Capstone Report:
Machine Learning for Functional Verification

Alberto Guasco

May 4, 2019

1 Definition

In this report is described the project realized as conclusion of Machine Learning Nanodegree from Udacity. The project aims to use reinforcement learning for improving functional verification.

1.1 Project Overview

The project consists in using reinforcement learning with actor-critic for improving the stress of a generic hardware. The project is under revision control using git at this remote (git version 2.7.4) and has been developed using Python 3.5.2. This can be considered as the final work for showing what I learnt during the nanodegree. In my opinion the most important topic was reinforcement learning with actor-critic. This is a really recent method and it has been really interesting to see that all the references found were under intensive research work. Moreover, reinforcement learning with acotr-critic is needed for our goal beacuse of the high number of states. The environment used for developing our algorithm has been develpoed during the project (described in next section). The main goal is to test this cutting-edge technology with a generic and parametrizable environment. The project has been developed mainly with what has been done at Udacity, and using internet for various needs and suggestions (in the repository there's a file called `useful_links.txt`).

1.2 Problem Statement

The problem we want to face is the functional verification of a random piece of hardware. Functional verification aims to ensure that a given RTL respects the specifications. The most difficult part of this work is to stress all possible states (and sequences of states) efficently, in order to save time and being sure that we are verifying mainly the most interesting states. What we normally want to ensure is that during verification RTL is continuosly stressed: this increase the probability to reach the bug condition. This is a very important topic because RTL is going to be more and more complex and the functional verification is the most time dispending phase of development. Reinforcement learning may help us in generating more intelligent stimuli. At each step (which corresponds to a clock cycle) we look at the state (which corresponds to the RTL register value) and then we decide the action (stimuli) to provide. What I would like to have is a generator which avoid invalid actions and is able to cover all possible states.

1.3 Metrics

In the domain of functional verification it's difficult to state that the design under testing (DUT) is fully verified. The level of verification is normally expressed using two indicators: how many tests have been run and the percentage of coverage reached. Coverage is a method in which a series

of conditions (states values) is defined; the software used for testing collects how many times the DUT reached each of the conditions previously listed. The DUT model used for this project is done for returing also the coverage value at each step. In order to have a generic DUT, the coverage is defined as the list of all possible states and all possible transitions between states (when connected). This approach ensure us to see if the actions are stimulating all the possible functions of the DUT. This metric is not used as reward during the reinforcement learning algorithm. The reward has been defined for ensuring an high coverage at the end of the test (high reward for high coverage) and for avoiding actions that doesn't change the state (negative reward for this actions, which are different for each state). The reward is used as feedback during reinforcement learning algorithm, while coverage is an index for understanding if at the end of a test (an episode) the DUT has been well explored: the goal is to have high exploration avoiding unallowed actions.

2 Analysis

2.1 Data Exploration

Before starting the project, a python class for modeling the DUT has been developed. The class in file `Dut.py` aims to model a generic piece of hardware, assuming that any sequential logic can be modeled as a Moore's machine, so as a graph. Each node of the graph is a value of the Moore's state register. If it exists an input value for moving from a state to another one, the model has added the corresponding directed edge, having weight equal to the input value. With a little effort we are able to have a dataset for running our reinforcement learning algorithm. This allows us to have a DUT as big as needed, for sure without any functional bug, and can be used just for our neeeds:

- efficient: simulate a DUT and collect the coverage are particularly slow operations: with this approach we concentrate our computational effort on reinforcement learning only, without using any slow and apying software;
- configurable: the Dut is generated with only two parameters, N_STATES (total number of states) and N_INPUTS (action bound is between 0 and N_INPUTS - 1);
- random: all connections are randomly generated;
- flexible: it is use for generating a dataset and as environment capable of returning us the next state, rewards and coverage.

As previously described, a problem of functional verification is to always provide an input capable of stress the DUT. We have easly modeled this issue with unconnected states: in case of an action doesn't change the state.

The project is done using three configurations:

	N_STATES	N_INPUTS
config_1	32	8
config_2	256	32
config_3	1024	128

Notice that the total number of coverage items is random.

Algorithm 1 shows how the DUT is built. Notice that the class Dut is able also to return the next_state given an action, and the coverage at each step. The main two dictionaries are DUT and

COMB. For a given state `s`, `DUT[s]` returns the list of states connected to `s`. If `s_0` and `s_1` are connected, `COMB[s_0, s_1]` returns the input needed for moving from state `s_0` to `s_1`.

Algorithm 1: DUT model

Input : N_STATES, N_INPUTS
Output: DUT, COMB

```
for all states s do
    ensure that s is reachable from almost one state
    define a random list l of connected states
    DUT[s] = l for each next state ns from list l do
        C
    end
    OMB[s, ns] = random(0, N_INPUTS)
end
```

2.2 Exploratory Visualization

We don't need a particular visualization of the Dut: the resulting graph may be particularly intricated (it is random) and in config_2 and config_3 it's really big. Moreover one of the initial hypothesis is that we don't need to know the DUT: we would like to build a reinforcement learning generator able to stress any DUT.

2.3 Algorithms and Techniques

The technique used for our goal is Reinforcement Learning. This approach has been chosen because of two reasons: it's the most interesting topic of the course, and it's the approach that most suits the issue, as we require an intelligent agent capable of generating a specific action for a given state.

Algorithm 2: Reinforcement Learning basic algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$;
foreach *episode* **do**
    Initialize S;
    foreach *step of episode* **do**
        Choose A from S using policy derived from Q Take action A, observe R, S';
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$;
        $S \leftarrow S'$;
    end
end

Algorithm 2 shows a version of Reinforcement Learning: a key point is how to choose action A. The method chosen for our project is the https://arxiv.org/pdf/1509.02971v2.pdf. With this approach we approximate both Q value function and the policy function with two neural networks. The training is done at each time step using a data saved in a dedicated buffer. The algorithm is particularly adapted for high dimensional environments. For the capstone project we have chosen this method because of it's really new and because DUT may have a very high number of states. An important point to underline is how reinforcement learning interacts with DUT:

- action: input value used for stimulating the DUT;
- state: value of the DUT sequential logic at a given moment;
- reward: indicator on the coverage;

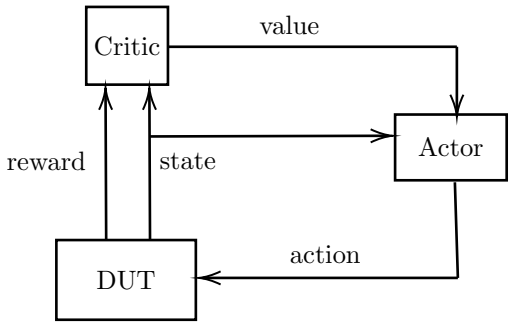


Figure 1: Schematic of actor-critic approach

- step: unit of time (defined by the clock) in which DUT sample the input and compute the next state depending on the actual state.

2.4 Benchmark

Reinforcement learning results will be compared with the coverage obtained using a random generator. This kind of generator is commonly used as stimuli generator for functional verification: this in an heuristic method for searching the corner cases. This kind of generator have limited knowledge of the DUT and, most important, are not able to forecast the next state. It's then difficult to ensure that the DUT is sufficiently and uniformly explored.

	coverage
config_1	0.7377
config_2	0.0667
config_3	0.0033

From the previous table we can see an important point: with bigger DUTs it's more difficult to have a good coverage, specially with completely random generation.

3 Methodology

3.1 Data Preprocessing

As previously explained, we don't have to do any data preprocessing. The DUT will be created at the beginning of the test and it uses a fixed seed for ensuring that the DUT is always the same at each run. We don't look at the DUT before the test and we don't have any need for searching or characterizing the dataset. The only property that we ensure at the creation of te DUT is that all states and all transitions are reachable.

3.2 Implementation

As suggested during the review of Quadcopter project, the starting point of the project is from https://github.com/pemami4911/deep-rl.git repository, where there the actor-critic algorithm has been implemented. Actor network has two hidden layers with 400 and 300 nodes each; critic network has two hidden layers for state input (400 and 300 nodes each), and one hidden layer for action input (300 nodes); the output (Q value approximation) is a combination of the two resulting

layers. All activation functions of hidden layers are `relu`. The main functionalities are coded using `tflearn` API.

An important point consists in listing and managing main parameters used along the algorithm:

- neural network parameters:
  - actor learning rate: learning rate of actor neural network;
  - critic learning rate: learning rate of actor neural network;
  - gamma: discount factor for critic updates;
  - tau: target weights update parameter;
- replay buffer parameters:
  - buffer-size;
  - minibatch-size;
- general:
  - max-episodes: number of episodes for learning;
  - max-episode-len: number of steps in each episode.

After having familiarized with the code, it was clear that we had to do big changes. First of all we have replaced the environment with the DUT: it has been easy becuase the DUT has been coded for being both a model and an environment.

The first thing we had to think about, was the action. DDPG algorithm is structured in order to return a continuous value between two bounds. For our propose we need another format: for each discrete action value we need a value between 0 and 1, representing the probability that this action for a given state, may return a positive reward. At the end of learning the actor network, for a given state, will return a vector of N_INPUTS elements, in which for each action there will be probability to have positive reward. This is needed because for a given state we may have multiple valid action to take, and we have to guarantee that our algorithm will take all valid actions for a given state.

A paper I used as reference has been https://arxiv.org/pdf/1512.07679.pdf, were the policy had to return only one discrete action. The actor network return one real value, called `proto-action`, then we choose k discrete values next to it, and with `KNN` algorithm they chose the discrete action with higher Q. This has been an important reference for me because I also need to transform the proto-action in something slightly different.

Another important topic is to manage the noise (which ensure us the randomness, so a good action exploration) with this specific action format. We can't use any discrete noise directly on the action to take, but we will add some noise on the probability of not-taken actions.

Similarly, state value has to be taken into account. Actor-critic consider the state as a continuous value, and the critic Q value approximation is a function of both states and actions. Compared to our application, we have to really important differences:

- states are discretes: this will not require any correction to the algorithm;
- Q value functions of two close states can be completely independent between them: this becase the connection with other states are (pseudo) random.

This is not the only reason why it has been difficult to model the Q value function. Differently from other reinforcement learning problem, in our project we don't have any fixed final state, or a clear optimal solution to follow, like a path in a maze. The optimal solution is the one which avoid unallowed actions, and that explore all possible states and connections, without doing too many times the same paths (quite similar to the traveling salesman problem).

3.3 Refinement

Here there is the description of the main steps taken for having the final solution. The results refer to config_1.

First step: make it works

The first thing to do is to replace the environment with the DUT, and set the neural network input dimensions properly. Then we have to define how the reward is assigned: I assumed that the first approach to take is to be capable to distinguish between allowed and not-allowed actions for each state. Then I assigned $reward = 1$ in case the action returns a new $next\_state$, and $reward = -1$ in case we don't change $state$ after a step. Other kind of rewards have been tried, but this symmeric approach was the one that worked better. The second thing to do, as previously said, is to manage the action and its format. That's why we changed the last layer of the actor network: instead of returning a continuous value, we have put a N_INPUTS softmax output. Our expectation was to train the network for returning the probability for each possible action, and then choose the action as the following:

$$action = np.random.choice(N\_INPUTS, p = softmax\_output)$$

The results was surprising: with only 6 episodes the algorithm reaches the maximum possible reward, but, unfortunatly, with a really low coverage. What I discovered is that softmax is not good for setting multiple most-likely action, but it tends to promote only one element. That's why we reach high reward: it recognizes one action to take for each state, and then it always takes the same. Another important point is that we obtain the same results also adding some noise on softmax.

Second step: change activation function

From previous step I assuemd that the approach taken was good, but we had to change (at least) the activation function. It implies that we can't use the network's output directly in the *choice* function, but a normalization is needed, in order to have a vector that sum to one. The activation function has been chosen looking at the rewards at the end of each episode, and at the actions predicted for a given state, compared to the connections described in `COMB` and `DUT` dictionaries.

- relu: it predicted quite well one or more actions to take, but Q value exploded too fast;
- relu6: it limits the Q value increase, but predictions saturated too fast without changing anymore;
- tanh: worked quite well, but negative values added too many noise on predictions;
- sigmoid: the one that worked better.

From udacity course we learned that the problem of sigmoid is that its derivative is too little when it tends to one or zero, resulting in a difficult learning. In order to avoid this problem, all activation function of actor hidden layers have been replaced using sigmoid.

The result was comparable to previous step, but quite randomly we had some degradation in terms of rewards. Looking at the Q values for each episode, we saw that it didn't increase: it was decreasing or randomic. It suggested us that there was some randomness in the algorithm to manage.

Third step: heuristics for action selection and promotion

We I had to improve is the action selection, which is described in algorithm 3. First of all we predict the action vector from the actor network; this is a vector of N_INPUTS elements, where each element is between zero and one. The action to choose (for performing the step) is randomly chosen using as probability weights a normalized form of *probs*. Until now we can say that we have chosen the action returned from the actor network, but with some noise due to the random choise. The step will return a reward that has to be associated to the state and the action: which action? For being sure that the *probs* vector corresponds to the action that returned a given reward, we have promoted the element *action*.

Algorithm 3: Action normalization and promotion

```
1 probs = actor.predict(state)
2 action_probs = [i/sum(probs) for i in probs]
3 action = np.random.choice(n_inputs, p = action_probs )
4 probs[action] = 1
```

Instead of using *choice* function, we tried also *argmax*: as long as it doesn't add any noise, this approach produces less performing results. After this step the rewards and coverage were increasing reaching results comparable to random generation. We clearly needed to improve the performace.

Fourth step: states encoding

As we discussed before, we have to put particular attention on states managing. At some point it was clear that we had to modify the networks in order to have each state independent with each other. We have discovered this needs looking at the most predicted actions per state: we saw that for a given state, an unallowed action had higher probability because of nearer states had that action as allowed. At the same time we don't want to build N_STATES neural networks: it would be too costly and we would loose all actor-critic advantages.

The solution we have taken has been to add an *one_hot_encoding* layer in both actor and critic network, right after the state input layer. With this approach for a given state, there will be just a portion of network that will be *turnedon*. It has been an easy solution (maybe is not the ideal one for bigger DUT configurations) and it has been fast to see a little increase in learning.

Fourth step: adding bonus

Bonus have been added keeping in mind two principles:

- avoid unallowed actions: negative rewards for unallowed actions;
- exploration: returns some bonus when some coverage targets are reached.

Algorithm 4: Reward function

```
if next_state == state then
    reward = -1
end
else
    reward = 1
end
if all actions have been covered for a given state then
    reward = 5
end
if coverage ≥ 0.5 then
    reward = reward + 5
end
else if coverage ≥ 0.6 then
    reward = reward + 10
end
else if coverage ≥ 0.7 then
    reward = reward + 100
end
else if coverage ≥ 0.75 then
    reward = reward + 500
end
else if coverage ≥ 0.8 then
    reward = reward + 2000
end
else if coverage ≥ 0.85 then
    reward = reward + 3000
end
else if coverage ≥ 0.9 then
    reward = reward + 5000
end
else if coverage ≥ 0.95 then
    reward = reward * reward
end
```

The final result is that we are able to increase both the rewards and the coverage. Please notice that in algorithm 4, the expression *coverege* $\geq$ means that the the coverage *just* overcome the threshold. The final result (commented in next section) was considered good, so we didn't search for any improvement.

4 Results

4.1 Model Evaluation and Validation

We didn't perform any parameters evaluation, and we keep the ones as suggested from the papers. Moreover we didn't need any data manipulation: the algorithm is validated with three different DUT configurations.

Model evaluation has been done using a script that launch three times the algorithm, saving the stdout. The resulting logs have been parsed for extracting the results showed later in this report. As shown later, I'm confident (and happy) about the results mainly becasue of the high

level learning trends: this translates in having both high rewards and coverage.

For a better characterization, we may run thousands of tests with random DUT (with the same configurations) and then see if coverage results and learning curves are similar.

4.2 Justification

In the following table is resumed the most important outcome of our project. We can see that for config_1 and config_2, coverage is better when reinforcement learning is used. For config_3 it's a little bit lower. We can state that this approach is good, it works as expected, but still needs some improvement for reaching higher coverage in smaller configurations, and for being more scalable for bigger ones.

	Random	Reinforcement Learning
config_1	0.7377	0.8096
config_2	0.0667	0.1004
config_3	0.0033	0.0028

The main justification is that we should scale the neural networks with the DUT size. Moreover my impression is that the layer *one_hot_encoding* doesn't scale with big N_STATES. Another important point is that we have to increase the number of step for bigger DUT: in the following table is resumed the number of coverage items per configuration, and is easy to see that 500 steps are not sufficient for config_2 and config_3 for reaching high coverage (as for random policy).

	Coverage items
config_1	167
config_2	1317
config_3	53847

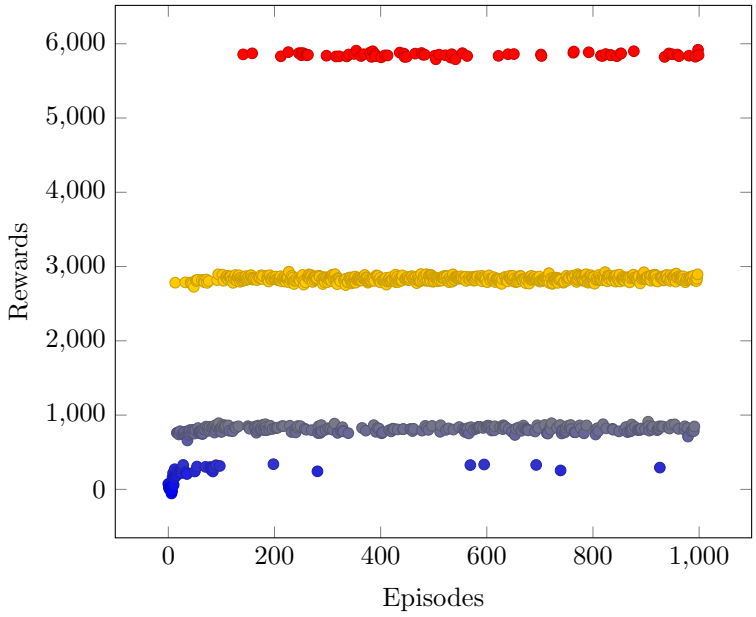
5 Conclusion

5.1 Free-Form Visualization

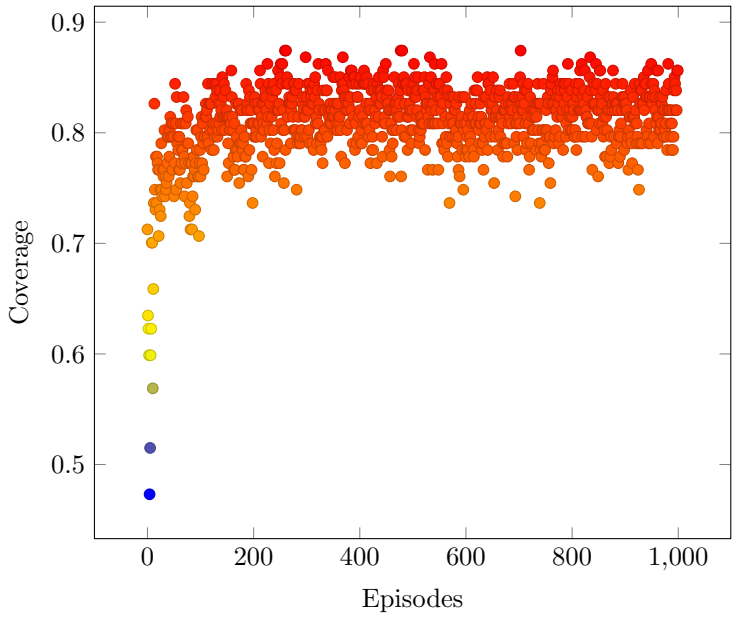
The following graphs shows the rewards and coverage for each configuration.

- config_1: the learning phase is really small; in the reward graph we can distinguish different areas due to the rewards bonus;
- config_2: the learning phase is really long, this may be due to the fact that 500 steps are too few for such DUT; coverage follows rewards expect for last 100 episodes (there isn't any reward bonus for this slow rewards);
- config_3: the neural networks are clearly too small for being able to model such a big DUT and both coverage and rewards are random.

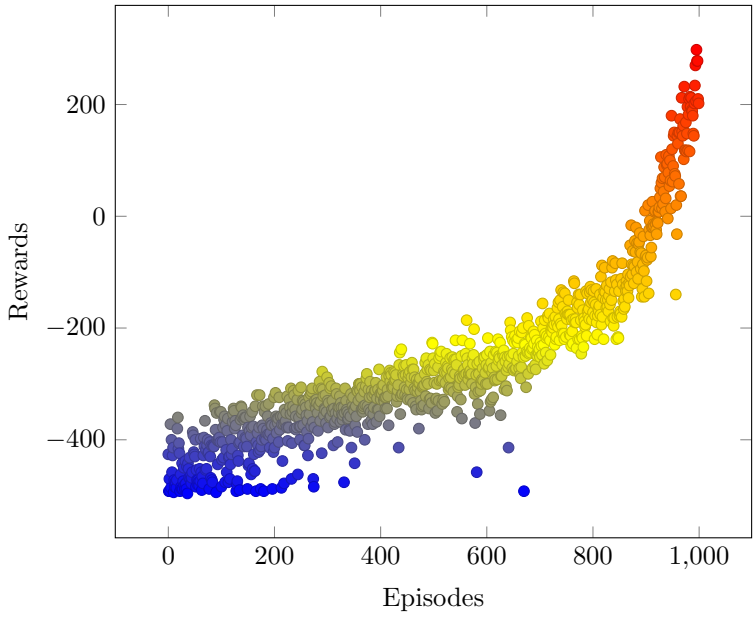
config 1 - Rewards



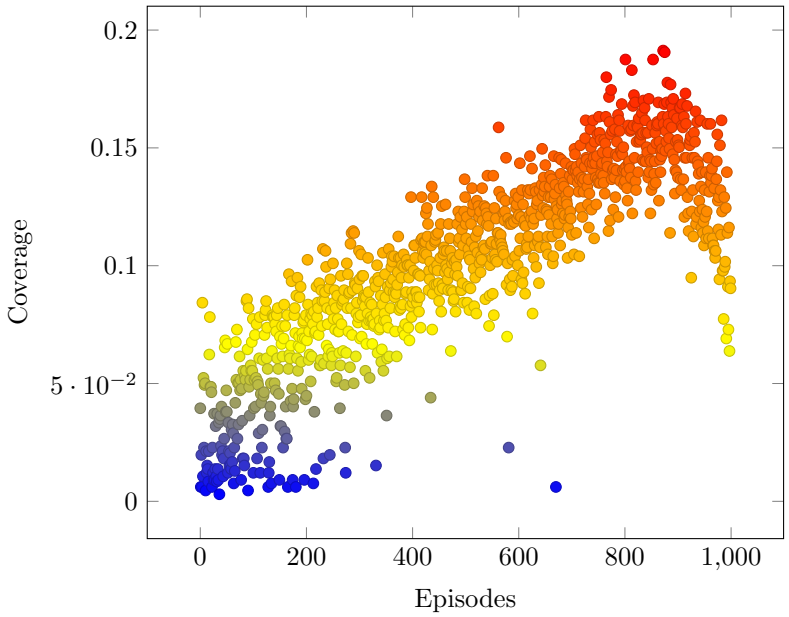
config 1 - Covereage

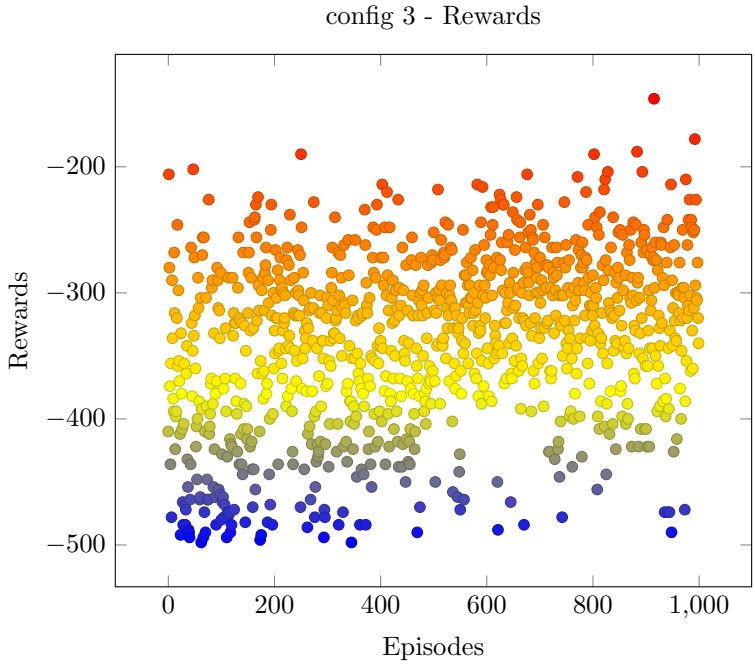


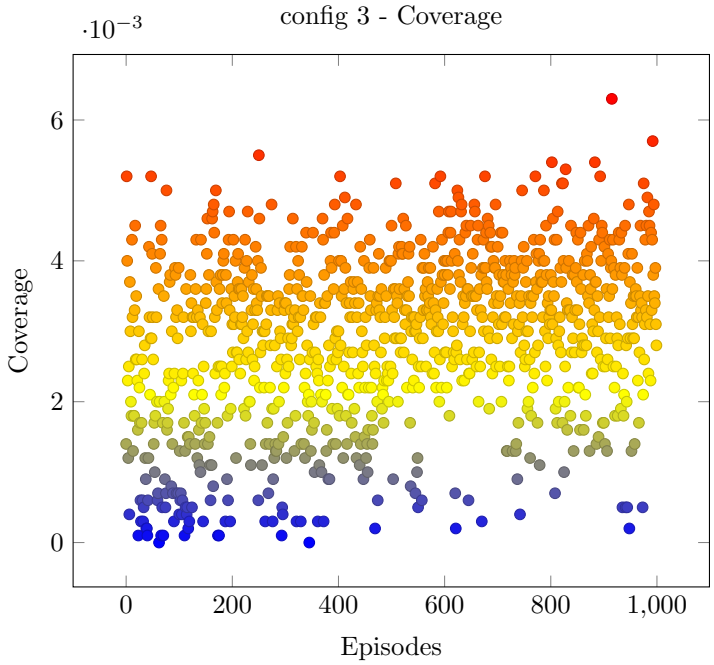
config 2 - Rewards



config 2 - Covereage







5.2 Reflection

The most interesting part of the project was try to imagine how each portion of the algorithm participate in learning, and then find a solution for improving each part. *Findasolution* means searching everywhere: udacity, google scholar, blogs, friends and, most important, our brain. Moreover it has been fashinating to see how modern this topics are, and that this area still needs some stable and robusts bibliography.

More specific, I was really surprised on how the reward function determine the learning: I think that in future there will be a sort of function approximation just for improving this function in order to have a better learning.

5.3 Improvement

The problem solution appears to be good, but there space for improvement. It may be nice now to try with a verilog RTL, and see if it is capable, for example, to full stimulate a real not-random sequence of operations. Another study would imply the sampling of portion of the state (partially observable environment) as for real applications is too costly (the tests would be too long) to probes all the registers at each clock cycles.

Another intereting study would be to use a recurrent neural network, and then search for any solution to travel salesman problem, as done in https://arxiv.org/pdf/1611.09940.pdf.