AI COURSEWORK OF GROUP 66

Jingyan Wang 2533494w, Liangyue Yu 2522553y, Tao Lin 2528179L, Jiahui Ren 2438565r

https://github.com/WayenVan/AiCourseWork

ABSTRACT

In this laboratory, our group focuses on the problem how to take actions to response COVID-19 virus, using AI learning agent to simulate the environment and getting corresponding actions, according to those rewards we can get the optimal strategy. We used several theory to complete the laboratory, such as policy search and q-learning, with 2 different storing methods, tabular and neural network function. After implementation of coding, we have got rewards of different problem situation, and based on them, we evaluate our agents from different aspects. And last, we conclude in several aspects of this laboratory and explore more possibilities of this topic.

Index Terms— COVID-19, policy search, q-learning, neural network, tabular

1. INTRODUCTION

We have all known that COVID-19 virus has affected human beings for almost an entire year, but we still have no idea how to get over it thoroughly so far. People can only reduce the frequencies of gathering to response the serious situation. Take Scotland as example, from 20th November, Scottish cities all lockdown for reducing the number of infected people. But the number still goes high today, almost nearing 1,000. So, to figure out which actions we should take to reduce the number of infected persons, we can use AI agent to simulate the environment and actions, and get the rewards to find the best responses. Our simulation environment is Virl, whose state has 4 dimension:

- susceptible people
- infectious people
- quarantined people
- recovered people

The action space is :

- none
- full Lockdown
- track & trace
- social distancing

Once per week, the agent obtains evidence of the state of the epidemic and can update its policy accordingly. Each episode ends after 52 weeks, irrespective of the remaining number of infected individuals at that time.

2. METHODS

2.1. Discrete state-space

Discrete state-space is a way to split data into several separated spaces. We could easily understood this method by thinking of chessboard, since every chess on the chessboard belongs to a square and every square represent a state in state-spaces. It depends on data how to define the state-space, and importantly, all data should have corresponding state in state-space.

2.2. RBF

In this project, we use an effective function called radial basis function (RBF) to predict the data in the Reinforcement learning. The learning type of RBF is linear regression for the classification. Each $(x_n, y_n) \in D$ will influence the h(x), and it is based on the radial part $||x - \mu_k||$. So, the fundamental form of the RBF can be shown as follows.

$$h(x) = \sum_r (w_k) * exp(-\frac{1}{r_r^2}||x - \mu_k||_2^2)$$

For the code of above formula equation, we could use the package numpy to implement it. Besides, we should set a fixed parameter μ_k to make predict as accurate as possible. ψ refers to the

$$exp(-\frac{1}{r_r^2}||x - \mu_k||_2^2)$$

2.3. Neural Network

Neural Network is a kind of algorithmic mathematics that imitates the behavioral characteristics of animal neural networks. These kind of networks rely on the complexity of the system and achieves the purpose of processing information by

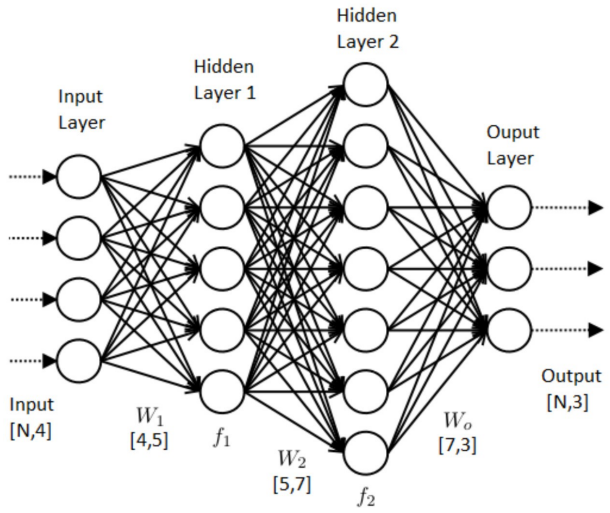

Fig. 1. neural networks

adjusting the interconnection between a large number of internal nodes. The Neural Networks consist of plenty of function. A single neuron represent weighted input with activations:

$$in = \sum_j w_j \cdot x_j = \mathbf{w}^\top \cdot \mathbf{x} = w_0 \cdot 1 + w_1 \cdot x_2 + w_2 \cdot x_1$$

A simple neural network shown by figure1, which includes numerous weights.It always include 2 steps:

- A forward step to propagate the input X to a predicted output Y through each of the layers
- A backward step to propagate the error between the predicted and expected output back through the network from layer L all the way back to layer1

2.4. Q-learning

The utility of one state is related to the action. Thus, to learn an action-utility representation, we define the Q value as the long term value of taking action a in state. Consequently, we get:

$$U(s) = \max_a Q(s,a)$$

Than, from the Bellman equations:

$$U^*(s) = R(s) + \gamma\, max_a \sum_{s'} p(s'|s,a) U^*(s')$$

we could formulate it in $Q(s,a)$ style:

$$Q(s,a) = Q(s,a) + \gamma \sum_{s'} p\left(s'|s,a\right) \max_{a'} Q(s',a')$$

To learn the Q value from environment, once we get a transition, we assumed this happens all the time:

$$p\left(s'|s,a\right) \overset{assume}{=} 1.0$$

Consequent we could get the TD target(assuming real Q value):

$$\text{target} = \hat{Q}(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

Than we calculate the diffrence between current Q value and the TD target:

$$\Delta_Q = \hat{Q}(s,a) - Q(s,a)$$

Finally, we let the current Q value take a small step to TD targets:

$$\hat{Q}(s,a)^{\text{new}} = \hat{Q}(s,a) + \text{stepsize} \times \Delta_Q$$

2.5. Policy search

Policy search is a policy-based method, which output the policy of the action, and represented by π. We use θ as parameters of Neural Network, during the policy search, finds a value of θ that results in good performance; the value could be found by linear function or a nonlinear function such as a neural network. Compared with Q-learning, policy search directly outputs the action value, the process is simpler.
A function which quantifies how good it is to take action in a state,which called $Q_\theta(s,a)$. For selecting the specific action,use the softmax function.

$$\pi(a|s,\theta) = e^{Q_\theta(s,a)} / \sum_{a'} e^{Q_\theta(s,a')}$$

Define a policy value $\rho(\theta)$ to evaluate the policy π is expected or not. Then calculate the full episode to compute the gradient policy by the following function:

$$\bigtriangledown_{\theta\rho}(\theta) \approx \frac{1}{N} \sum_j \frac{(\bigtriangledown_\theta \pi_\theta(s,a_j)) G_J(s)}{\pi_\theta(s,a_j)}$$

Then, gradient step to improve the policy using a suitable step-size α and the approximated gradient:

$$\theta \leftarrow \theta + \alpha \cdot \bigtriangledown_\theta \rho(\theta)$$

Repeat the episode and gradient-based search until convergence.

3. RANDOM AGENT

The working tree of random agent:

```
./
├── .data
├── random_agent.py .2 run_eval.ipynb
└── run_random.ipynb
```

where `run_random.ipynb` implement a simple random agent with random action go throw every episode:

```
for t in range(max_steps_per_episode):
      next_state, reward, done, _ = env.step(
  action=np.random.choice(env.action_space.n))
```

The episodes rewards of random agent is shown by figure2

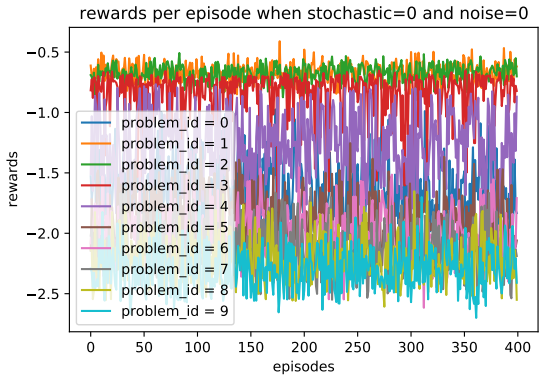

Fig. 2. random agent when S=0 N=0

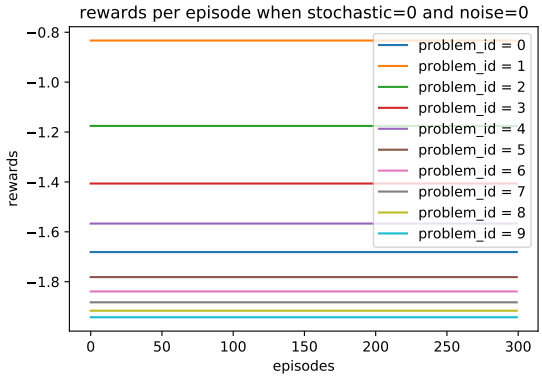

Fig. 3. deterministic agent when S=0 N=0

4. DETERMINISTIC AGENT

The working tree of deterministic agent

```
./
├── .data
├── deterministic_agent.py
├── run_deterministic.ipynb
└── run_eval.ipynb
```

This agent simply apply no-interval action (s[0]):

```
for t in range(max_steps_per_episode):
    next_state, reward, done, _ = env.step(action
    =0) #no interval at all!
```

The episodes rewards of this agent is show as figure3

5. Q-LEARNING WITH TABULAR METHODS

5.1. Implementation

My **working tree** is consisted by a data folder and some other `.py` or `.ipynb` files. `q_learning_tab.py` is a class designed for training agent and save Q table. `run_qlerntab.ipynb` is for running the agent and save training data into data folder. `evaluation.py` is for plot different analysis figures which focus on different points. `run_eval.ipynb` is for running all plots in a single program. As for data folder, there are several sub-folders and log files. `q_table` is for storing trained Q-tables of different situations of environments. `stats_training` is for storing trained rewards data of different situations of environments. `training.log` is for storing training information of every time start training.

```
./
├── evaluation.py
├── q_learning_tab.py
├── run_eval.ipynb
├── run_qlerntab.ipynb
└── data
```

Q-table (as shown below) of my implementation is a discrete state=space table. It is designed according to environment data. We found every time we step forward, the output is always an array formed by 4 kinds of data, so we can confirmed that data has 4 different properties. And the highest value is below $6e+08$, hence we can dived a single property into **7** different state from $[0, 6]$. Moreover, we could get there are 4 actions to take. Therefore, we could get a $(7 \times 7 \times 7 \times 7 \times 4)$ tables including all states. My way is to use a number as a state, so we can define a dictionary to build a link between state and index.

```
        1         2         3         4
0     0.0  1.000000  2.000000  3.000000
1     0.0 -0.031416 -0.012097 -0.007621
2     0.0  0.000000  0.000000  0.000000
3     0.0  0.000000  0.000000  0.000000
4     0.0  0.000000  0.000000  0.000000
...   ...       ...       ...       ...
2397  0.0  0.000000  0.000000  0.000000
2398  0.0  0.000000  0.000000  0.000000
2399  0.0  0.000000  0.000000  0.000000
2400  0.0  0.000000  0.000000  0.000000
2401  0.0  0.000000  0.000000  0.000000
```

Q-learning is a theory for AI learning, and it stores learning results in some ways and according to the results to get the optimized solutions.
I used discrete state-space, q-table we mentioned above, to store the training results, and I take the equation as reference to train my agent. And in my code, I write my agent based on the equation above and make little changes because of data formats. And eventually, it partially shows below.

```
# q-learning
Q.loc[s, a] += alpha * (r + gamma * np.max(Q.loc[
    s1_index, :].values) - Q.loc[s, a])
```

For running the agent, I set several arguments to treat different requirements and situations.

Table 1. arguments of agent

args	description
`env`	data environment
`num_episodes`	number of episodes
`max_step_week`	max steps in single episodes
`args`	stochastic, noisy and problem id

5.2. Evaluation

I evaluate my agent from the following angles:

Rewards analysis: observing the rewards is the most direct way to detect whether a agent is effective. So that we can choose training data and trained data of a problem to compare the differences. we can see from training data and trained data (as shown in figure 4) of problem 0. Take problem 0 as an example, we can find that the rewards of training period $[0, 600]$ shows badly at first but nearing 0 at the end, it typically shows our agent has learned to make better decision through training.

Robustness: the way we determined whether the robustness is great or not, is to observe whether rewards of the agent converge and become stable start from some point. As shown in figure 4 and figure 5, it is obvious that the rewards converge nearing $[200, 400]$ in figure 4 and become stable afterwards, it also happened in figure 5 where rewards converge at $[200, 400]$.

Training rewards versus Test rewards: in the 4 we can also find another two graphics based on problem 0 but taking different properties, one of them adds noise for comparing. The comparing result is very significant: the rewards of training shows badly at first but converge at the end; the rewards of test adding noisy shows badly but floating in a period $[-2.0, -1.0]$ stably; the rewards of test adds nothing shows perfectly, almost like a horizontal line, rewards of this rest complete converged. This consequence shows that training agent is really working in this situation. For ensuring the conclusion, we can see another graphics, figure 5, to compare with the former one.

Comparing with random and deterministic agent: Random agent has no learning process, or learning ability, so we can compare results of q-learning agent with random agent, and then we can easily figure out whether our agent is efficient or not. As shown in figure 9, this figure shows average rewards value of the random agent is around $[-0.7, -0.6]$ and our test rewards float at $[-0.4, -0.3]$. It means: comparing with random agent our q-learning agent is better than random agent.

Noisy versus No noisy: The rewards of no noisy environment and noisy environment is totally different. We can find rewards of no noisy environment converge completely, but rewards of noisy environment float strongly. It indicates that our agent works well in different environment.

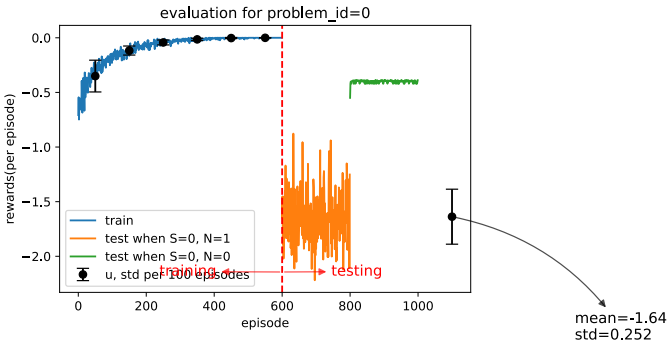

Fig. 4. Evaluation for problem id = 0

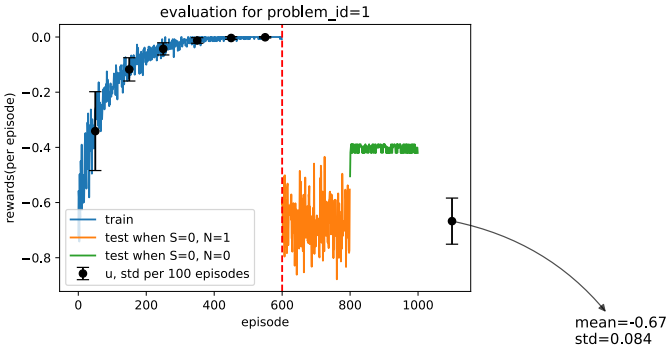

Fig. 5. Evaluation for problem id = 1

6. Q-LEARNING WITH NEURAL NETWORK FUNCTION APPROXIMATION

6.1. Implementation

To implement a Q-learning agent with neural network, I organize all files as:

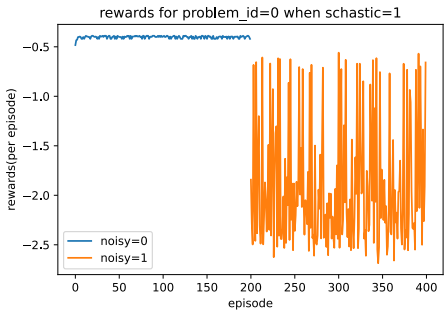

Fig. 6. Rewards for problem id = 0 when stochastic = 1

```
./
├── approximator.py
├── evaluation.py
├── q_learning_nn.py
├── run_eval.ipynb
├── run_qlernn.ipynb
└── data
```

The `run_qlernn.ipynb` file run the training phase all 10 instances with all data saved in `data` directory. The `run_eval.ipynb` run the test phase for all 10 instances and load these data to visualize them. The `evaluation.py` consists of some functions to plot as well as manipulate variables. The `run_qlernn.ipynb` consists of Q learning agent, neural network function approximator and any other class for replaying episodes or storage datas(training log and stats).

When design a Q learning agent, neural network could be used as a powerful tools to implement a function approximator. Specificly, I use the implementatioini approximtor class Joint with Keras `NNFunctionApproximatorJointKeras` **from lab**.

```
alpha = 0.004
nn_config = [36,36]
nn_func_approximator =
    NNFunctionApproximatorJointKeras(alpha,
    d_states, n_actions, nn_config)
```

This class use the Keras to construct a state-to-Q network, throw which a 4-dimensional state generate a 4-dimensional Q table. Our goal is to optimize the hidden weights by using gradient descent on Huber loss function. In respect to parameters,the reason of choosing 2-layer model is that this structure is enough to represent any decision boundary with any precision in a relative small scalar. Selecting a proper learning rate is a vital step for a superior model. At the beginning, I applied alpha with 0.01, which leads to a explosion of gradient. Thus, I gradually shrink this value into 0.004 to see a obvious convergence.

To design a Q learning process, I use the `q_learning_nn` function **from lab** with a sightly change (normalization):

```
#Training
BATCH_SIZE = 80
BUFFER_SIZE = 8000
num_episodes = 1000
stats = q_learning_nn(env_train,
    nn_func_approximator,
    nn_func_approximator_target, num_episodes,
    use_normalization = True,
    max_steps_per_episode=500,discount_factor=0.9,
    epsilon_init=0.1,epsilon_decay=0.99995,
    epsilon_min=0.001,
    use_batch_updates=True, BATCH_SIZE=BATCH_SIZE,
    fn_model_in=None, fn_model_out=r"./data/
    approximator_training/cartpole0"+str(
    num_instance)+".h5", BUFFER_SIZE=BUFFER_SIZE)
```

When acting Q learning process, every step we firstly predicate the Q table by the approximator, than take an action by greedy policy(action with best Q value). For each step we save all transitions in replay memory. After that, we sample data from replay memory, calculate the TD(temporal diiferentiate) targets as the "real sample value", finally feed it into the neural network to do gradient descent step. Concerning to the COVID, we prefer a long term reward. Thus, I set the discount factor as 0.9. Exploration steps are also reflected in he epsilon value, which is decreasing follow the training process. The batch updating strategy is used for improving performance by using extended data set.

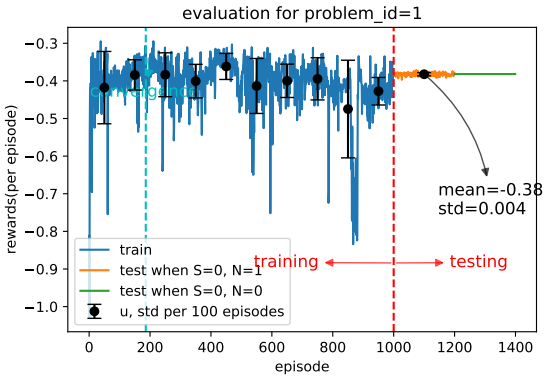

Fig. 7. rewards all in one of problem instance 1

6.2. Evaluation

To evaluation performance of a agent, we choose the reward per episode as metrics. (as shown by figure7. The model was trained when noise=False, stochastic = true.From the figure we could know the model converged around 186 episode. Once it entered test environment with alpha=0(stop learning), the expectation of rewards reach -0.38. Moreover, once we set noise = 0, the standard deviation shrink into almost zero. However, there is a few huge fluctuations during training phase, I assume that is result from the random action when we apply our policy. To test the robustness of my agent, I trained all the problem instances, which saved in `run_eval.ipnb`. Here is another 2 example (as shown by figure8). The standard deviation in training phase of these 3 instances are different, but they all converged finally. Thus, we could say it has robustness to a certain extent.

To prove this model works better, I compared it with the random agent and deterministic agent(always no interval). Testing result of these 2 agent on problem instance 1 are plotted as figure9. The expectation rewards of these 2 are around Sadly, however, this agent seems not suitable for stochastic problems. When set the stochastic=True the result is totally messive (as shown by figure10

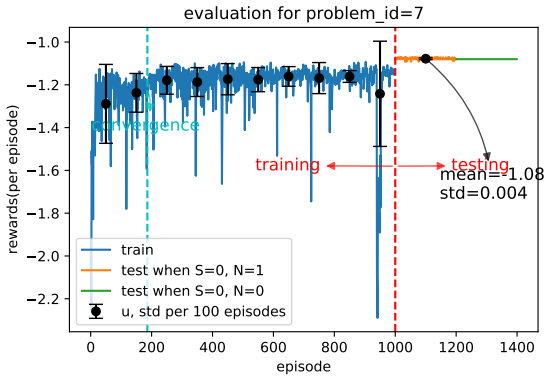

Fig. 8. rewards all in one of problem instance 7

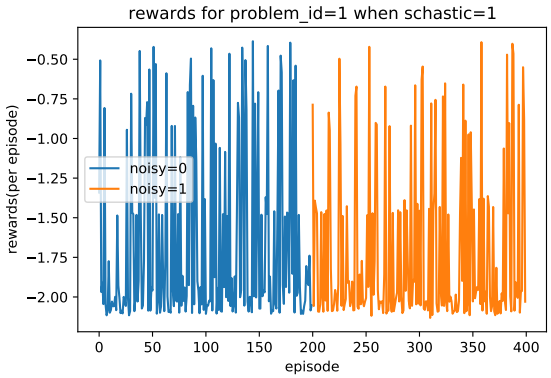

Fig. 10. rewards of problem instance 1 when open stochastic

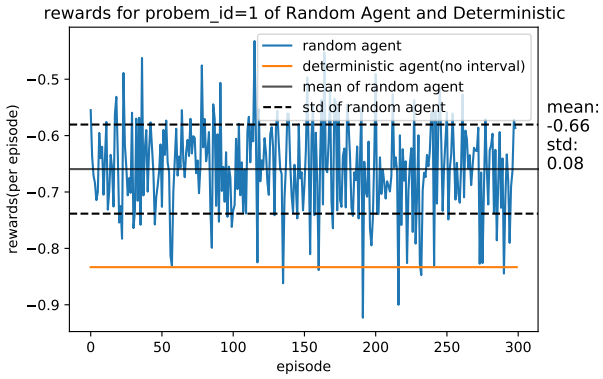

Fig. 9. rewards in testing phase of random & deterministic agent

7. POLICY SEARCH WITH LINEAR FUNCTION APPROXIMATION

7.1. Implementation

I use the **working tree** to show the files and their functions, the types of procedure file are consists of `.py` and `.pynb`

```
./
├── policy_search_RBF.py
├── run_eval_RBF.ipynb
├── run_policy_search_RBF.ipynb
├── data
└── picture
```

The trained model of env has been stored in the file `data`, and this file also store the stats (length and awards) for training phase and test phase. `'run_policy_search_RBF.ipynb'`: Plot the episode rewards of training phase. Execute the policy research, create the function approximation class and set the reinforce learning function. `'policy_search_RBF.py'`: Store the policy research function, function approximation class and the reinforce learning function. It is used be invoked by another file. `'run_eval_rbf.ipynb'`: Evaluate the agent to analysis the results. In addition, The episode rewards has been plotted in the test phase. `'evaluation\_RBF.py'`: the function is used to plot the figure.

RBF function is used as an effective agent to implement the agent. As we mentioned above, To set the different ψ, we could plot different bump shapes. The following pseudo code is about the RBF formulate function.

```
# pseudo code
w[0] + np.sum(w *  np.exp(-np.Linalg.norm(x-\mu_k)
    **2/ gamma**2))
])
```

In order to implement The RBF be more efficient, the python has packaged the toolbox *sklean*

```
# RBF regression model
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(observation_examples)
transformer = sklearn.pipeline.FeatureUnion([
("rbf1", BFSampler(gamma = 5.0, n_components =
    100))
("rbf1", BFSampler(gamma = 2.0, n_components =
    100))
("rbf1", BFSampler(gamma = 1.0, n_components =
    100))
("rbf1", BFSampler(gamma = 0.5, n_components =
    100))
])
```

The function approximator is come from **Lab** and this class is used for getting the prediction and update the model. Besides, it is also used for handle the function approxiamtor for the reinforce function that is based on the **Lab** I use the package *SGDRegressor* to implement the optimal square loss for the improvement of performance. Aiming at optimizing the loss of w

$$w* = argminLoss(w; ((x_n), (y_n))^{n=1.8}$$

```
# Regression
model = SGDRegressor(learn_rate = learn_rate, tol
    = le-5, max_iter = le5, eta0 = eta0)
```

The fixed parameter: epsilon = 0.1, discount_factor = 0.9, epsilon_delay = 0.995.

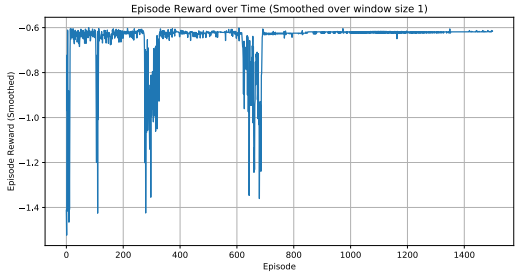

Fig. 11. Training phase for id 3

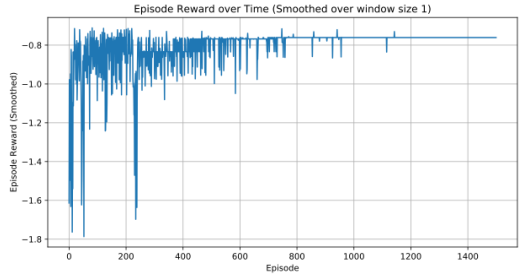

Fig. 12. Training phase for id 4

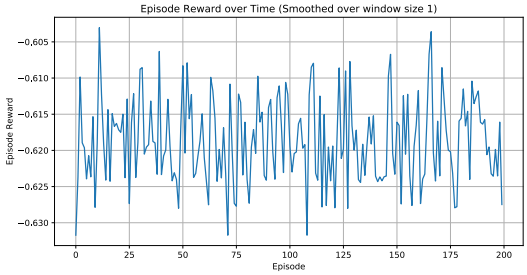

Fig. 13. Evaluation phase for TF_id 3

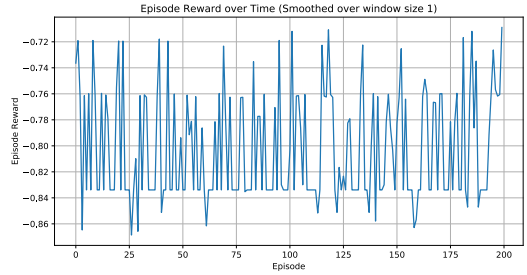

Fig. 14. Evaluation phase for TF_id 4

7.2. Evaluation

All problems' id has been saved into the file../picture/train. I take two typical examples (id = 3 and id = 4) (as shown by figure7.2 and figure12) to analysis the rewards of the training phase. For the problem id3, when the episode increases by 700, this problem id become more convergent and the episode rewards are approximate to -0.6. For the problem id4, it achieves an effective regression around the 600 episode. The average of episode rewards is about -0.75. At the evaluation process, The parameter epsilon and eta0 is set to zero. Additionally, a new test rewards has been calculated and saved by handling the four trained models.

```
# Setting parameters
approximator = FunctionApproximator(env, scaler,
    feature_transformer, eta0= 0.00, learning_rate
    = "constant", read_approximator = path)
stats = reinforce(env, approximator, 200,
    use_training=False, epsilon = 0)
```

All evaluation rewards of problem id have been saved, I attached the problem of id3 and id4 in the condition Noisy = True and stochastic = False to analysis. (as shown by figure7.2 and figure7.2) Compared with the training figure, the average of evaluated id3 is around -0.618 and the evaluated id4 is around -0.79, which is approximate to the training phase awards for the id3 and id4. At the evaluation part, I plot all test rewards of problems in one figure to compare their episode rewards and the degree of convergence (as shown by figure15). From the figure15, the id1, id2, id3 and id4 have a high astringency. Although the id 6, 7, 8, 9 converges very quick, the episode rewards is nearly -2.0. Obviously, it is a wrong data for the id 6, 7, 8, 9. In some degree, the reward results are not stable. I hold the view that problems that id = 0, 1, 2, 3, 4, 5 for evaluation phase have better performance in robustness owing to the average, variance and amplitude values of episode rewards and the rate of convergence. For the other condition: noisy = False and stochastic = False, the figure shows a straight line. For the condition: stochastic = True and noisy = True, the generated awards could not be convergent.

8. POLICY SEARCH WITH TABULAR METHODS USING DISCREDITED STATE-SPACE

8.1. Implementation

The policy research could be implemented by following files:

```
./
├── data
├── policysearch_DS_evaluation.ipynb
├── run_policy_search_ds.ipynb
└── policy_search_ds.py
```

The `run_policy_search_ds.ipynb` file run training phase and testing phase with all data saved in `data` directory. The `policysearch_DS_evaluation.ipynb` file polted the reward of both train result and test result figures, which could be edited by change the name of file. During the process of policy search with tabular method, the state-space has

Fig. 15. The evaluation rewards of all problem ID

been segmented into 7 different sate for [1,8], after that, $(7 \times 7 \times 7 \times 7)$ tables including all states, the out put will be the state and their corresponded index number. Observed the code **from lab9**, need to zero the table before the input, and then only set one to the current input state to activate the neural network. The code for the sampling process located in `policy_search_ds.py` as follows:

```
def one_hot(state, tabular):
    state = state/1.0e8
    split = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
    7.0]
    state_ds = np.digitize(state, split)
    state_ds = tuple(state_ds)
    state_oh = np.zeros((1, len(tabular)))

    state_oh[0,tabular[state_ds]] = 1.0
    return state_oh
```

In order to implement the policy search algorithm, which need to build up the neural network with **loss function**,which need to be minimized. This part of setting also refer from **lab9**. After training, the average reward is approx to -1.38.

8.2. Evaluation

In the train section,All of the reward of problems id has been saved into the file :/data/data_train. The train evaluation rewards of two different problem id1 and problem id2 was selected with the same leaning rate $\alpha = 0.002$ and the same discount factor$\gamma = 0.99$, after 1200 times episodes,the result figure could be shown as Figure 14 and Figure 15. In the test section, there are four environment variables combination could be changed in the environment, there has been codded respectively : stochastic=True, noisy=False; stochastic=True, noisy=False;stochastic=False, noisy=True;stochastic=False, noisy=False; All of the test evaluation rewards will stored in the file : /data/data_test. During this part, the addition of noise and stochastic with the same problem id=2 ,set the learning rate at 0 ,with the same episode number of 200, the result figure could be shown as figure 18 and figure 19.

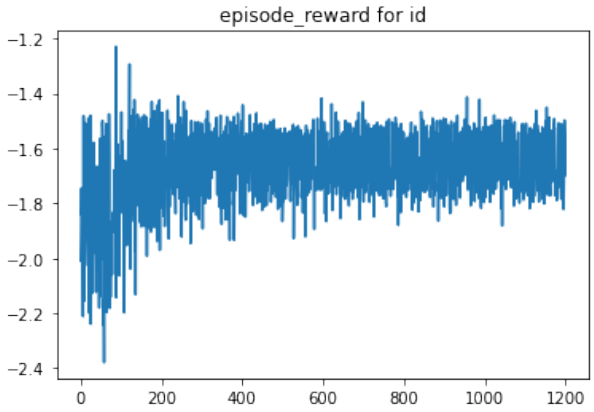

Fig. 16. Training phase for id 1

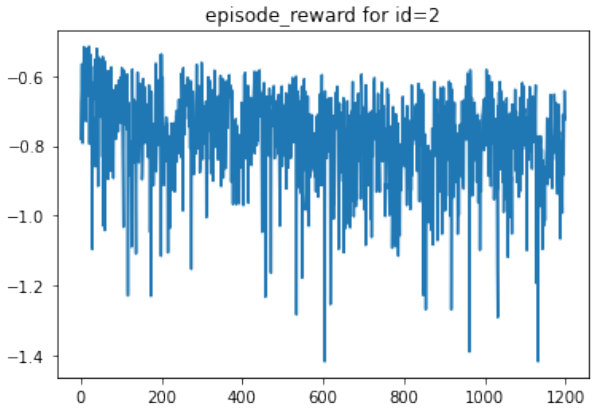

Fig. 17. Training phase for id 2

9. CONCLUSION

In this project, we implement six agents totally. The detailed information about the performance of the agent has been analyzed. We use a wide range of reinforcement learning approaches to predict the rewards in terms of discrete state-space, RBF, neutral network, Q learning and policy search.

Although we generate various decent models, there are also some deficiencies. For example, we could not generate effective models under the stochastic is 'False'. In addition, although we have trained our model so many times, it could not get the perfect model. The agent could not solve all problems owing to the augmentability. Improving the accuracy and performance of our agent is our aim in the future.

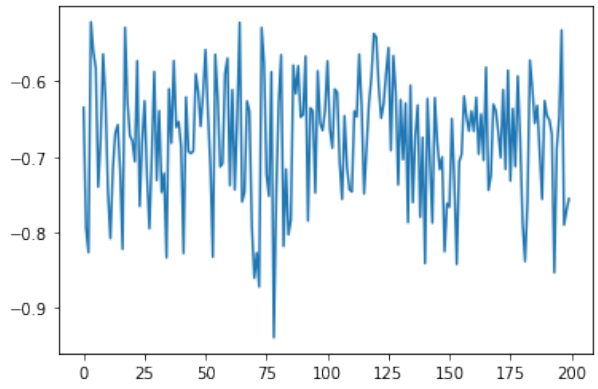

Fig. 18. rewards in testing phase of S=0 N=1 with problem id=2

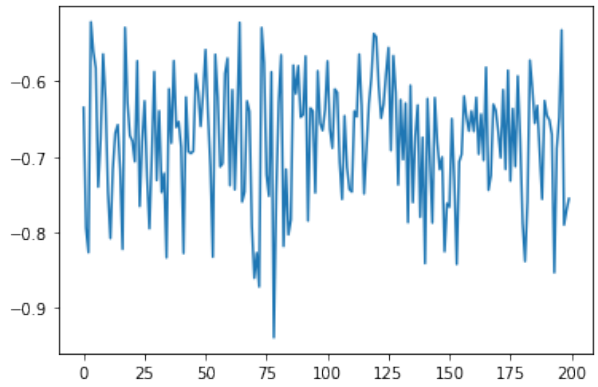

Fig. 19. rewards in testing phase of S=1 N=0 with problem id=2