Programming Support for Autonomizing Software

Wen-Chuan Lee
Purdue University
USA

Peng Liu
IBM Research
USA

Yingqi Liu
Purdue University
USA

Shiqing Ma
Purdue University
USA

Xiangyu Zhang
Purdue University
USA

Abstract

Most traditional software systems are not built with the artificial intelligence support (AI) in mind. Among them, some may require human interventions to operate, e.g., the manual specification of the parameters in the data processing programs, or otherwise, would behave poorly. We propose a novel framework called Autonomizer to autonomize these systems by installing the AI into the traditional programs. Autonomizer is general so it can be applied to many real-world applications. We provide the primitives and the runtime support, where the primitives abstract common tasks of autonomization and the runtime support realizes them transparently. With the support of Autonomizer, the users can gain the AI support with little engineering efforts. Like many other AI applications, the challenge lies in the feature selection, which we address by proposing multiple automated strategies based on the program analysis. Our experiment results on nine real-world applications show that the autonomization only requires adding a few lines to the source code. Besides, for the data-processing programs, Autonomizer improves the output quality by 161% on average over the default settings. For the interactive programs such as game/driving, Autonomizer achieves higher success rate with lower training time than existing autonomized programs.

CCS Concepts • **Software and its engineering → Frameworks**; **Runtime environments**.

Keywords AI; Deep Learning; Software Autonomization; Dynamic Program Analysis

ACM Reference Format:
Wen-Chuan Lee, Peng Liu, Yingqi Liu, Shiqing Ma, and Xiangyu Zhang. 2019. Programming Support for Autonomizing Software. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314593

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314593

1 Introduction

Autonomous software systems have achieved incredible success in *specialized* domains. For example, the world is shocked when the self-learning AlphaGo program defeated the human champions [1–4]. The Waymo self-driving car has driven flawlessly 25,000 miles each day on complex city streets [5], which is valued at around $175B [6]. Inspired by the existing autonomous systems, we are intrigued by the question whether the *general* software engineering problems can benefit from the autonomization. In this work, we take the initiative to answer this question and share the results.

1.1 Autonomization: Bringing the Intelligence to Traditional Programs

In the following, we present two sets of general software engineering problems, which are representative of the problems that the autonomization potentially applies to.

Parameterized Programs. Many traditional software systems, especially data processing, machine learning and scientific computation programs, often carry the parameters that affect the quality of the results. However, different inputs require different configurations to achieve the ideal results, i.e., no parameter configuration universally applies. Therefore, the users need to manually configure the parameters, which is difficult for normal users due to the great domain expertise required and sometimes even difficult for the experts if the parameter value space is huge (consider the neural network hyperparameters [7]). To mitigate this problem, the users may use the autotuning tools [8] to tune the configuration. In either case, the users are faced with the dilemma that they either need to tune the parameters for each input, which prohibits the application to large volume of inputs, or have to tolerate the unsatisfactory results.

Artificial intelligence, specifically the supervised learning (SL), excels at learning the target values (i.e., the ideal parameter values) for different settings (i.e., inputs). According to the experiments (Section 6), the parameter values predicted by SL improve the quality of results over the default values by around 70% for Canny, a popular edge detection program.

Besides, the prediction of SL is very fast, in contrast with the manual specification or the autotuning, which means the programs equipped with SL can process large volume of inputs fast while offering good results. Therefore, we propose to install AI into the traditional parameterized Programs.

Interactive Programs. Many software systems interact with the environments, e.g., the monkey testing software which interacts with the mobile UI environment [9], the Mario game agent that interacts with the simulated game environment [10], the software that controls the cooling of the data center [11] and many other cyberphysical system (CPS) software. However, these pieces of software usually follow the random behaviors [9] or the simple heuristics [11], which do not behave effectively in practice.

We propose to install the AI, in particular the reinforcement learning (RL) which is designed for the action selection, into the software systems such that they behave smart and achieve better results, while offering the full automation. Deepmind researchers pioneer the study: They reduced Google's data centre cooling bill by 40% by leveraging the AI in place of the simple heuristic-based control [11]. Earlier, they built the control through deep reinforcement learning and demonstrated it can surpass the human-level performance on a set of Atari games [12]. Another work integrated RL with the monkey testing and shown the effectiveness [13].

1.2 Problems and Challenges

We discuss potential problems and challenges of program autonomization in the following.

Autonomization is Tedious and Not Portable. In existing autonomous systems [1, 10–19], autonomization is implemented manually, which requires a lot of engineering efforts. In general, it needs to construct the neural network model and select the feature variables, of which the runtime values are used as the inputs of the model. Besides, at runtime, it needs to (1) collect the feature values, (2) save them to database and load them back in batches, (3) feed them to the model for training and prediction, (4) integrate the predicted result into the execution, and (5) provide program checkpointing/restore logic when the training (esp., reinforcement learning) enters the ending state. Even worse, when moving to a new application, the above process needs to be repeated.

Challenge on Feature Extraction. Similar to many other neural network applications, the crucial challenge lies in the feature selection, i.e., selection of the feature variables for predicting the target variables specified by the users. The target variables are the variables of which the values are to be predicted and affect the quality of results, e.g., the parameters in the parameterized programs or the actions in the interactive programs.

Existing works usually use the raw program inputs (e.g., images) as the feature variables. For example, the autonomous game agent work [12] uses raw images to train AI models, which leads to very slow training process (e.g., ~83 hours of training to make an AI competitive with respect to human [20, 21]). The problem is that the neural network needs several preprocessing layers, i.e., the *convolutional neural network* (CNN) layers [22], to derive the high-level information from the raw program inputs (e.g., images).

Our key insight is the high-level information (e.g., the position of Mario or the image histogram information of Canny) is already derived through the code logic and stored as program variables. Therefore, we propose to use the program variables (which provide the rich information) as the feature variables, thereby obviating the preprocessing layers.

However, large number of program variables exist. Given the user-specified target variables, it is important yet challenging to select the variables that are most semantically relevant to the target variables as their feature variables. Manual extraction of the feature variables [16] would impose heavy burden on the programmers. We designed multiple automated strategies atop the program dependence graph and found through the extensive experiments that (1) the selected feature variable and the target variable should be correlated (share some common dependent), (2) the selected feature variable closer to the common dependent leads to better prediction quality. The feature extraction algorithms are discussed in Section 4.

1.3 Our Design

In this paper, we propose a novel programming framework Autonomizer which consists of the primitives [1] and the runtime support, where the primitives abstract the common components aforementioned while the runtime support does the heavy lifting and realizes them transparently.

As shown in the experiments (Section 6), autonomizing the programs, such as a widely used edge detection program Canny [23] and a speech recognition program Sphinx [24], only requires adding a few lines to the original source code. For SL programs, the autonomization output quality is improved by 161% on average over the baseline with execution overhead no more than 0.64X. For RL programs, the training procedures only need 3.5-20.36 hours for the AIs to be competitive with human players. Comparatively, prior art typically requires at least 83 hours of training [21] to be competitive with respect to human, which is not efficient.

Contributions. We made the following contributions.

- We proposed a novel idea of autonomizing traditional software programs, which applies to the parameter configuration of parameterized programs, the action selection of interactive programs and many other potential applications.
- We designed a novel programming framework which consists of the primitives and the runtime support. The

[1] It is also possible to use the language constructs instead of the library API. However, it would require a specialized compiler and sacrifice the usability.

primitives abstract common tasks of autonomization and the runtime support realizes them transparently.
- We designed the strategies for feature extraction.
- We implemented our approach and evaluated it against nine real world programs. The results are promising as described above: the programs can be autonomized with little effort; the autonomized version leads to much better results while incurring the tolerable execution slowdown.

2 Autonomization Framework Overview

In this section, we show how to autonomize the `Mario` [25] game, which is a representative of a large set of interactive software applications that do not have autonomization in consideration during design. We will explain how to annotate and autonomize the game with reinforcement learning using our proposed primitives. The game is autonomized for two different purposes. We first autonomize the game to play by itself normally and compare the results with the model borrowed from DeepMind [12] which uses raw image screenshots as model input. Then we show how to autonomize the game to do coverage testing. Note that here we are not comparing our work with DeepMind but rather leveraging its model. DeepMind's contributions are orthogonal to Autonomizer. DeepMind demonstrates human-like learning by observing raw images. It does not focus on identifying an efficient way to train a model to play games.

Primitives. The primitives are listed in Fig. 1, which are the library calls in the same programming language as the source program. While more details of the primitives will be discussed in Section 3, we will explain them when they are used in the example.

```
Library Calls :
@au_config(modelName, modelType, algo., layers, neuron1, ...) |
@au_extract(extName, size, data) |
@au_NN(modelName, extName1, ..., wbName1, ...) |
@au_write_back(wbName, size, data) |
@au_serialize(data1, ...) |
@au_checkpoint() |
@au_restore()
```

Figure 1. Primitives

Running Example. In this section, we show how to autonomize an interactive program, i.e., the Mario game, such that it achieves the decent score without human assistance. In Section 6.3, we further show that we can autonomize the parameterized programs to achieve ideal parameter configurations automatically on the fly.

In Fig. 2, we show how to autonomize the Mario game, where the primitives are highlighted. Lines 24-50 show the main game loop function of Mario. In each iteration, multiple functions (lines 1-23) of the program are orchestrated to make the game work. For example, lines 5-13 handle minion collisions and lines 19-23 update Mario's position.

First, the user specifies with the primitive *au_write_back*() at line 44 the target variable (i.e., the output of the neural network model) , which is the variable `actionKey` that holds Mario's action (line 46). Autonomizer then automatically extracts some program variables as the feature variables (i.e., the inputs of the model), which are also annotated with the primitive *au_extract*() at lines 9-10, 17, and 21-22. The feature variables, e.g., the positions of Goombas at lines 9-10, contain the important and relevant information for predicting the target variable, e.g., Mario's action at line 46. We refer the readers to Section 4 for the feature extraction algorithms.

While executing the primitive *au_extract*(), the Autonomizer runtime automatically records the values of the feature variables into a database and assigns names to them for later reference. While executing the primitive *au_write_back*(), e.g., at line 44, the Autonomizer runtime updates the target variable *actionKey* with the predicted value of the target variable. Note the value 5 means there are 5 possible actions.

During initialization, the neural network is configured with the primitive *au_config*() (line 2). In our example, the model has two hidden layers with 256 and 64 neurons, respectively. The size of the input and output layers is automatically computed based on the input fed to the network and the output to be predicted. We also provide a callback function in which the users can create arbitrary neural networks from scratch with Tensorflow, which is omitted due to space limit.

The program interacts with the neural network via the primitive *au_NN*() at line 40. It works in two modes: the training mode and the deployment mode. In the training mode, the program sends data to train the model, in addition to generating the predicted value. In the deployment mode, the program sends data solely to generate the predicted value. In practice, we produce two versions for the modes. Autonomizer automatically writes the output value predicted by the model to the database and index it with the name `output` specified at line 43.

If Mario enters the end state (i.e., Mario dies), it is important to roll back to a previous checkpoint to avoid the expensive full restart. We provide two primitives to achieve this: The primitive *au_checkpoint*() checkpoints the program state at line 27. The primitive *au_restore*() would restore the program state at line 48. Note that the neural network states of Autonomizer are not affected by this pair of primitives. More details are explained in Section 5.

Execution Model. The simplified execution model is shown in Fig. 3. Given the user-annotated target variable Y, Autonomizer first extracts the feature variable X for predicting Y (Section 4). The variables X is then annotated in the program.

The runtime execution is as follows. The original main process executes normally until it reaches the *au_extract*() primitive (①). At this point, the value of the feature variable X is saved to the database (②).

The main process continues its execution until it reaches the primitive *au_NN*(). At this point, the main process transfers the control of the execution of the original program (③) to the execution of a piece of Python code (④), which is

```
1 void initGame() {
2   // ... Init game ...
3   au_config("Mario", DNN, QLearn, 2, 256, 64);
4 }
5 void minionCollision() {
6   for (int i=0; i<Minion.size(); i++) {
7     for (int j=0; j<Minion[i].size(); j++) {
8       // ... Update minion collision ...
9       au_extract("MnX", 1, minion[i][j]->X);
10      au_extract("MnY", 1, minion[i][j]->Y);
11    }
12  }
13 }
14 void checkObj() {
15   if (checkObj(player.front) == "PIPE")
16     ...
17   au_extract("OBJ", 1, player.front);
18 }
19 void updatePlayer() {
20   // ... Update player.x and player.Y ...
21   au_extract("PX", 1, player->X);
22   au_extract("PY", 1, player->Y);
23 }
24 void gameLoop() {
25   while (true) {
26     terminated = 0;
27     au_checkpoint();
28     // Reward calculation
29     if (moveForward(player)) reward = 2;
30     else reward = -1;
31
32     if (reachFlagPole(player)) {
33       reward = 10; terminated = 1;
34     } else if (dead(player)) {
35       reward = -10; terminated = 1;
36     }
37     // This line is only added for self-testing
38     if (checkNewCoverage()) reward = 30;
39
40     au_NN("Mario",
41       au_serialize("PX", "PY", "MnX", "MnY", "Obj"),
42         reward, term,
43         "output");
44     au_write_back("output", 5, actionKey);
45     // ... Act based on returned data ...
46     act(actionKey);
47
48     if (terminated) au_restore();
49   }
50 }
```

Figure 2. Autonomizing `Mario`. The highlighted statements are added. Autonomizer primitives start with au.

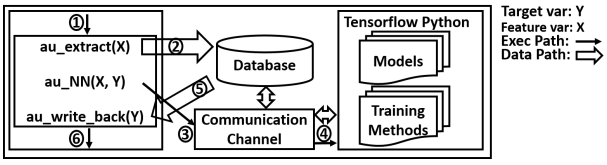

Figure 3. Execution model

generated by Autonomizer based on the annotations and performs the model training and testing using Tensorflow. Besides, Autonomizer also feeds the input data X stored in the database to the model, uses the model to predict the output value (which corresponds to the target variable Y) and writes the value to the database.

Afterwards, the control of the execution is transferred back to the main process. Note the output value is now stored in the database. The primitive *au_write_back*() (⑤) loads it from the database to update the program variable Y.

The main process continues its normal execution (⑥) with the updated Y value. Note that Autonomizer supports multiple model instances in one execution.

Result and Comparison. Videos of the training process and the autonomization result can be found at [26]. It took around 5.7 hours to train Mario to have reasonable behavior without using GPU. Furthermore, we compare the results between our model (i.e., the model using extracted program states) and the DeepMind model [12, 27] (i.e., a model using raw image screenshots).

Low Engineering Efforts. With the support of Autonomizer, the users achieve the autonomization with very few annotations, as highlighted in Fig. 2. The automatic feature extraction of Autonomizer further alleviates the burden of specification. Comparatively, existing work for Mario autonomization [10] spends great engineering efforts in the common tasks such as data collection, data saving/loading, integration of the model and the original program.

Better Result and Faster Training. We stop the training if the score of Mario is comparable with human (i.e., difference $< 20\%$) or if the training time exceeds 24 hours. Here the score refers to the stage clearance rate of 10 runs.

According to the experiments, the DeepMind model is trained for 8000 epochs before exceeding the 24 hours limit. Comparatively, Our model is trained for only 2000 epochs. Note each epoch corresponds to 100 iterations of the loop. The results show that the DeepMind model achieves the score 40% after 24 hours' training, while our model achieves the score 80% after only 5.7 hours' training.

The reason for the difference lies in the feature selection. The DeepMind model uses the raw images as the model inputs and applies multiple convolution layers to derive high-level information from the raw images. In particular, each image is an 84x84x4 input array (after preprocessing). The neural network has three convolution layers, each followed by a max pooling layer, and finally two hidden layers with 256 and 64 neurons. Due to the complexity of the network, the training requires very long time to achieve good result, or equivalently, achieves bad result within a short time.

Comparatively, our key insight is that the programmers derive the high-level information through the code and store them in some internal program variables. Therefore, our model directly uses such high-level information as the model inputs, thereby obviating the need for the three convolution layers in the DeepMind model. Our simpler model achieves better results while requiring shorter training time.

The screenshots illustrate the difference more intuitively. In Fig. 4, following our model, Mario jumps only when it's necessary. In Figure 5, the Mario following the DeepMind model keeps jumping all the time, indicating the model is still at the early stage of the training. Intuitively, if Mario jumps too often, it is less likely to stay on the ground where control can be applied, i.e., the chance of controlling Mario

becomes lower. Thus, it easily hits the Goombas and dies as shown in Fig. 5. The relevant videos can be found at [26].

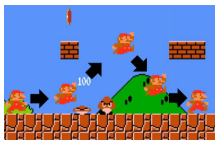

Figure 4. Using internal data

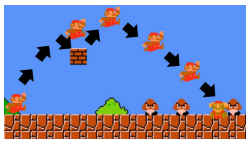

Figure 5. Using raw data

Autononmization for Software Self-Testing. To further demonstrate the capability of software autonomization and the flexibility of Autonomizer, we autonomize the Mario game in a way that it performs the testing. All we need to do is to update the reward so that it reflects the code coverage improvement, in addition to the original reward which reflects the stage clearing. Line 38 in Fig. 2 show the added reward for code coverage, where the code coverage is collected using gcov [28]. Intuitively, any improvement of code coverage results in large reward. Note that we also need the original reward to ensure Mario survives long enough to reach the complex game logic. After training for 10 hours, Mario is able to make many unexpected moves that lead to good code coverage quickly. In 30 seconds of game play, ~65% code coverage can be achieved. In contrast, the previous AI model (which is not designed for testing) cannot reach a similar code coverage after 10 mins, not to mention the random testing in which Mario easily dies within seconds.

Fig. 6 shows the screenshot of the testing. Observe that Mario has more interesting behaviors, e.g., Mario jumps backward to eat the mushroom (①-(③)) and then jumps into the ditch (④). The AI even found two bugs during self-testing. The videos of both bugs can be found at [26]. Fig. 7 shows one of the bugs. Before falling to the ground of the dungeon, Mario moves in some unexpected ways such as jumping forward. As a result, Mario reaches the ceiling of the dungeon. Then it tries to further jump forward and goes out of the screen, which crashes the program. Code inspection discloses that the developer missed a boundary check. This case study illustrates the capabilities of Autonomizer in enabling future research along this line.

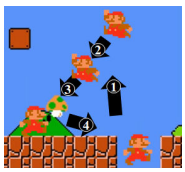

Figure 6. Coverage testing

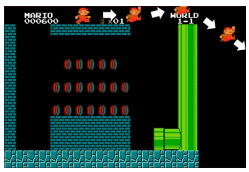

Figure 7. Bug

3 EXECUTION MODEL: SEMANTICS

Autonomizer features a novel set of primitives and a unique execution model that are particularly designed for software autonomization. After compilation and linking with Autonomizer runtime, an executable with two execution modes is generated. One for training, and one for deployment (or production run). Training is piggybacking on normal software execution to derive an AI model (or multiple models). During production run, the model is used to replace human interactions/decisions. Supervised and reinforcement learning are supported by default. Autonomizer is also extensible through its interface with Tensorflow to support other methods.

To support learning from software operation, Autonomizer needs to monitor software execution, trace values of feature variables that would be used by the model to make decision, record the desirable decisions made by human users to serve as the objective of learning (i.e., desirable model output), and roll back software execution state but not the model state (during reinforcement learning). In supervised learning, model training is conducted offline after execution. In reinforcement learning, model training is conducted online and the training execution interleaves with software execution. Intuitively, Autonomizer collects model inputs/outputs for a window of time (e.g., a few game loop iterations) and then invokes the training method to use the collected data.

During production runs, Autonomizer intercepts values of feature variables and passes them to the model to make prediction. The predicted results are then copied to some program variables (such as the variable denoting the next move of Mario) to drive execution. Note that all the aforementioned complexities are transparent to the users. In the following, we discuss the semantics of individual primitives, which is important to precisely understand how Autonomizer works.

3.1 Definitions

Definitions related to the operational semantics are presented at the top of Fig. 8. Autonomizer has two stores, the *Program Store* σ for original program states, which can be intuitively considered as a hash map that projects a variable to its current value, and the *Database Store* π which stores the extracted feature variable values. It is a mapping from a string to a list of values. The value of a program variable extracted by the primitive *@au_extract*() will be appended to a list in the database store indexed by a string name. Furthermore, the outputs returned by the underlying model are also put in the database store before they are written back to target program variables with the primitive *@au_write_back*() (to affect the execution). The two stores are isolated. Transferring data between the two should be explicitly requested by the programmer through the primitives.

We abstract a neural network model θ as a mapping from a model name to a list of parameter values. Intuitively, one can think of them as the weight values of matrices in individual model layers. By default, Autonomizer supports two model types: fully connected neural network (DNN) as well as CNN and two popular algorithms: Q [29] for RL as well as AdamOpt [30] for SL. Execution mode ω denotes the two

Figure 8. Operational Semantics

modes supported: *TR* for training and *TS* for production runs (or testing). Autonomizer runtime provides a list of API functions denoted as statement extensions (e.g., **buildModel()** for initializing a model inside Tensorflow). The semantics of many primitives are resolved to these API functions.

3.2 Rules

The semantics rules are in the lower part of Fig. 8. As indicated by the configuration (in the box), each rule is a transition of statement s to s' while updating the two stores and the model. We organize the rules to two groups: (i) model construction/training/testing and (ii) checkpointing/restore.

3.2.1 Model Construction/Training/Testing

Model Construction. In Rule [$CONFIG - TRAIN$], Autonomizer in the training mode creates a new model if model exists in memory by executing the statement **buildModel**, where the model name *modelName*, the model type δ, the training algorithm α, the number of layers l and the number of neurons $n1$ are specified by the programmer. In Rule [$CONFIG - TEST$], Autonomizer in the testing mode simply loads an existing trained model with the specified model name *modelName* by executing the statement **loadModel**.

Model Training and Testing. In Rule [$EXTRACT$], Autonomizer appends the variable value(s) to a list in the database store indexed by a unique name *extName*. Note that if the primitive *au_extract*() is inside a program loop that iterates multiple times before invoking the *au_NN*() primitive, the list contains multiple values. In Rule [$TRAIN$], Autonomizer trains the model by gradient descent [31], i.e., updating model parameters $\widetilde{Parm}$ along the largest gradient. Then, the model output generated by executing the statement **runModel** on the model input retrieved by $\pi(extName)$ is put in the database store with index *wbName*. Afterwards, the model input is reset to an empty list (by mapping *extName* to $\perp$ in π). Rule [$TEST$] is similar to rule [$TRAIN$] except that it does not update the model. It simply uses the model.

In Rule [$WRITE - BACK$], Autonomizer writes the value with name *wbName* from the database store back to the program variable x. Rule [$SERIALIZE$] concatenates multiple lists of values into a single list through primitive **concat**(). The names of those lists are also concatenated through *strcat*(). This feature helps to combine multiple extracted values into one list and feed it to the underlying model. Note that neural network models only take vector inputs. For example, at line 41 of Fig. 2, six lists of extracted values are combined into

one list of values and fed to the neural network as input. Note that Autonomizer supports serializing multiple lists.

3.2.2 Checkpointing/Restore

In Rule [*CHECKPOINT*], Autonomizer checkpoints the states of current program store and database store by making the snapshot through the statement **mkSnapshot**(). Note that although model state in θ is part of the software process, it is not checkpointed because we want the model to accumulatively learn. In Rule [*RESTORE*], Autonomizer restores the states of program store and database store with the previously made snapshot through **rtSnapshot**(). Note that the states between both stores need to be consistent, so their states have to be checkpointed and restored together.

4 Feature Variables Extraction

In this section, we discuss how to extract program variables that correspond to important features. The values of these variables will be extracted as model inputs. The analyses are different for supervised learning and reinforcement learning. We adopt dynamic dependency analysis instead of static analysis which incurs too many false positives.

Supervised Learning. In our settings, the supervised learning (SL) is used to predict the ideal value of the parameters (i.e., the *target variables*) that affect the quality of the result based on the relevant internal program states (i.e., the *feature variables*). While the target variables are specified by the users, which is an easy task according to our experiments, it is non-trivial (e.g., labor-intensive and error-prone) for the users to specify the feature variables. To lift the burden, we automatically extract the feature variables by combining heuristics and program analysis. We also conducted extensive experiments to validate the effectiveness of the heuristics (Section 6).

First, we observed the ideal values of the target variables (or parameters) vary for different program inputs, meaning that they are sensitive to the inputs. Therefore, we identify the input variables and those that transitively depend on them as the *candidate* feature variables. Furthermore, we conduct correlation analysis to determine the subset of candidate feature variables correlated with each target variable. Intuitively, we say two variables are correlated if they are depended upon by the same variable. Lastly, to refine the subset of feature variables, we rank the feature variables heuristically according to their "distances" to the correlated target variable, and select the top-ranked variables for prediction. According to the experiments (Table. 3), the refinement leads to better prediction results.

In the following, we explain the automatic extraction and the involved terms in details.

Algorithm 1 takes three inputs: In, Trg, and G_{Dep}. In is the set of input variable set, Trg is the set of target variable, and G_{Dep} is the pre-computed dynamic dependency graph.

Algorithm 1 Automatic SL Feature Extraction

Input: In, Trg, G_{Dep}
Output: $Feature$
1: $Candidate \leftarrow In \cup dep(In)$
2: $Feature \leftarrow Map()$
3: **for each** $v \in Trg$ **do**
4: **for each** $w \in Candidate$ **do**
5: **if** $dep(w) \cap dep(v) \neq \emptyset$ **then**
6: $Feature[v] \leftarrow Feature[v] \cup \{w, \infty\}$
7: **for each** $w, dist \in Feature[v]$ **do**
8: $dist \leftarrow BFS(G_{Dep}, w, first(dep(w) \cap dep(v)))$
9: $Feature[v] \leftarrow \{w, dist\}$
10: $Sort(Feature[v])$
11: **return** $Feature$

First, we construct the candidate set, which consists of the input variables and their transitive dependents.

$Feature$ is a map that maps a target variable to its feature variables, which is returned eventually. For each target variable v in Trg, if a candidate feature variable w shares some common dependent with v (line 5), then w is a feature variable correlated with v. For prediction purpose, w is not considered as feature variable if it depends on v.

To rank a candidate feature variable w, we use its dependency graph distance, which is defined as the number of edges between w and the first common descendent of w and v. In lines 7-9, the shortest distance from each w to the common descendent is found by BFS on the dependency graph G_{Dep}. In line 10, the feature variables are sorted according to $dist$, which allows us to further select the top-ranked feature variables. Intuitively, the shorter the distance, the more abstract (and the more important) the feature variable.

Fig. 9 demonstrates the example of extracting feature variables in `Canny`. Variable *lo* is a target variable and the remaining are candidate feature variables. Variable *hist* is ranked first to predict *lo* because it has distance 1 to first common descendent *result*. Feature variable *sImg* has distance 2 so it is ranked lower than *hist*.

Reinforcement Learning. We propose a feature variable identification technique for the reinforcement learning (RL) applications. Unlike SL programs, RL programs are usually not for one-time data processing like `Canny`. Instead, they often have some main loop that continuously updates program states, such as the game loop in game applications and the control loop in autonomous vehicle control software. Intuitively, variables that represent these continuously updated states are *candidate* feature variables. Note that they may not be dependent on external inputs (e.g., user key strokes).

For a target variable, other program variables that share common descendent with it are considered correlated with it. Those program variables are candidate feature variables. According to our observation, variables that correlate with target variable contain program states that affect the prediction result of target variable in RL applications. Our experiment results in Section 6 also justify the observation. For simplicity, Autonomizer only checks variables that are

Algorithm 2 Automatic RL Feature Extraction

Input: Trg, $UseFunc$, $ProgVar$, ϵ_1, ϵ_2
Output: $Feature$
1: $Feature \leftarrow Map()$
2: **for each** $v \in Trg$ **do**
3: $\quad Candidate \leftarrow Map()$
4: $\quad$ **for each** $w \in ProgVar$ **and** $w \neq v$ **and**
$\quad\quad UseFunc[dep(v)] \cap UseFunc[w] \neq \emptyset$ **and**
$\quad\quad dep(v) \cap dep(w) \neq \emptyset$ **do**
5: $\quad\quad Candidate[w] \leftarrow Scale_{0-1}(Tracing(w))$
6: $\quad$ **for each** $w, Trace_w \in Candidate$ **do**
7: $\quad\quad$ **for each** $x, Trace_x \in Candidate$ **and** $x \neq w$ **do**
8: $\quad\quad\quad$ **if** $EucDist(Trace_w, Trace_x) \leq \epsilon_1$ **then**
9: $\quad\quad\quad\quad Delete(Candidate[x])$
10: $\quad\quad$ **if** $Variance(Trace_w) \leq \epsilon_2$ **then**
11: $\quad\quad\quad$ **continue**
12: $\quad\quad Feature[v] \leftarrow Feature[v] \cup w$
13: **return** $Feature$

used in the same functions as variables that depend on the target variable. After finding all candidates, Autonomizer prunes redundant or unchanging variables based on runtime values of each variable's trace according to two thresholds set by the user. The remaining variables are combined and returned as features. Note that ranking is not as effective as in SL because most feature variables would have loop-carry dependencies due to the iterative updates.

Algorithm 2 takes five inputs: Trg, $ProgVar$, $UseFunc$, ϵ_1 and ϵ_2. Trg is the target variable set, and the $ProgVar$ set contains all program variables. Map $UseFunc$ maps a variable to its usage functions. Thresholds ϵ_1 and ϵ_2 are used to prune redundant and unchanging variables respectively.

At line 4, if program variable w is used in the same function as target variable v's dependent variable, and both v and w have common descendents, then w is considered correlated with v and is added to the map $Candidate$ with its runtime trace $Trace_w$ which contains w's runtime values in a profiled time sequence. The sequence of trace values are scaled [32] between 0 and 1 (line 5). In lines 8-9, the similarity between w and x is computed according to the distance between $Trace_w$ and $Trace_x$ using the euclidean distance formula.[2] For example, assume $Trace_w$ contains [0.1, 0.3, 0.4] and $Trace_x$ contains [0.1, 0.2], the similarity is $\sqrt{(0.1-0.1)^2+(0.3-0.2)^2+(0.4-0)^2} = \sqrt{0.17}$. If the similarity is less than ϵ_1, x is considered redundant and pruned. In lines 10-12, if the variance of w's trace values is smaller than ϵ_2, w is considered unchanging and pruned. Intuitively, a rarely changing variable is not a good feature. Real pruning examples are shown in the `TORCS` autonomization case study.

Fig.10 shows an example of extracting feature variables in `Mario` to predict the target variable *right*, which makes Mario move right. Variable *Player->X* is a feature variable because it depends on itself and shares the same descendent with *speed* (and transitively with *right*). Another feature variable is *Minion->X* as it shares the descendent *collide* with *pX* (and transitively with *right*). Variable *mX* is pruned by ϵ_1 because it is a duplicate of *Minion->X*.

[2]If the sequences' lengths are different, we append zeros to the shorter one.

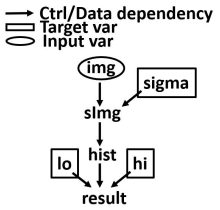

Figure 9. Alg.1 on `Canny`

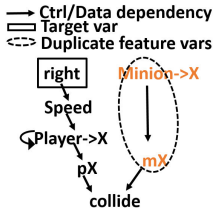

Figure 10. Alg.2 on `Mario`

5 Implementation

We leverage Tensorflow [33] to support model training and execution. Tensorflow is an open source software library with strong support for machine learning applications. Autonomizer essentially stitches the execution of Tensorflow with the execution of the original software. In other words, both executions occur in the same process space. Since Tensorflow has a comprehensive Python interface, at compile time, Autonomizer generates a Python template for each injected model. The template essentially provides the API functions used in the semantics (e.g., **runModel**()). These functions further invoke Tensorflow functions to realize their functionalities. Templates have to be generated based on primitive annotations because different model structures, model input sizes, and learning algorithms (in *au_config*() and *au_NN*()) lead to different templates. To make a source program interact with its Python template, a communication channel (Fig. 3) is implemented with the Python C/C++ extensions [34]. Details are elided.

Checkpointing and Restore. In our context, we need to checkpoint very complex software states in addition to memory states (e.g., thread and socket), and memory mapped I/O states. Simple software checkpointing by forking does not work. Even existing process level virtualization techniques (e.g., criu [35] or docker [36]) do not provide the guarantees of arbitrary-scale program states checkpointing/restore. We hence leverage KVM [37] to create checkpoints.

As mentioned earlier, we only checkpoint software states and database states but not model states. However, this is difficult to achieve as all these states are in the process space and indistinguishable for KVM. As such, before restoring to a checkpoint, Autonomizer saves the current model states to persistent storage. After restoring, it overwrites the model states with those saved in storage. Details are elided.

6 Evaluation

In Autonomizer, the feature variable extraction component is implemented using Valgrind-3.14.0 [38, 39]. The language and runtime are implemented in C++/Python. It is publicly available at [26]. Experiments were run on a machine with Intel i7-2640M 2.80GHz processor, 16GB RAM. We use a

Table 1. Program analysis statistics

Program	LOC	Added LOC	Trg Vars	Candidate Vars	Feature Vars
[SL] Canny	1.1K	8	3	26	1/23/23
[SL] Rothwell	1.3K	6	3	8	1/8/8
[SL] Phylip	12K	7	3	42	1/1/28
[SL] Sphinx	28K	37	2	107	13/14
[RL] Flappybird	0.8K	40	2	19	4
[RL] Mario	21K	73	5	345	25
[RL] Arkanoid	1K	39	-	-	-
[RL] TORCS	150K	89	2	370	20
[RL] Breakout	153K	65	-	-	-

Arkanoid and Breakout are emulator games so we annotate the emulator and use the exported game information directly.

Table 2. Model statistics

Program	Raw		Med		Min		Raw/Min		Checkpoint Time(s)	Restore Time(s)
	Trace Size(MB)	Model Size(MB)	Trace Size	Model Size	Trace Size	Model Size	Trace Size	Model Size		
[SL] Canny	48	215	48	215	14	113	3.43	1.9	-	-
[SL] Rothwell	49	215	143	215	143	215	0.34	1	-	-
[SL] Phylip	4	21	1	1.8	1.2	1.9	3.33	14	-	-
[SL] Sphinx	157	172	36	15	36	15	4.36	11.66	-	-
	Raw				All		Raw/All			
[RL] Flappybird	781042	0.58	-	-	13.93	0.25	56069.06	2.32	27.38	6.23
[RL] Mario	68359	0.58	-	-	100.41	0.51	680.79	1.13	25.11	7.51
[RL] Arkanoid	351562	0.59	-	-	57.13	0.23	6153.72	2.57	26.82	6.21
[RL] TORCS	25313	1.9	-	-	714.8	0.47	35.41	4.04	25.33	6.91
[RL] Breakout	304018	0.62	-	-	11.44	0.21	26575	2.95	26.12	6.71

GPU of NVIDIA GeForce 1060 with 6GB RAM for supervised learning tasks. We use a wide variety of C/C++ benchmarks in our experiments. All programs have multiple datasets that can be found online or come with the program. We only selected those that have the ground truth (for SL).

In Section 6.1, we present the statistics that expose details about our approach. In Section 6.2, we conduct a comparative study of the effectiveness of our approach. In Section 6.3, we conduct in-depth case studies for more insights.

6.1 Statistics

The statistics of our approach are presented in Table 1.

Lines of Code Added. Column 2 shows the lines of code (LOC) of the programs and Column 3 shows the lines of code added for autonomization. According to Column 3, only a few lines are required for autonomization. In other words, with the help of our language support, the users introduce the advanced AI capability to the programs with little effort.

Feature Variables. Column 4 shows the number of target variables, column 5 shows the number of candidate feature variables, and column 6 shows the feature variables available for selection.

Supervised Learning. Any feature variables in column 6 are available for use. To assist the selection, they are ranked as discussed in Section 4. For example, only around 15 out of the 100+ candidates are available in the Sphinx application. The results suggest that a lot of candidates are pruned.

Reinforcement Learning. For RL applications, we made similar observation that a large number of candidates are pruned during the selection. By pruning the redundant and unchanging candidate feature variables, our approach keeps the prediction focused, thereby making it more effective (as confirmed in Section 6.2). All feature variables are combined to predict multiple target variables due to the large overlap of the feature variable sets.

Model Construction. Table 2 shows the statistics related to the model, including (1) the size of the trace collected which consists of the input to the model, and (2) the size of the trained Tensorflow model.

Supervised Learning. To study the effect of distance (Algorithm 1) upon the model statistics, we compare three versions: *Raw* in Columns 2-3 which selects feature variables with the maximum distance (i.e., input variables), *Med* in Columns 4-5 which selects the feature variables with the median distance, and *Min* in Columns 6-7 which selects the feature variables with the minimum distances. For fairness, all versions use the same neural network architecture except for the input layer which accounts for different input size.

In columns 8-9, we show the ratio of *Raw* and *Min* in terms of the trace size and the model size, respectively. Both the trace size and the model size of *Raw* is much larger than *Min*. The reason is that the model of *Raw* usually has a larger number of input neurons than *Min* because the model inputs of *Raw* are typically raw data which are usually larger. Specially, for Rothwell, *Min* has larger trace size. This is because *Raw* and *Min* have a similar number of inputs but *Raw* represents them with the char type while *Min* represents them with the float type.

Reinforcement Learning. Results are compared between two settings: *Raw* and *All*. *Raw* uses the DeepMind model [12], which takes the scaled images as the inputs. *All* uses a four-layer fully connected neural networks. It combines and uses all feature variables identified by Algorithm 2 as inputs. For the RL programs, since game executions do not terminate as SL programs, we collected the statistics for a time window of a fixed length. In columns 8-9, we show the ratio (*Raw*/*All*) in terms of the trace size and the model size, respectively. According to column 8, the size of the trace collected by *Raw* is 35-56069 times the size of *All*. This is because *Raw* collects raw images which are typically much larger than the (extracted) internal states collected by *All*. For instance, in Flappybird, a raw images collected by *Raw* is 700x800x4 bytes while a vector of internal data collected by *All* is only 32 bytes. The fact that RL usually requires many iterations further amplifies the difference. Besides, according to column 9, the model size of *Raw* is larger than *All*. The reason is that the model of *Raw* has more layers where the first few layers extract the visual features from the raw images. Comparatively, *All* does not need such layers because it takes the

program states of the feature variables as the input, which already hold the important feature information.

Checkpointing/Restore. Column 10 and 11 represent the time for creating and restoring a checkpoint. Only RL programs need checkpointing/restore. Creating a checkpoint takes around 26 seconds. Although it takes non-trivial time, it only needs to be done once at the beginning. After that, Autonomizer can restore the checkpointed state when ending states (e.g., Mario dies) are encountered during training. Restoring takes around 7 seconds.

6.2 Effectiveness

In this section, we present the study of the effectiveness of our approach. In particular, we are interested in how our autonomization affects the quality of the results. The results are shown in Table 3.

6.2.1 Experiment Settings

We first discuss the settings for gathering the results.

Comparisons. For the SL applications, we compare four versions: the baseline, *Raw*, *Med*, and *Min*, where the baseline refers to the execution with the default parameter configurations and the others three versions are explained in Section 6.1. For the RL applications, we compare three versions: the players version, *Raw*, and *All*, where the players version accounts for the average of 10 human players and the remaining two versions are explained in Section 6.1. Each experiment is repeated 10 times to compute the average.

Scoring. The quality of the results is measured with the score assigned to the results. We will explain how the score is assigned soon. Note that higher quality does not necessarily correspond to higher score. We put a mark in Column 1 to specify whether higher quality corresponds to a higher score or a lower score. For the SL programs, we use the built-in score functions shipped with the programs. For the RL programs, the score functions are not available. Instead, we define the score for each program that accounts for the progress or the success rate. For Flappybird, the score stands for the progress (i.e., how far the bird flies in terms of the percentage of the whole distance). For Mario, the score is a pair in the X/Y form, where X stands for the progress (i.e., how far Mario goes) and Y is the success rate (i.e., the rate of taking down the flag). For Arkanoid, the score is also a pair X/Y, which respectively stands for the progress (i.e., the percentage of cleared bricks) and the success rate (i.e., the rate of clearing all bricks). For Torcs, the score represents the driving progress (i.e., how far the car drives) without bumping the wall before finishing. For Breakout, the score represents the number of hit bricks before missing the ball. Note all scores are averaged over 10 runs.

Training. For SL programs, we train each version (except the baseline version which does not need training) until convergence (i.e., the score stops changing). For RL applications, training RL applications is normally considered non-stationary and hard to converge. Thus, we force the training to time out after 24 hours.

For each setting, we show the training time and the execution time, which correspond to the time taken by the training run and the testing run, respectively. Specifically, for the RL applications, the execution time stands for the time taken by each iteration of the game loop.

6.2.2 Experiment Results

Here we discuss the efficiency and effectiveness of using program internal features extracted by Autonomizer through execution time, training time, and evaluation score.

Comparing to Baseline and Human Players. We compare a baseline with the corresponding best setting. (*Min* for SL programs and *ALL* for RL programs).

Supervised Learning. The *Min* version (Columns 11-12) improves the baseline results by 161% on average. Besides, the overhead is less than 0.64X. It shows that autonomization improves the data processing results with small overhead.

Reinforcement Learning. The extracted feature variables help the *All* version (Columns 11-12) to achieve scores close-to/better-than the results of human players. The execution overhead ranges from 0.89X to 6.14X. Although 6.14X looks substantial, the incurred overhead does not cause any noticeable delay[3] because the execution time is computed for each time frame and the overhead is not human perceptible if the number of frames that the program can handle in a time unit exceeds a certain threshold.

Comparison among Different Settings. We compare the results between different autonomization settings for SL and RL programs.

Supervised Learning. Although all settings (Columns 5-6, 8-9, and 11-12) outperform the baseline, the quality of improvements are different. Specifically, *Min*, *Med*, and *Raw* versions improve the baseline results by 161%, 141%, and 120% on average respectively. It shows that feature variables that close to the target variable are more important.

Reinforcement Learning. The *All* version (Columns 11-12) achieves good performance. On the other hand. the *Raw* (Columns 5-6) version cannot achieve similar score (difference $< 20\%$) of human players and times out after 24 hours of training for most benchmarks. Furthermore, the execution overhead of *Raw* is higher (3.16X-23X) than *All* because Algorithm 2 helps *All* to prune many redundant feature variables and only retain the most representative ones. We further compare the *All* version with the *Raw* version using the most representative Breakout benchmark from DeepMind [12]. The *Raw* version uses the model in DeepMind. Observe that both the *Raw* and the *All* versions can compete with the human players and the *All* version has higher

[3]The execution videos can be found at [26]

Table 3. Benchmark experimental results.

Program	Baseline		Raw			Med			Min			Training Time Raw/Min
	Exec. Time(s)	Score	Train Time(s)		Score	Train Time(s)	Exec. Time(s)	Score	Train Time(s)	Exec. Time(s)	Score	
[23] [SL] ↑[1] Canny	1.32	0.45	1791.52	1.58	0.543	1719.34	1.39	0.69	817.62	1.47	0.763	2.4
[40] [SL] ↑ Rothwell	1.14	0.49	1978.1	1.63	0.64	1540.68	1.62	0.68	1616.37	1.53	0.705	1.22
[41] [SL] ↓ Phylip	1.49	1.013	46.91	2.23	0.96	6.079	2.01	0.63	5.56	2.15	0.54	8.44
[24] [SL] ↑ Sphinx	0.86	0.108	4001.52	1.52	0.57	3898.22	1.61	0.581	141.73	1.41	0.6323	28.23
	Players		Raw						All			Raw/All
[42] [RL] ↑ Flappybird	0.0101	91.4%	t/o[2]	0.071	1.3%	-	-	-	12781.32	0.028	95.7%	-
[25] [RL] ↑ Mario	0.0243	92%/90%	t/o	0.101	63%/40%	-	-	-	18465.42	0.046	84%/80%	-
[43] [RL] ↑ Arkanoid	0.0041	77.2%/60%	t/o	0.049	1.5%/0%	-	-	-	41328.13	0.015	88%/60%	-
[44] [RL] ↑ TORCS	0.003	100%	t/o	0.072	7.8%	-	-	-	73323.59	0.018	100%	-
[45] [RL] ↑ Breakout	0.0071	29.8	68902.14	0.058	25.3	-	-	-	34452.74	0.012	28.5	1.99

1. ↑: Higher scores are better; ↓: lower scores are better.
2. "t/o" means using raw data cannot achieve the similar score (i.e., difference < 20%) of 10 human players.

score. Note that DeepMind aims to demonstrate feasibility of human-like learning (from raw images). It does not focus on efficient training or automating the procedure.

Training Time. For SL programs, training *Min* only take $\frac{1}{28} \sim \frac{1}{1.22}$ of the time taken by *Raw*. For RL programs, the training time for *All* version ranges from 3.5 to 20.36 hours while *Raw* times out for most RL benchmarks except the Breakout benchmark. The reason that the *Raw* version can be trained within the time limit for this benchmark is that the playing field for this game is not as complex as other benchmarks (e.g., Mario). Besides, after following the preprocessing steps (e.g., greyscale conversion and image cropping) in DeepMind [12], the input images become much less noisy. These factors make the *Raw* version model training easier compared to other RL benchmarks. For the *All* version, its training overhead is 1.99X less than the *Raw* version, which demonstrates the advantage of using internal program states.

6.3 Case Studies

In this section, we study the details of autonomizing two representative programs with SL and RL.

Canny. Canny [23], a popular edge detection tool, carries the parameters that affect the quality of the edge detection. To achieve the ideal result, each input image requires a specific parameter configuration, i.e., no universal optimal parameter configuration generally applies. Thus, users either have to manually tune [46] or auto-tune [47] the configuration for each image, which prohibits the application from handling a large volume of images with satisfactory results. Comparatively, Autonomizer automatically predicts the proper parameter values on the fly without human intervention. With the support of Autonomizer, Canny can process a large volume of diverse images and provide satisfactory results.

Ease of Use. Fig. 11 shows the user specification for autonomization, specifically for the *Min* version. Initially, the user only needs to annotate the three important parameters of Canny (i.e., the target variables): `low`, `high` and `sigma`, where the former two are for edge traversal (lines 6-7) and the last one is for Gaussian smoothing (line 18). Then our extraction algorithm automatically recommends *image* (line 19) as the feature variable for predicting `sigma` and *hist* (line 9) as the variables for predicting `low` and `high`, which are annotated at lines 16 and line 4, respectively. In total, we need only 9 lines of extra code (i.e., the highlighted ones) as shown in Fig. 11.

```
1 char *hysteresis(mag, lo, hi)
2 {
3   hist = computeHist(mag);
4   au_extract("HIST", 32767, hist);
5   au_NN("MinNN", "HIST", "LO", "HI");
6   au_write_back("LO", 1, &lo);
7   au_write_back("HI", 1, &hi);
8
9   return do_hysteresis(hist, lo, hi);
10 }
11
12 void canny(image, sigma, lo, hi) {
13   // 1. Gaussian smooth
14   au_config("SigmaNN", DNN, AdamOpt, 6, ...);
15   au_config("MinNN", DNN, AdamOpt, 6, ...);
16   au_extract("IMG", 62500, image);
17   au_NN("SigmaNN", "IMG", "SIGMA");
18   au_write_back("SIGMA", 1, &sigma);
19   sImg = smooth(image, sigma);
20
21   // 2. Magnitude computation
22   mag = magnitude(sImg);
23
24   result = hysteresis(mag);
25 }
```

Figure 11. Canny. Autonomizing with the *Min* version. The highlighted statements are added.

Running Example of Algorithm 1. We also created the *Raw*, *Med*, and *Min* versions, which use different feature variables to predict the target variable. Consider Fig. 9, given the target variables `low` and `high`, Algorithm 1 determines `hist`, `mag`, `sImg`, and `image` as the candidate feature variables because they share the common dependent `result` with the target variables. Algorithm 1 further sorts them based on their distance (i.e., 1, 2, 3, 4 respectively) to the dependent. Accordingly, *Min* uses the variable `hist` with the minimum distance as the feature variable, *Med* uses the variable `sImg` with the medium distance and *Raw* uses `image` with the maximum distance.

Usefulness. To demonstrate how autonomization helps improve data processing results, we show the results of baseline, *Raw*, *Med*, and *Min*. For fair comparison, all versions use the

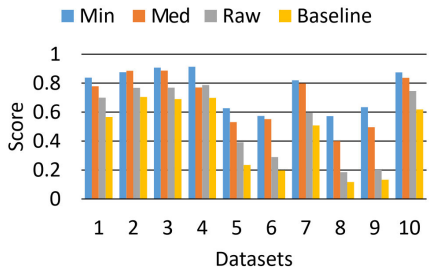

Figure 12. Canny predictions of 10 datasets

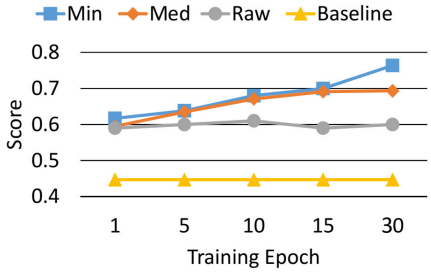

Figure 13. Canny prediction score variation

same neural network structure, i.e., a six-layer fully connected neural network inspired by [48], except for the input layer. In particular, for input layer, both *Raw* and *Med* have 62500 neurons whereas *Min* has 32768 neurons. They differ because *Min* uses `hist` for prediction, which is of a smaller size than `image` and `sImg` used by *Raw* and *Med*.

We use the images from [49] to train the neural networks with SL. We use 10 images from [46] for testing which are associated with the ground truth specified by experts. Better result has a higher score (the SSIM score [50]).

Fig. 12 shows the test scores of baseline, *Raw*, *Med*, and *Min*. Each model is trained around 30 epochs. On average, the improvement of *Min* over baseline is 70%, which clearly shows that Autonomizer significantly improves the quality of the result. Meanwhile, the improvement of *Raw* and *Med* over the baseline is around 20% and 53%. It shows that Algorithm 1 (in particular, the ranking) is useful for extracting the most relevant feature variables.

Fig. 13 shows the change of the score along with the increase of the number of the training epochs. *Min* consistently has higher scores than all the rest versions. Furthermore, as shown in Table 3, the training time of *Min* is about half of *Raw* and *Med*, which is because the feature variables it adopts have a smaller size.

Fig. 14 shows a list of sample images denoting the edge detection results. Clearly, *Min* provides the outcome most similar to the ground truth.

TORCS. TORCS [44] is an open source C++ 3D car racing simulator. Many works [51–53] use it to study the application of reinforcement learning to self-driving cars. In this study, we autonomize TORCS by using the program internal

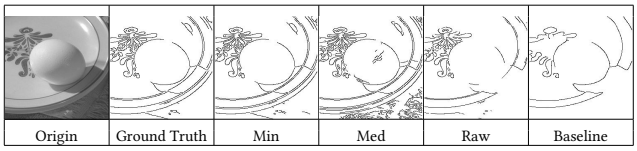

Figure 14. Canny results

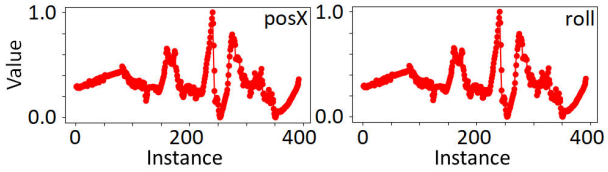

Figure 15. EucDict ≈ 0

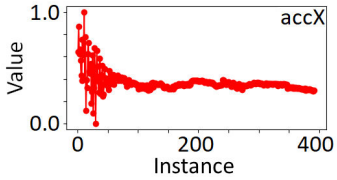

Figure 16. Variance ≈ 0.007

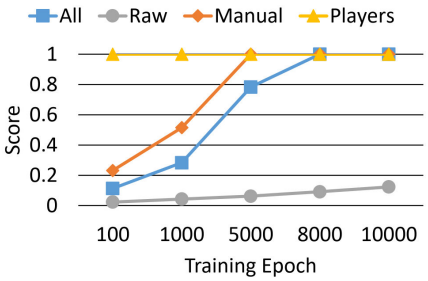

Figure 17. Driving score

states Autonomizer automatically extracts as the feature variables. Comparatively, existing works either use the manually extracted features [16], or the raw image features [15].

Ease of Use. To autonomize Torcs for the *All* version, we annotate the variable `steer` for steering control as the target variable, which determines the turning of wheel. We run Algorithm 2 by setting ϵ_1 to zero and ϵ_2 to 0.01. If the traced values of two variables are similar (i.e., the euclidean distance $\leq \epsilon_1$), we can prune one of them. As shown in Fig. 15, the traced values of the candidate feature variables *posX* and *roll* are almost the same ($EucDict(posX, roll) \approx 0$), so *roll* is pruned. Besides, we prune the candidate feature variables whose values rarely change. For example, as shown in Fig. 16, the variable *accX* is pruned because the variance of its values is ~0.007, which is less than ϵ_2 (0.01). In total, twenty feature variables are automatically extracted. The annotation is similar to Mario (Section 2) and hence elided.

Usefulness. To demonstrate how autonomization helps self-driving without human intervention, we compare four settings: Players, which represents the average score from 10 human players, *Raw*, which uses an existing model [15] that takes screenshots as the input data, *All*, which is our version, and *Manual*, which uses an expert model [16] with manually extracted/preprocessed program variables as the input data. All models have the same output consisting of three actions: left turn, right turn, and no turn. We compare the results using the following criterion: how far the car drives without bumping to the wall before finishing.

Fig. 17 presents the scores of all settings after training for the same number of epochs. It also includes the average of 10 human players as a reference. Observe that *All* consumes around 8000 epochs (20.3 hours) and *Manual* consumes 5000 epochs (14.5 hours) to have close-to human performance. Although *Manual* learns quicker than *All*, it requires non-trivial human effort (~2000 lines of code) to extract and preprocess input data. On the other hand, Autonomizer only uses 89 lines of code as shown in Table 1.

For *Raw*, after training for 10000 epochs (~40 hours), it still performs bad. Furthermore, the improvement is really slow. According to the author [15], it takes around 200000 epochs for *Raw* to learn reasonable behavior. Each epoch consists of 100 neural network updates.

7 Related Work

Autonomizer is related to many proposed systems for program autotuning or input selection [8, 47, 54–67]. Specifically, [8] is a white-box tuning framework that tunes program parameters regarding program internal variables. However, with Autonomizer, no tuning or searching is needed for the autonomized program to process each new input, so the application to large volume of inputs becomes feasible. Multiple works were proposed to observe program internal behaviors [68, 69], but Autonomizer proposes a systematic way to select important program variables as model inputs.

Some other works were proposed for solving software engineering problems with machine/deep learning techniques, such as test generation [70–73], fuzzing [74–76], and bug repair [77–79]. Autonomizer has the potential of allowing some learning tasks to piggyback on software operation.

Machine learning and deep learning techniques have a wide range of applications such as computer vision [80–82], speech recognition [83, 84], and bioinformatics [85, 86]. For individual application, the developers have to compose the learning procedures from scratch and use raw data. In fact, many of these problems have been extensively studied and they have existing solutions based on programs. The problem is that these programs are often heavily parameterized and require human interventions. Autonomizer provides a general technique to autonomize such programs.

While there are some works that leverage internal data (e.g., in training AIs to play Mario [87] and a first-person shooting game [88]), they are application specific.

Some frameworks have been proposed to train models for playing games such as OpenAI Gym [17], Arcade Learning Environment [89], and Mario AI competition [90]. However, these frameworks are limited to specific platforms and the training is a stand-alone process isolated from the original system operation. In contrast, Autonomizer is general and the training is piggybacking on software operation.

In [91], researchers propose a technique to play NES games automatically. Unlike Autonomizer that works on source, it works on executable. It first identifies the locations of simulated NES memory that stores the progress (i.e., score or game level) of the NES game. Then the objective function is derived and learned accordingly in order to make progress. Unlike Autonomizer that aims to support various software systems, the work is specific to playing NES games.

8 Conclusion

We propose a novel technique that supports autonomization of existing software systems that require human interactions/interventions. It features a novel execution model facilitated by several programming primitives. The developer can instruct an AI model to learn from the normal operations of the software by adding few invocations to these primitives in the source code. Our system Autonomizer then transparently weaves the model training and deployment into program execution, hiding all the complexities such as collecting data, extracting features, and replacing user interactions/interventions with the AI model. Our experiments on nine real world programs show that very little human effort is required to autonomize these programs with Autonomizer. The autonomized versions produce results with higher quality. Autonomized games can play by themselves and have performance competitive with human players with less training time.

Acknowledgements

We thank our shepherd Zheng Zhang and the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1748764 and 1409668, Sandia National Lab under award 1701331, and ONR under contracts N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[2] (2018) Alphago versus lee sedol. [Online]. Available: https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol

[3] (2018) GoogleâĂŹs alphago defeats chinese go master in win for a.i. [Online]. Available: https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html

[4] (2018) Alphago at the future of go summit,. [Online]. Available: https://deepmind.com/research/alphago/alphago-china/

[5] (2018) Waymo. [Online]. Available: https://waymo.com/

[6] (2018) Waymo. [Online]. Available: https://www.cnbc.com/2018/08/29/waymo-alphabets-self-driving-unit-morgan-stanley-forecast-to-highs.html

[7] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1611.01578

[8] W.-C. Lee, Y. Liu, P. Liu, S. Ma, H. Choi, X. Zhang, and R. Gupta, "White-box program tuning," ser. CGO 2019, 2019.

[9] (2018) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/monkey

[10] S. Bling. (2018) Mario bizhawk emulator. [Online]. Available: https://pastebin.com/u/SethBling

[11] (2018) Deepmind ai reduces google data centre cooling bill by 40. [Online]. Available: https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[13] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *Int. J. Inf. Syst. Model. Des.*, vol. 6, no. 3, pp. 46–83, Jul. 2015. [Online]. Available: http://dx.doi.org/10.4018/IJISMD.2015070103

[14] A. Jung. (2018) mario-ai: Playing mario with deep reinforcement learning. [Online]. Available: https://github.com/aleju/mario-ai

[15] (2017) Torcs for reinforcement learning. [Online]. Available: https://github.com/YurongYou/rlTORCS

[16] (2016) Using keras and deep deterministic policy gradient to play torcs. [Online]. Available: https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html

[17] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[18] (2018) Skydio. [Online]. Available: https://www.skydio.com/

[19] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep face recognition," in *BMVC '15*, 2015.

[20] A. Irpan. (2018) Deep reinforcement learning doesn't work yet. [Online]. Available: https://www.alexirpan.com/2018/02/14/rl-hard.html

[21] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, 2017.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[23] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, 1986.

[24] P. Lamere, P. Kwok, E. Gouvea, B. Raj, R. Singh, W. Walker, M. Warmuth, and P. Wolf, "The cmu sphinx-4 speech recognition system," in *ICASSP '03*, 2003.

[25] L. Jakowski. (2018) jakowskidev/umario_jakowski: umario c++/sdl2 game by lukasz jakowski. [Online]. Available: https://github.com/jakowskidev/uMario_Jakowski

[26] Autnomizer. (2018) Autonomizer. [Online]. Available: https://github.com/ProjectDemooo/Autonomizer

[27] (2018) Using deep q-network to learn how to play flappy bird. [Online]. Available: https://github.com/yenchenlin/DeepLearningFlappyBird

[28] d. t. GNU. (2018) Using and porting the gnu compiler collection (gcc): Gcov. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

[29] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, 2014.

[31] R. Hecht-Nielsen, "Neural networks for perception," 1992, ch. Theory of the Backpropagation Neural Network.

[32] (2018) sklearn scale. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.minmax_scale.html

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI '16*, 2016.

[34] Python2.7. (2018) Extending python with c or c++. [Online]. Available: https://docs.python.org/2/extending/extending.html

[35] d. t. CRIU. (2018) Criu. [Online]. Available: https://criu.org/Main_Page

[36] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[37] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[38] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI 2007*, 2007.

[39] W. M. Khoo. (2013) wmkhoo/taintgrind - github. [Online]. Available: https://github.com/wmkhoo/taintgrind

[40] C. A. Rothwell, J. L. Mundy, W. Hoffman, and V. D. Nguyen, "Driving vision by topology," in *ISCV '95*, 1995.

[41] D. Plotree and D. Plotgram, "Phylip-phylogeny inference package (version 3.2)," *cladistics*, vol. 5, 1989.

[42] (2017) Flappy bird c++. [Online]. Available: https://github.com/Gear-Code/Flappy-Bird

[43] AndreaOrru. (2017) Cycle-accurate nes emulator. [Online]. Available: https://github.com/AndreaOrru/LaiNES

[44] (2016) Torcs, the open racing car simulator. [Online]. Available: http://torcs.sourceforge.net/

[45] S. A. Bradford W. Mott and T. S. Team. (1995) Stella: A multi-platform atari 2600 vcs emulator. [Online]. Available: https://stella-emu.github.io/

[46] M. Heath, S. Sarkar, T. Sanocki, and K. Bowyer, "Comparison of edge detectors: a methodology and initial study," *Computer vision and image understanding*, vol. 69, no. 1, pp. 38–54, 1998.

[47] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *PACT '14*, 2014.

[48] D. stutz, "Introduction to neural networks," *Selected Topics in Human Language Technology and Pattern Recognition*, 2014.

[49] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik, "Contour detection and hierarchical image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 5, pp. 898–916, 2011.

[50] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 13, no. 4, 2004.

[51] V. Mnih, A. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, 2016.

[52] Y. You, X. Pan, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," *CoRR*, 2017.

[53] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *ICCV '15*, 2015.

[54] T. Katagiri, K. Kise, H. Honda, and T. Yuba, "Fiber: A generalized framework for auto-tuning software," in *HPC '03*, A. Veidenbaum, K. Joe, H. Amano, and H. Aiso, Eds., 2003.
[55] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *ICML '15*, 2015.
[56] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *GECCO '15*, 2015.
[57] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: A framework for engineering self-tuning self-adaptive software systems," in *FSE '10*, 2010.
[58] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *IPDPS '09*, 2009.
[59] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *SIGPLAN Not.*, vol. 46, no. 3, 2011.
[60] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT, Tech. Rep., 2009.
[61] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, 2009.
[62] V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed, "Enabling self-managing applications using model-based online control strategies," in *ICAC, 2006*.
[63] F. Chang and V. Karamcheti, "A framework for automatic adaptation of tunable distributed applications," *Cluster Computing, 2011*.
[64] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *SIGPLAN Not.*, vol. 45, no. 6, Jun. 2010.
[65] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *PLDI 2009*, 2009.
[66] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *PLDI 2015*, 2015.
[67] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "Siblingrivalry: Online autotuning through local competitions," in *CASES 2012*, 2012.
[68] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *MICOR '97*. Washington, DC, USA: IEEE Computer Society, 1997.
[69] D. Notkin and T. Xie, "Checking inside the black box: Regression testing by comparing value spectra," *IEEE Transactions on Software Engineering*, 2005.
[70] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *ICSE '17*, 2017, pp. 643–653.
[71] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *ACM Sigplan Notices*, vol. 48, no. 10, 2013, pp. 623–640.
[72] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *ICSE '15*, 2015.
[73] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *ASE '16*, 2016, pp. 63–74.
[74] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *ASE '17*, 2017, pp. 50–59.
[75] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *PLDI '17*, 2017, pp. 95–110.
[76] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *CoRR*, 2018.
[77] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning." in *AAAI '17*, 2017, pp. 1345–1351.
[78] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *CoRR*, 2017.
[79] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports," in *ASE'15*, 2015, pp. 476–481.
[80] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS '12*, 2012.
[81] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. Lecun, "Pedestrian detection with unsupervised multi-stage feature learning," in *CVPR '13*, 2013.
[82] W. Shen, X. Wang, Y. Wang, X. Bai, and Z. Zhang, "Deepcontour: A deep convolutional feature learned by positive-sharing loss for contour detection," in *CVPR '15*, 2015.
[83] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on , Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012.
[84] A. Graves, A. r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *ICASSP '13*, 2013.
[85] Y. Wang, H. Mao, and Z. Yi, "Protein secondary structure prediction by using deep learning method," *Knowledge-Based Systems*, vol. 118, pp. 115 – 123, 2017.
[86] Y. Chen, Y. Li, R. Narayan, A. Subramanian, and X. Xie, "Gene expression inference with deep learning," *Bioinformatics*, vol. 32, no. 12, pp. 1832–1839, 2016.
[87] Y. Liao, K. Yi, and Y. Zhe, "Cs229 final report reinforcement learning to play mario," 2012.
[88] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, 2016.
[89] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.
[90] (2012) Mario ai championship 2012. [Online]. Available: http://www.marioai.org/
[91] D. Tom and M. V. P. D, "The first level of super mario bros. is easy with lexicographic orderings and time travel... after that it gets a little tricky." 2013.