1 RBF Model Regression

The dataset for experimentation with RBF gradient descent is the *QSAR fish toxicity* [1] data set. It is a regression problem with 908 instances, 6 features and 1 target. First, a linear model was fit using the pseudo-inverse on the RBF features for a range of J and γ_{RBF} as an attempt to find the optimal pair (see Figure 1) in terms of lowest mean squared error (MSE). The means of the basis functions were computed using k-means clustering, and $\sigma^2 = \gamma_{RBF}^{-1}$ was the same for all basis functions. The model was always prioritising a low γ_{RBF} and a high J, which means that it was trying to create a basis function for each data point (i.e. over-fitting). A better approach would be to optimise these parameters on validation data but since the data set was already small, and the model would be later used in an on-line setting which does not provide validation data, the parameters were arbitrarily set to $J = 50$ and $\gamma_{RBF} = 0.3$ for the experiment below.

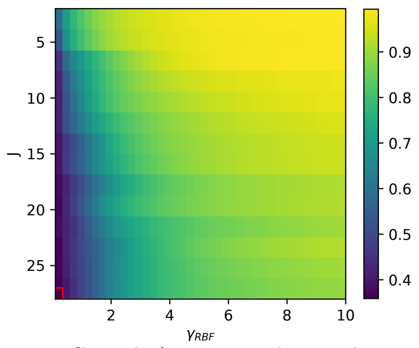

Figure 1: Search for optimal J and γ_{RBF} in the *QSAR fish toxicity* problem.

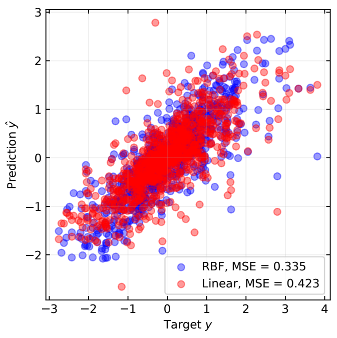

Figure 2: Predictions against targets for a linear and a RBF model.

The RBF features were then used to train a linear model with parameters $\boldsymbol{\omega}$ via stochastic gradient descent (SGD). A batch size of 1 was used since that would best represent the on-line Q-learning situation in next section. Figure 3 shows the learning curves for different learning rates, the best one of which is 0.01. The MSE of SGD comes close to the analytic solution and the similar results can also be seen in Figure 4. Figure 2 compares a linear model fit on the raw features against the RBF model and the RBF model performs slightly better, although this might be due to over-fitting.

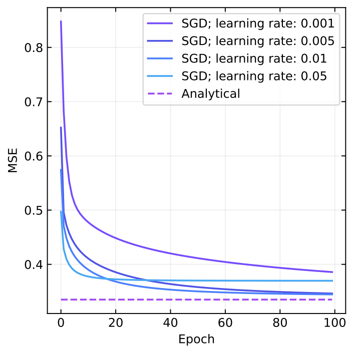

Figure 3: Training curves of the RBF model on the *QSAR fish toxicity* problem.

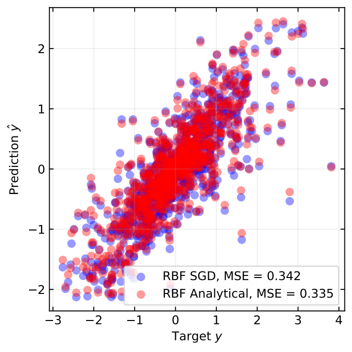

Figure 4: Predictions against targets of the analytical and SGD RBF models.

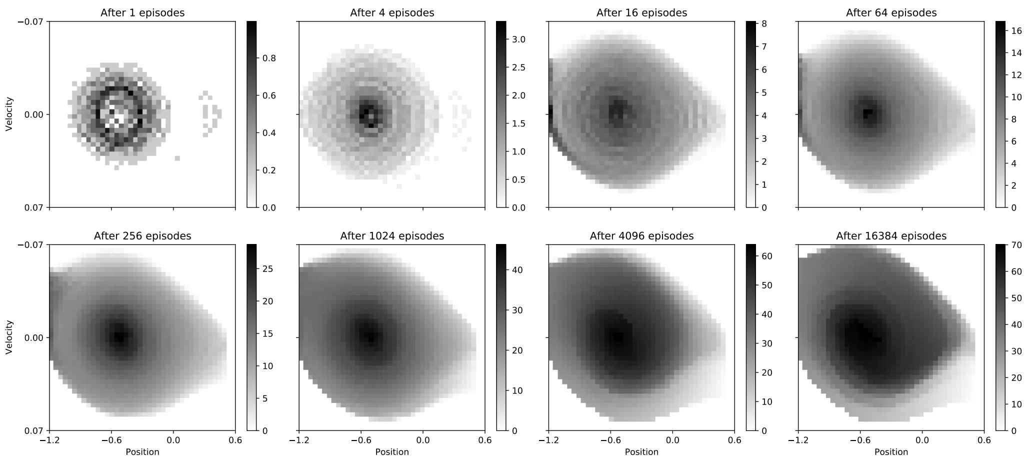

Figure 5: Evolution of the cost-to-go function from the tabular method.

2 Mountain Car Problem

First, an agent was trained using the tabular discretisation method with a Q-table of size 40×40, hyper-parameters $\alpha = 0.2$, $\gamma = 0.99$, $\epsilon = 0.1$ and a Q-learning update rule. Figure 6 shows the total reward for each episode during training, which converges at approximately -155. Figure 5 shows the evolution of the cost-to-go function $-max_a \hat{q}(s, a)$ during training. Even at episode 1, the agent discovered the rough shape of the true cost-to-go function, although the estimated values are optimistic; minor change can be see between episodes 4096 and 16385. Interestingly, the agent never visits the states where the position is greater than 0.5 (since the episode terminates) and near a velocity of 0.07 (the car never goes that fast). Hence, this model can not generalise to these boundary states.

Next an RBF model was trained off-line to fit the Q-table. Three separate sets of weights were used for each action. There are multiple ways to find the centres of the radial basis functions: k-means; the method in §3; random sampling. However, since the state space for this problem is known, the following experiments use centres from a uniform two-dimensional grid and J is equal to the square of number of centres along any dimension. To be able to use the same region of influence σ for the position and velocity, those were normalised to $[-1, 1]$. Figure 7 shows the MSE of the RBF approximation of the q-table for a range of J and γ_{RBF} values. A similar problem to §1 is that the model performs better with an ever increasing J. Thus, J was arbitrarily set to 784 (a uniform grid of size 28×28) and γ_{RBF} was set to 115 according to Figure 7.

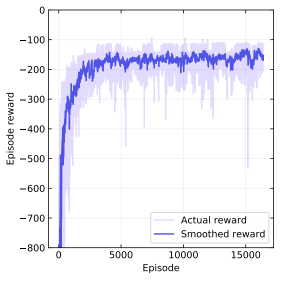

Figure 6: Total reward against episode from the tabular method.

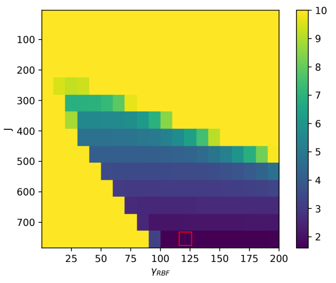

Figure 7: Search for optimal J and γ_{RBF} when fitting the q-table.

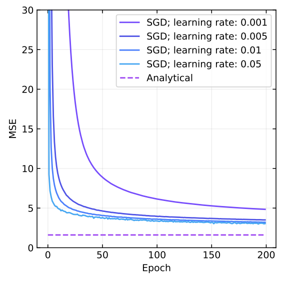

Figure 8: Training curves of the RBF model when fitting the q-table.

Figure 8 shows the training curves of the aforementioned RBF model using SGD with a batch size of 1 for a range of learning rates, the best of which is 0.05. The last column of Figure 11 shows the resulting cost-to-go function, which closely resembles that of Figure 5. Probably due to the high number of basis functions and small region of influence, there are some visible wave-like artefacts when the cost-to-go is near 0, whilst a smoother RBF approximation would be preferred. Nevertheless, like in the tabular method, the RBF approximation managed to drive the car to the top of the hill.

Once the model parameters were chosen off-line, the same model was trained on-line using a Q-learning update rule with $\alpha = 0.2$, $\gamma = 0.99$, and $\epsilon = 0.1$. Figures 9 and 10 show the total reward at each episode and the loss at each training step respectively. The function approximation method converges much quicker (approx. 200 episodes) than the tabular method (approx. 5000 episodes) and there is less variance in the reward between episodes, probably because the learnt policy is smoother.

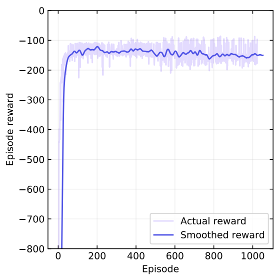

Figure 9: Total reward against episode from the RBF approximation method.

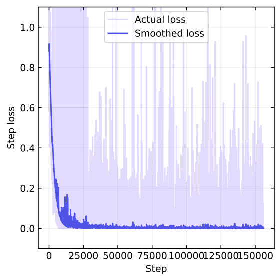

Figure 10: Training loss against step from the RBF approximation method.

The first three columns of Figure 11 show the evolution of the cost-to-go function. When compared to the tabular method and the RBF approximation of the q-table, the on-line Q-learning method produced a smoother policy with more realistic cost values. This model also managed to drive the car to the top of the hill.

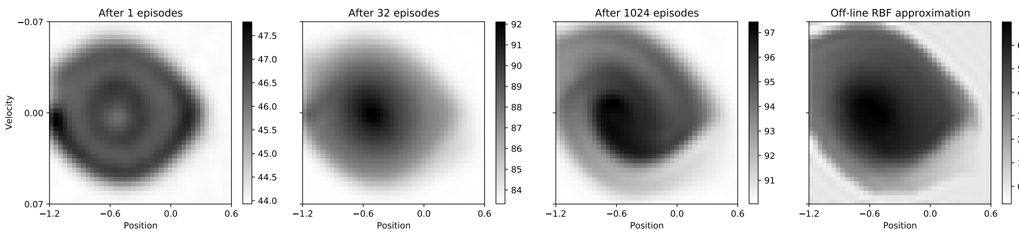

Figure 11: Evolution of the cost-to-go function from the on-line (first three columns) and off-line (last column) RBF approximation methods.

Lastly, the greedy policies of each model are investigated in Figure 12. All three policies look relatively similar. Probably due to the incomplete convergence, the tabular method produced a noisy policy. The off-line RBF approximation smoothened that policy, but generated the wave-like artefacts mentioned before.

The on-line RBF approximation generated the smoothest policy and qualitatively performed best. The three policies were tested for 100 runs and the mean reward for each of them was recorded. The tabular method finishes with an mean reward of -155.14, the off-line approximation with -119.22 and the on-line approximation with -104.59. This shows that the smooth on-line approximation better resembles the physics of the environment.

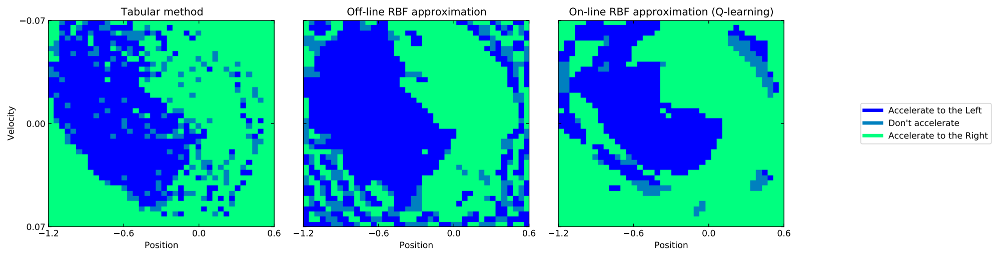

Figure 12: Greedy policies from each of the three models.

3 Resource Allocation Network and RL

So far, we have either used k-means or a two-dimensional grid to find the centres of the Gaussian units in the RBF kernel by arbitrarily setting J and σ. The experiments above show that this could lead to over-fitting, poor performance and an unnecessarily complex model. [6] suggests an ad-hoc way of adding new centres with a corresponding region of influence to cover the whole space that has been seen with the minimal amount of basis functions. The pseudo-code below combines on-line Q-learning approximation with the RAN algorithm from [6]. The function $\hat{v}(S, \boldsymbol{w}_{A'})$ is $f(\boldsymbol{x})$ from the assignment; the same notation as in the assignment is used. $U(S)$ generates RBF features based on the state S; M is a matrix of means $\boldsymbol{m}$.

```
On-line Q-learning approximation with RAN

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \mapsto \mathbb{R}$ such that $\hat{v}(terminal, \cdot) = 0$
Algorithm parameter: step size $\alpha > 0$, desired accuracy $\epsilon_{RAN} > 0$, discount factor
$\gamma \in (0, 1]$, epsilon-greedy parameter $\epsilon \in [0, 1]$, scale resolution $\delta_{min}, \delta_{max}$

Loop for each episode:
    Initialize $\mathcal{S}$
    Loop step $\tau$ until $\mathcal{S}$ is terminal:
        If $r \sim \mathcal{U}(0, 1) < \epsilon$:
            $A \leftarrow$ random sample from $\mathcal{A}$
        Else:
            $A \leftarrow \text{argmax}_{A'}\, \hat{v}(S, \boldsymbol{w}_{A'})$
        Take action $A$, observe $R$, $S'$
        Calculate target $y \leftarrow R + \gamma \max_{A'} \hat{v}(S', \boldsymbol{w}_{A'})$
        Calculate prediction $\hat{y} \leftarrow \hat{v}(S, \boldsymbol{w}_A)$
        Calculate error $e \leftarrow y - \hat{y}$
        Find distance to nearest mean $d \leftarrow \min_i \|\boldsymbol{m}_i - S'\|$
        If $\|e\| > \epsilon_{RAN}$ and $d > \delta$:
            Allocate a new unit $M \leftarrow [M, S']$ and $\boldsymbol{w}_{A'} \leftarrow [\boldsymbol{w}_{A'}, e]$ for $A' \in \mathcal{A}$
            If this is the first unit allocation:
                Allocate unit width $\boldsymbol{\sigma} \leftarrow [\boldsymbol{\sigma}, k\delta]$
            Else:
                Allocate unit width $\boldsymbol{\sigma} \leftarrow [\boldsymbol{\sigma}, kd]$
        Else, do gradient descent:
            $\boldsymbol{w}_A \leftarrow \boldsymbol{w}_A + \alpha e U(S)$
            Equation (2.12) from [6]: $\boldsymbol{m}_i \leftarrow 2\frac{\alpha}{\sigma_i}(S - \boldsymbol{m}_i)\boldsymbol{u}_i(ew_i)$ for $i \in |M|$
        Decay the scale of resolution $\delta \leftarrow max(\delta_{min}, \delta \cdot exp(-1/\tau))$
        $S \leftarrow S'$
```

4 Resource Management With Deep RL

The concrete problem that [4] tried to solve was that of online multi-resource cluster scheduling. They trained a RL agent (called DeepRM) to control a simulated computer cluster with CPU and memory resources that need to be efficiently distributed between jobs that require access to these resources. Their model assumes no malleability, which means that as soon as resources are allocated for a particular job, they remain continuously used by that job until the job is finished. Effectively, the role of the agent is of a scheduler that allocates resources for the awaiting jobs, based on the available resources and the requirements of each job. The authors highlight that similar problems are usually solved by creating a simplified model of the problem and hand-crafting heuristics. One such heuristic is a packing scheme that was inspired by Tetris [2]. Some problems with these "manual" methods are: that they require intricate fine-tuning and expert knowledge of the field of application; they are largely problem specific and might not perform well, when the environment changes slightly. Since schedulers are usually optimised for an easily quantifiable metric, such as the *average job slowdown*, and can be abstracted using an agent-environment setting, a RL framework naturally fits the problem. Furthermore, RL would offer flexibility since the same algorithm would apply to variations of the initial problem and these variations could even be learnt on-line (e.g. after an upgrade of the cluster).

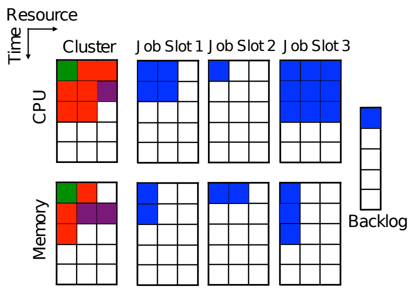

Figure 13: State representation visualisation of DeepRM. (reproduced from [4])

The **state space** is defined by a set of images (see Figure 13). It defines the current resources in the cluster that are being in use and M awaiting jobs (there is a hard limit on the number of awaiting jobs, since a constant size input is required by the policy network; $M = 20$ in the experiments). Any jobs beyond the count of M are counted in the backlog (60 jobs can be counted in the backlog), which makes the problem a partially-observed MDP. The other images in Figure 13 represent discrete time steps as their rows (of length 20), and the number of required resources as their columns. The leftmost column of images in Figure 13 identifies the resources currently in use, where one job is expressed by one colour. The other images specify the time and hardware requirements for each of the awaiting M jobs (e.g. awaiting job 1 requires 2 CPU and 1 memory for 2 time steps).

The **action space** is a vector $\{\emptyset, 1, \dots, M\}$, where the action $a = \emptyset$ means skip to the next time step, while $a = i$ for $i \in [1, M]$ means add the job at slot i to the cluster. Unlike the typical RL setting, at each time step the agent can take multiple actions (i.e. allocate resources for multiple jobs). Actions are continuously executed until either: the selected action is $a = \emptyset$; or the selected action is invalid and the job requirements do not fit in the cluster (as is the case for job slot 3 in Figure 13).

The **reward** for the agent is the negated sum of job slowdowns. This means that the objective of the agent is to minimise the *average job slowdown*. To be precise, the *average job slowdown* for a job j is $S_j = C_j/T_j$ where C_j is the time between the job request and the end of execution of that job, and T_j is the ideal time (i.e. without waiting) for the job to complete. Thus, the reward function is defined as $\sum_{j \in \mathcal{J}} -1/T_j$.

The **policy** is a neural network that takes as inputs the images of the state space and outputs a probability distribution over all the actions. The neural network has a single hidden layer with 20 units and a total of 89,451 parameters

The authors use episodic training. In each episode, a fixed number of jobs arrive and the episode finishes when all jobs finish executing. The algorithm (Figure 3 in [4]) that is used is a modification of REINFORCE, which is a *Monte Carlo Method*. One of the differences from the original REINFORCE algorithm is that there is a set of episodes for multiple examples of job arrival sequences called *jobsets*. The policy network weights are only update after the gradients have been aggregated over all episodes of all jobsets. The motivation for this is to generate a policy that generalises to different types of job arrival sequences. Furthermore, the authors modify the typical update equation of REINFORCE (equation (2) in [4]) by adding a baseline value term that is subtracted from the return values. This term is calculated for each time step t as the average return value over all episodes for a given jobset, and the motivation for its addition is to reduce the variance in the gradient estimates.

The authors measure the performance of different schedulers in terms of *average job slowdown* and *average completion time*. They compare their algorithm, DeepRM, to other hand-crafted alternatives such as *Shortest Job First*, *Packer* [2] and *Tetris** [2]. They also compare the performance of DeepRM, when trained with two different objectives: minimise the *average job slowdown*; and minimise the *average completion time*. When optimised for *average job slowdown*, DeepRM scores first in the *average job slowdown* category, but fourth in *average completion time*. When optimised for *average completion time*, DeepRM scores second in the *average job slowdown* category and first in the *average completion time* category, which makes it a viable alternative to the heuristic-based algorithms. To this date, [4] has been cited 372 times (based on `scholar.google.com`). Some successes of follow-up work using deep reinforcement learning are: adaptive bitrate generator for video [3]; TensorFlow computational graph placement on distributed CPUs and GPUs with sequence-to-sequence modelling [5]. This suggests that deep reinforcement learning could be seen more often as a solution to resource distribution and decision making. In the near future it is not unlikely to see RL agents controlling load distribution of power plants, thread schedulers in operating systems and train schedulers, to name a few.

References

[1] M Cassotti et al. “A similarity-based QSAR model for predicting acute toxicity towards the fathead minnow (Pimephales promelas)”. In: *SAR and QSAR in Environmental Research* 26.3 (2015), pp. 217–243.

[2] Robert Grandl et al. “Multi-resource packing for cluster schedulers”. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 455–466.

[3] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. “Neural adaptive video streaming with pensieve”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 197–210.

[4] Hongzi Mao et al. “Resource management with deep reinforcement learning”. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. 2016, pp. 50–56.

[5] Azalia Mirhoseini et al. “Device placement optimization with reinforcement learning”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2430–2439.

[6] John Platt. “A resource-allocating network for function interpolation”. In: *Neural computation* 3.2 (1991), pp. 213–225.