COMP6247 LabFour submission

Edvinas Piaulokas
ep1g20@soton.ac.uk | 32422938

May 21, 2021

1 Background

In this report, several reinforcement learning techniques will be implemented and tested on an artificial Mountain Car problem from Open AI's gym library. First, Radial Basis Functions will be looked into in more detail as they will be used extensively during model training later on. Then, the Mountain Car problem will be attempted using a set of techniques of increasing complexity. Finally, all of the approaches will be compared and some discussion provided on possible improvements to the techniques for better generalisation.

2 Radial Basis Functions

Before applying them in a reinforcement learning setting, Radial Basis Functions (RBFs) will be tested on a simple linear regression task. An example dataset on house prices in Boston was downloaded from the UCI repository using a built-in function in the sklearn library (sklearn.datasets.load_boston()). The dataset contains 506 observations, each with 12 numerical features describing various neighborhoods and a target feature - the median property value.

The design matrix X (506, 12) was then transformed onto a new space U (506, 200) using 200 Radial Basis Functions, as follows:

$$u_{ij} = exp(-||x_i - m_j||/\sigma_j) \tag{1}$$

where m_j and σ_j are means and standard deviations of the RBF functions. In this particular case, the 200 means were found by applying the KMeans algorithm on the design matrix X, while the standard deviations were all set to $\sigma = SD(X)$.

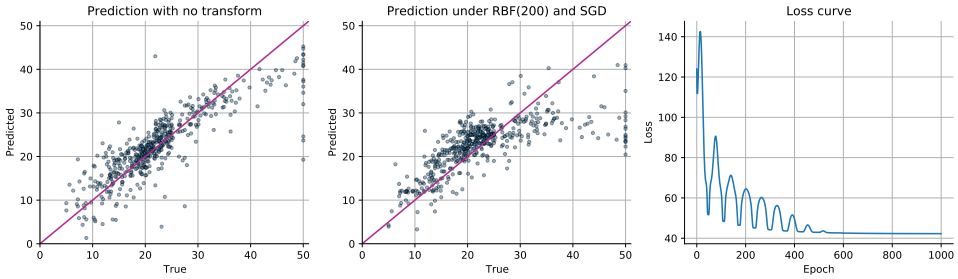

Figure 1: Closed form solution using X (left) and SGD results with RBF transform (mid, right).

With a design matrix in place, the optimal set of linear regression weights can be found in a closed form using a pseudo-inverse of the design matrix. Alternatively, these weights can be iteratively approximated using stochastic gradient descent (SGD). Both approaches were implemented, some of the results are presented in Figure 1.

A closed form solution on the original data X yielded predictions that had an MSE of 24. Using an RBF transform with 200 functions improved the model's performance by almost halving the MSE to a value of 13. However, when optimising the RBF weights with SGD, performance dropped with MSE at 42. Scheduling the learning rate to follow an oscillating and constantly decreasing sinusoidal curve only marginally improved the results. Most of the error, however, comes from the extreme values in the data.

3 Mountain Car - Tabular method & RBF approximation

The Mountain Car problem has a two-dimensional state space with position and velocity ranging $[-1.2, 0.6]$ and $[-0.07, 0.07]$ respectively. There are three actions that an agent can take at any time: drive right, left or do nothing. Initially, the agent is not aware of its environment and has equal preferences for all states and actions. The agent is then initiated under an ϵ-greedy policy where it takes a random action with a probability ϵ and otherwise takes whatever action is of the highest value for the current state. The agent then moves around the states updating the discretised state-action values at every step using the following formula:

$$Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma \max_a Q(S', a)) \tag{2}$$

where Q is the state-action quality function, α is the learning rate, γ is the discounting constant, and the letters S, A, R represent the state, action and reward respectively.

After 5000 episodes the agent has explored a significant amount of state-actions many times over. While the first episode took upwards of 30k steps to complete, the agent quite quickly improved thanks to a better understanding of the environment (Fig. 3 in red). The discretised quality function table now contains reasonable approximations of the true values. This table can now be transformed using a number of RBFs and then approximated via linear regression.

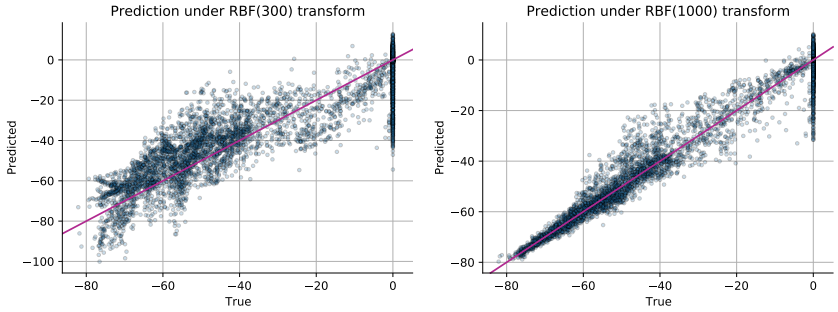

Figure 2: Prediction performance under RBF transform. Closed form solution.

RBF regression was implemented for 300, 500 and 1000 basis functions. Results from two of them in terms of True vs Predicted scatterplots are displayed in Figure 2. Quite clearly RBF(1000) performs significantly better, with the majority of the data points situated quite close to the identity line. Predictions using RBF(300) are much more scattered. In both cases, however, there is a strip of True values at 0 that fit the model poorly. That is because these points represent the states which the agent has never visited, hence the value of 0 – unchanged from initialisation.

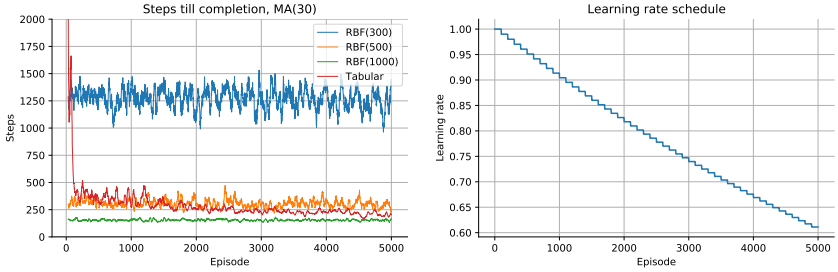

Figure 3: Moving average steps till completion for the Tabular and the RBF approximation methods.

The agent was now put back in the car for few more rounds. This time, the ϵ-greedy policy would choose the best action to take based on the value approximations acquired through RBF regression. The leftmost plot in Figure 3 displays steps taken per episode of 4 different agents. Consistently, agents using more RBFs take fewer steps to complete the task. Additionally, a higher number of RBFs seems to produce more consistent results as the data is a lot less volatile. The Tabular method, on the other hand, struggled at first, but by episode 5000 reached comparable results to the RBF(1000) agent.

4 Mountain Car - SARSA & Q-Learning

The RBF methods explored earlier were all offline. Agent first explored the environment to collect the data about the states and only then RBFs were fitted which approximated the quality table. Instead, some online learning methods such as SARSA and Q-Learning can be used which learn the weights of the RBF model iteratively, one step at a time. These models share some similarities with the Tabular approach. The agent continues to follow ϵ-greedy policy. However, this time, action-state values are not read off a table but instead approximated using RBF regression. Furthermore, as there is no quality table, learning happens through updates to the RBF weights directly.

For the SARSA algorithm, the weight update logic is shown in (3). At every step, the weight update happens according to the first line if the agent completed the task, or using the second one otherwise.

$$\begin{aligned} &\text{if done } \mathbf{w} \leftarrow \mathbf{w} + \alpha(R - Q(S, A, \mathbf{w}))\nabla Q(S, A, \mathbf{w}) \\ &\quad \text{else } \mathbf{w} \leftarrow \mathbf{w} + \alpha(R + \gamma Q(S', A', \mathbf{w}) - Q(S, A, \mathbf{w}))\nabla Q(S, A, \mathbf{w}) \end{aligned} \tag{3}$$

The only difference to the algorithm for Q-Learning is that the quality function of state S' is always maximised. More specifically, $Q(S', A', \mathbf{w})$ turns into $\max_a Q(S', a, \mathbf{w})$ regardless of what action was actually chosen under the policy.

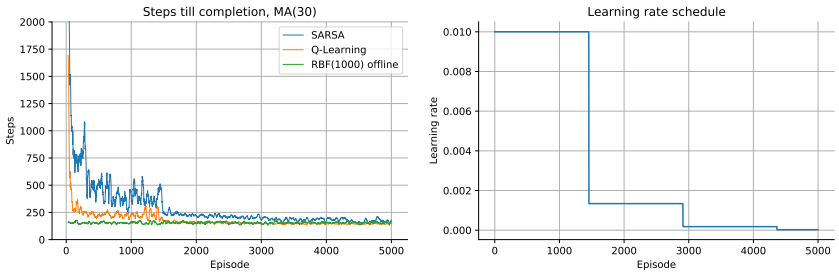

Figure 4: Moving average steps till completion for the online learning methods +offline RBF(1000).

Both online learning algorithms were run for 5000 episodes with steps taken till completion recorded. The learning rate was scheduled to follow a stepped exponential decay function. The number of RBFs was set at 1000. The Q-Learning agent started off by very quickly learning the environment as the steps till completion dropped to around 250 just 100 episodes in before stabilising around 144. SARSA agent performed much worse at first, before stabilising at 170 – a reasonably good result, though still significantly higher than of Q-Learning. For comparison, the offline RBF agent reached similar values to Q-Learning - 154 steps.

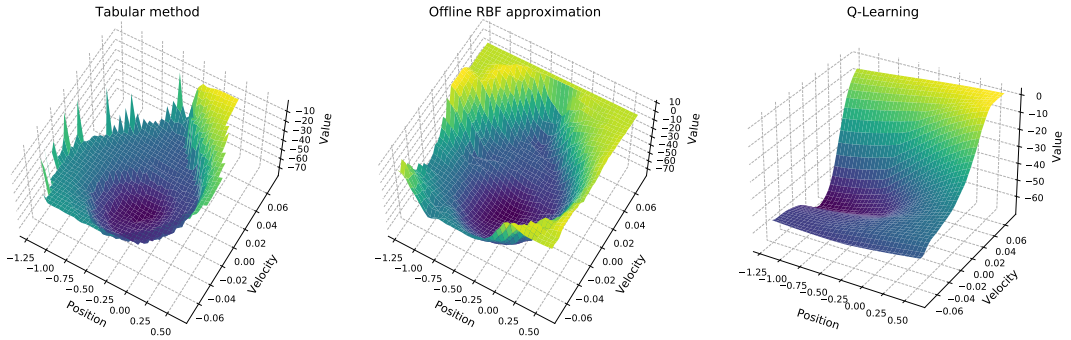

Figure 5: State quality surfaces from different approaches.

The methods applied in this report approximate state values for every possible action. These approximations can be used to create state space quality surfaces. Some of these surfaces are displayed in Figure 5. Interestingly, all of the shapes are rather distinct, yet have some key common features. For the Tabular method, there is a valley in the center with heights around 0.5 position and high velocity. Some of the state values around the borders of the surface are missing – agent never explored there. Offline RBF

approximation looks very similar, yet it has values assigned for the unvisited states. The Q-Learning quality surface is completely different though. It is very smooth, with heights around high velocity.

5 Resource Allocation Network

The RBF approximation based methods implemented so far required specifying the number of functions a priori. This is inherently problematic as any set value will either result in a suboptimal accuracy (in case of too few functions) or in a model which generalises poorly (due to overfitting because of too many functions). A better approach would be to construct an algorithm that adapts to the complexity of the data and allocates more functions on demand.

A Resource Allocation Network (Platt 1991) does exactly that. Platt's proposed solution, adapted for the Mountain Car problem, using SARSA update rules in a pseudo-code form is as follows:

Algorithm 1: Resource Allocation Network for the SARSA method

Require: Initialise $\mathbf{w}$, ϵ and $\delta_{min}, \delta_{max}$ – weights, desired accuracy and min/max values for the length scale between centers. Initialise α, γ, τ and κ – learning rate, discounting factor, scale decay constant and overlap factor.

$\delta = \delta_{max}$
for *every episode* **do**
  Initiate the agent with some S and A
  for *every step* **do**
    Take action A, observe S', R and *status*
    if *status=done* **then**
      $y = R$
    else
      Choose an action A' according to the policy
      $y = R + \gamma Q(S', A', \mathbf{w})$
    end
    $\mathbf{x} = [S, A]$
    Create $\mathbf{u}$:
      $u_j = \exp(-||\mathbf{x} - \mathbf{c}_j||/\sigma_j^2)$
    $\hat{y} = \mathbf{u}^T\mathbf{w}$
    $e = y - \hat{y}$
    $d = \min_j ||\mathbf{c}_j - \mathbf{x}||$
    if $|e| > \epsilon$ *and* $d > \delta$ **then**
      $\mathbf{c}_{new} = \mathbf{x}$
      $\sigma_{new} = \kappa d$
      $w_{new} = e$
    else
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y - Q(S, A, \mathbf{w}))\nabla Q(S, A, \mathbf{w})$
      Update all centers $\mathbf{c}_j$:
        $\mathbf{c}_j \leftarrow \mathbf{c}_j - 2\frac{\alpha}{\sigma_j}(\mathbf{x} - \mathbf{c}_j)\big[(y - \hat{y})\mathbf{w}\big]$
    end
    $\delta = \max(\delta_{min}, \delta \exp(-1/\tau))$
    if *status=done* **then**
      break
    else
      $A \leftarrow A'$
      $S \leftarrow S'$
    end
  end
end

$Q(S, A, \mathbf{w})$ and $\hat{y}$ denote exactly the same thing using different notation.

6 Real world application

6.1 Summary of a paper on RL enabled drug design system.

The key ideas behind the Reinforcement Learning methods implemented in the toy car simulation earlier are transferable to a wide array of practical problems. One such novel application is presented in a publication on Deep Reinforcement Learning for de novo drug design by Popova et al. (2018). In the paper, researchers explain how they have combined two deep neural network architectures in a reinforcement learning setting in order to generate never seen before chemical compounds with certain desired properties.

Manufacturing drugs is an expensive and resource intensive task. This is partially due to the fact that researchers often go through many iterations of trial and error testing various substances for their chemical properties until they find one that is promising. Automated approaches in creating new chemical compounds with desired properties that can be used in manufacturing drugs offer an opportunity to speed up the process. It was estimated, however, that there could be between 10^{30} - 10^{60} synthetically feasible drug-like particles (Polishchuk, 2013). A brute force approach, therefore, is out of the question. A Reinforcement Learning solution was proposed instead.

Researchers have trained two neural networks. A generator network that generates strings of characters that encode the chemical structure of a molecule, according to the simplified molecular-input line-entry system (SMILE). And a predictor network that takes the generated SMILE strings and predicts the properties of these novel compounds. Both of these networks were trained offline separately at first. The generator network had to learn to produce synthetically feasible compounds while the predictor had to learn how to accurately predict their properties. Once the pre-training was done networks were combined in a Reinforcement Learning setting. The generator network acted here as the agent and its action space was the entire alphabet of all possible characters of the SMILE strings. At every step, the agent would take an action which would be a selection of one character from the alphabet. This character would then be appended to the string of all previous actions (characters). The string itself is the state which changes (grows) with every action. All future actions are conditioned on the current state which in turn is conditioned on all past actions. In other words, actions are stochastically chosen from a conditional distribution, while states are always deterministic. Both states and actions are discrete. Such input-output interaction was possible thanks to a recurrent architecture of the network. Once a string is complete the episode is concluded. At that time the string is passed on to the predictor network (the critic) which predicts the compound's properties and allocates some reward to the agent. No reward is given for intermediate steps, only for a completed string, thus the terminal state. The size of the reward is a function of the generated compound's performance as well as the preferences over some desired characteristics. The reward function can be tweaked to favour compounds of a specific kind.

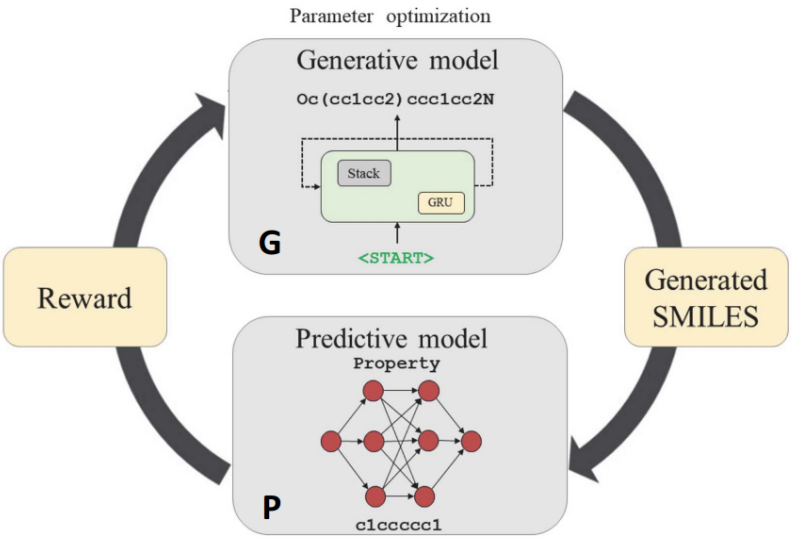

Figure 6: General pipeline of the RL system.

Researchers have tested the system in several different scenarios and found promising results. At first, they made sure that the generative network is performing as expected. The main criteria were that the outputs are actually feasible chemical compounds and that they are actually novel - substantially different from the training set. Researchers generated over 1M new compounds and found that over 95% of them were legitimate. The validity check was done using a structure checker from ChemAxon. The structural novelty of the generated compounds was confirmed by comparing the content of the Murcko scaffolds of the generated structures against the ones found in the training set. It was found that less than 10% of the scaffolds from the generated compounds were present in ChEMBL.

The system was then tested at generating compounds with specific, user-defined properties. The specific requirements were passed onto the model through a modified reward function. For example, the reward function was set such that it would reward compounds that had an octanol-water partition coefficient logP between 1-4 as values higher than 5 are outside a favourable drug-like range. Under such modification, the generator proceeded to create a library of compounds where 88% of them satisfied this constraint. A number of other structural biases such as melting point range or minimising/maximising half maximal inhibitory concentration (pIC50) were introduced and tested. In all cases, the system responded well to the modified reward function and generated compounds as per defined requirement.

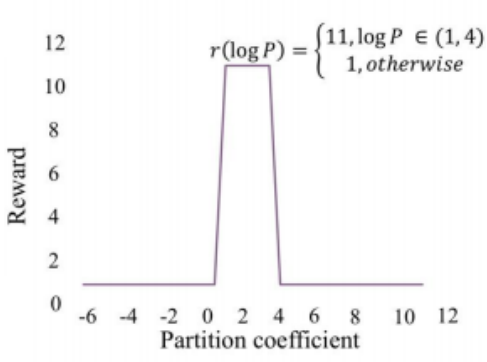

Figure 7: Reward function optimising for a specific range of $\log P$.

This paper is another great example of using Reinforcement Learning in practice. It showcases a unique application of RL techniques coupled with Deep Learning methods. The system designed and tested in the paper delivered great results by creating large amounts of novel chemical compounds. One of the key features of this system is its flexibility. It can be set up to produce compounds of a specific kind with certain desired properties. This provides researchers a tool with which they are able to explore the chemical compound space. If utilised well by drug manufacturers, it could greatly reduce the time required for compound discovery. However, as promising as it is, the system can only generate chemical compounds. This still leaves the rest of the drug manufacturing pipeline intact. Perhaps future improvements of the model could take a step further and model biological properties and effects it has on humans (and bacteria, viruses or cancerous cells for that matter) alongside the chemical ones. That said, the learning paradigm developed in this paper is rather general in nature and could very well be applied in unrelated fields to explore some other data spaces.