# ACCELERATION OF HARDWARE VERIFICATION USING DEEP REINFORCEMENT LEARNING

*A Project Report*

*submitted by*

**SHILPA N**

*in partial fulfilment of the requirements*
*for the award of the degree of*

**BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY**

**DEPARTMENT OF ELECTRICAL ENGINEERING**
**INDIAN INSTITUTE OF TECHNOLOGY MADRAS.**

**JUNE 2021**

# THESIS CERTIFICATE

This is to certify that the thesis titled **ACCELERATION OF HARDWARE VERIFICATION USING DEEP REINFORCEMENT LEARNING**, submitted by **Shilpa N**, to the Indian Institute of Technology, Madras, for the award of the degree of **Dual Degree(B.Tech+M.Tech)**, is a bona fide record of the research work done by her under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Prof. Pratyush Kumar**
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

**Prof. Janakiraman Viraraghavan**
Department Co-Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 28th June 2021

# ACKNOWLEDGEMENTS

Thanks to all friends, family and faculty of IIT Madras who made this project possible.

# ABSTRACT

KEYWORDS: Hardware verification; Reinforcement learning; Open source; Metric of Performance; Automation algorithm

Traditional hardware verification which was based on constrained pseudo-random stimulus proved to be inefficient with respect to the time taken for completing the verification of functional correctness of designs. There has been significant work done in recent years on using machine learning for intelligently suggesting inputs to the test generator so that rare and hard to hit events are targeted more often during the verification task. This project proposes a open source solution on using Deep Reinforcement Learning for achieving the same. A metric of performance to evaluate the quality of the event-coverage results achieved in the Automated-RL verification over the pseudo-random verification process and an algorithm which automates the process of carefully supplying the reward structure and other hyper-parameters to the RL agent in-order to increase this metric value of the Automated-RL run over the metric value of the pseudo-random run are proposed. Additionally the report includes experimental results of this algorithm tested on two designs - RLE-Compressor and COO-Compressor.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABBREVIATIONS

| | |
|---|---|
| **RL** | Reinforcement Learning |
| **Deep RL** | Deep Reinforcement Learning |
| **ML** | Machine Learning |
| **RLE** | Run Length Encoding |
| **COO** | COOrdinate list |
| **CDTG** | Coverage Directed Test Generation |
| **TG** | Test Generator |
| **DQN** | Deep Q-Network |
| **DDPG** | Deep Deterministic Policy Gradient |
| **SAC** | Soft Actor Critic |
| **LR** | Learning Rate |
| **TF** | Train Frequency |

# CHAPTER 1

# INTRODUCTION

As digital designs are getting increasingly complex and sizeable, the time for their verification, which usually takes up a major amount in their production time, also goes up significantly. For the verification of hardware designs, the techniques of formal verification(which includes model checking, theorem proving etc.) or simulation-based verification or a hybrid of the two are adopted. Simulation-based verification broadly involves using random input stimulus or carefully designed set of input signals to test the design for bugs. The output of the design is compared to the output of a reference golden model, to identify bugs in the design.

Traditionally, for simulation-based verification, constrained random stimulus, which is the process of random input generation with constraints on their values based on design specifications, used to be the norm. But as the size of these digital circuits is going up, the number of inputs required to fully verify the circuit blows up exponentially and the constrained random stimulus verification method might take forever to ensure correctness of the designs. So, in this project we model the task of verification as a reinforcement learning(RL) problem, where the goal of the RL agent is to suggest more of such inputs to the design-under-test(DUT) which will drive the design to more uncommon and faulty internal states. This will help us check the system's behaviour more rigorously and enable us to catch bugs easily.

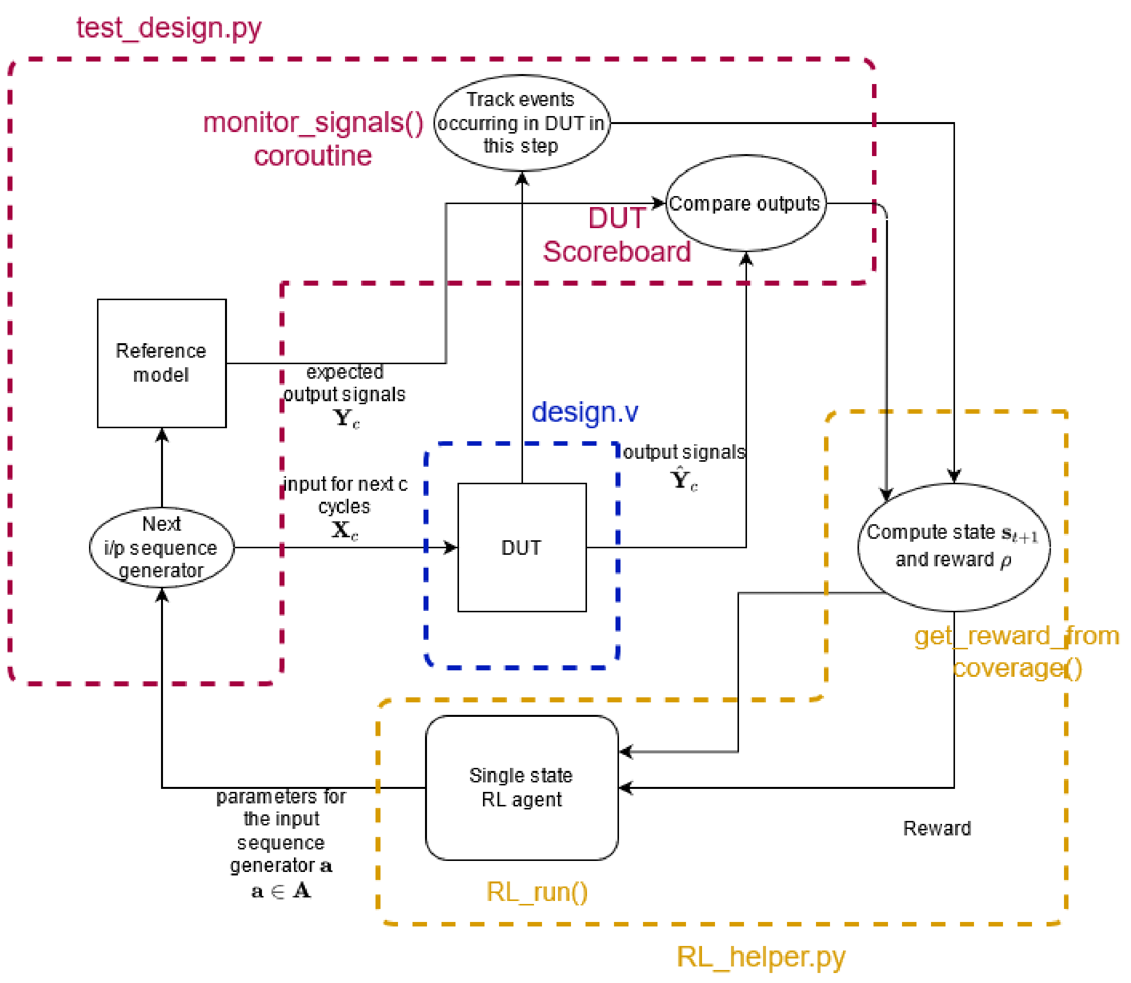
This report gives a brief about hardware verification method commonly in use, the ML/RL algorithms which are used to aid it, the open-source framework that is adopted as part of this project for the task of verification - which involves a DUT, a cocotb layer integrated with a RL layer and an automation algorithm which runs on top of it. The report also includes experimental results of the framework tested on two designs - RLE-Compressor and COO-Compressor.

# CHAPTER 2

# THEORY AND BACKGROUND

Simulation-based hardware verification which was traditionally done using constrained pseudo-random stimulus turned out to be highly inefficient with respect to the time it took to fully verify designs. Now, there are test benches which work on Coverage Directed Test Generation(CDTG) (3). In this method, after a round of inputs are sent to the test generator(TG) or a set of parameters that directly affect input generation is sent to the test generator(TG), the outputs and the results of coverage(of internal states) from the DUT is collected. Based on these results the next set of inputs/parameters to the TG are carefully designed such that the next round of result would close the coverage holes of the previous round. Researches, both in academia and industry, have now turned to Machine Learning to help close this gap between coverage results and test case generation.

Among the Machine Learning algorithms, this project focuses on using Reinforcement learning(RL) for the above process (2) (1). RL methods have a Markov State Space defined as part of it and this can be used to capture the internal states of the digital designs at any point of time. Also, since the number of possible internal states and available action choices of these designs is huge, we could use Deep RL algorithms to handle it effectively. In the single-state case, which is the most generic method to handle most designs, these algorithms take the form of the multi-armed bandit problem.



test_design.py
monitor_signals() coroutine
Track events occurring in DUT in this step
Compare outputs
DUT Scoreboard
Reference model
expected output signals $\mathbf{Y}_c$
design.v
output signals $\hat{\mathbf{Y}}_c$
input for next c cycles $\mathbf{X}_c$
Next i/p sequence generator
DUT
Compute state $s_{t+1}$ and reward $\rho$
get_reward_from coverage()
Single state RL agent
parameters for the input sequence generator $\mathbf{a}$ $\mathbf{a} \in \mathbf{A}$
Reward
RL_run()
RL_helper.py

Figure 2.1: Block diagram of the RL framework developed

## 2.1 Deep RL algorithms used

The Deep Reinforcement learning algorithms used in the verification framework developed by us are DQN(Deep Q-Network), SAC(Soft Actor Critic) and DDPG(Deep Deterministic Policy Gradient). Their respective implementations from the $Stable-Baselines3$ package have been used directly.

For designs requiring parameters in the continuous action space, only DDPG and SAC are used(as DQN deals with discrete actions).

In designs with parameters in the discrete action space, DQN, DDPG and SAC are used. This is possible by mapping the actions of DDPG and SAC from continuous space to discrete space.

## 2.2 The Framework developed

The framework developed consists of four layers - DUT, cocotb, RL and the automation layer. The innermost layer is the DUT(written in Verilog). The Cocotb layer is the next one and it consists of the test bench(written in python) which provides inputs to the DUT and can be used to monitor the internal signals and the outputs of the DUT. The RL layer interacts with the cocotb layer to intelligently suggest inputs to the DUT such that the coverage of certain targeted events(generally uncommon/rare events) can be increased. The automation layer, which acts as a wrapper on the RL layer, takes as input the accumulated coverage of all monitored events after each round and suggests reward schemes and hyper-parameters(to direct the testing to certain targeted events) to the RL layer such that the coverage holes of the last run are met.

The automation algorithm in this project predicts which is the next rare event to be targeted by the RL agent based on the coverage results obtained. A brief of the algorithm adopted is given in the next section.

## 2.3 The automation algorithm

The automation algorithm has two main objectives:

- Among the events monitored, we give greater weightage to increasing the probability of occurrence of relatively rare events and thereby attempting to reach a more uniform distribution of visited events.
- Drop out from trying to explore events that don't show much increase in their probability of occurrence with the aid of RL.

The automation algorithm works based on the metric of performance we define. The algorithm must target those events during each iteration to increase this metric value such that it is higher or preferably much higher than the metric value of the random run. To explain it better, let us first define the metric.

### 2.3.1 Metric definition

In a given DUT, say, $n$ events are monitored. The pseudo random test-case generation, when run for sufficient time on the DUT, would show a certain probability of occurrence of these `n` events. Let the individual probabilities of occurrence of these events in the pseudo random test-case generation be denoted by the vector,

$$P = [p_1, p_2.., p_n]$$

Note that the values $p1, p2$ etc. are individual probabilities and do not form a probability distribution, i.e., $p_1 + p_2 + ..p_n = 1$ may or may not hold.

Now, we run the Automated-RL test bench on the DUT for the same number of iterations and we note down the new probabilities of occurrence of these events as vector,

$$Q = [q_1, q_2.., q_n]$$

NOTE : These individual probabilities are calculated via the `monitor_signals()` function. The `monitor_signals()` function keeps a track of the occurrence/non-occurrence of these $n$ events by sampling every clock cycle. So the occurrence probability of an event is calculated as the number of clock cycles for which the event occurred in our duration of consideration divided by the total number of clock cycles for which the `monitor_signals()` sampled for events in the duration of our consideration.

For defining the metric, we consider the following criteria :

- The random baseline value of the metric should be constant for a given design.
- For a given same proportionate increase in the probabilities of two events from their respective random test-case generation probabilities, the metric of the Automated-RL run should increase(w.r.t the metric for the random baseline run) more for the increase in occurrence probability of the rarer event.

After consideration of multiple mathematical functions, the metric of performance has been defined as(4):

$$metric_{Automated-RL} = \sum_{i=1}^{n} -(1-p_i)^2 * log(p_i) * (q_i/p_i)$$

The equation can be broken down as:

$$metric = \sum_{i=1}^{n} factor(p_i) * Proportionate\ Change\ In\ Probability\ Of\ Event\ i$$

where,

$$factor(p_i) = -(1-p_i)^2 * log(p_i)$$

$$Proportionate\,Change\,In\,Probability\,Of\,\,Event\,i = (q_i/p_i)$$

Now, $Proportionate\,Change\,In\,Probability\,Of\,\,Event\,i$ refers to the proportionate change in probability of event $i$' in the Automated-RL run when compared to the random test-case generator run. $factor(p_i)$ is a kind of relevance measure given to the proportionate change in probability of event $i$. It is a decreasing function w.r.t $p_i$. Hence, higher this value, more will be the increase in the metric of performance due to a given proportionate change in the probability of an event.

The metric of the random test-case generator run can be found by substituting $q_i = p_i$ in the metric equation of Automated-RL run to obtain,

$$metric_{random} = \sum_{i=1}^{n} -(1-p_i)^2 * log(p_i)$$

The metric value of the pseudo random run on a design is constant for a given design in comparison to the metric value of different Automated-RL runs on the design which keeps changing with the RL algorithm and other hyper-parameters adopted.

Exception of $p_i = 0$ is handled by placing a 0.1 in place of the 0 in the $i^{th}$ coverage count bin, i.e. $p\_i = 0.1/total\ number\ of\ events\ sampled$. This is chosen as an optimistic probability estimate of $p_i$, i.e., we are expecting that the event will show up at least once when simulated 10 times more than the current number of iterations.

## 2.3.2 Algorithm

The automation layer takes as input the accumulated coverage of all monitored events after each round and suggests reward schemes and hyper-parameters(to direct the testing to certain targeted events) to the RL layer of the framework developed such that the coverage holes of the last run are met.

A descriptive pseudo-code version of the algorithm used is given below.

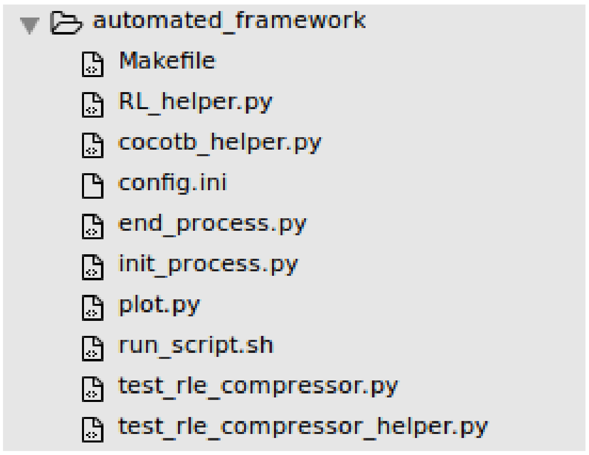
```
#initialization
NUM_EVENTS = 5
```

```
MAX_ITERATIONS = 50
start_value = 0.05
factor_array = [start_value] * NUM_EVENTS
reward_function = [0] * NUM_EVENTS
all_events = [0, 1, 2, 3, 4]

for(i=0; i<MAX_ITERATIONS; i++)
{

   Run the code with the parameters loaded in the "config
      .ini" file using "make SIM=verilator" command

   After the run, collect the accumulated coverage of the
       events till this point of time in terms of their
      occurrence probabilities(Q), and note down the
      reward structure used, R. Also note down the
      accumulated occurrence probabilities till before
      this run, call it Q_prev.

   Since the reward structure is such that only a single
      event is rewarded in one run, using the R value,
      note down the chosen event for rewarding for this
      iteration, call it E

   Calculate s_i = -((1-q_i)**2)*log(q_i) for each event
      i. Call [s_1, s_2.. s_5] = S #exception of q_i=0 is
      handled

   if(i>0)
   {
      Calculate t = scaled_sigmoid((Q[E]-Q_prev[E])/Q[E])
          #exception of Q[E]=0 is handled
      factor_array[E]=t;
```

```
  }

  Event chosen for rewarding in the next run,
  E_next = random.choice(all_events, probability=
     normalize(S*factor_array))

  Update the reward_function in the "config.ini" file by
  "R_next = [0] * NUM_EVENTS; R[E_next] = 1".
  Also, update other RL run hyperparameters as required.

}
```

### 2.3.3 Implementation in Code

The file organization in the automated-RL framework developed for the verification task is as given below:



```
automated_framework
    Makefile
    RL_helper.py
    cocotb_helper.py
    config.ini
    end_process.py
    init_process.py
    plot.py
    run_script.sh
    test_rle_compressor.py
    test_rle_compressor_helper.py
```

Figure 2.2: File organization in the Automated-RL framework

NOTE: The code for the automation algorithm has been inserted into the function called $automation\_algorithm\_and\_logging()$ in the $RL\_helper.py$.

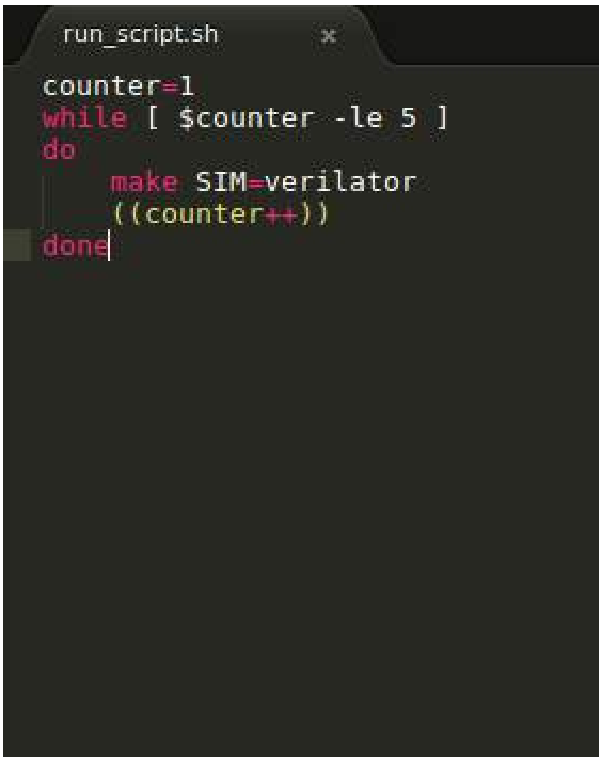
### 2.3.4 How to Run Code

To start the whole verification process, the user must enter into the *automated_framework* directory and run the *init_process.py* file using the command:

```
python init_process.py
```

This file initializes all variables and other hyper-parameters used in the automation run.

After this, the user must open the $run\_script.sh$.



```
counter=1
while [ $counter -le 5 ]
do
    make SIM=verilator
    ((counter++))
done
```

Figure 2.3: run_script.sh file contents

Replace the "5" given in this file with the total number of iterations for which you want the automation algorithm to run, say T iterations. Now run the $run\_script.sh$ file using the commands:

```
chmod +x run_script.sh
```

followed by

```
.\run_script.sh
```

NOTE: If the user wishes to stall the automation algorithm and run using their own desired set of hyper-parameters and reward structures for each iteration, then they have the provision to do so as the partial coverage results for the runs till then are stored in a file. They can do so by changing the $config.ini$ file with their desired set of parameters and then running the command $./run\_script.sh$ again.

If the user is done with the verification task, they should run the $end\_process.py$ file using the command:

```
python end_process.py
```

This file saves the output log from all the runs along with the timestamp and also deletes all intermediate files which were created.

## 2.4 Designs used for experimentation

### 2.4.1 RLE-Compressor

The goal of this hardware is to take in a sequence of natural numbers and compress it using run length encoding. The hardware design thus attempts to reduce the length sequence by representing the consecutive zeros in the input sequence using their count.

### 2.4.2 COO-Compressor

COO-Compressor or COOrdinate List compressor compresses the input map supplied to it based on the index and value of only the non-zero elements in the input map. The two design configuration parameters $word\_width$ and $index\_width$ are the only variables that we alter during the verification process.

# CHAPTER 3

# EXPERIMENTS AND RESULTS

This chapter gives the experimentation results on the two designs discussed above.

## 3.1 RLE-Compressor

In this design we monitor 5 events for the purpose of our experiment.

- Event-0 is *dut.rg_word_counter.value == 16*.
- Event-1 is *dut.rg_zero_counter.value == 64*.
- Event-2 is *dut.rg_counter.value == (2**count_width - 2)*.
- Event-3 is *dut.rg_next_count != 0*.
- Event-4 is *(dut.rg_zero_counter.value == 64) and (dut.rg_next_count != 0)*.

The input suggested by the test generator is a probability value called Z, which is used to determine the probability by which each of the elements in the input activation map(of size 400) supplied to the RLE-compressor is 0. So Z=1 implies all 400 elements of the activation map is 0. This particular choice of action space was because it was identified that the special functioning of the compression algorithm which treats zeros in a particular way and any other non-zero number in another particular way(but all non-zero numbers are treated similarly in the compression algorithm).

### 3.1.1 Pseudo random verification

When the pseudo random test-case generation is used on the RLE-Compressor design for 50 iterations/runs, we notice the following occurrence probabilities of events.

Table 3.1: Occurrence probabilities of events in pseudo random verification process

| Event | Probability |
|---|---:|
| Event-0 | 0.104 |
| Event-1 | 019 |
| Event-2 | 0.00022 |
| Event-3 | 0.015 |
| Event-4 | 0.012 |

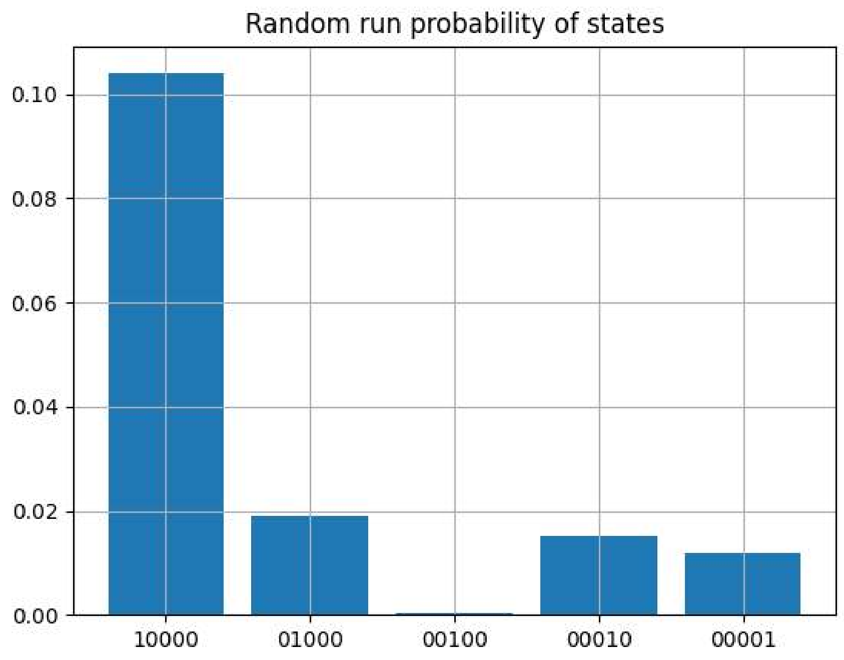


Figure 3.1: Bar graph of occurrence probabilities of events in the random run

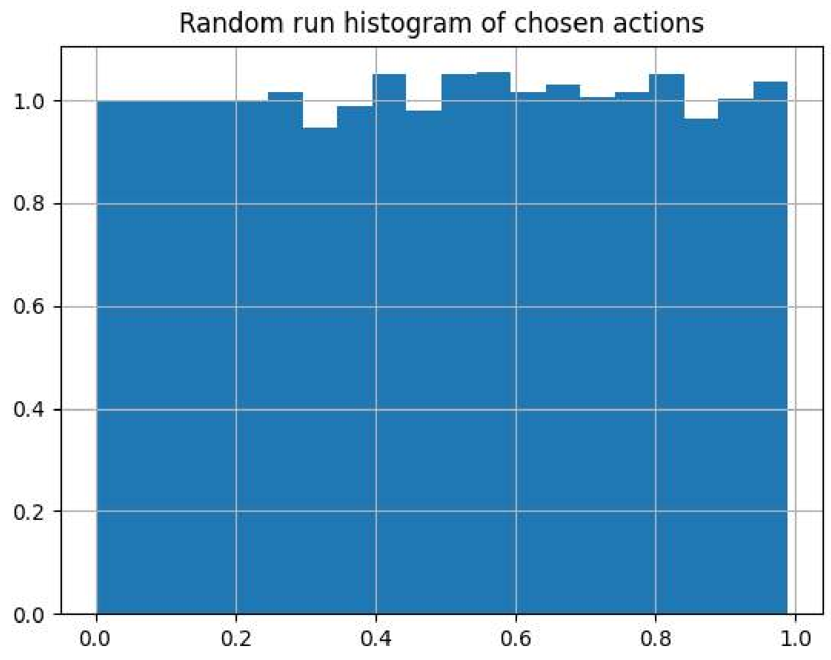


Figure 3.2: Histogram of all chosen actions(Z values) by the test generator during the random run

Now we use the Automated-RL framework to predict the Z value. Since Z is a probability, its value is in the continuous action space. Hence, we use Deep RL methods of SAC and DDPG, which deal with continuous action space.

We consider two settings - the first with a single probability knob predicting a Z value with which the entire input activation map is decided, the second with two probability knobs. The two knobs predict the next element in the activation map based on the value of the element immediately preceding it, i.e., the first knob predicts the probability of the next element being 0 if the previous element is 0 and the second knob predicts the probability of the next element being 0 if the previous element was a non-zero number. Let us call the probability values in the 2 knob setting as $Z = [Z_1, Z_2]$.

So, for this design, we run the automation algorithm for 4 settings in total - SAC-1 knob, SAC-2 knob, DDPG-1 knob and DDPG-2 knob.

The 5 events are denoted by 10000, 01000, 00100, 00010 and 00001 and they respectively correspond to Event-0, Event-1, Event-2, Event-3 and Event-4.

### 3.1.2 Results obtained

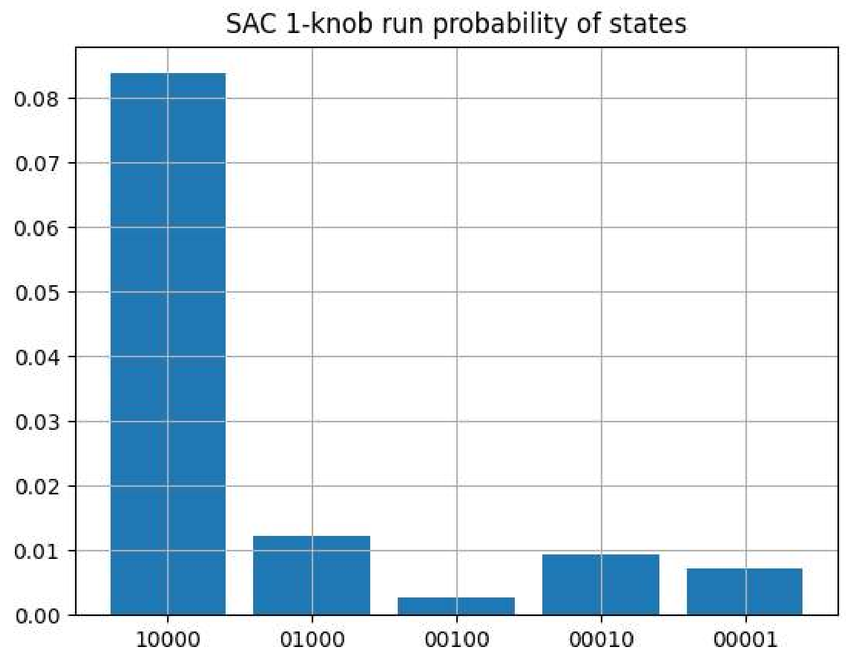


Figure 3.3: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

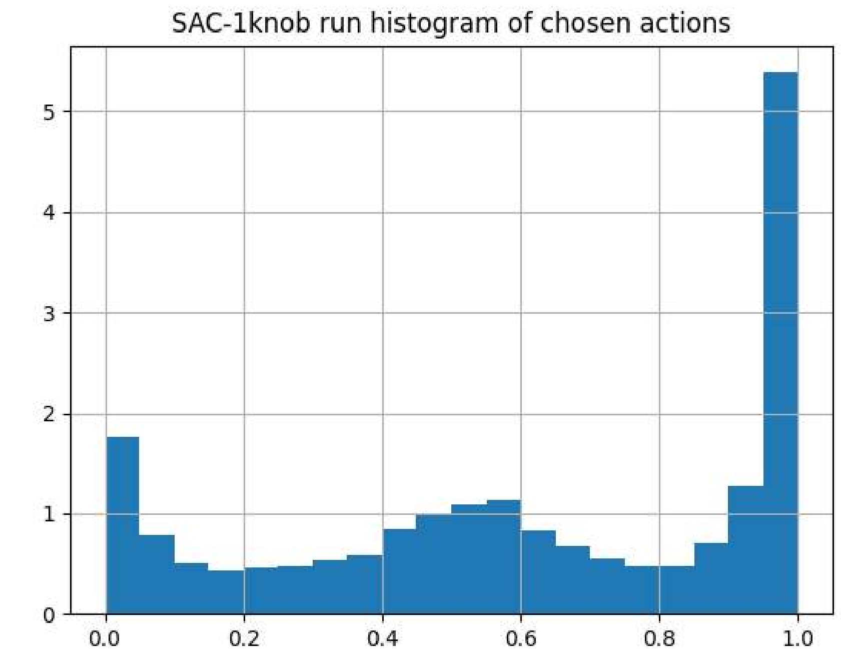


Figure 3.4: Histogram of all chosen actions(Z values) by the test generator during the run

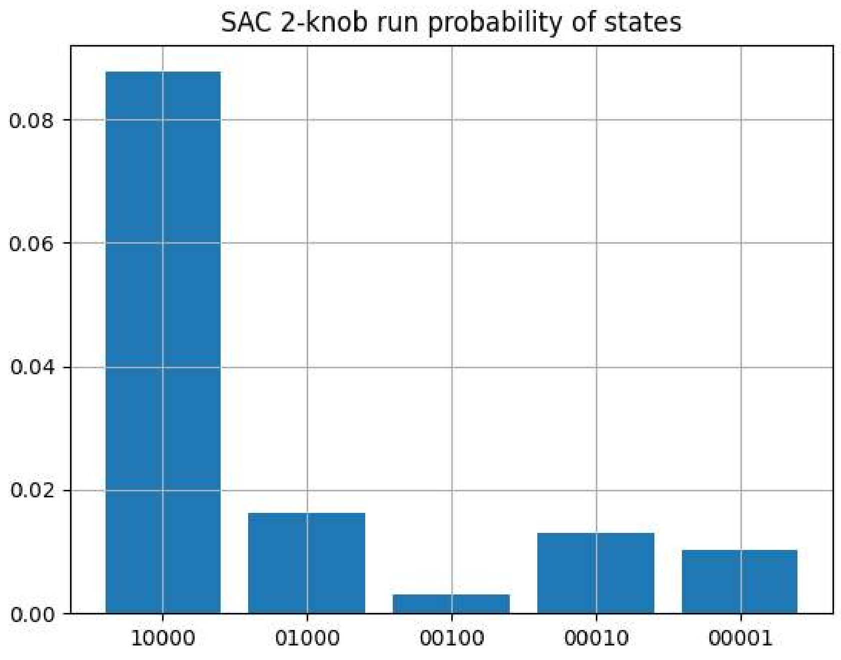


Figure 3.5: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

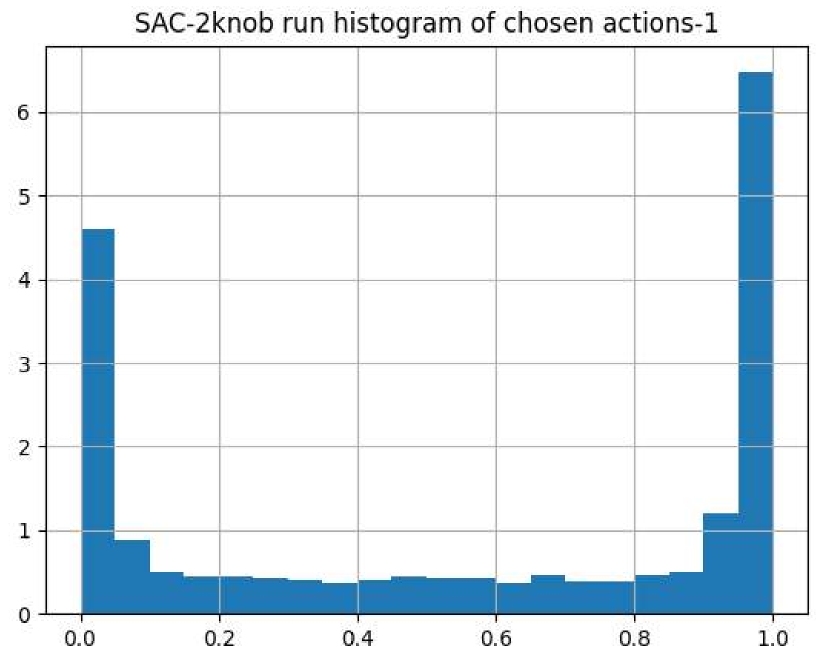


Figure 3.6: Histogram of all chosen actions($Z_1$) by the test generator during the run

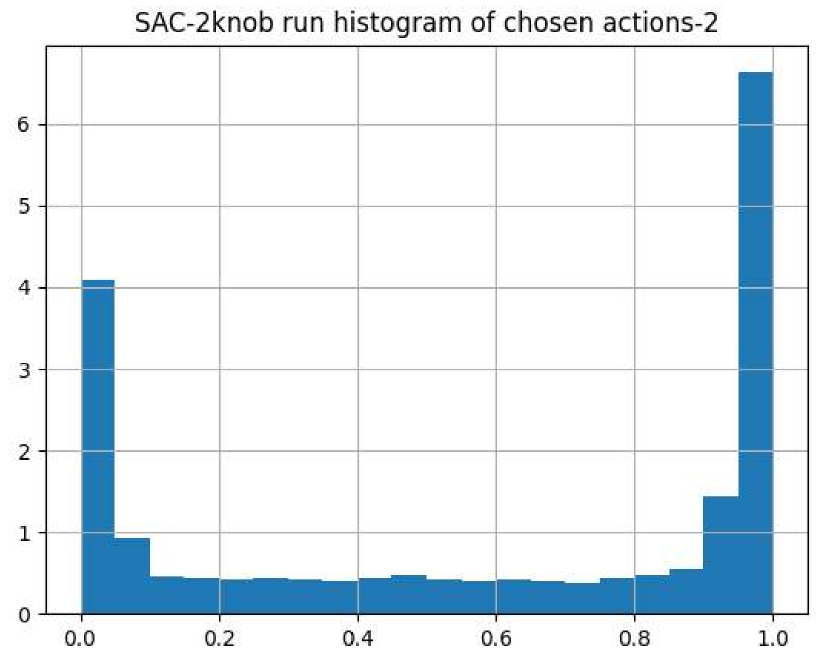


Figure 3.7: Histogram of all chosen actions($Z_2$) by the test generator during the run

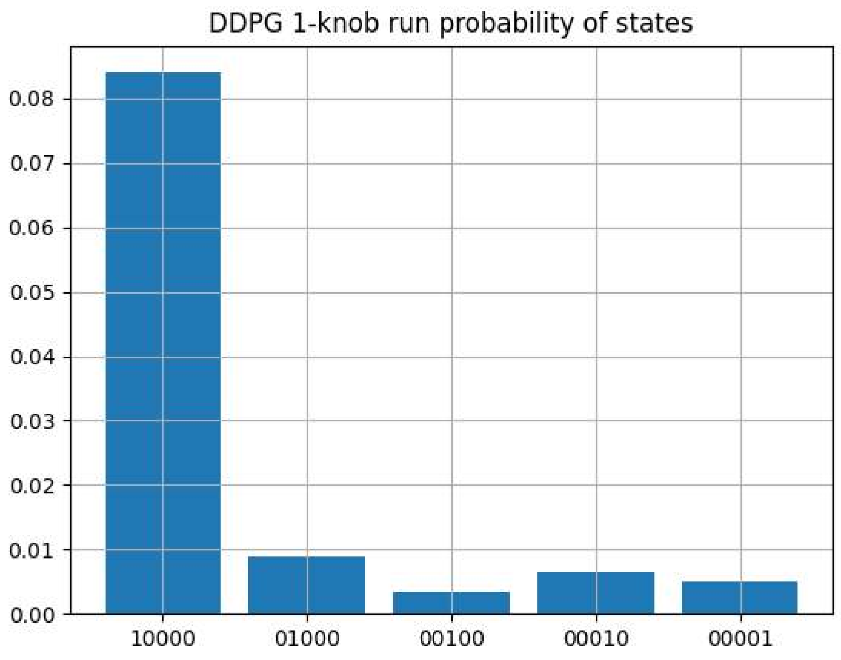


Figure 3.8: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

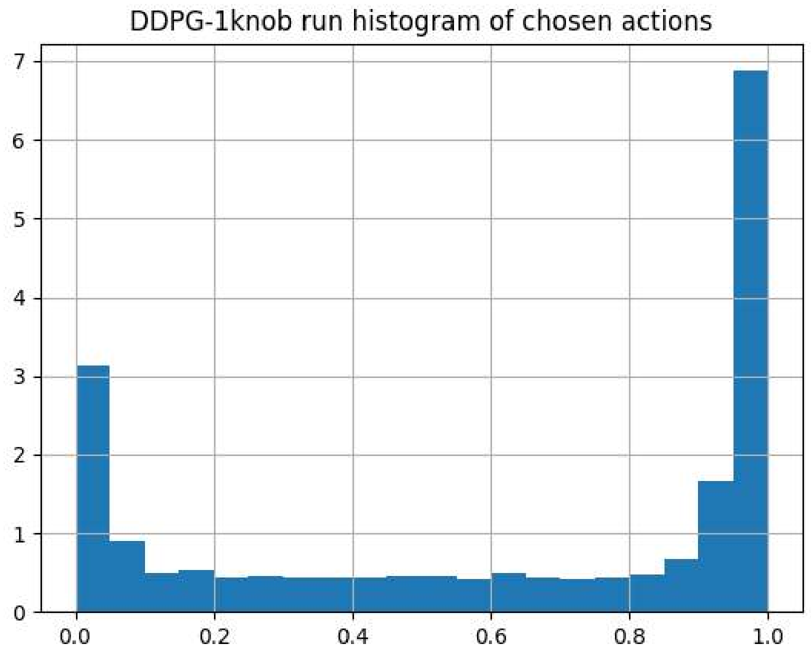


Figure 3.9: Histogram of all chosen actions(Z values) by the test generator during the run

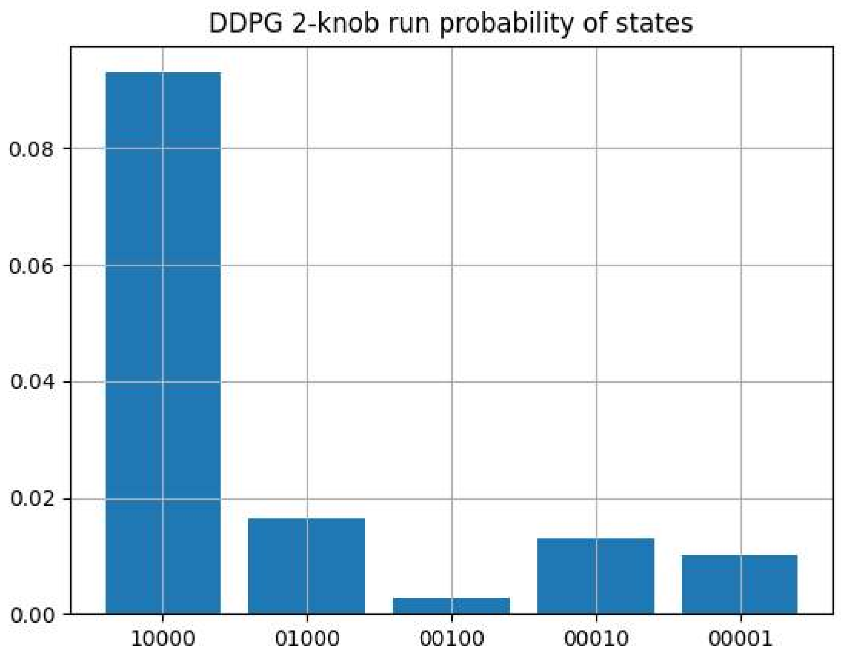


Figure 3.10: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

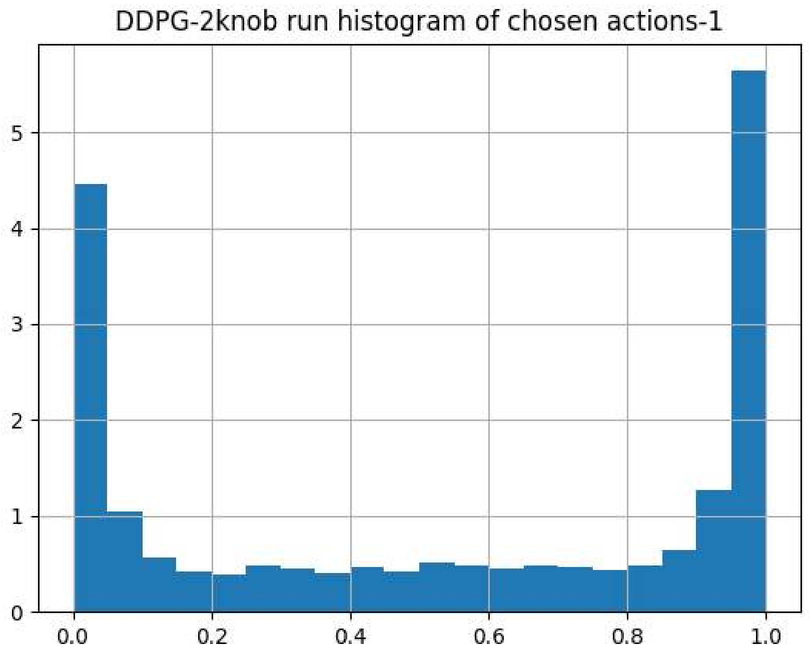


Figure 3.11: Histogram of all chosen actions($Z_1$) by the test generator during the run

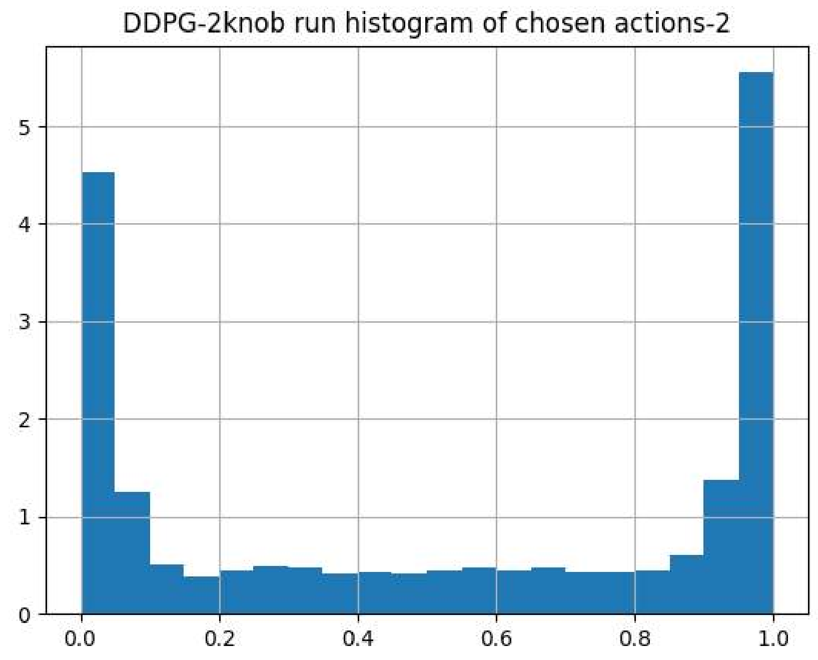


Figure 3.12: Histogram of all chosen actions($Z_2$) by the test generator during the run

### 3.1.3 Occurrence probabilities observed for events in DDPG-1 knob setting

DDPG-1 knob setting showed the highest increase in metric of performance of the Automated-RL run. The occurrence probabilities of each event in the run are given below:

Table 3.2: Occurrence probabilities observed for events in the best setting - DDPG 1-knob

| Event | Probability |
|---|---:|
| Event-0 | 0.084 |
| Event-1 | 0.0085 |
| Event-2 | 0.0034 |
| Event-3 | 0.0064 |
| Event-4 | 0.00496 |

### 3.1.4 Maximum probabilities observed for events

Now, by varying the algorithms(options - SAC, DDPG) used and also by varying the available hyper-parameters like learning rate or LR(options - 0.0001, 0.001, 0.01, 0.1) and train frequency or TF(options - (1,'episode'), (2,'episode'), (4,'episode')) of the RL agent, the maximum probabilities for each event observed was tabulated.

Table 3.3: Maximum probabilities observed

| Event | Maximum Probability | RL setting for maximum probability | Mode of chosen actions |
|---|---:|:---:|---:|
| Event-0 | 0.1479 | SAC,1-knob,LR=0.01,TF=1 | Z=0 |
| Event-1 | 0.0325 | SAC,2-knob,LR=0.01,TF=1 | Z=[0,0] |
| Event-2 | 0.00545 | DDPG,1-knob,LR=0.001,TF=1 | Z=1 |
| Event-3 | 0.0265 | SAC,2-knob,LR=0.01,TF=1 | Z=[0,1] |
| Event-4 | 0.0210 | SAC,1-knob,LR=0.01,TF=1 | Z=[1,1] |

### 3.1.5 Evolution of probabilities of various events over the iterations

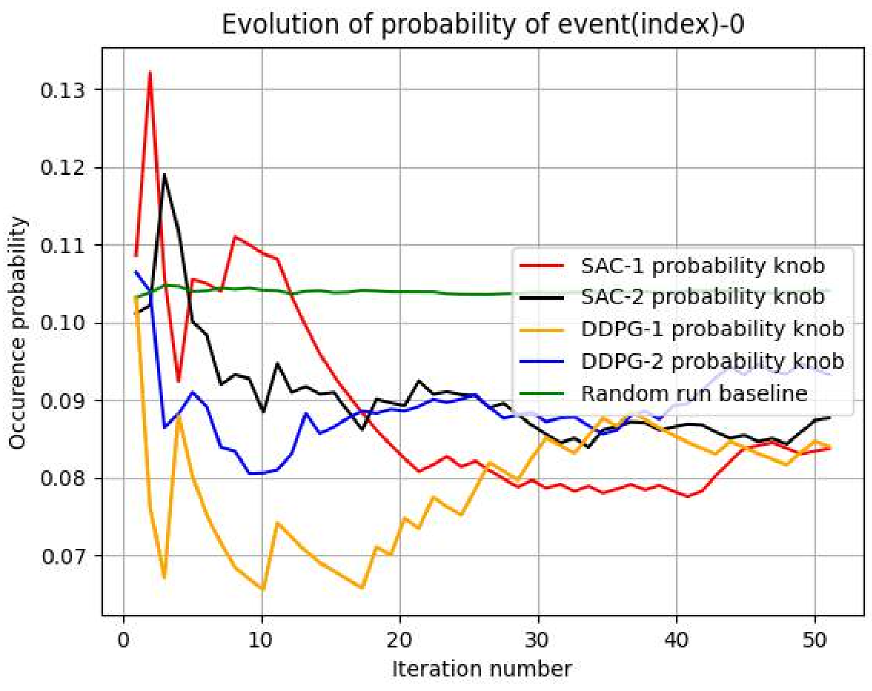


Figure 3.13: Evolution of probability of Event-0 over the iterations

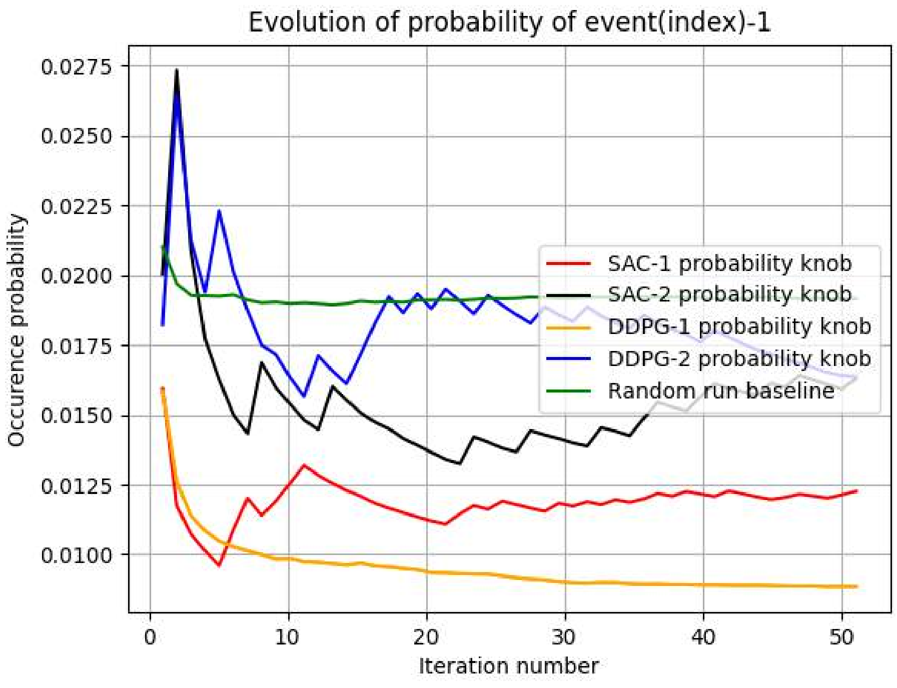


Figure 3.14: Evolution of probability of Event-1 over the iterations

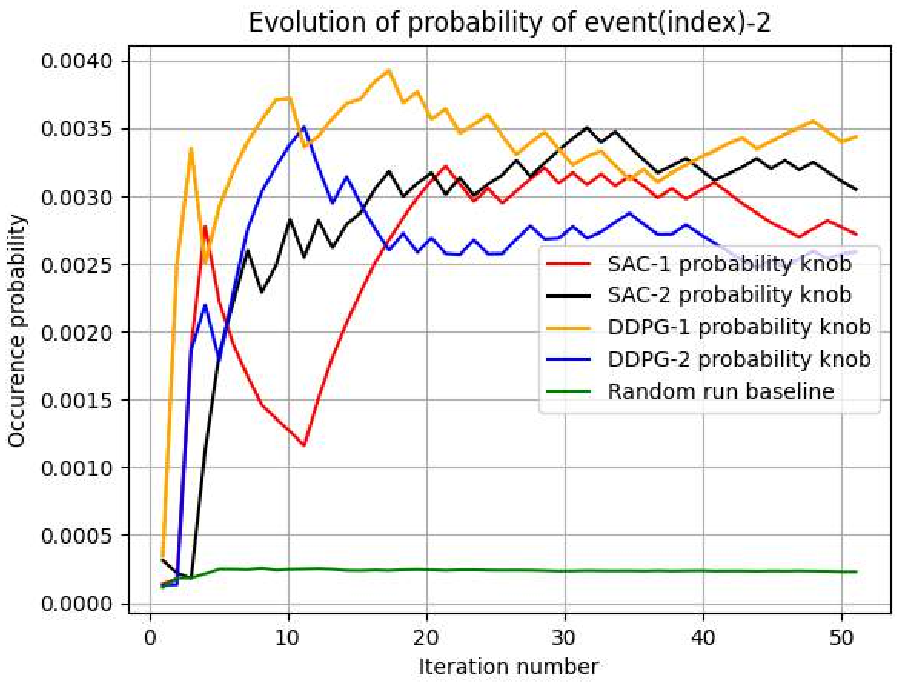


Figure 3.15: Evolution of probability of Event-2 over the iterations

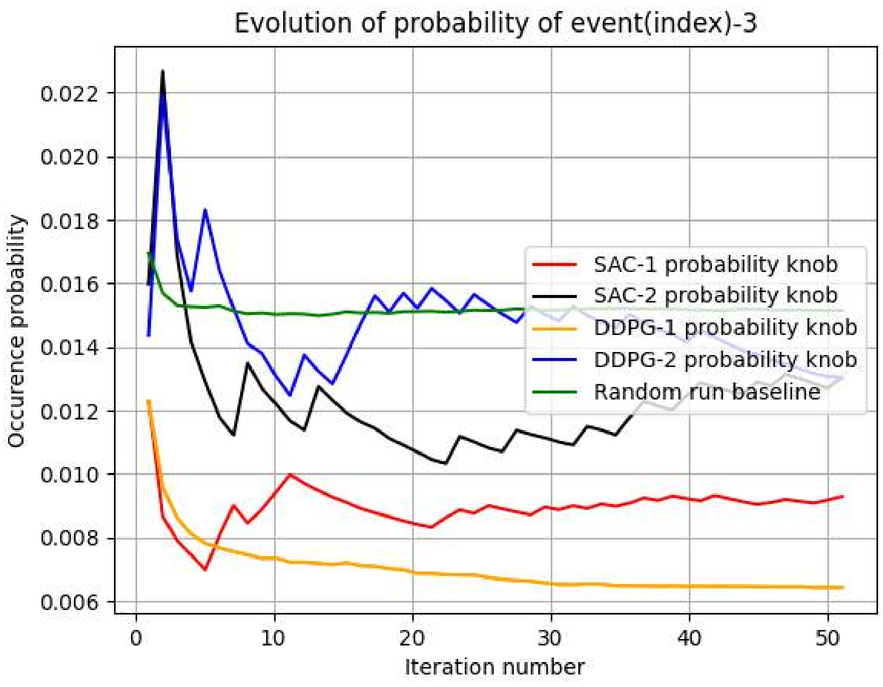


Figure 3.16: Evolution of probability of Event-3 over the iterations

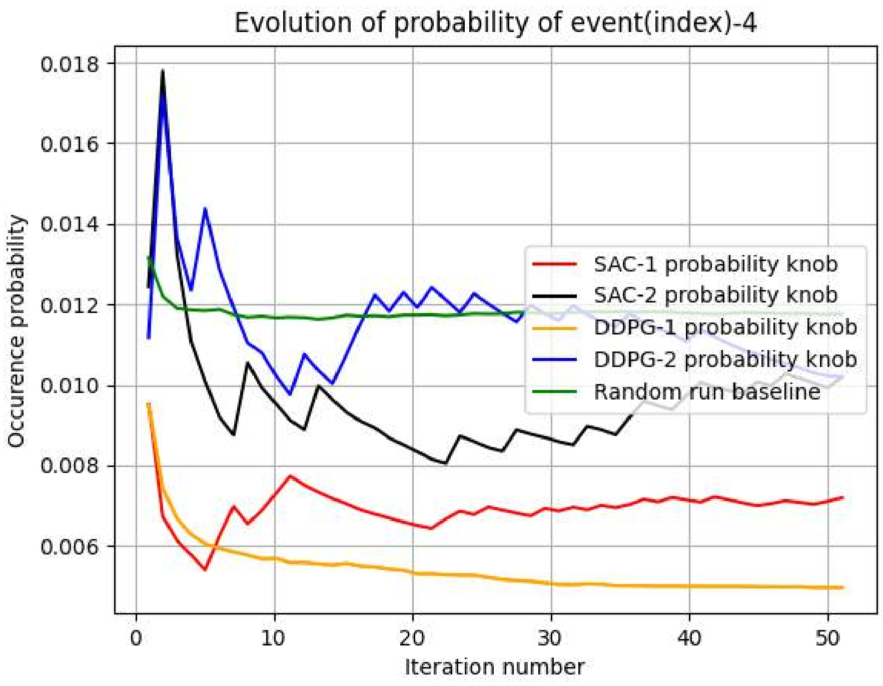


Figure 3.17: Evolution of probability of Event-4 over the iterations

### 3.1.6 Metric Evolution over the iterations

The metric value after each iteration is calculated according to the metric defined for the Automated-RL run given in Section 3.3.1.

The metric corresponding to the pseudo random test generator is calculated and is set as a baseline which is constant across iterations. This curve is named as "Random RL run baseline"

The metric evolution curves corresponding to runs SAC 1-knob, SAC 2-knob, DDPG 1-knob and DDPG 2-knob have been plotted.

The curve named as "Maximum metric" is calculated by taking the probability values of the events from the Maximum probabilities observed table(Table 4.2). These probability values are plugged into the Q array defined in Section 3.3.1 and used in the equation for $metric_{Automated-RL}$. These best probabilities for each each don't occur in one single run, but then these can be used as a good upper bound on the best achievable metric. The curve corresponding to the metric evolution over iterations is given below.

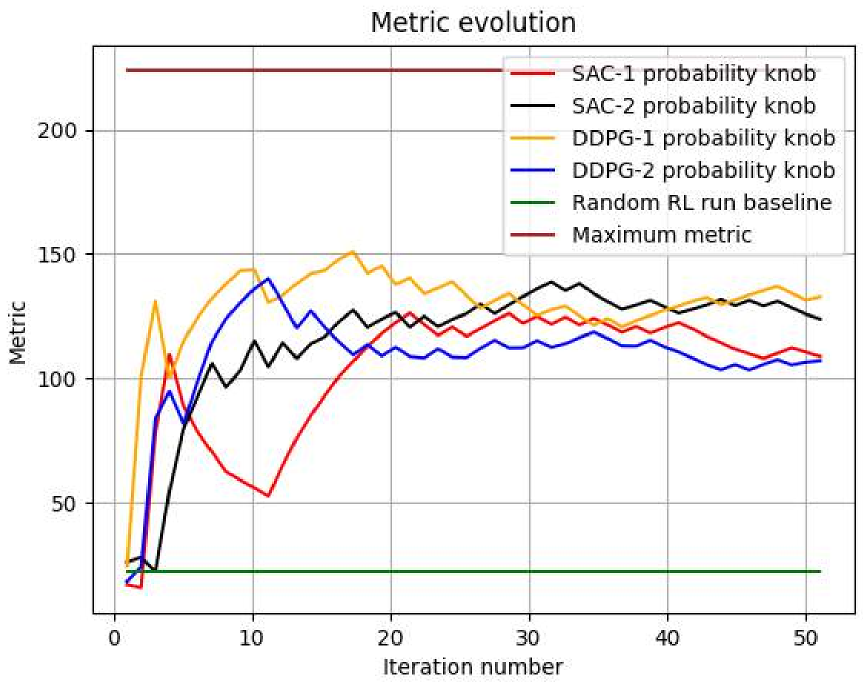


Figure 3.18: Metric Evolution over the iterations for the RLE-Compressor design

### 3.1.7 Comments

- As observed from the probability evolution over iterations curves, we can see that the metric evolution follows the trend of evolution of probability of event-2, the rarest event.
- For event-2, we see from the Maximum probabilities table(Table 3.3) that the mode of chosen actions for run favoring event-2 is Z=1(i.e., its most often predicting the next number in the activation map to be zero).
- From the histogram of chosen actions curves, we see that DDPG-1 knob predicts Z=1 more often that other RL settings and hence this design setting shows the highest increase in the metric of performance over random run.
- Though it maybe puzzling that 1-knob setting does better than the 2-knob setting, it is mainly because of our time constraint that we set the total number of episodes of experience collection of the RL agent to 100 and 2-knob setting would require more iterations to learn the 2-parameters in the setting when compared to the 1-knob setting which needs to learn only 1-parameter in the same time. The number of iterations required grows exponentially with number of knobs.

## 3.2 COO-Compressor

In this design we monitor 3 events for the purpose of our experiment.

- Event-0 is *dut.rg_block_counter.value == 16.*
- Event-1 is *dut.rg_block_length.value % 4 != 0.*
- Event-2 is *dut.rg_next_count != 0.*

The input suggested by the test generator are 2 values - $word\_width$ and $index\_width$. Both these parameters have their valid values in the discrete action space of $\{1, 2..8\}$. So for this design we supply actions in the space $\{1, 2..8\} * \{1, 2..8\}$. Hence, we have 64 sets of valid actions of the form - (1,1), (1,2) .. (8,7), (8,8).

### 3.2.1 Pseudo random verification

When the pseudo random test-case generation is used on the COO-Compressor design for 50 iterations/runs, we notice the following occurrence probabilities of events.

Table 3.4: Occurrence probabilities of events in pseudo random verification process

| Event | Probability |
|---|---:|
| Event-0 | 0.2034 |
| Event-1 | 0.7205 |
| Event-2 | 0.1696 |

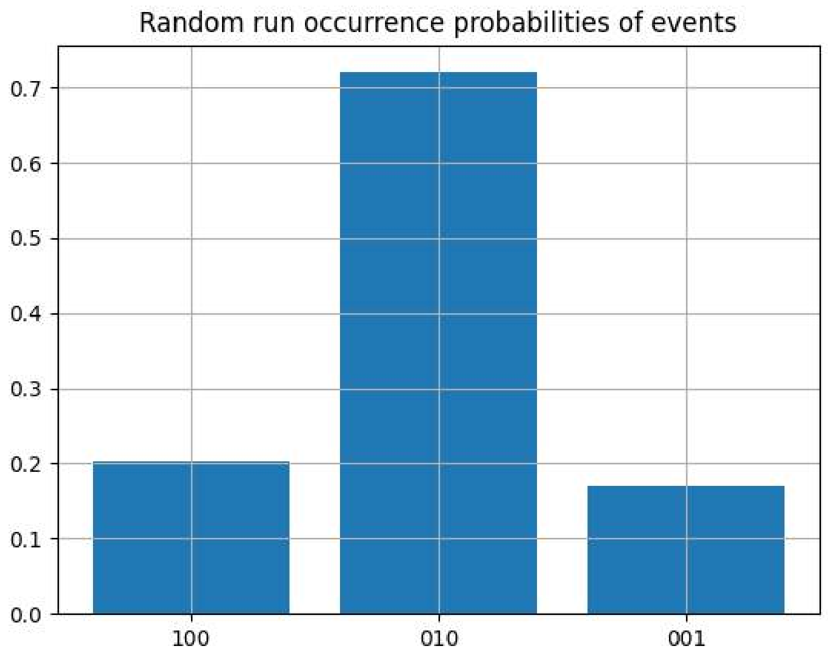


Figure 3.19: Bar graph of occurrence probabilities of events in the random run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

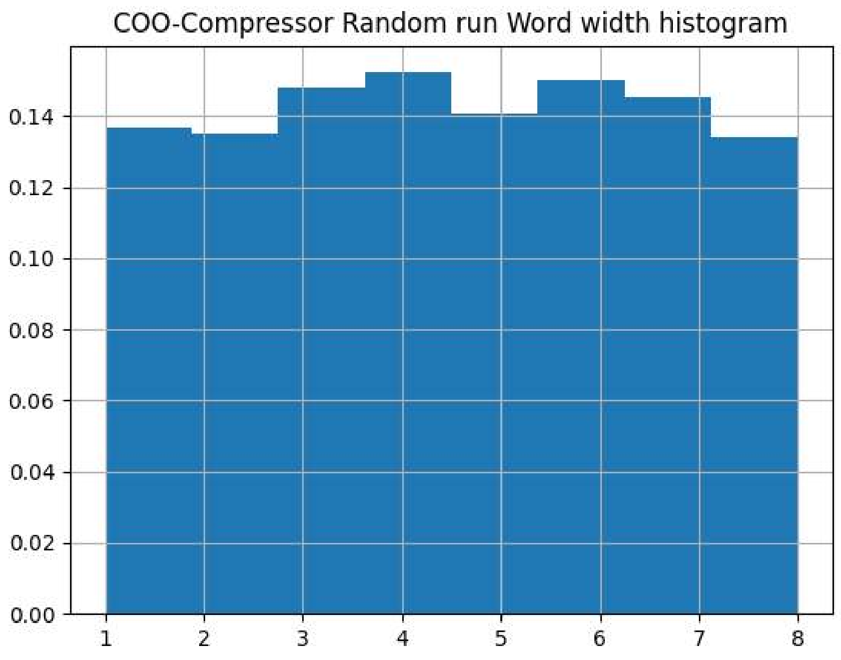


Figure 3.20: Histogram of all chosen actions($word\_width$) by the test generator during the run

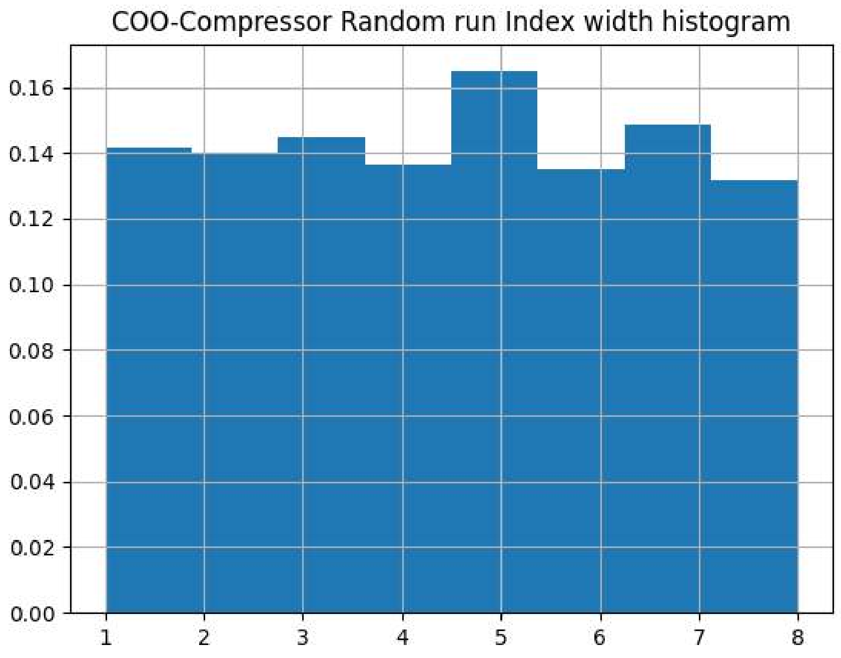


Figure 3.21: Histogram of all chosen actions($index\_width$) by the test generator during the run

Now we use the Automated-RL framework to predict the index_width and word_width.

We use Deep RL methods of SAC, DDPG and DQN for this experiment. We model the action space as discrete set of 64 values. For DQN we directly use the discrete action predicted, and for DDPG and SAC which predicts optimal actions in the continuous space, we map it to one of the 64 discrete actions(by scaling followed by using $math.ceil()$ function in Python). We then map the discrete set of 64 actions(say, a discrete action in this set is A) into a word_width and index_width by:

$$word\_width = A\%8$$

$$index\_width = A/8$$

The 3 events are denoted by 100, 010 and 001 and they respectively correspond to Event-0, Event-1 and Event-2.

### 3.2.2 Results obtained

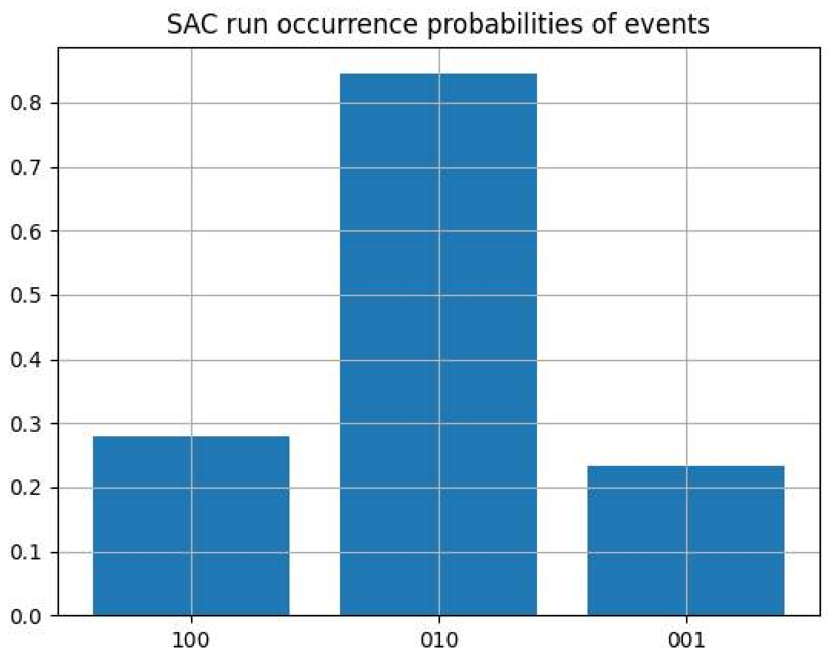


Figure 3.22: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

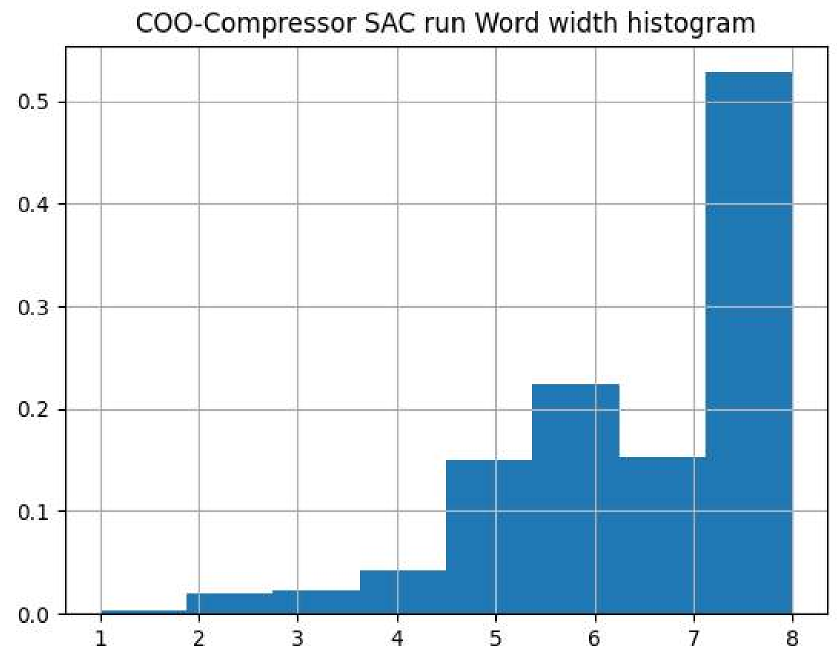


Figure 3.23: Histogram of all chosen actions($word\_width$) by the test generator during the run

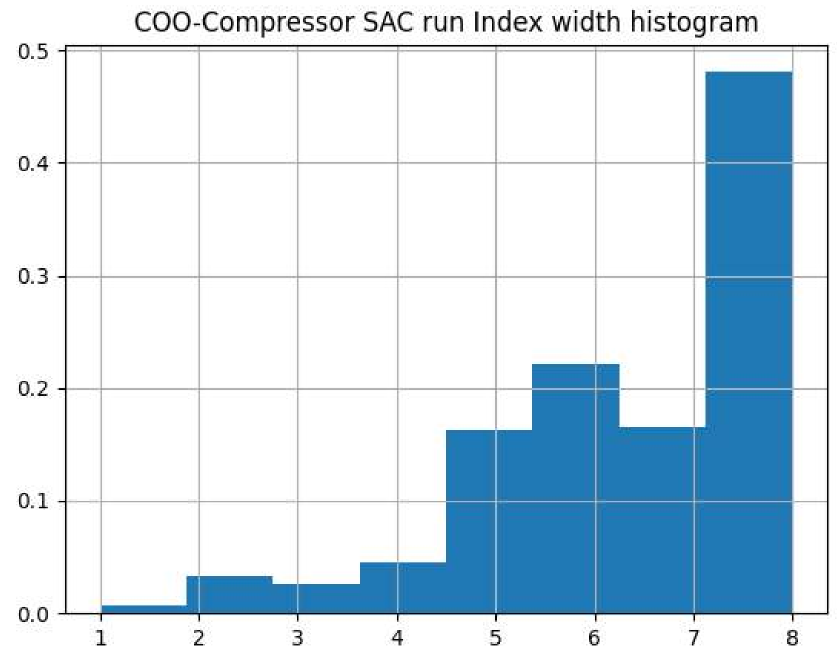


Figure 3.24: Histogram of all chosen actions($index\_width$) by the test generator during the run

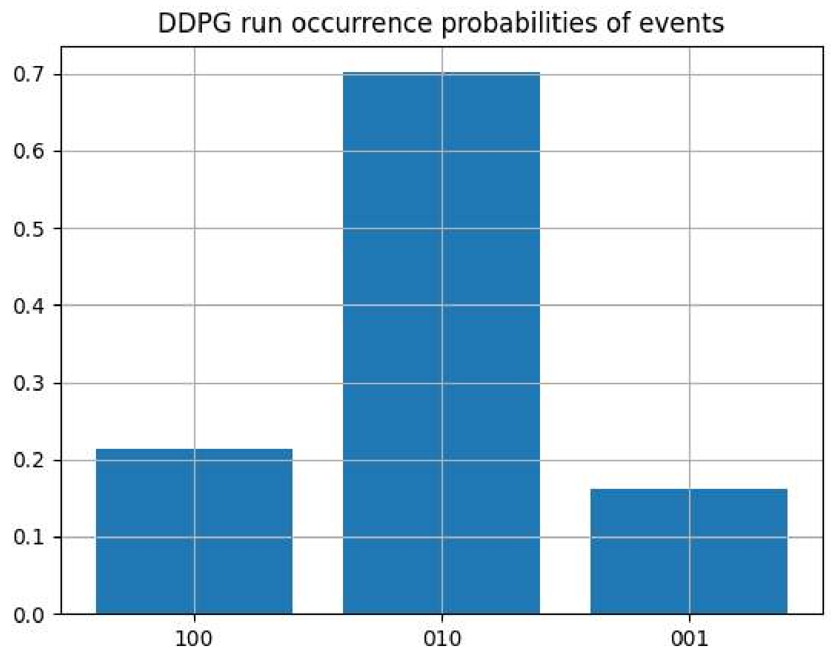


Figure 3.25: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

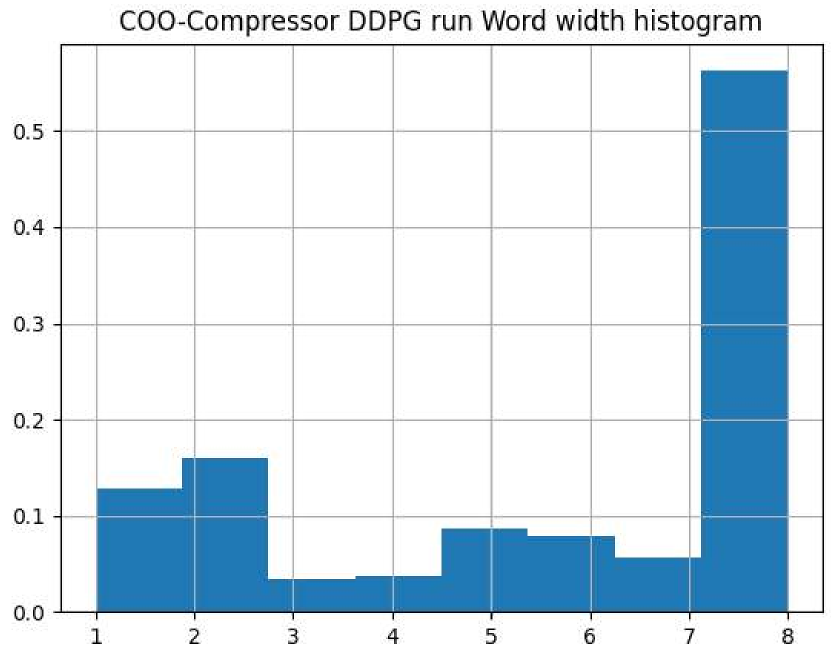


Figure 3.26: Histogram of all chosen actions($word\_width$) by the test generator during the run

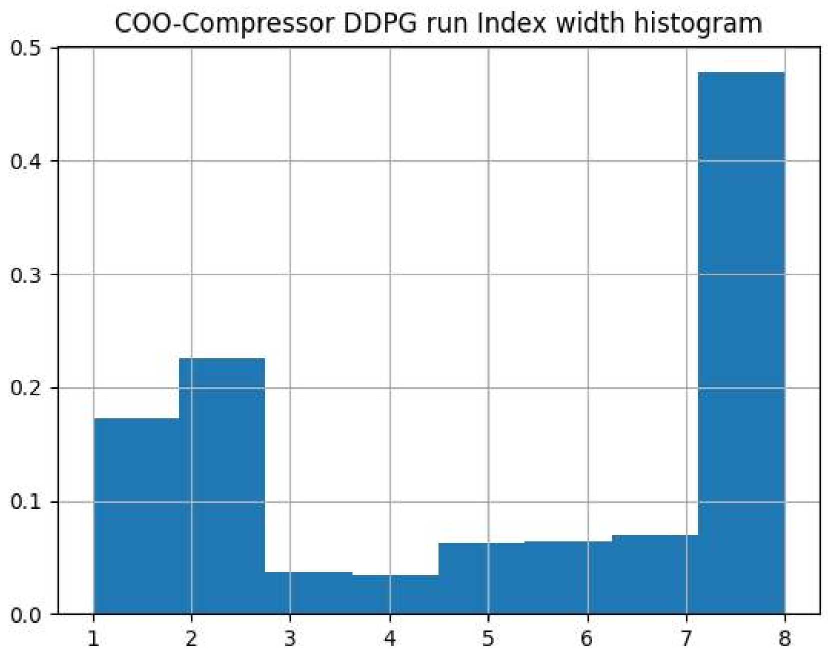


Figure 3.27: Histogram of all chosen actions($index\_width$) by the test generator during the run

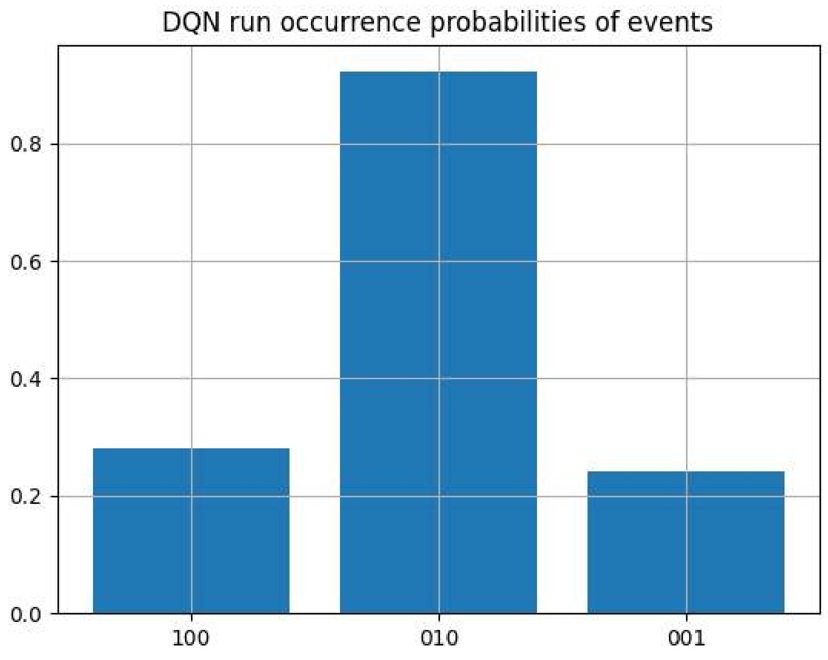


Figure 3.28: Bar graph of occurrence probabilities of events in the run, X-axis denotes the various events monitored, Y-axis denotes the probability of occurrence of those events

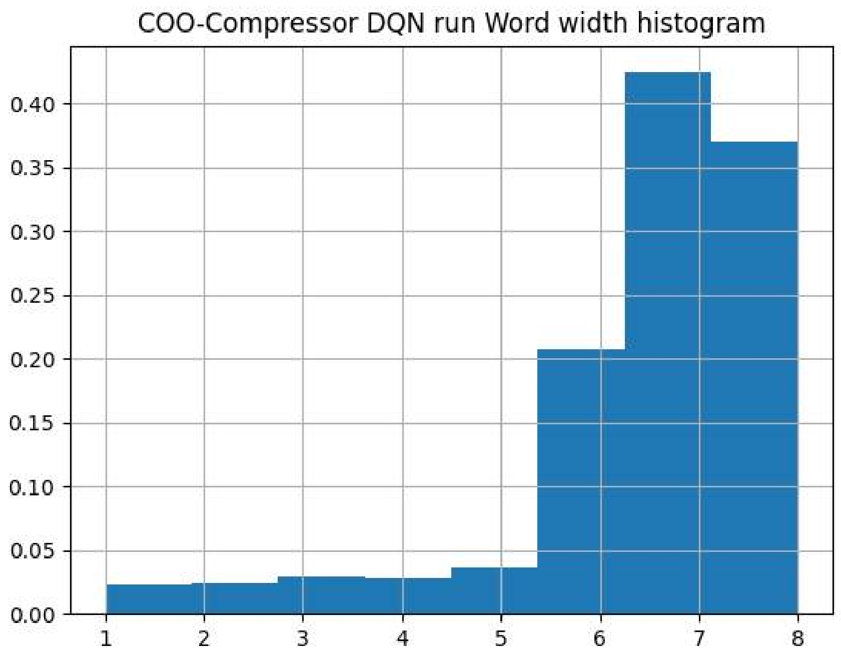


Figure 3.29: Histogram of all chosen actions($word\_width$) by the test generator during the run

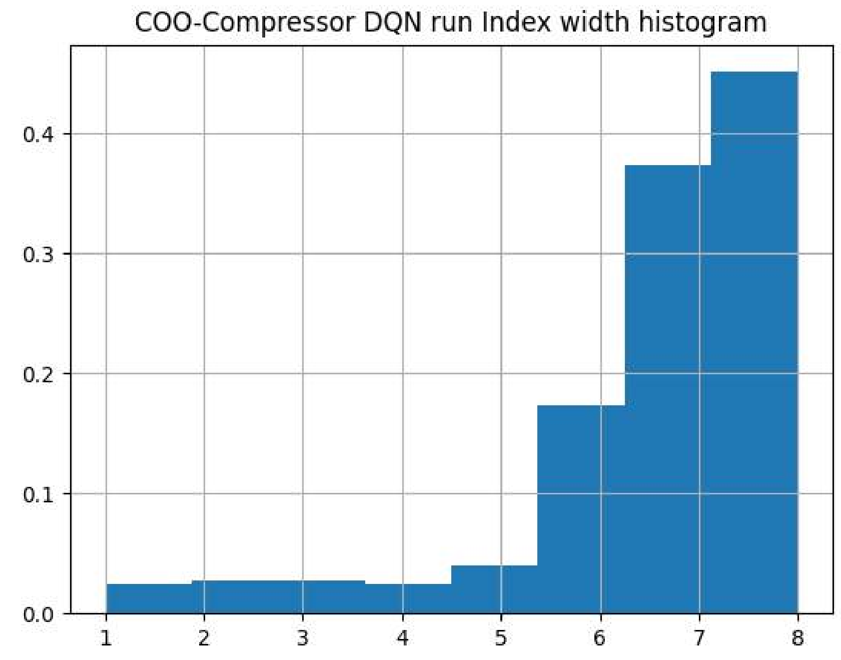


Figure 3.30: Histogram of all chosen actions($index\_width$) by the test generator during the run

### 3.2.3 Occurrence probabilities observed for events in DQN setting

The DQN setting showed the highest increase in metric of performance of the Automated-RL run. The occurrence probabilities of each event in the run are given below:

Table 3.5: Occurrence probabilities observed for events in the best setting - DQN

| Event | Probability |
|---|---:|
| Event-0 | 0.2804 |
| Event-1 | 0.9215 |
| Event-2 | 0.2425 |

### 3.2.4 Maximum probabilities observed for events

Now, by varying the algorithms(options - SAC, DDPG, DQN) used and also by varying the available hyper-parameters like learning rate or LR(options - 0.0001, 0.001, 0.01, 0.1) and train frequency or TF(options - (1,'episode'), (2,'episode'), (4,'episode')) of the RL agent, the maximum probabilities for each event observed was tabulated.

Table 3.6: Maximum probabilities observed

| Event | Maximum Probability | RL setting for maximum probability | Mode of chosen actions |
|---|---|---|---|
| Event-0 | 0.2853 | SAC, LR=0.01, TF=1 | $word\_width = 8, index\_width = 8$ |
| Event-1 | 0.9346 | DQN, LR=0.01, TF=1 | $word\_width = any, index\_width = any$ |
| Event-2 | 0.2693 | DQN, LR=0.01, TF=1 | $word\_width = 7, 8, index\_width = 7, 8$ |

### 3.2.5 Evolution of probabilities of various events over the iterations

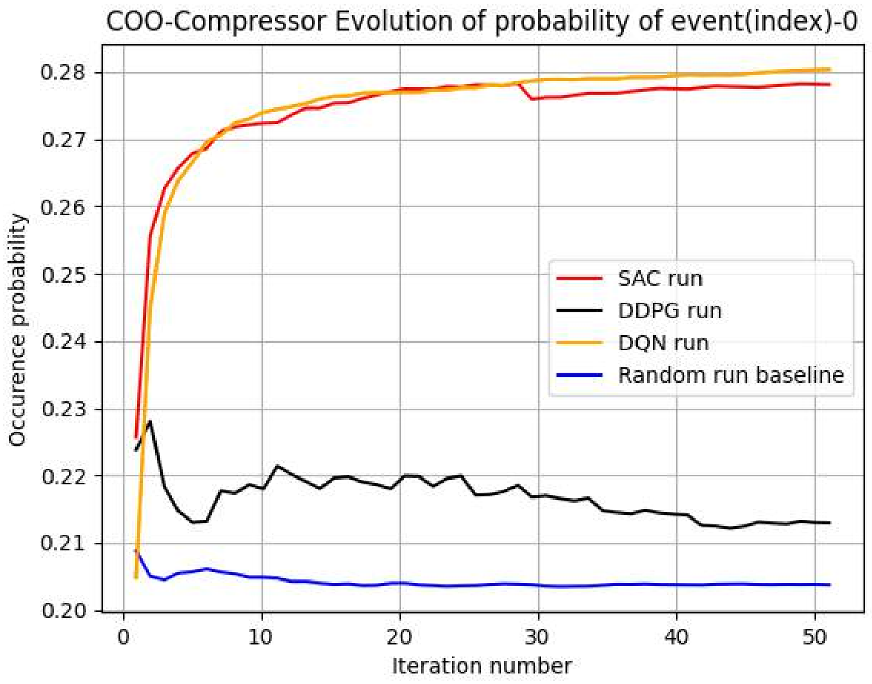


Figure 3.31: Evolution of probability of Event-0 over the iterations

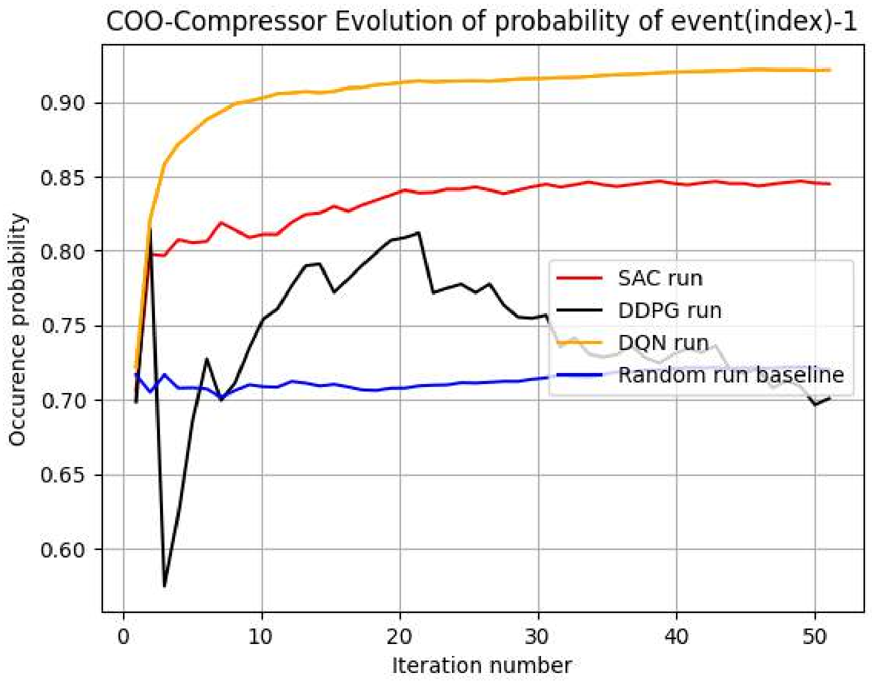


Figure 3.32: Evolution of probability of Event-1 over the iterations

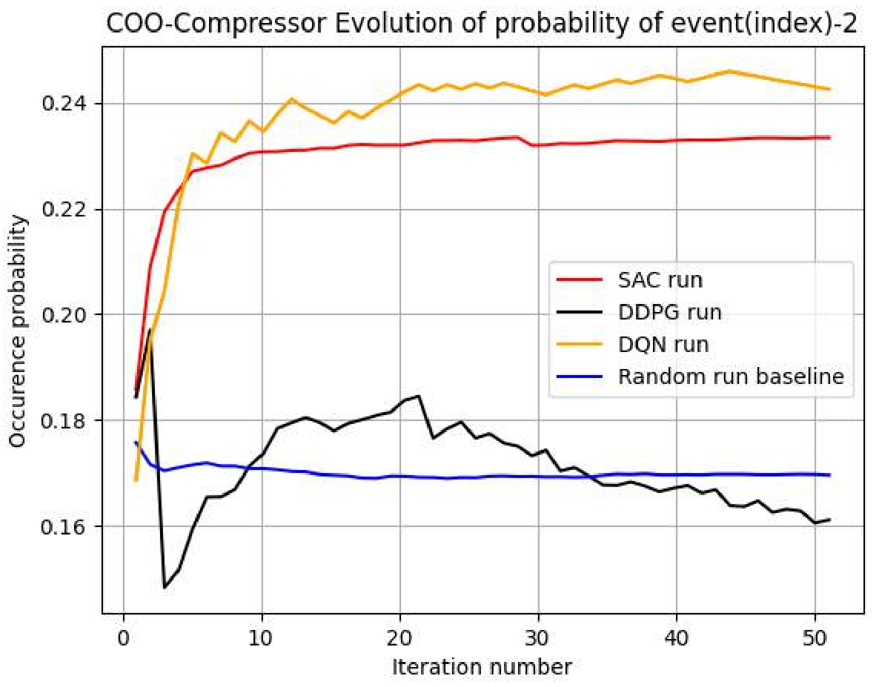


Figure 3.33: Evolution of probability of Event-2 over the iterations

### 3.2.6 Metric Evolution over the iterations

The metric value after each iteration is calculated according to the metric defined for the Automated-RL run given in Section 3.3.1.

The metric corresponding to the pseudo random test generator is calculated and is set as a baseline which is constant across iterations. This curve is named as "Random run baseline"

The metric evolution curves corresponding to runs of SAC, DDPG and DQN are plotted.

The curve named as "Maximum metric" is calculated by taking the probability values of the events from the Maximum probabilities observed table(Table 3.6). These probability values are plugged into the Q array defined in Section 3.3.1 and used in the equation for $metric_{Automated-RL}$. These best probabilities for each each don't occur in one single run, but then these can be used as a good upper bound on the best achievable metric. The curve corresponding to the metric evolution over iterations is given below.

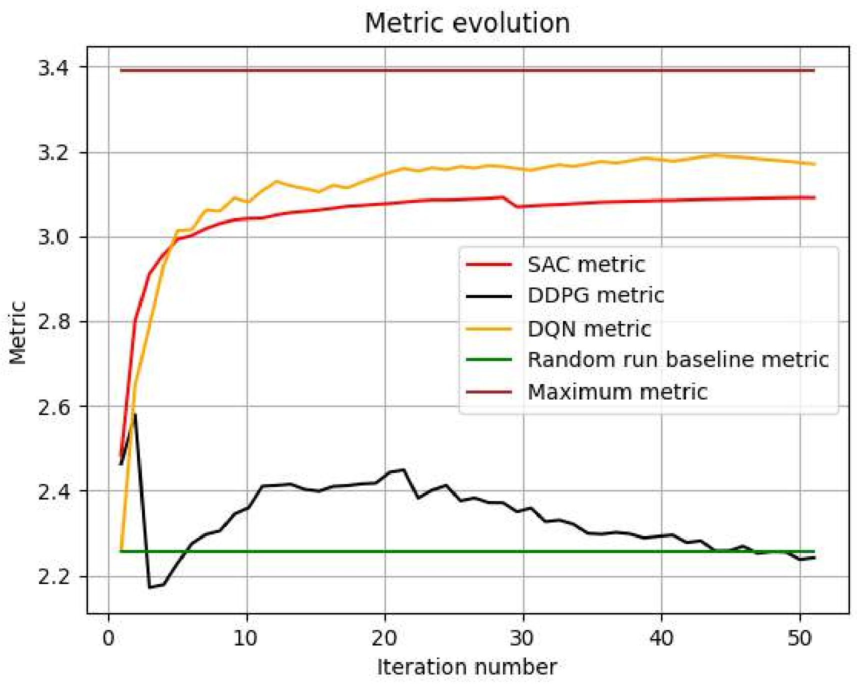


Figure 3.34: Metric Evolution over the iterations for the COO-Compressor design

### 3.2.7 Comments

- It can be observed from the metric evolution plot that the metric values for the COO-Compressor design are much lower than that of the RLE-Compressor. This is because the event occurrence probabilities in the random test case generation case of the COO-Compressor are much higher than their counterparts in the RLE-Compressor example.
- As observed from the probability evolution over iterations curves, we can see that the metric evolution follows the trend of evolution of probability of event-2, the rarest event.
- For event-2, we see from the Maximum probabilities table that the mode of chosen actions for run favoring event-2 is $word\_width = 7, 8, index\_width = 7, 8$.
- From the histogram of chosen actions curves, we see that the DQN-automated run predicts the optimal actions in the most similar fashion(to the above discussed mode of chosen actions) than other RL settings and hence this design setting shows the highest increase in the metric of performance over random run.
- Since the design configuration parameters($word\_width, index\_width$) are in the discrete space and since DQN's action space is also discrete, DQN was able to show highest increase in metric of performance.

# CHAPTER 4

# FUTURE WORK

- Multi-state RL based experiments - meaningful choice of what constitutes as the Markov states of the RL environment
- Curiosity and Hindsight Experience Replay as methods to deal with sparse rewards - coverage holes
- Dealing with large discrete action spaces
- Model based RL and need for good priors

# CHAPTER 5

# SUMMARY

- The automation algorithm with the desired objectives was successfully implemented in code.
- The algorithm was tested out on two designs-RLE-Compressor and COO-Compressor, and the results were presented.
- A metric to quantify the performance of the automation algorithm over the pseudo random test case generator was proposed and was used for evaluating the results of the two designs.
- The Automated-RL setting which did best in terms of the metric of performance for each design was identified and meaningful insights on the results were presented.

# REFERENCES

[1] **Böttinger, K.**, **P. Godefroid**, and **R. Singh** (2018). Deep reinforcement fuzzing.

[2] **Hughes, W.**, **S. Srinivasan**, **R. Suvarna**, and **M. Kulkarni** (2019). Optimizing design verification using machine learning: Doing better than random.

[3] **Ioannides, C.** and **K. I. Eder** (2012). Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, **17**. ISSN 1084-4309. URL `https://doi.org/10.1145/2071356.2071363`.

[4] **Tsung-Yi Lin, R. G. K. H. P. D., Priya Goyal** (2017). Focal loss for dense object detection.