

Part 1: Radial Basis Functions

1.1 RBF Regression

To explore the use of radial basis functions (RBF) for function approximation in reinforcement learning, we started by implementing an RBF model for solving a regression problem. An RBF model is consisted of nonlinear basis functions, typically a gaussian function, and a set of learnable weights. For a gaussian RBF model, the hyperparameters includes the number of basis functions (J), the location of each basis function (m_j), and the width of each basis function (σ_j). Since the basis functions are linearly combined, the gaussian RBF model are *linear* in parameters. Therefore, in addition to being able to incrementally updating the weight using gradient descent, it can be solved in closed-form using pseudo inverse method, and there exists a true global minimum.

For this task, the Wine Quality dataset from UCI repository is selected as a regression problem. The goal is to predicted the quality of the wine given the attributes, such as acidity, pH value, and alcohol of each wine. There are a total of 11 features and a target (quality score between 0 and 10). The steps of solving this problem using Gaussian RBF are summarised as follows.

1. The features are normalized to have zero mean and unit variance, so that all features are of equal importance, and this would also help gradient descent converges faster.
2. The number of basis function (J) is initially set to be 20, then later improved using k-fold validation.
3. The width of the basis function (σ_j) is selected using the average of pairwise distance from data.
4. K-means clustering algorithm is used to find the center of each basis function (m_j).
5. Design matrix (U) is then constructed based on given data and the gaussian radial basis functions.
6. The parameters (w) are optimised using stochastic gradient descent (SGD) with momentum.

The code for SGD with momentum training loop is shown below. Figure 1 shows the the convergence of error from using 100 gaussian basis functions (J).

```
MaxIter = 5000
lRate = 0.01
w = np.random.randn(J)
v = np.zeros_like(w)
beta = 0.9
for iter in range(MaxIter):
    j =
        np.floor(np.random.rand()*N).astype(int)
    xj = U[j,:]
    yj = y[j]
    v = (beta * v) + (1-beta) * (xj.T * (xj @
        w - yj)) # velocity
    w = w - lRate * v
```

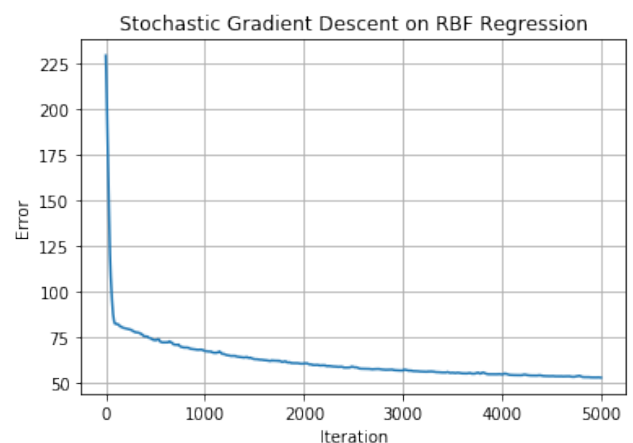


Figure 1: Plot of the loss curves

1.2 Mountain Car

For the following tasks, a gaussian RBF is used to approximate the action value function of a mountain car problem using OpenAI Gym environment. The goal of this problem is to get the under-power car to the top of the mountain. The states consisted of 2D continuous values of position (-1.2 to 0.6) and velocity (-0.07 to 0.07), and the available actions are move left (0), stay (1) and move right (2). For each time step, there is a reward (penalty) of -1 until the goal is reached. This is a control problem where the agent needs to iteratively improve its policy in order to achieve its goal.

1.2.1 RBF Approximator from Tabular Discretization Method

In this task, the mountain car is solved using Q-learning and tabular discretization method. The discretization of states is done by dividing the state space into 40 by 40 grids (velocity and position). In order to ensure that the agent can find the optimal policy, extensive exploration of the state is needed. Therefore, epsilon-greedy approach (with epsilon of 0.05) is used to select the next action. Then, Q-learning, an off-policy model-free control algorithm, is applied to iteratively improve the action value table. For this method, in the initial episode, it took more than thousands steps (achieving rewards of less than -1000s), and then gradually improve to achieve the rewards of around -200 in the 1000th episode.

The next task is to use a gaussian RBF to approximate the action value function obtain from the above method, and determine if the policy obtained from the approximation is capable of driving the car to the top of the mountain. The steps to use a gaussian RBF for function approximation are summarised as follows.

1. The 3D Q tables are converted into 2D matrix where the columns consisted of position, velocity, action, and Q value. This process makes the data applicable for training where the position, velocity and action become the features, and Q value becomes the target for regression.
2. The number of basis functions, center and width of each basis function are selected the same way as mentioned in section 1.1.
3. The optimisation is done in closed-form using Pseudo-inverse method to find the best weight (w).

Table 1 summarised the performance of an RBF model for value function approximator using different number of basis functions. As shown in the table, the policy derived from an RBF with 100 and 200 basis functions were unable to drive the car to the top of the mountain. In addition, the loss keeps decreasing with higher number of basis functions. However, the average reward achieved from RBF with 400 basis functions is less than an RBF with 300 basis functions. This could be due to the problem of overfitting where the model with too many basis functions overfit the discretized state features (the interpolation becomes less effective).

| No. of Basis Functions | Loss | Goal Achieved | Ave. Reward |
|------------------------|--------|---------------|-------------|
| 100 | 365.66 | No | None |
| 200 | 323.40 | No | None |
| 300 | 302.54 | Yes | -118.19 |
| 400 | 290.08 | Yes | -131.41 |

Table 1: Comparison of performance between different number of basis functions

1.2.2 Online Learning using Q and SARSA Update Rule

Instead of using the obtained Q table to learn the weights of an RBF, they can be learned online using SGD with SARSA or Q-learning update rule. Compare to the previous approach, we can use the raw continuous states without making them discrete, which will be beneficial for problems where the state space are large. In addition, learning online with function approximation also helps generalize the model for unseen states. The process to learn the weight of RBF, as an action value function approximator, online can be summarised as follows.

1. The state space are sampled randomly for 10000 data points in order to generate the initial data for finding the hyperparameters for Gaussian RBF, such as the center and the width.
2. The sampled data are normalized to zero mean and unit variance.
3. Using the sampled data, the number of basis functions, center and width of each basis function are selected the same way as mentioned in section 1.1.
4. The RBF models are created based on the number of action (action-out approach), which means action are not part of the features. So for a mountain car problem, there will be a total of three RBF models, one for each action.
5. The next action is selected based on epsilon-greedy policy (with epsilon = 0.05).
6. For each time step, the weights of the RBF are updated using SGD with SARSA/ Q update rule.

Figure 2 shows the reward of each episode over time using Q-learning update rule with 200 basis functions, fixed learning rate of 0.01 and a discount factor of 0.99. As shown in figure 2, the reward converges to around -130 after just 5 episode. With better tuning of hyperparameters, such as learning rate, discount factor, and epsilon, the average reward would definitely increase further. Figure 3 shows the 3D plot of the value of the state space, in which the value are calculated using $\max_a \hat{q}(s, a, w)$. As expected, the value is high on the top right of the figure, which is where the position is near the goal and the velocity is high. Another interesting point is that the value is higher on the left position compare to the value from middle position, because of the fact that the car has built more momentum to drive to the top of the mountain. The snippet of code for this section including the initialisation of RBF, SARSA update, Q-learning update and epsilon-greedy policy are shown in appendix A.1.

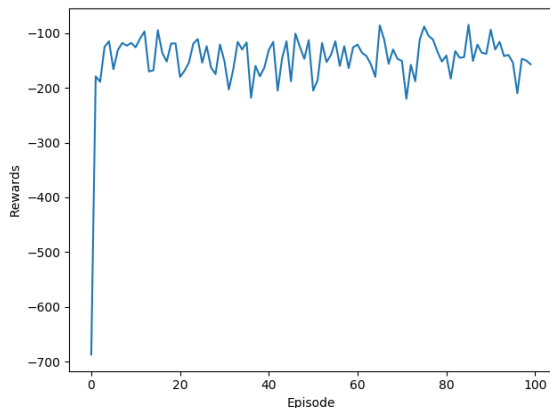


Figure 2: Episode reward over time

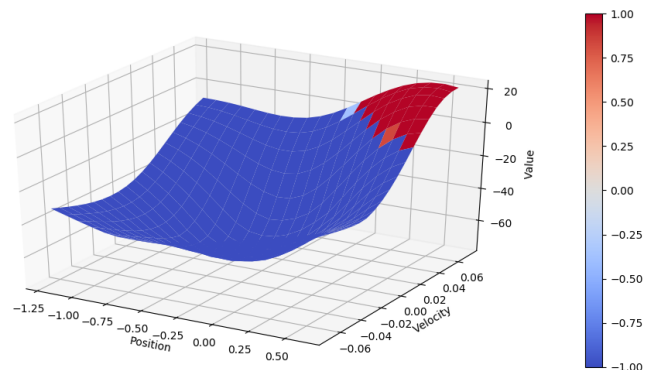


Figure 3: 3D plot of the value function after 1000 episodes

1.3 Resource Allocation Network

Resource Allocation Network (RAN) is an adaptive model that allocate a number of computational units (gaussian basis function) based on the difficulty of the problem. Compared to the previous approach, where the number, the width and the location of the basis function are fixed, RAN's dynamic nature helps alleviate the problem of underfitting or overfitting. This is because fixed-sized networks either use too few units, which make the network memorise poorly, or too many units, which make the model unable to generalize. In RAN, the allocation of new unit is done based on two conditions: the distance between new input and existing centers (δ), and the desire accuracy (ϵ). If the conditions are not met, then the weights are optimised normally using gradient descent.

To incorporate RAN for solving the mountain car problem, we first need to specify the distance (δ), and the desire accuracy (ϵ). These parameters can be chosen arbitrarily and then later improve by performing systematic parameter search, or we can select them based on statistics obtained from the RBF method, such as average distance and average error. The overall processes of using RAN with SARSA are summarised in the following pseudo code.

Algorithm 1: Pseudo-code for integrating RAN for solving a mountain car problem

```
 $\epsilon, \delta_{max}, \delta_{min} \leftarrow$  initialise desire accuracy, minimum and maximum desired distance;  
 $\delta = \delta_{max}$ ;  
for each episode do  
   $S, A \leftarrow$  initialise state and action of episode;  
  while episode is not end do  
    Take action  $A$ , observe  $R, S'$ ;  
    Choose next action  $A'$  as a function of  $\hat{q}(S, ., w)$  using  $\epsilon$ -greedy;  
    Evaluate output from current approximated Q function;  $q_{cur} = \hat{q}(S, A, w)$ ;  
    Compute error using SARSA target;  $error = R + \gamma \hat{q}(S', A', w) - q_{cur}$ ;  
    Find distance to nearest center;  $d = \min_j ||c_j - S||$ ;  
    if  $||error|| > \epsilon$  and  $d > \delta$  then  
      Allocate new unit with center equal to the location of current state;  $c_{new} = S$  with  
      width equal proportionally to the distance to nearest center;  $width = kd$   
    else  
      Perform gradient descent update;  $w \leftarrow w + \alpha(error) \nabla \hat{q}(S, A, w)$   
    end  
    if  $S'$  is terminal then  
      Go to next episode  
    end  
     $S \leftarrow S'$ ;  
     $A \leftarrow A'$ ;  
    if  $\delta > \delta_{min}$  then  
       $\delta \leftarrow \delta * \exp(-1/\Gamma)$   
    end  
  end  
end
```

Part 2: Summarising a recent scientific study

Human-level control through deep reinforcement learning

In this paper, Mnih et al. show how to combine reinforcement learning (RL) with deep learning as a function approximator in a stable manner. They developed an RL agent called deep Q-Network (DQN), which combined Q-learning with deep convolutional neural networks (CNN). Their aim is to create a general learning algorithm to solve various tasks. To demonstrate this, 49 different Atari 2600 video games were used as a platform for learning and evaluating the performance of the DQN agent. The Atari 2600 is a video game console which contains several popular arcade games, such as Pong, Breakout and Space Invaders. Due to the popularity and the challenges these games posed, they have been attractive as a testbed for studying and evaluating RL algorithms. With DQN, they show that an RL agent can achieve high performance in different kinds of game, without using any problem-specific features.

The main idea of this agent is that the CNN, which processes raw pixels of the game, are used to approximate the Q function directly without modeling the state transition (model free approach). To represent the input state of the deep CNN, raw images of the game are preprocessed by converting a 210x160 colour images into a 84x84x4 image (using the last 4 frames). The last layer of the CNN are fully connected layer which outputs the value for each action, representing the state-action value. The actions are defined according to the joystick of the game console. Depending on the game, the number of valid actions varied between 4 and 18. The reward signal are standardized across all games, in which the reward is +1 whenever the score of the game increased, -1 whenever the score decreased, and 0 otherwise. As mention in the paper, this standardized of rewards helps the model learn without adjusting the learning rate between each game. To ensure extensive exploration of state-space, the action is selected using ϵ -greedy policy, with ϵ decreasing linearly over the first million frames.

The main improvements from a DQN agent over previous methods, which make training deep reinforcement learning stable are the use of *experience replay* and *fixed Q target*. Traditionally, training deep neural network to approximate the value function is known to be difficult and often diverges due to the correlation between the sequence of training data (observation). Experience replay technique helps reduces the correlation between training examples, where the experiences are randomly sample instead of using the full sequence of frames as training data. In addition, instead of computing the gradient of the Q function with respect to itself, the fixed old weights from previous C iterations are used to compute the Q-learning target. The overall optimisation objective can be summarised in the following equation:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim D_i} [(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \quad (1)$$

where D_i is the set of experiences, θ_i is the weight of the CNN network at iteration i, and θ_i^- is the fixed weight of the duplicate network at iteration i.

As a result, the DQN agent was able to play the game at the level of human or better in 29 out of 49 games. This development is a major step forward in creating an artificial intelligent system that excel in diverse and challenging tasks without problem-specific features. However, there are still many limitations in this approach. For instance, the DQN is still not able to multi-task, as the weight are learn separately for each game. In addition, even though the agent was able to solve various kinds of games, the tasks are quite similar in which the raw pixels are taken as input, and the output is the optimal action to control the games. Despite these limitations, I believed that DQN has shown us the potential of combining RL with deep learning, and open up many possibilities for practical real-world applications, such as industrial robots, self-driving car, and power systems management.

A Snippet of code

A.1 RL agent using gaussian RBF for action value function approximator

```
# generate basis function center using K-means clustering and sampled environment states.
def init_rbf(self, env, n_basis=100):
    observation_examples = np.array([env.observation_space.sample() for x in
        range(10000)])
    scaler = StandardScaler()
    scaler.fit(observation_examples)
    scaled_examples = scaler.transform(observation_examples)
    kmeans = KMeans(n_clusters=n_basis, random_state=0).fit(scaled_examples)
    return kmeans.cluster_centers_

# get Q value from a given state and action
def q(self, state, action):
    features = self.feature_transformer.featurize_state(state)
    return np.dot(features, self.w[action])

# given state choose next action based on epsilon greedy policy
def epsilon_greedy(self, state):
    y = self.predict(state)
    if np.random.uniform(low=0, high=1) < epsilon:
        chosen_action = env.action_space.sample()
    else:
        chosen_action = np.argmax(y)
    return chosen_action

# given state, predict value for all actions
def predict(self, state):
    features = self.feature_transformer.featurize_state(state)
    return features @ self.w.T

# update the weight using SARSA update rule
def sarsa_update(self, lr, reward, gamma, state, action, next_state, next_action):
    feature = feature_transformer.featurize_state(state).flatten()
    self.w[action] += lr * (reward + gamma * self.q(next_state, next_action) -
        self.q(state, action)) * feature

# update the weight using Q-learning update rule
def q_update(self, lr, reward, gamma, state, action, next_state):
    feature = feature_transformer.featurize_state(state).flatten()
    next_q_max = np.max(self.predict(next_state))
    self.w[action] += lr * (reward + gamma * next_q_max - self.q(state, action)) * feature
```
