COMP6208 Reinforcement and Online Learning Coursework Report

MSc.Artificial Intelligence
Student: Wei Lou
wl4u19@soton.ac.uk
Supervisor: Dr Mahesan Niranjan

I. Introduction

THE radial basis function method can be thought as an extension of linear regression on non-linear problems. The basis functions are defined by:

$$f(\boldsymbol{x}) = \sum_{j=1}^{J} w_j \phi\left(\|\boldsymbol{x} - \mathbf{m_j}\| / \sigma_{\mathbf{j}}\right)$$

The non-linear function $\phi(.)$ is usually a Gaussian function and σ_j offers the flexibility to control the region of influence of the basis functions. When having a training set $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$, the RBF can construct an N*J matrix U and the weight W of RBF function can be calculated by

$$\boldsymbol{w} = \left(U^T U\right)^{-1} U^T \boldsymbol{y}$$

Adaptively estimating the w by a gradient search method is the key to the use of the model in the reinforcement learning setting.

II. Solve a regression problem by using RBF model.

The dataset using in this task is the house price data. The training data is a 506*13 matrix and the target is a 506*1 matrix. The first way to implement the RBF model using the whole dataset. First, calculate the centres using Kmeans and the standard deviation based on the data distribution. Then construct the design matrix U, update the weights w. Finally, get the prediction = U*w
The following 2 images are the linear regression result of training data and the RBF prediction result.

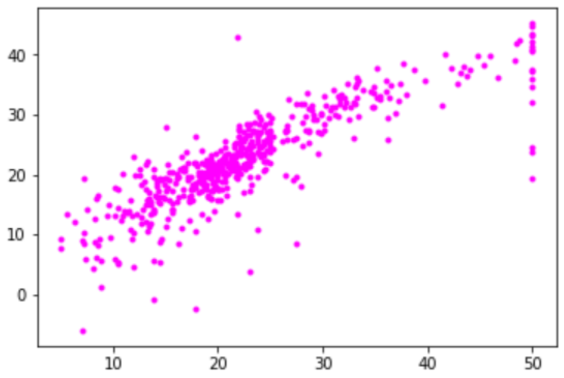


Fig. 1: The error between Linear prediction and original data is about 110.58

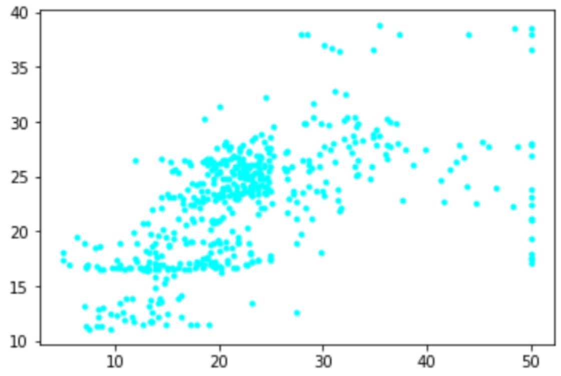


Fig. 2: The error between RBF prediction and original data is about 168.29

We can find that the RBF model performs not so good even compare with the linear model because the randomness of Kmeans initialization.

A. RBF with gradient descent

The other way is to implement a gradient descent method to update the parameters of the RBF model sequentially.
1. First, normalize the input data.

2. Set the hidden unit numbers as n_hidden and batch size as 1
3. Random Initialize centre m, sigma $\sigma_\mathbf{j}$, w
4. Calculate the output by using:
$f(\boldsymbol{x})=\sum_{j=1}^{J} w_j \phi\left(\|\boldsymbol{x}-\mathbf{m_j}\| / \sigma_\mathbf{j}\right)$ and the Error = y - f(x)
5. Update the parameters:
m = m + alpha * dm * Error
$\sigma_\mathbf{j}$ = $\sigma_\mathbf{j}$ + alpha * d $\sigma_\mathbf{j}$ * Error
w = w +alpha * dw *Error
6. Calculate the prediction by using the new m, $\sigma_\mathbf{j}$ and w The model converge error curve is shown in Fig.3

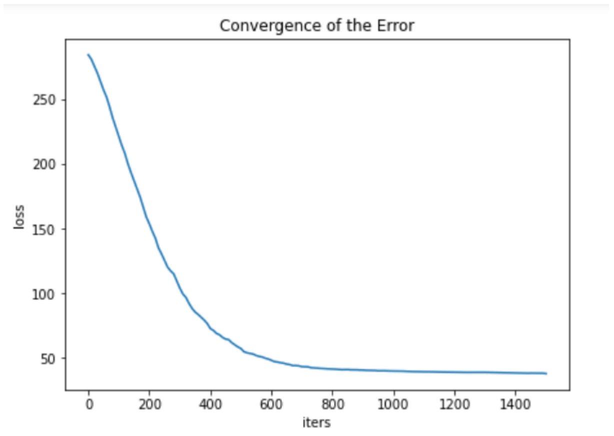


Fig. 3: The Convergence Error curve

III. Solve the mountain car problem

Mountain Car, a standard testing domain in Reinforcement Learning, is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill.

A. *RBF with tabular discretization method*

The basic idea of this solution is to combine tabular discretization method with RBF approximator. And then evaluate the performance when using different number of hidden units.
The discretization method scales the observed position and speed as integers which can be shown on girds. Create a q_table to retore the q_value for each state and corresponding action.
The whole process of each episode is:
1. Initialize the number of hidden unit n_hidden, learning rate lr, gamma, sigma, w, centres m.
2. For each iteration, observation is discretized and normalized as the state1.
3. Get the action a from argmax(RBF.predict(state1)) and get the new state2 and reward by using env.step(a) and discretize state.
4. Update the q_table by using new state2 and reward.
5. Train the RBF model by using state1 and q_table[state1] and update the centre m, weight w and sigma.
6. If terminate, break the iterations and output the cumulated reward
With different model complexity, the performance of RBF model for this task is different, in this lab, the number of hidden units are chosen as 3,5,10,20,40,80 for comparison. In this comparison, I recognize the episode cost more than 1000 steps as fail and get to the top within 1000 steps as success. The performance difference with various hidden units numbers are shown in Table.1.
We can find from the table, the very small or very large hidden units would decrease the performance of RBF model and achieve fewer success episodes in the testing stage. And if reach the mountain top, in this method, the time cost is similar.

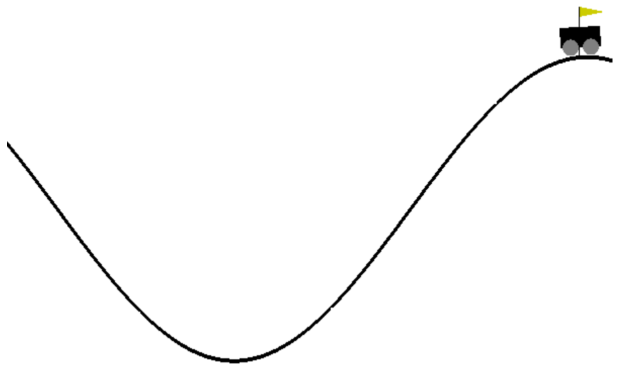


Fig. 4: The Final state of the car

B. *RBF with Q-learning*

When using continuous observations to update q value in Q-learning method. The q value update function becomes: $Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right)+\alpha\left[R_{t+1}+\gamma \max_a Q\left(S_{t+1}, a\right)-Q\left(S_t, A_t\right)\right]$
And most of the processes of the q-learning algorithm are the same with the last methods. Except, in this case, the observations are not discrete integers but real float numbers. And the $Q\left(S_{t+1}, a\right)$ is the output of RBF.predict(new state)
The whole process of each episode is:
1. Initialize the number of hidden unit n_hidden, learning rate lr, gamma, sigma, w, centres m.

TABLE I: The performance of tabular RBF with different complexity

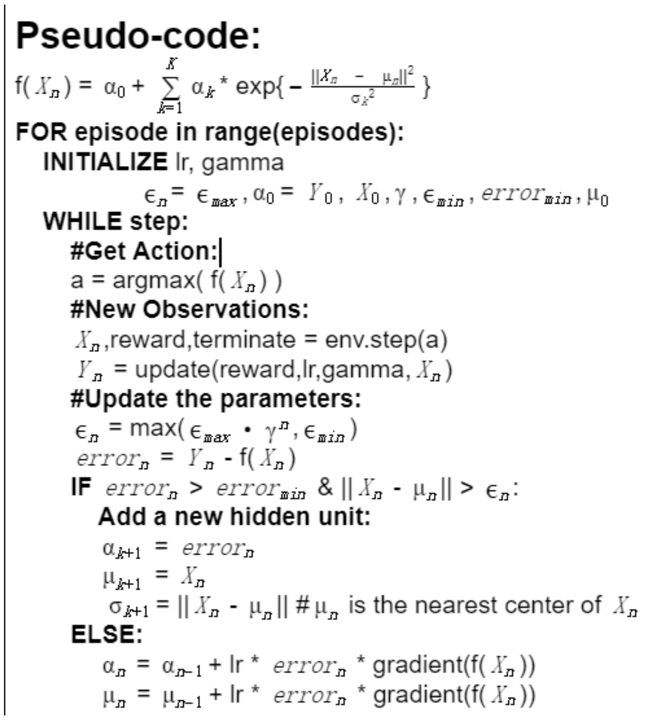
Hidden size	Average steps	Success numbers of 10 episode	Average cumulated reward
3	329	1	107
5	503	3	132
10	471	5	551
20	537	7	134
40	391	6	114
80	235	1	59

2. For each iteration, observation is normalized as the state1.
3. Get the action a from argmax(RBF.predict(state1)) and get the new state2 and reward by using env.step(a) and normalized state.
4. Calculate the $Q\left(S_{t+1}, a\right)$ using RBF.predict(state2) and update the q_value[state][a]
5. Train the RBF model by using state1 and q_value[state] and update the centre m, weight w and sigma.
6. If terminate, break the iterations and output the cumulated reward

When using hidden unit number as 20, the final comparison result is shown in Table.2. This time there is no step limitation.

IV. Incorporated RAN into implementation

For now, even we have tried different sizes of hidden units, every time the model is still training with fixed model size. Sometimes, it's better that the complexity of the model can change adaptively with more observations. So we can have a minimal network for a sequential problem. Implement RAN for this task:



```
Pseudo-code:
$f(X_n) = \alpha_0 + \sum_{k=1}^{K} \alpha_k * \exp\{-\frac{||X_n - \mu_n||^2}{\sigma_k^2}\}$
FOR episode in range(episodes):
  INITIALIZE lr, gamma
          $\epsilon_n = \epsilon_{max}, \alpha_0 = Y_0, X_0, \gamma, \epsilon_{min}, error_{min}, \mu_0$
  WHILE step:
    #Get Action:
    a = argmax( f( $X_n$ ) )
    #New Observations:
    $X_n$,reward,terminate = env.step(a)
    $Y_n$ = update(reward,lr,gamma, $X_n$)
    #Update the parameters:
    $\epsilon_n = \max(\epsilon_{max} \cdot \gamma^n, \epsilon_{min})$
    $error_n = Y_n - f(X_n)$
    IF $error_n > error_{min}$ & $||X_n - \mu_n|| > \epsilon_n$:
      Add a new hidden unit:
      $\alpha_{k+1} = error_n$
      $\mu_{k+1} = X_n$
      $\sigma_{k+1} = ||X_n - \mu_n||$ # $\mu_n$ is the nearest center of $X_n$
    ELSE:
      $\alpha_n = \alpha_{n-1}$ + lr * $error_n$ * gradient(f( $X_n$ ))
      $\mu_n = \mu_{n-1}$ + lr * $error_n$ * gradient(f( $X_n$ ))
```

Fig. 5: Pseudo-code for implement RAN

TABLE II: The performance comparison between tabular method and Q-learning

Hidden size	Average steps	Success numbers of 10 episode	Average cumulated reward
Tabular	1274	7	360
q-learning	1584	10	306

V. Paper Summery

The paper is chosen for this section is 'Resource management with deep reinforcement learning' [1]

A. Problem Statement

Resource management problems are commonly encountered in modern computer system or networks. Most of the hardware devices have multiple resources like CPU, Memory, GPU, I/O. When running different tasks, how to efficiently allocate these resources is vital to maximising the usage of these workloads.
However, real-world resource management has many challenges, the base system is very complex and can not be modelled as a fixed model. The jobs can intervene on shared resources. So, in practice, the allocation needs to be made with online inputs and work well under diverse conditions. Now most of the time, we are still using human-made heuristic methods to solve the issue.
This leads to a problem that if the machine can learn how to manage resource themselves? And this paper shows a solution called DeepRM by using reinforcement learning.

B. Why Reinforcement Learning

Reinforcement learning deals with agents that learn to make decisions directly from past experience interact with the environment which perfectly meets the requirement of this task.
With more detail, those systems and networks are making lots of repeatable decisions which helps the RL model to learn underlying rules efficiently. And nowadays, RL is always implemented with Deep neural network which shows great inference ability. It's possible that model complex systems and decision making policy as DNNs. Besides, DNN can add noise and raw signals into training data which can increase the generalization ability to make the model run in a real environment. Also, when setting different rewards, it's possible to train accurate models on specific objectives like small jobs and low loads.

C. The design of DeepRM

1) Model : Consider an environment which has d kinds of resource types(CPU,I/O, Memory……). The jobs arrive at the cluster in an online fashion in discrete timesteps. The resource profile of resources requirement of each job are $\mathbf{r}_j = (r_{j,1}, \ldots, r_{j,d})$ and the duration of job j is T_j. And for simplicity, there is no preemption and fixed allocation profile. Besides, the average job slowdown— $S_j = C_j/T_j$is used as the system objective. Cj is the completion time of the job and Tj is the ideal duration of the job j.

2) State Space : The state of the system contains:
1. The current allocation of cluster resources
2. The resource profiles of jobs waiting to be scheduled
Ideally, we can have as many job slot waiting for service in the state, but as the input of a Deep neural network, it's desired to have a fixed size. So only the first M jobs in the waiting list are maintained. This approach adds the advantage of constraining the action space which makes the learning process more efficient.

3) Action Space: The action space is given by $\{\emptyset, 1, \ldots, M\}$, if a = i mean 'schedule the job at i-th slot' and if a = $\emptyset$, it's an invalid action so the scheduler will not schedule any jobs. With a valid action, the agent then observes a state transition:
The scheduled job is moved to the appropriate position in the cluster image. With an invalid action, time proceeds.

4) Rewards: The reward at each time step is: $\sum_{j \in \mathcal{J}} \frac{-1}{T_j}$. The $\mathcal{J}$ is all the jobs in the system. If set $\gamma = 1$, the cumulative reward over time coincides with the sum of job slowdowns, so maximize the cumulative reward is the same as minimize the average slowdown.

D. Learning paradigm

In this paper, a policy network is trained with states and outputs a probability distribution over all possible actions. At each iteration, the action can be got by the output of the pocicy network based on the current state. And the state, action and rewards will be recorded to train the neural network using reinforce algorithm:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta (s_t, a_t) v_t$$

In order to reduce the high variance, the v_t needs to minus its average value called baseline bt.

$$\Delta\theta \leftarrow \Delta\theta + \alpha \nabla_\theta \log \pi_\theta \left(s_t^i, a_t^i\right) \left(v_t^i - b_t^i\right)$$

And with different reward types, this RL network can realize other objectives.

E. Result

After comparison with other management methods on some aspects like the average job slowdown and the average job completion time.
The result shows the DeepRM is often better than all heuristics methods and it can learn directly from experience without any prior knowledge. Besides, the DeepRM method is customizable for different objectives. Also, the DeepRM converges quicker and learns strategies that are more sensible.

F. Potential Usage

In real-world applications, this method may not only be used on computer systems, but also used on other large systems like city public transport resource management systems or factory production chain management. By learning the allocation policy from past experience itself, the systems may work more efficient and smart than the system under human's arrangement.

References

[1] Mao, H., Alizadeh, M., Menache, I. and Kandula, S., *Resource management with deep reinforcement learning*, Proceedings of the 15th ACM Workshop on Hot Topics in Networks: 50-56, 2016.