Test Input Generation Using Deep Q-Learning Based Generators

Sicheng Jiang **Caroline Lemieux**

April 29, 2020

ABSTRACT

Generator-based testing tools such as QuickCheck can effectively generate random test inputs for programs. However, simple hand-written generators used with these tools may fail to capture the semantic structure of the inputs generated and therefore generate mostly invalid inputs for some programs. A reinforcement learning based approach called RLCheck can generate inputs that tend to be both valid and diverse using Q-table based generators. RLCheck works well but has several limitations. One is that it requires the user to provide a good state representation function, which can have a significant impact on the unique rate of the inputs generated. The other limitation is due to the fact that it has to save every state it meets in the Q-table, which makes the memory usage grow rapidly.

In this work, we replace the Q-table in RLCheck with a neural network and try to use the Deep Q-Learning to overcome the two limitations above. We make the neural network to learn a proper state representation function, and use a curiosity-driven reward to encourage the agent to explore different states. Finally, we use a recurrent neural network (RNN) to encode a variable-length state into a fixed-length representation so that the generator can utilize more information before it selects the next action.

1 Problem Setup

We use O to represent a domain of objects, and G to represent a generator which is a non-deterministic program returning an object in O every time it is called. And we use $V \subseteq O$ to represent all objects in O that are valid inputs for the program under test. Notice that V is a black box in our setting and we expect G to learn the semantic structure of V. Let L represent an oracle (or learner) that is initialized with a fixed action space C_L and non-deterministically returns an action in C_L every time it is called. And we use C to represent the union of all learners L_i, i.e. $C = \bigcup_i C_{L_i}$

We expect the client of the testing framework to write a relatively simple program P_o to specify O. And in P_o, the user will ask the oracle L to select a proper action at each choice point. Notice that P_o and L together make up G.

The goal of this paper is to implement an oracle L that learns to generate unique objects in V using neural networks. And here is an example of an XML generator client code where O = a set of XML documents:

```
class XML_RLGenerator:
    max_num_children = 3
    max_num_attributes = 2

    def __init__(self, dict_items):
        self.max_depth = 4 # restrict documents to depth 4
        self.dict_items = dict_items # XML tags

    def generate(self, oracle):
        self.depth = 0
        return self.generate_node(0, oracle)

    def generate_node(self, depth, oracle):
        tag = oracle.select(self.dict_items, 1)
        cur_node  = etree.Element(tag)
        num_attributes = oracle.select(range(self.max_num_attributes),
2)
        for _ in range(num_attributes):
            cur_node.set(oracle.select(self.dict_items, 3),
                             oracle.select(self.dict_items, 4))
        if depth < self.max_depth and oracle.select([True, False], 5):
            num_children = oracle.select(range(self.max_num_children),
                                          10)
            for i in range(num_children):
                child = self.generate_node(depth + 1, oracle)
                cur_node.append(child)
        elif oracle.select([True, False], 6):
            text = oracle.select(self.dict_items, 7)
            cur_node.text = text
        return cur_node
```

2 Generator using simple feed-forward neural network

In this section, we present an implementation of a generator using a group of 2-layer fully connected feed-forward neural networks.

2.1 Architecture

In this approach, each choice point with index *i* (where *i* is usually the line number) will be backed by a different learner L_i, and each learner is a 2-layer neural network with the following architecture:

```
Linear(input_dim, hidden_dim)
ELU(alpha=0.5)
Normalize()
Linear(hidden_dim, output_dim)
```

where hidden_dim is set 12 in our examples. input_dim = $C \times w + 1$, where *w* is the number of actions we used to determine the current state of the learner, and the extra '+1' is used to indicate whether the current state is a terminal state. And output_dim is simply C_{L_i}. And we use $Q_i : R^{D_{in}} \to R^{D_{out}}$ to denote this neural network, where *R* is the set of real numbers.

2.2 Training

We use on-policy training with experience replay inspired by [2]. The generator will generate an input in each episode. In every episode, when a choice point *i* is executed in the generator, a corresponding learner L_i will be called. L_i stores all the previous actions it performed in this episode in a list S_i, and it looks at *w* prior actions and output the next action. We define $s_t = S_i[-w:]$ as its current state, where the right side of the equation is the last *w* terms of S_i. Then, s_t is one-hot encoded into a vector $\phi_t = E(s_t)$, where *E* is an encoding function, and ϕ_t is passed into Q_i and generates an output $\varphi_t = Q_i(\phi_t) \in R^{D_{out}}$, where $D_{out} = |C_{L_i}| = \text{the size of the action domain of } L_i$. And an action *c* is chosen as follows (where ε is set to 0.25 in our implementation):

```
if UniformRandom() < $\varepsilon$ then
    $c \leftarrow \text{Random}(C_{L_i})$
else
    $c \leftarrow C_{L_i}[\text{argmax}_{j \in \{0,1,\ldots,D_{out}-1\}} \varphi_t[j]]$
```

And *c* is then appended to S_i and returned by the learner L_i. Also, every time after a transition from one state φ_t to another state φ_{t+1} due to an action c_t, the transition $\eta_t = (\varphi_t, c_t, \varphi_{t+1})$ is appended to a list *T*.

After an episode has finished, each L_i will receive a reward r_E indicating whether the input it has just generated is uniquely valid, just valid, or invalid. Besides that, an additional intrinsic reward $r_I^{(j)}$ will be calculated for each transition η_j. The details about the intrinsic reward are discussed in section 2.3.

Then, the transitions stored in *T* will be grouped into mini-batches, and a loss will be calculated for each mini-batch. Then, an optimization step will be performed on the loss.

```
**for** mini-batch $(\varphi_j, c_j, \varphi_{j+1})$ in $T$ **do**
    Set $y_j = r_E + r_I^{(j)}$ for terminal $\varphi_{j+1}$
    Set $y_j = r_E + r_I^{(j)} + \gamma \max_c Q(\varphi_{j+1})[c]$ for non-terminal $\varphi_{j+1}$
    Perform an optimization step on $(y_j - Q(\varphi_{j+1})[c])^2$
**end for**
```

2.3 Using curiosity-driven intrinsic reward

To encourage the learner to explore more states, we use an additional intrinsic reward inspired by [3] and [4]. And we chose to use Random Network Distillation described in [4] to compute the intrinsic reward. In particular, we have a randomly initialized neural network $F : R^{D_{in}} \to R^m$ which encode an state into a feature space with dimension *m* and is fixed after the initialization. And there is another neural network $P : R^{D_{in}} \to R^m$ we are going to train to approximate F. The hidden layers of *P* have a smaller dimension than those of *F* to prevent overfit.

Every time we update the Q network after each episode, we also train *P* to approximate *F*. Then the loss we compute in this process can be used as an indicator of how familiar is the inputted state to the learner. The intuition is that if the learner has encountered a state a lot, the *P* net will have a low prediction loss on the output of *F*. Then, we normalize this prediction loss by its running average and running standard deviation and set this as the intrinsic reward $r_I^{(j)}$ of a transition $\eta_j = (\varphi_j, c_j, \varphi_{j+1})$, because a state is not very interesting if the learner has met it a lot of times.

```
**for** mini-batch $(\varphi_j, c_j, \varphi_{j+1})$ in $T$ **do**
    $l_{j+1} \leftarrow (F(\varphi_{j+1}) - P(\varphi_{j+1}))^2$
    $\mu_l \leftarrow \alpha\mu_l + (1-\alpha) l_{j+1}$        where $\mu_l$ is the running average of $l$
    $v_l \leftarrow a v_l + (1-\alpha) {l_{j+1}}^2$        where $v_l$ is the running variance of $l$
    $r_I^{(j)} \leftarrow (l_{j+1} - \mu_l) / \sqrt{v_l}$
    Perform an optimization step on $l_{j+1}$
    Set $y_j$ and perform a optimization step on $(y_j - Q(\varphi_{j+1})[c])^2$ as discribed in 2.2
**end for**
```

2.4 Memory replay

For some examples that are difficult to generate a valid input randomly, we have to use experience replay introduced by [2], where we store all transitions that lead to a valid generated input in a memory *M* for each learner. The old transitions will be removed from *M* if *M* is full in order to allow new transitions to be stored.

If the rate of valid inputs is lower than a threshold, then before the updating procedure described in section 2.2 happens, some transitions will be sampled from *M* (we only sample one transition) and append to *T* to allow the learner to replay some successful experience it had before.

Notice that the reward of these sampled transitions should be adjusted accordingly, i.e. they do not share the same reward with the transitions produced in that episode, but have a different reward for producing valid inputs.

2.5 Bootstrapping

Besides memory replay, we also use bootstrapping to first train the learner offline for several thousands of episodes by executing the actions of an RLCheck learner instead of our Deep Q-Learner because RLCheck learner usually can start to generate many valid inputs before the Deep Q-Learner. This is probably because the neural network has multiple layers to learn and therefore need more training for the gradients to propagate throughout the entire network.

Bootstrapping is shown to be helpful for the Maven XML example but it makes no difference for some other examples such as the JavaScript example.

2.6 Results

We tested the deep q-learner on the Maven XML example and the JavaScript example and compared it to QuickCheck, RLCheck, and Zest (for JavaScript example only).

We use the last 4 actions as the abstract state for both RLCheck and Deep Q-Learner in both examples. And the rewards are $r_{unique} = 20$, $r_{valid} = 0$, $r_{invalid} = -1$ for both methods, but the rewards for the Deep Q-Learner is normalized before used for updating.

2.6.1 Maven XML

Table 1: Maven XML Results

Method	Num Valid	Num Unique	Num Total	Time Total (s)	Unique to Total Ratio	Unique per second
Random	52	2	50000	5.64	0.00004	0.3546
RLCheck	22250	8553	50000	40.12	0.17106	213.2
DeepQ-Learner (no intrinsic reward)	18847	10751	50000	1076.83	0.21502	9.980
DeepQ-Learner	17688	11619	50000	1122.67	0.23238	10.35

The first 1000 episodes of the Deep Q-Learner are bootstrapped by RLCheck.

The result shows that the Deep Q-Learner can produce more unique inputs than QuickCheck and RLCheck for the same number of total inputs generated. However, it runs almost 200 times slower than QuickCheck and 30 times slower than RLCheck, therefore it produces much less unique inputs per second than RLCheck. Also, the intrinsic reward is shown to improve the unique rate a little bit in this example.

Figure 1: Maven XML input distributions

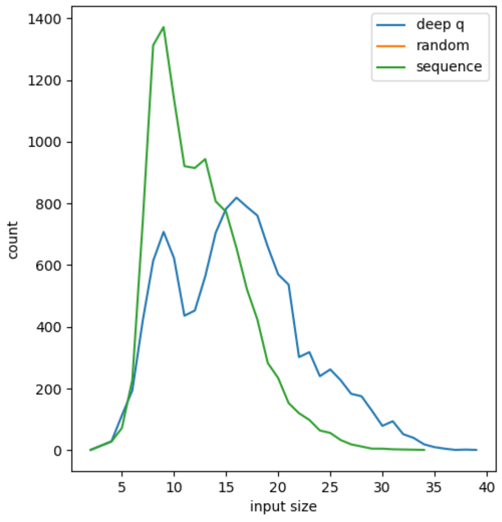
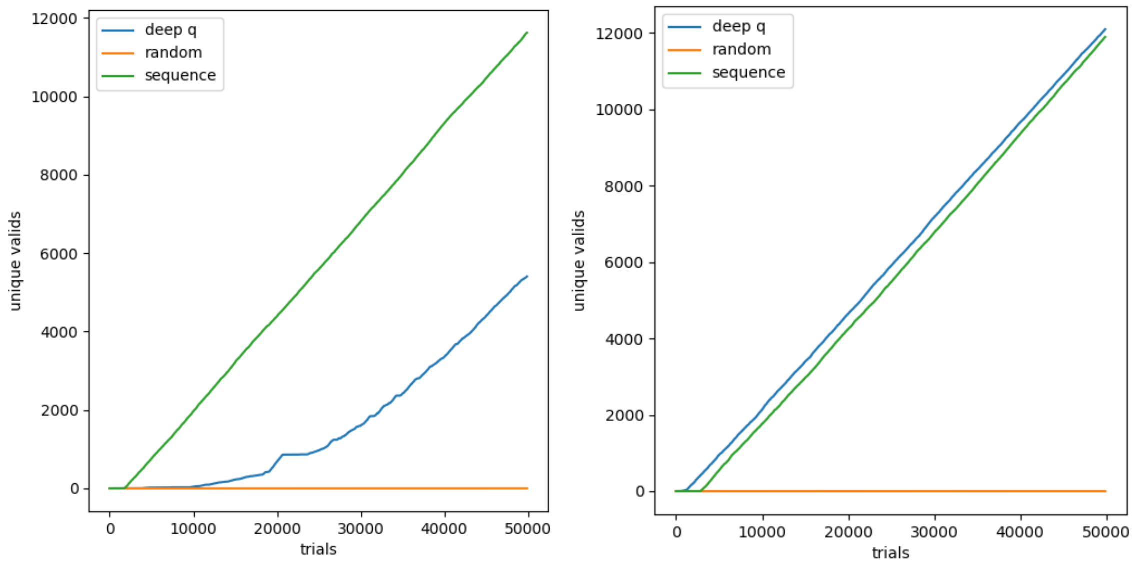


Figure 1 shows the distribution of the length of test inputs generated by the three methods, where the green line is RLCheck and the blue line is Deep Q-Learner (bootstrapped). We can see that the length of the inputs generated by Deep Q-Learner is more widely distributed and tend to be longer than those generated by RLCheck. This indicates that Deep Q-Learner may be able to generate more diverse inputs than RLCkeck and QuickCheck in some situations.

Figure 2: Effect of Bootstrapping for Maven XML



The Deep Q-learner in the left figure is not bootstrapped, and the one on the right is bootstrapped for the first 1000 episodes.

Figure 2 shows that bootstrapping can help the learner to quickly learn the valid syntax of Maven and generate unique valid inputs at its maximum rate from the beginning.

2.6.2 JavaScript

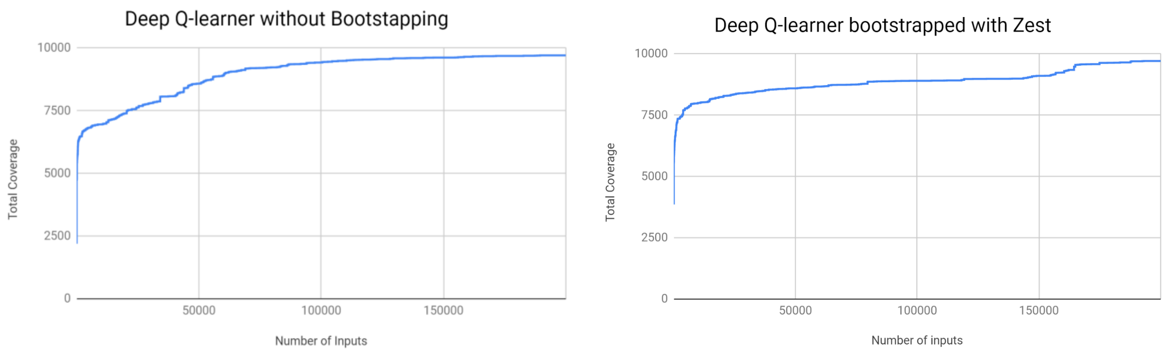
Table 2: JavaScript Results

Method	Num Valid	Num Unique	Num Total	Time Total (s)	Unique to Total Ratio	Unique per second	Total Coverage
Random	19707	4205	100018	1253	0.04204	3.356	8553
RLCheck	72809	54593	100000	1477	0.54593	37.73	8906
Zest	36775	16889	100000	1253	0.16889	13.47	9249
DeepQ-Learner (no intrinsic reward)	81311	67780	100000	-	0.67780	-	8886
DeepQ-Learner	46336	18936	100000	19179	0.18936	0.9592	9428

The Deep Q-Learner is not bootstrapped in this example.

The results show that Deep Q-Learner without intrinsic reward can generate the most unique inputs in the first 100k episodes, but Deep Q-Learner with intrinsic reward is able to cover more branches. Also, we notice that although Deep Q-Learner has comparable performance to RLCheck and Zest, it still runs much slower than these methods.

Figure 3: Effect of Bootstrapping for the JavaScript Example



The Deep Q-learner on the right is bootstrapped with Zest for the first 100,000 episodes. The left one is not bootstrapped. They end up at almost the same coverage after 200,000 episodes.

Figure 3 shows that bootstrapping does not improve the performance of the JavaScript example. This could be due to the fact that it is relatively easier to randomly generate a valid JavaScript program.

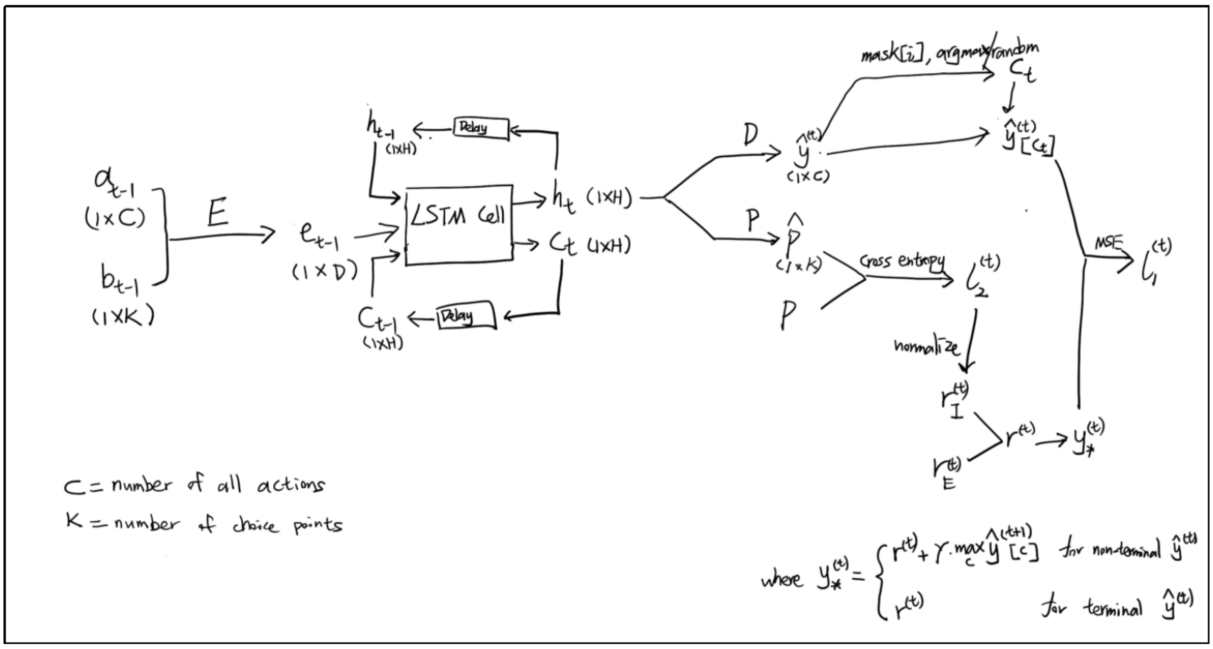
3 Generator using recurrent neural network

In this section, we discuss an alternative approach using a recurrent neural network, so that the learner can take all previous actions as the current state.

3.1 Architecture

Since the RNN takes much longer to train, we decide to use the same neural network for all choice points.

Figure 4: RNN Learner



When the generator calls the RNN learner at a choice point *i*, the learner takes the previous action c_{t-1} and the previous choice point index *k* as the input, encodes them into two vectors a_{t-1} and b_{t-1} correspondingly, and passes them through an encoding network $E\colon R^{C+K} \to R^{D}$, which outputs a vector e_{t-1}. Then, e_{t-1} goes through an LSTM cell, and the outputted hidden layer h_t is used as the input of a decoder *D* and a predictor *P*. *P* is used to predict the index *i* of the current choice point, and the prediction loss ($l_2^{(t)}$) will be used as the intrinsic reward of the previous transition. The output of *D* is the score of each action of the current choice point *i*. A mask is used before taking the argmax to make sure the action chosen by the learner is within the domain of the current choice point.

The training process is very similar to the previous approach. We calculate the MSE loss ($l_1^{(t)}$) for the action c_t that the learner chose, and perform an optimization step on $l_1^{(t)} + l_2^{(t)}$. The intuition is that a small $l_1^{(t)}$ will make the learner learn to generate unique valid inputs, and optimizing $l_2^{(t)}$ will help the network to learn the semantic information behind the choices and also generate an intrinsic reward.

3.2 Results

We directly connect a Q-network with a LSTM cell in our approach. There is no similar previous work, and we did not know whether this would work before we tested it. In fact, this RNN Learner does not work as we expected and its behavior is unstable and unpredictable, which indicates that the design above is probably wrong and needs some modifications before proceeding further. Table 3 lists the results of some different runs of the RNN Learner for the JavaScript Example.

Table 3: RNN Learner results for the JavaScript Example

Sample	Num Valid	Num Unique	Num Total	Time Total (s)	Unique to Total Ratio	Unique per second	Total Coverage
1	2079	201	2489	146	0.0807	1.3767	6661
2	13	9	358	240	0.0251	0.0375	5739
3	1113	174	1455	114	0.1196	1.5263	6807
4	176	104	1257	101	0.0874	1.0297	6526
5	35	23	597	670	0.0385	0.0343	5889
6	2809	719	14706	707	0.0489	1.0170	7081

We observe that the RNN learner usually fails to learn to generate valid inputs, and the generating speed differs greatly each time. It sometimes tries to produce inputs containing thousands of choices, which makes the run time significantly longer. This behavior tends to appear more often when running on a GPU.

We have made many attempts to make it stable, such as adding an extra reward to control the length of the episode, turning off the intrinsic reward, using the same reward for unique valid and valid inputs, reducing the size of the network, but none of these really work.

4. Future Work

4.1 Algorithm Changes

For the RNN Learner, we can first train an RNN to represent the state using the transitions of another generator such as RLCheck, then fix that RNN and train a Q-net using the approach in section 2. This approach simplifies the problem and also allows the learner to take a variable-length state as input.

4.2 Other Approaches

Transformer Models Transformer models are widely used in natural language translation and are computationally cheaper than RNN [6]. We can try to replace the RNN with a transformer because RNN may not work well when the input sequence is too long (when there are long-term dependencies) while the transformer does not suffer from this problem because it uses attention.

Graph Neural Networks We notice that RNN may still not be able to fully represent the semantic information of a test input object, which could be a graph in some complex examples such as JavaScript and PDF. Therefore GNN might be a better choice in these cases because it can directly represent a graph.

Generative Adversarial Networks We can mutually train two networks to play against each other. A generative network generates test inputs, and a discriminator network rejects the input generated if it has seen a similar input before. In this way, the generative network might be able to generate more diverse inputs based on the feedback from the discriminator network.

REFERENCES

[1] Sameer Reddy, Caroline Lemieux, Rohan Padhye, Koushik Sen. "Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning" In Proceedings of ICSE 2020.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning" arXiv:1312.5602v1 [cs.LG] 19 Dec 2013

[3] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, Trevor Darrell. "Curiosity-driven Exploration by Self-supervised Prediction" arXiv:1705.05363 [cs.LG] 15 May 2017

[4] Yuri Burda, Harrison Edwards, Amos Storkey, Oleg Klimov. "Exploration by Random Network Distillation" arXiv:1810.12894v1 [cs.LG] 30 Oct 2018

[5] Sameer Reddy, Caroline Lemieux, Koushik Sen. "Reinforcement-Learning Based Tuning of Generators for Fuzzing" UC Berkeley Technical Report, 2019.

[6] John Canny. UC Berkeley CS182 Spring 2020 Lecture 13.