use policy in Section 5.3.3;
- reset CARLA environment with random weather conditions;

**end**

divide training set and test set;

offline model learning as in Algorithm 2;

decision-time planning as in Algorithm 3

### 5.2 Formulation of Problem

We borrow the classic MDP problem definition of a tuple $< S, A, T, R >$, where $S$ is the set of states and is further defined as: $S = < C, P, W >$

$C = < \text{Position}, \text{Velocity}, \text{Acceleration} >$

$P = < \text{Position}, \text{Velocity}, \text{Acceleration} >$

$W = < \text{Cloudiness}, \text{Precipitation}, \text{RainDeposit}, \text{SunAzimuth}, \text{SunAltitude} >$

$A$ is the set of actions which is defined by a change in $W$. System dynamics is represented by transition model $T$ that $T(s, a, s') = p(s' | s, a)$. The agent’s goal is to choose a series of actions so that the total rewards are maximised and since the length of episodes is finite, we omit the discounting factor.

Since variables in state space are of float type and have infinite domain space, we can’t develop an explicit representation of transitions from each state-action pair. Instead, we devise an non-parameterised statistical model to approximate the real transition function. In Section 5.3, we show the detailed definition of the model and how could it be used as a generative model to sample outcomes of the approximated latent transition function.

At each state, the agent receives an immediate reward defined by $R(s)$. A policy $\pi = s \rightarrow a$ assigns an action $a$ to each state $s$. The value function at each state $s$ is thus defined as $V_\pi(s) = E[\sum_{t=0}^{k} R(s_t)]$ which specifies the total expected reward under policy at each state $s$. Due to the infinite state space we won’t be able to obtain an explicit representation of the policy, instead we devised a derivation of Monte-Carlo Tree Search to produce an online mapping of state and action whose detailed implementation will be provided in Section 5.4. A generative model that approximates the transition function $T$ is needed for this planning process.

Once we have a representation of policy $\pi$ the agent selects action given by $a = \pi(s)$ repeatedly until it reaches a final state.
5.3 Offline Model Learning

5.3.1 PILCO Gaussian Processes

Probabilistic Inference for Learning Control (PILCO) is a model-based policy search algorithm proposed by [3] and has demonstrated unprecedented speed of learning for robotics control [4]. We adopt the method PILCO follows in model learning to utilise its data efficiency.

Transition model in PILCO is implemented as a GP, or a set of conditionally independent GPs if the target space is multivariate. PILCO GPs define the training inputs as tuples of state and action \((s, a)\) and the training target as differences between states before and after transition \(\delta_s = s - s'\). A square exponential kernel is defined to describe the covariance of the random variables.

\[
\Sigma(x, x') = \sigma^2 \exp \left(-\frac{||x - x'||^2}{2\lambda^2}\right)
\]  

Variance of the latent function is denoted as \(\sigma^2\) and the diagonal length-scales characteristics matrix \(\lambda^2\) are fit by maximising the marginal likelihood using conjugate gradient [3]. Posterior distribution of the latent function \(p(\Delta_s|x_*)\) given an input \(x_*\) is completely defined by mean and variance. Given \(n\) training input \(X = [x_1, ..., x_n]\) and corresponding training targets \(y = [\Delta_1, ..., \Delta_n]\)

\[
\mu(\Delta_s) = \Sigma(x, x^*) \left(\Sigma(x, x) + \sigma^2_n I\right)^{-1} y
\]  

\[
\sigma^2(\Delta_s) = \Sigma(x^*, x^*) - \Sigma(x^*, x) \left(\Sigma(x, x) + \sigma^2_n I\right)^{-1} \Sigma(x^*, x)
\]
5.3.2 Sample from Latent Function

Once we obtained a PILCO GPs with parameters optimised over some input $X_*$, we compute the mean and variance of the posterior distribution $p(\delta_*|x_*)$ as outlined in Section 5.3.1. Plugin the mean and variance to a normal distribution allows us to sample state transitions. In our software, we leverage on Rontsis and Polymenakos’s [13] implementation of PILCO with GPflow for the construction of GPs. Sampling from posterior GPs is thus simplified to invocation on GPflow’s built-in routines: `predict_f_samples` for one-step sampling directly from the posterior of the latent function, or `predict_f` which computes the mean and variance of the posterior distribution.

5.3.3 Data Gathering Policy

As a sufficiently large set of training input $X_*$ is needed for optimising PILCO GPs, a dedicated workflow is conducted to interact with CARLA for gathering such data. We devise an arbitrary policy that produces actions that modifies the Sun Altitude weather parameter by sampling from a Bernoulli Process that gives the probability of two actions: increase Sun Altitude by 2, or decrease it by 2.

$$P(A = 2) = p = 0.8, P(A = -2) = 1 - p$$  \hspace{1cm} (9)

To ensure a good coverage on the domain of state-action space, after RESET the CARLA environment at the end of each episode run with the data gathering policy, we randomly initiate the Sun Altitude using a uniform distribution over its range of [-90, 90].

5.3.4 Offline Model Learning Algorithm

The overall algorithm of our offline model learning component consists of two parts:

1. collection of $X_*$ and $Y_*$ using data gathering policy

2. batch-based optimisation of PILCO GPs with random restarts and hill-climbing.

Optionally, we also perform tests on the prediction accuracy of PILCO GPs after completion of each training batch.
Algorithm 2: Batch Optimisation of PILCO GPs

Result: Optimised hyper parameters of GPs

initialise X, Y to empty array;

while $i$ is less than target batches do
    load training input $x$, training target $y$ with length of batch size from data;
    add $x$ to X;
    add $y$ to Y;
    best parameters = optimise by maximising likelihood with X, Y;
    while $j$ is less than number of restarts do
        new parameters = randomise;
        new parameters = optimise by maximising likelihood with X, Y;
        if new parameters better than best parameters then
            best parameters = new parameters;
            break;
        end
    end
end
5.4 Online Planning

5.4.1 ρUCT

The ρUCT method defines a tree with two types of interleaved nodes: decision node which represents the state, and chance node which represents an action. Each node in the tree contains a count of visits and the estimation of expected future reward i.e. $V(s)$ for decision nodes and $Q(s, a)$ for chance nodes.

![Structure of ρUCT](image)

ρUCT’s algorithm reflects the classic four phases in other MCTS methods [17]. The selection phase starts from the root decision node $r$ and traverses to a leaf chance node $n$ by applying the selection strategy (tree policy) and the generative model (transition model) in turn. In particular, a selection strategy returns the best child action, represented by a chance node, of the current decision node. In the following expansion phase, a new decision node $v$ is sampled from the generative model and added to the selected chance node $n$. Next, in the simulation stage, a rollout strategy (policy) is utilised and, together with the generative model, samples a trajectory of transitions until reaching the final state or hitting the horizon limit and produces an estimated reward. Finally, in the backpropagation stage, the value estimated in $v$ is used to update the estimated future reward of $n$, and hence triggers a path of recursive updates until the root node $r$.

UCB1 is used by ρUCT as the selection strategy, mainly aims to leverage on the strategy’s capability of providing a good balance between exploration and exploitation. We adopt the same definition of UCB1 with [17] as listed below:

\[
\frac{1}{m(\beta - \alpha)} Q(s, a) + C\sqrt{\frac{\log(H(s))}{H(sa)}}
\]  

(10)

In the equation above, $m$ denotes the remaining planning horizon and $C$ denotes a constant.
factor that we choose to balance exploration and exploitation. \(H(n)\) denotes the number of visits received by an arbitrary node \(n\).

The overall algorithm of our implementation for adopted from [17] is listed below:

**Algorithm 3: \(\rho\)UCT Planning**

Result: recommended action \(a\)

initialise UCT node \(r\);

while \(i < \text{rollouts}\) do
    sample as in Algorithm 4
end

return children of \(r\) with highest estimated value;

The recursive process of sampling is implemented as:

**Algorithm 4: \(\rho\)UCT Sample**

Result:

initialise reward to 0;

if \(\text{horizon equals 0 OR is final state}\) then
    return reward;
end

if \(\text{is decision node}\) then
    if \(\text{not visited}\) then
        reward = heuristics value;
    else
        select chance node \(A\) from action list using UCB as per defined above;
        reward = UCT Sample(\(A\), Horizon-1);
    end
else
    decision node \(S\) = sample from transition model;
    add \(S\) to children of \(A\) if not existing;
    reward = discount factor * UCT Sample(\(S\), Horizon-1) - step penalty;
end

mean reward = \((\text{reward} + \text{mean reward} \times \text{visits}) / (\text{visits} + 1)\) visits += 1;

### 5.4.2 Model Error and Heuristics

Although UCB1 is largely considered capable of consistently approximating the optimal policy by running a sufficiently large number of samples and given that the model is perfect [8]. In practice, we wouldn’t expect the model learned to be perfect and certain degree of error is inevitable. With the presence of model error, the planning process could compute a suboptimal policy. Although it is argued that when the approximate model is optimistic, that overstates the possible reward or predicts transitions that are not possible, it could be quickly rectified when the policy tries to exploit such advantages [15], such conditions do
not apply to our setup. There is no guarantee that the model resulted from our learning process is overall optimistic and furthermore, due to the offline nature of model learning, we do not incorporate the decision-time feedback to model, thus making the error correction impossible.

Hence, other than using rollouts to estimate future rewards at each decision node, we could also leverage on heuristics to reduce the impact of model error by introducing bias into our system. From our early observations, when the vehicle crashes into the pedestrian, sun altitude tends to close to zero i.e. the sun is barely above the horizon. Thus, we compute a linear regression relationship between sun altitude and the position of vehicle for episodes of crashing, and use this linear model to estimate the heuristics value for a state.

5.5 Software Architecture

Our software implementation is separated into two spaces: CARLA space and Agent space, which will be run in different processes with independent runtime configuration. We define a communication endpoint in each space respective - in CARLA space it is CARLA client and in Agent space it is CARLA interface.

Segregation of the processes allows us to decouple our agent with CARLA and the AI controller under test, thus providing our system the flexibility of swapping out a different AI controller for assurance purposes or even swapping out CARLA as whole. In the latter case, the only change we need to make is the interface component of the Agent space. Such flexibility is critical to our solution, since our extended goal is to devise a general framework for assurance of self-driving vehicle controllers, which should not be bound by a particular choice of AI controller nor a particular simulation technology.

5.5.1 Agent Space

In the Agent space a CARLA interface is devised to provide a level of abstraction of interactions with the CARLA space and converts observations from CARLA to the agent’s environment, which gives the definition of state and actions. ZeroMQ is used for communication between the CARLA client in the CARLA space and the CARLA interface in the
Agent space. ZeroMQ provides light-weight APIs and convenient json messaging interfaces. The CARLA interface also persists state-action pairs produced during interactions between the agent and CARLA in a shared file storage.

Two types of messages are defined:

1. **RESET**, which reconfigures all actors in CARLA runtime to the initial position and resets weather parameters that correspond to initial state defined by the agent.
2. **ACTION**, which causes a change in weather parameters as defined by the agent’s action.

The communication between CARLA client and CARLA interface is synchronised, that CARLA client will not generate the next frame until a message from the agent has been received. This guarantees the task environment to be static.

Frame rate of the CARLA client is fixed at 20 per second. Though the rate is chosen mainly to accommodate the computing capacity of experiment infrastructure, a fixed frame rate is necessary to reduce the difficulty of learning the transition model.

The workflow of offline model learning and online planning is organised as per the top level algorithm defined in Section 5.1.

### 5.6 CARLA Space

Though CARLA provides a set of Python APIs that allows convenient extraction of runtime information, the AI vehicle controller under experiment may have an entirely different set of Python dependencies compared to the Agent space. Hence, the CARLA client serves as a facade that abstracts functionalities in the CARLA space and exposes a communication endpoint. CARLA client operates in a “server mode” in relation to the Agent space that it is the receiving end of network connection and executes in an event loop reacting to messages from the Agent space.
Algorithm 5: CARLA Client Algorithm

Result:
start ZeroMQ server;
connect to CARLA runtime;
initialise CARLA map, actors and weather parameters;
load test AI controller;

while true do
    message = blocking receive message from Agent;
    if message is RESET then
        reinitialise CARLA actors and weather parameters;
        send observations to Agent;
    end
    if message is ACTION then
        apply action as per message;
        apply control from test AI controller;
        send observations to Agent;
    end
end

CARLA client defines handling functions for RESET and ACTION messages respectively. On RESET, the client signals CARLA runtime to clear all actors in the map, reset all weather parameters and recreate actors including vehicle and pedestrian to ensure every episode starts in the same initial state with the exception of data gathering stage for offline learning, where we start with random weather parameters to improve coverage of the data as described in Section 5.3.3.

5.7 Data Processing

Other than converting raw observations from CARLA to our definition of state space, we also perform normalisation of state information. The importance of normalisation is twofold: firstly, a similar order of magnitude provides numerical stability in Cholesky decomposition that is frequently used in our dealing with GPs. Secondly, we do not want the difference in magnitude impacts our learning of the transition model.

We perform normalisation by applying division of the raw data over its range, thus forcing the data into a range of 0 to 1. Data whose domain includes a range in negative numbers will be added an offset to shift its range to non-negative before the division.

We define the bidirectional conversion between normalised state-action and raw state-action. As raw data are more human readable, we retain the knowledge of state-action in raw format for the purpose of information displaying and configuration of hyper parameters or constants.
6 Results

6.1 Experiment Setup

We conduct a baseline experiment to validate the effectiveness of our solution as an assurance framework for self-driving vehicle controllers. For simplicity, we set up the CARLA environment with a single vehicle moving towards a destination in a straight lane and a pedestrian that is placed in the path of the vehicle and stands still, which simulates the dynamics between a moving car and a pedestrian who is crossing the street. Furthermore, out of the 6 weather parameters configurable in CARLA, in this baseline experiment we will only include Sun Altitude in our state-action space given its biggest impact on the lighting condition.

This baseline scenario, although simple enough in number of actors involved, adequately captures the most important factors in a crash accident of self-driving vehicles, of which lighting conditions seem to be the most important influence [18].

Since the vehicle is moving on a straight lane, dimensions of Position, Velocity and Acceleration could be reduced to 1. Thus, together with reduced weather parameters, the dimensions of our state space is simplified to 4.

\[
P = \langle \text{Position}, \text{Velocity}, \text{Acceleration} \rangle
\]

\[
W = \langle \text{SunAltitude} \rangle
\]

With such configuration in CARLA, we gathered 200 episodes of data using the algorithm described in Section 5.3.3 and performed offline model learning. The resulted PILCO GPs are then utilised in decision-time planning with \( \rho \text{UCT} \), final reward of each episode produced through the interaction between CARLA and \( \rho \text{UCT} \) planning is collected and analysed in later sections.

In the experiments we define reward functions as below:

1. vehicle crash into pedestrian: 10
2. step penalty: -1

Our experiments are conducted in a Dell Precision 3630 Desktop Tower, with an Intel(R) Core(TM) i7-8700 CPU of six 3.20GHz cores and an NVIDIA GTX 1060 GPU of 6GB graphics memory.
6.2 Model Learning

PILCO GPs trained with a varied number of training sets (or episodes) are validated using 10 episodes of test sets, where each state-action in the episodes are fed into the transition model to produce one-step prediction of change in state, which is then compared with actual change in state to measure the prediction accuracy.

The average percentage of error in prediction is used as a metric to measure the performance of each configuration of PILCO GPs. \( s_i \) denotes the \( i \)th variable in state \( s \).

\[
M_i = E\left[\frac{|s_i - \hat{s}_i|}{s_i}\right]
\]

We observe that although prediction accuracy is increasing along with more training inputs, our learning method seems to be more effective in learning the dynamics of position compared with that of velocity. We are able to reduce the prediction error to about 5% for change in position after 200 episodes of training input whereas for velocity the error percentage remains higher than 40%.

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Error in Position</th>
<th>Error in Velocity</th>
<th>Error in Sun Altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12.70%</td>
<td>54.81%</td>
<td>26.46%</td>
</tr>
<tr>
<td>50</td>
<td>7.66%</td>
<td>63.16%</td>
<td>12.57%</td>
</tr>
<tr>
<td>100</td>
<td>5.92%</td>
<td>46.78%</td>
<td>9.73%</td>
</tr>
<tr>
<td>200</td>
<td>5.22%</td>
<td>43.49%</td>
<td>8.60%</td>
</tr>
</tbody>
</table>

Table 1: Prediction Accuracy Results

It might be premature to conclude that our PILCO GPs approach is inadequate in learning the dynamics of velocity, as despite a trend of convergence after 200 episodes of training as evidenced by diminishing magnitude of improvements, there is no clear sign that what we have obtained so far is close enough to the upper bound.

Nevertheless, even though further training with more than 200 episodes of input could potentially increase our prediction capability, it may not be economically worthwhile both in terms of training time and sampling time.

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Optimisation Time (seconds)</th>
<th>Sample Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.03</td>
<td>0.017</td>
</tr>
<tr>
<td>50</td>
<td>60.94</td>
<td>0.23</td>
</tr>
<tr>
<td>100</td>
<td>1332.06</td>
<td>1.40</td>
</tr>
<tr>
<td>200</td>
<td>10504.21</td>
<td>8.13</td>
</tr>
</tbody>
</table>

Table 2: Model Learning Time

We have observed super-linearly increase in both optimisation time for maximising likelihood in PILCO GPs and sample time for the posterior distribution to produce a sample in
state change. Notably, with 200 episodes of training input each optimisation takes longer than 10,000 seconds and each sampling takes roughly 8 seconds, which makes planning using this configuration of PILCO GPs intractable. The configurations of PILCO GPs with 50 or 100 training sets instead seem to be economic in computation and provide prediction accuracy that is not far from the 200 episodes configuration.

What we are more interested in is the capability of our approximate transition model to perform multiple-step predictions, which is directly impacting the quality of search by ρUCT, as a high error would increase the chance of overvaluing or undervaluing a possible action.

Figure 8: Multiple-step prediction results using PILCO-GPs with 50 training sets

As shown in Figure 8, our prediction starts to diverge from the actual change in state after 10 steps of transitions and the error becomes significant with a horizon higher than 15. Thus, planning results using PILCO-GPs of 50 training sets after horizon 15 becomes of less credence.
6.3 Planning

To measure our solution’s capability in decision-time planning, we conduct 10 episodes of experiments by running MCTS over $\rho$UCT with various configurations of PILCO GPs. The same set of hyper parameters for MCTS are applied in the experiments.

<table>
<thead>
<tr>
<th>Number of Samples per Step</th>
<th>Horizon</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3: Hyper Parameters for MCTS

Final reward for each episode of MCTS against a configuration of PILCO GPs is collected. We compute the average of the final reward for each configuration of PILCO GPs to compare the impact of the approximate transition model in our planning capability.

Figure 9: Decision-time Planning Results

We observe that PILCO GPs of higher prediction accuracy indeed increases the planning capacity of our $\rho$UCT, as evidenced by the increased average final reward from 0.1 to 1.2 for the group of 10 training sets and 50 training sets. However, the best result of 1.2 average final reward that we have achieved so far is still having a significant gap from the theoretical optimal reward of 10. Model error still seems to be a limiting factor to our planning capability.
6.4 Discussion

6.4.1 Assurance Discoveries

One major observation we draw from cases where vehicles crashing into pedestrians happens is that sun altitude in these scenarios are low or close to zero. Lighting conditions under these scenarios could be much less ample compared with during daytime. The AI controller under test seems to be less effective in dealing with such lighting conditions and hence produce hazardous controls. It could be worthwhile to test these environment configurations against other AI controllers to examine whether our observation could be generalised and thus of more value to be considered as a checkpoint in an assurance framework.

Figure 10: Failure happens when sun altitude is low
7 Conclusion

7.1 Conclusion

In this work we have proposed an reinforcement learning system that is capable of systematically identifying the weakness of autonomous controllers through configuring the simulated environment which lead to the latter producing hazardous controls. Our work serves as the first step to the construction of an assurance framework for general autonomous controllers.

From the results, our reinforcement learning agent is capable of learning, to a certain degree, the dynamics of a vehicle controlled by an arbitrary autonomous controller under an environment whose conditions are subject to changes. Exploiting these learned dynamics, we utilise planning to find the most possible environment changes that could lead to failure of the autonomous controller under test. The results indicate that sun altitude is an important influence factor for the stability and consistency of the test autonomous controller in that certain configurations of sun altitude produce lighting conditions that cause more frequent failure of the controller. Moreover, as we do not make any assumption of the controller under test, our framework could apply to potentially any autonomous controller without change in formulation of the problem.

In the experiments we have only considered the effect of changing sun altitude, but the framework we proposed certainly does not limit future researchers to include other environment parameters into the state-action space and explore the impact of change in these parameters over the stability of autonomous controllers.

In this work we have not included a way of using new experiences obtained during the planning time to update the transition model mainly due to efficiency concerns. Model errors from offline learning are rather mitigated by introducing heuristics at planning time. Future researchers could look into a different GP implementation or a different model learning methods that potentially allows the feedback mechanism. Overall, we conclude that the framework we proposed has great potential in ultimately leading to a general assurance framework for autonomous controllers. Our design provides the flexibility of switching out any component in the system, being it the simulator, controller under test or methods used for model learning and planning. Future researchers using this framework could explore a different implementation for any of these components in seeking of a better outcome.
7.2 Future Work

7.2.1 Prediction

One of the bottlenecks we face is the pro-longed training time and sampling time of PILCO GPs. The PILCO implementation we leveraged on in this experiment utilise GPflow, an open source package for Gaussian Processes in Python. The shortcomings of the package that ultimately give rise to long training time and sampling time are that:

1. optimisation of the GP’s hyper parameters are not done in incremental-basis
2. intermediate results of sampling from the posterior distribution are not cached

thus every sampling requires a Cholesky decomposition of the covariance matrix. Future researchers may look into devising custom implementation of Gaussian Processes to overcome these shortcomings.

7.2.2 Scenario Simplicity

Although our baseline scenario has captured major factors that influence a self-driving vehicle to crash on pedestrians, we make a compromise in trade for simplicity that the pedestrian is standing still during the entire course of the episode. In reality, pedestrians crossing the road are rarely standing still in face of an approaching vehicle. Future researchers might want to incorporate moving pedestrians into the model and account for the complexity introduced by the trajectories of pedestrians and their impact over the dynamics of the vehicle.
References


8 Appendix 1: Study Contract
INDEPENDENT STUDY CONTRACT
PROJECTS

Note: Enrolment is subject to approval by the course convenor

SECTION A (Students and Supervisors)

UniID: 6646917
SURNAME: QUAN
FIRST NAMES: KEE
PROJECT SUPERVISOR (may be external): Hanna Kurniawati
FORMAL SUPERVISOR (if different, must be an RSSCS academic): Hanna Kurniawati
COURSE CODE, TITLE AND UNITS: COMP8755, Individual Computing Project, 12 Units

COMMENCING SEMESTER □ S1 ☑ S2 ☑ YEAR: 2019 Two-semester project (12u courses only): ☑

PROJECT TITLE:
Assurance of Self-driving Cars

LEARNING OBJECTIVES:
Explore alternatives and scalable approach for detecting sequence of inputs that are likely to cause catastrophic issue in self-driving cars

PROJECT DESCRIPTION:
Unexpected response to rarely seen sensor inputs have caused catastrophic (driverless) car crash. Could we design an automatic testing mechanism to reduce the chances of such potentially disastrous behaviors from occurring? To start answering this question, we will explore a simple planning and learning approach to automatically and efficiently generate simulated sensor inputs that are likely to trigger undesirable behavior.

The project aims to build a software system that utilise planning to generate sequences of inputs that resembles real-life error-leading sensor inputs. The first goal is to generate simple inputs that encoded in real number, an extended goal would be to generate graphics. Reinforcement learning could also be utilised in the learning part of signal generation.
ASSESSMENT (as per the project course’s rules web page, with any differences noted below).

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: style: research report (e.g. research report, software description...)</td>
<td>50</td>
<td>24/5/20</td>
<td>Awe Tiu</td>
</tr>
<tr>
<td>Artefact: kind: software &amp; exp. result (e.g. software, user interface, robot...)</td>
<td>40</td>
<td>24/5/20</td>
<td>Hanna Kurniawijati</td>
</tr>
<tr>
<td>Presentation:</td>
<td>10</td>
<td>24/5/20</td>
<td>course convenor</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

STUDENT DECLARATION: I agree to fulfil the above defined contract:

[Signature] 26/7/2019

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project. I nominate the following examiner, and have obtained their consent to review the report (via signature below or attached email)

[Signature Hanna Kurniawijati] 24/7/2019

Examiner: [Awe Tiu]  
Name: [Awe Tiu]  
(Nominated examiners may be subject to change on request by the supervisor or course convenor)

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course convenor approval)

[Signature] 26/7/19

Research School of Computer Science  
Form updated Jun 2018
9 Appendix 2: Software Details

Our implementation of the reinforcement learning agent utilises the work of [13], a python implementation of PILCO, mainly for the purpose of construction a set of GPs and as an interface to update as well as optimise the learned models. Most of times we use this library solely as a wrapper of the underlying open-source library GPflow, which introduces the actual implementation of GPs in Python.

For ease of implementation, we started our work from a fork of [13]’s repository and we organised our original work in a separated top-level folder mc_sda_pilco.

Below files and folders are present in the mc_sda_pilco root directory:

1. carla_interface.py, the implementation of CARLA interface as described in Section 5.1.
2. environment.py, an abstraction of the CARLA environment.
3. mct.py, the implementation of ρUCT.
4. pilco_gp.py, our interface to PILCO GPs.
5. sda.py, implementation of the agent as described in Section 5.1.
6. util.py, defines various util functions.
7. carla, includes the CARLA interface carla_controller.py and test_client.py which incorporate the data gathering policy and writes received observations to files. carla_controller.py is adopted on the basis of a test script provided by CARLA’s Python distribution as mentioned in the file’s disclaimer.
8. output, logs, figures are temporary folders that our program will write runtime data to, including dump of learned model, system logs, figures for performance etc.
9. data folder contains rather long-life files that we think will be helpful for users, including a list of pre-trained model dumps under data model and sample sets of state-action data under data training_set as well as data test_set.

For the purpose of integration test, we have devised a test mode in the program, which loads a model dump from the data model folder and executes a series of prediction using test sets under data test,et, a successful running should resemble what we have presented in Section 1.

Details of experiments could be found in Section 6.
10 Appendix 3: Software Manual
Disclaimer

Our original work in this program including source code and data files are all under mc_sda_pilco.

Files under examples, pilco, tests are of the credit of Nikitas Rontsis and Kyriakos Polymenakos, who authored the Python implementation of PILCO. Our work in this project builds on top of the fork of their GitHub repository: https://github.com/hrontsis/PILCO

Ke Quan, 12 June 2020

Dependencies:

python 3.7.1
numpy 1.15.4
gpflow 1.5.1
tensorflow 1.13.1
pyzmq 17.1.2
gym 0.15.4
mujoco-py (optional) 2.0.2.9

Build

1. make sure above dependencies are in place.
2. pilco to be build as per README_PILCO.md

File Structure

mc_sda_pilco/carla/carla_controller.py is the CARLA client resides in CARLA space. It is wired up with synchronised MQ communication to ensure each step of rendering in CARLA is only done when action is received from sda_client.

mc_sda_pilco/carla/test_client.py is a script embeded with Data Gathering Policy - a random policy that either increase or decrease SunAltitude in each step, with a bias towards decrease.

mc_sda_pilco/carla_client.py is the CARLA interface defined in Agent space, invoked by the agent implementation sda.py.

mc_sda_pilco/sda.py implements the reinforcement learning agent that described in the report.

mc_sda_pilco/pilco_gp.py is a wrapper of PILCO's MGPR package, which essentially constructs multiple GPs.

mc_sda_pilco/environment.py abstracts the CARLA environment for agent's internal operation, mainly for planning. Reward function is defined in this class.

mc_sda_pilco/mct.py is our implementation of rho-UCT.

mc_sda_pilco/util.py collects multiple util function that we used for file I/O, conversion of data types etc.

Other than source code listed above, following folder in mc_sda_pilco stores run-time or design-time files that are important to the application:

1. data/models stores a list of pre-trained dumps of PILCO GPs. When working in online planning mode (-m 3) the program will load a model dump as specified by -l model_dump_name and use that for planning.
2. data/training_set contains state.txt and action.txt which contains 200 episodes of data (each episode has about 50 state-action pairs) that could be used for offline model learning
3. data/test_set contains a smaller set of data (about 20 episodes) for verification the prediction accuracy of model trained using training data
4. logs, figures and output are folder where runtime information are dumped into. It's worth noting that model parameters learned during offline model learning (-m 2) are dumped into output folder.
**Introduction**

The main class or entry point of our program is `sda.py` which accepts a list of arguments:

- `-m` the mode which application shall run. There are essentially 3 modes supported in our program:
  1. `-m 1` runs a benchmark test, which loads a pre-trained model as specified by `-l model_dump_name` and performs a batch of prediction accuracy test against test data
  2. `-m 2` is the model for offline model learning. At high level, it works in batches and for each batch it loads a set of data from training files and optimise the GPs parameters. Model dumps are created and write to `output` folder after completion of each batch.
  3. `-m 3` is the mode for online planning, which load a model dump from `data/models` and initialise a rho-UCT as defined in `mct.py` to interact with the CARLA environment in decision-time.

**Data Gathering**

To run the data gathering policy, we need to copy the `carla_controller.py` to `Carla_dev/Carla 0.9.6/Python API/examples`. Make sure the line 384 is uncommented, which gives randomised sun altitude for better coverage on model learning. Once these are done follow below steps to gather training/test data:

1. launch the Carla server
2. in a separate terminal, activate conda environment `CAL_latest`, go to `Carla_dev/Carla 0.9.6/Python API/examples` then python `carla_controller.py`
3. in `mc_sda_pilco/carla` run python `test_client.py`

Change the hyper parameters defined in `python test_client.py` for number of episodes.

**Offline Model Learning**

Once we have the training data and test data ready under `mc_sda_pilco/data` we could run the Offline Model Learning mode to train up a PILCO GPs model.

```
python sda.py -m 2 -r 50 -b 10 -t 5 -d true
```

Execute above will tell our agent to run offline model learning by loading `-r 50` episodes from training data, break them down into 5 batches given batch size `-b 10` and after each batch, we test the prediction accuracy by compare our one-step prediction results with `-t 5` test sets. `-d true` specifies that our agent should dump the learned model parameters after completion of each batch. These files will be dumped into `output` folder.

**Planning**

To perform decision time planning, the procedure is similar to Data Gathering. But we need to make sure `carla_controller.py` configures the same starting weather parameters on restart, check line 384 for mode details.

1. launch the Carla server
2. in a separate terminal, activate conda environment `CAL_latest`, go to `Carla_dev/Carla 0.9.6/Python API/examples` then python `carla_controller.py`
3. in `mc_sda_pilco` run python `sda.py -m 3 -l model_dump_name`

As our connection with `carla_controller` is anytime, for start a new planning we simply just need to rerun `<sda.py` it will send message to `carla_controller` to reset the environment.

**Running Test**

For purely testing the prediction accuracy of learned models, use:

1. in another terminal, go to `mc_sda_pilco` make sure you have all the dependencies for our program, execute python `sda.py -m 1 -l name_of_model_dump -t 5`