CMPUT 655 Assignment 6

Seth Akins

September 2024

1 Code

All the code for this assignment can be found here.

2 Part 1

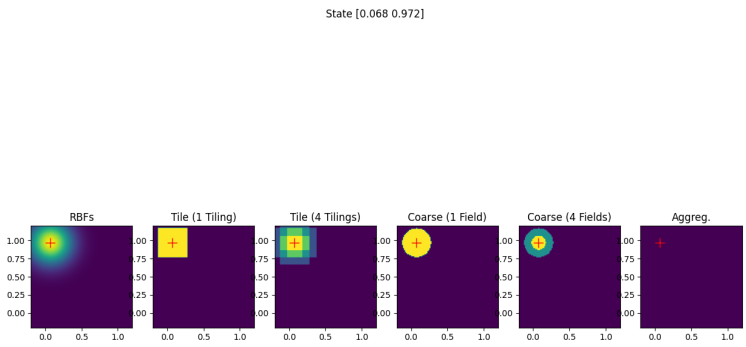

Figure 1: Feature Heat Maps

In RBFs, the hyperparameters are the centers and the sigmas. More centers means there are more features that can be approximated, whereas the sigma for each RBF controls the width of that feature. Wider RBFs will result in more generalization, whereas narrower ones will focus on a more specific area.

In tile coding, the number of centers controls the number of tiles, and width controls the width of each tile, and the offset controls the number of tilings

we have. Having more offsets means more tilings, so there is more interaction between different features that we capture. Increasing the number of tiles means that each tile captures a smaller area, so each tile is less general.

In polynomials the hyperparameter is the degree of the function. A higher function degree means that more complex functions can be modeled, as the higher degrees can work to stretch the function in more directions. This means that more features can be modeled; however, due to the continuous nature of functions, this is difficult in practice, and training becomes difficult with large degrees.

In state aggregation, the hyperparamater is the number of centers. More centers means that there are more tiles, so a larger number of tiles means that each tile is smaller, so the points in that tile are less generalized and more similar to each other; however, too wide and the points well become so averaged that they really have nothing in common.

3 Part 2

3.1 Function 1

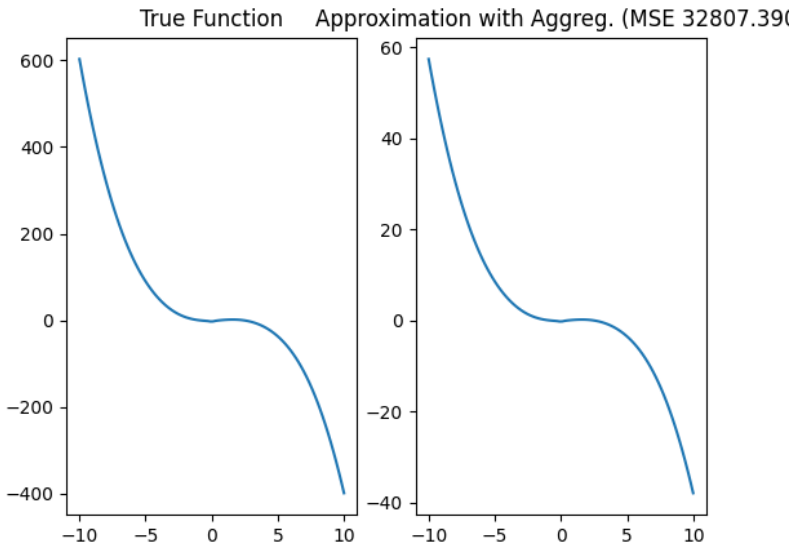

Figure 2: Function 1 Aggregation Features

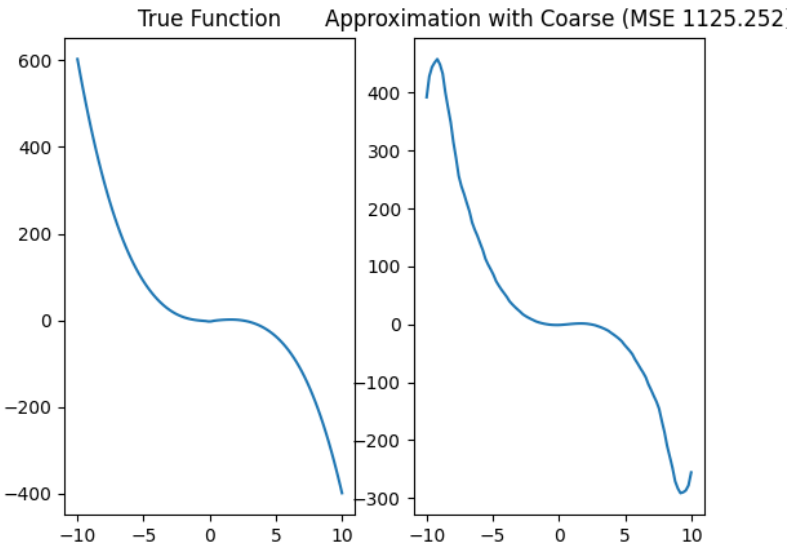

Figure 3: Function 1 Coarse Features

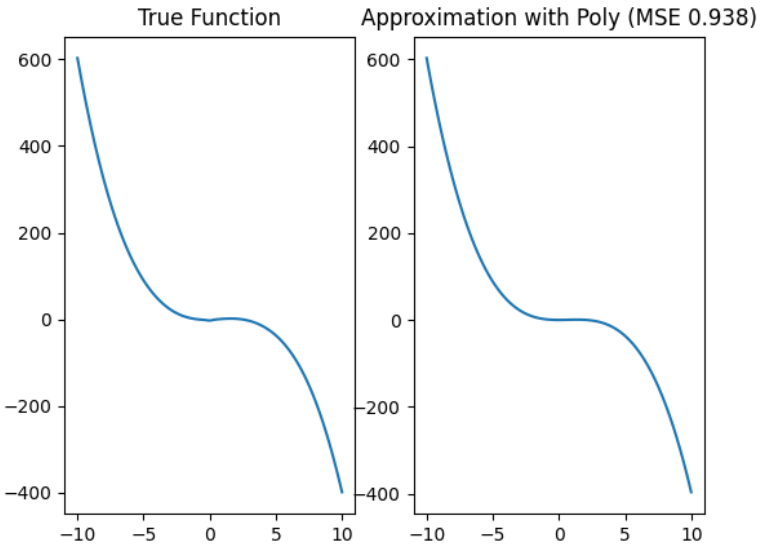

Figure 4: Function 1 Poly Features

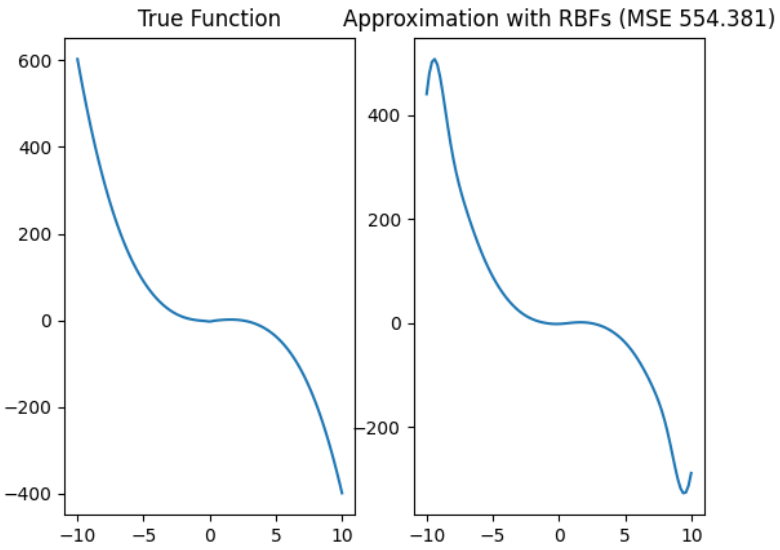

Figure 5: Function 1 RBFs Features

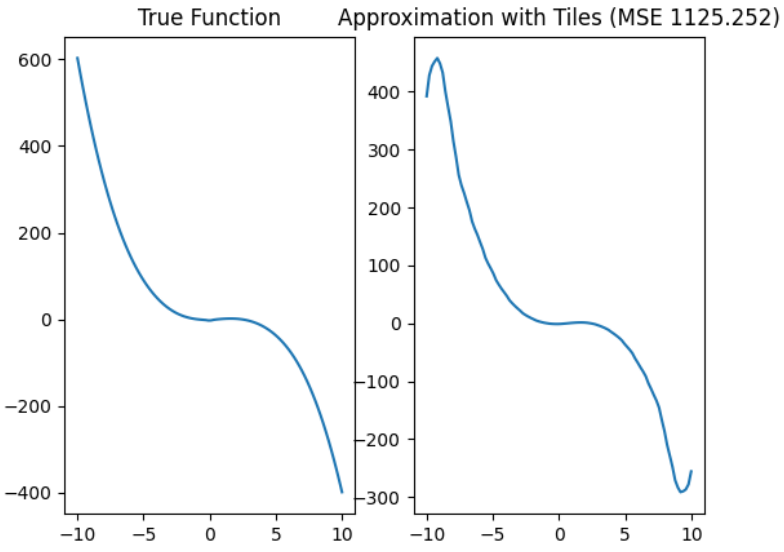

Figure 6: Function 1 Tile Features

If another dimension was added, I would increase both the widths and the number of features. In a larger space, if we kept our features the same, they would have to capture a larger area and be more general, likely making our fit worse. For the widths/sigma, it would depend on the function, if the function is quite regular, it would make sense to have wider areas, but if it is more irregular, smaller widths would likely be needed to capture the nuances of the function.

3.2 Function 2

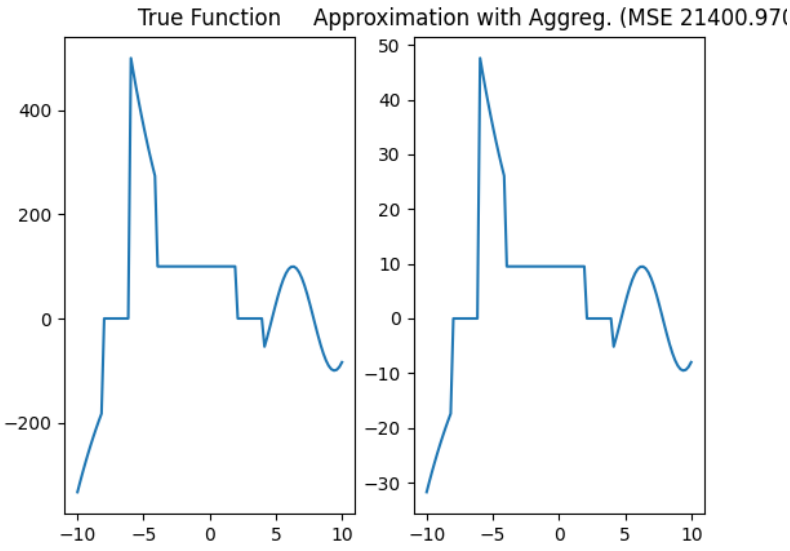

Figure 7: Function 2 Aggregation Features

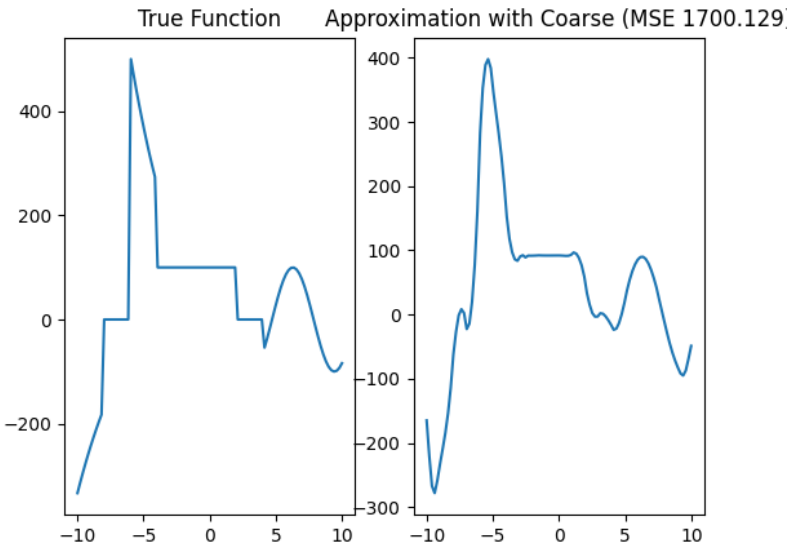

Figure 8: Function 2 Coarse Features

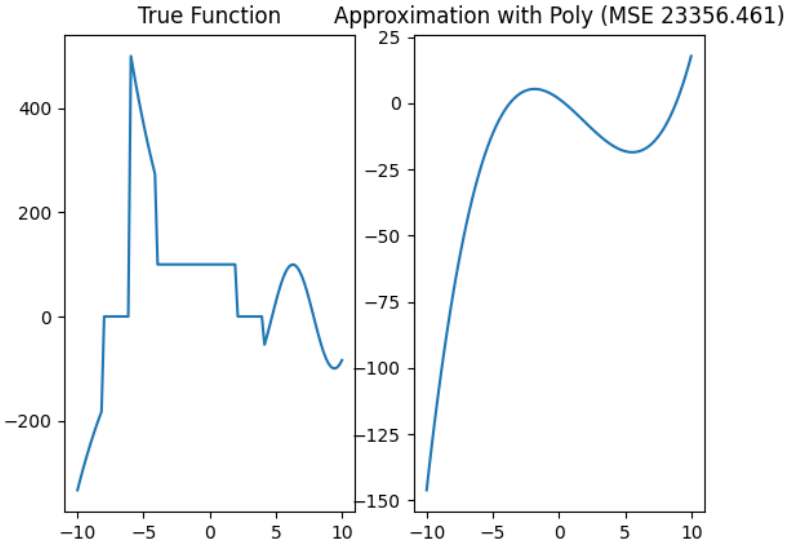

Figure 9: Function 2 Poly Features

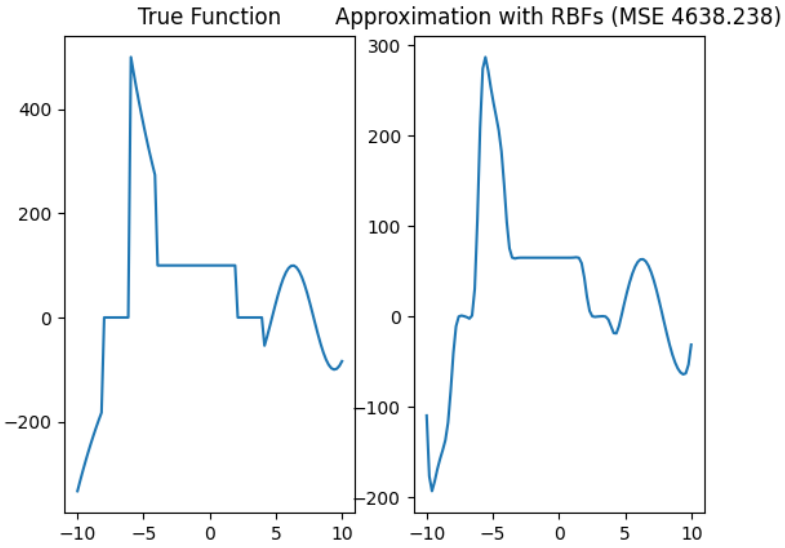

Figure 10: Function 2 RBF Features

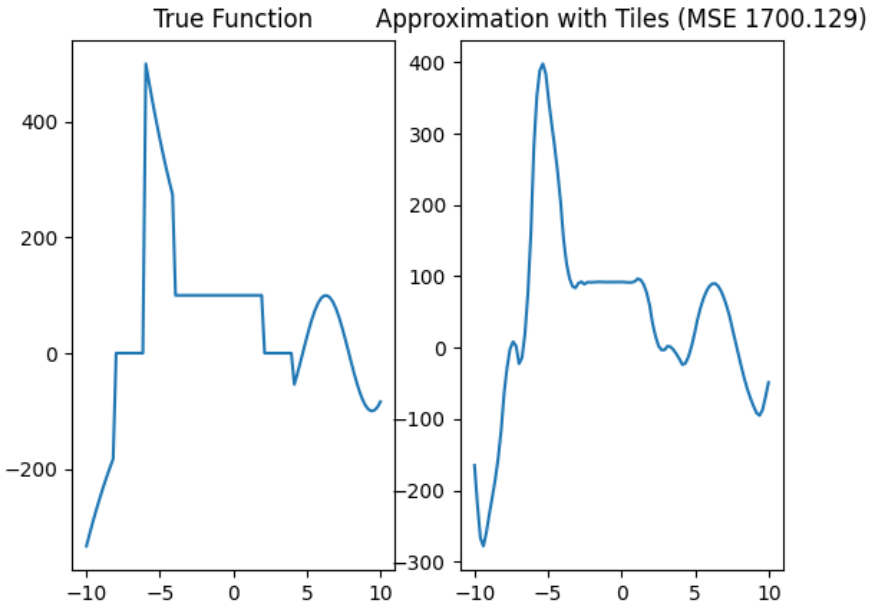

Figure 11: Function 2 Tile Features

For approximating function 2 compared to function one, I kept the number of features I had the same, but I used much narrower features. This is because the function is much less regular, so I thought that smaller features would do better.

For the number of features, if we added a second dimension, the total number of features would be equal to the number of features in the first dimension times by the number of features in the second dimension. This is because we now have all combinations of the two sets of features (all the x, y pairs in this case).

4 Part 3

For the grid world, poly features would be a bad choice. This is because the function would be quite irregular, and constrained to a very small area, so it would be difficult to learn a good continuous function to approximate the values.

Tile features would be an excellent choice for the grid world. As the grid world is tile based, each of the tiles in the approximation could approximate a tile in the grid world, and layering them with offsets could allow you to model the transitions between the tiles and create generalization between different states. The tile based nature of the problem lends itself to using tile features.

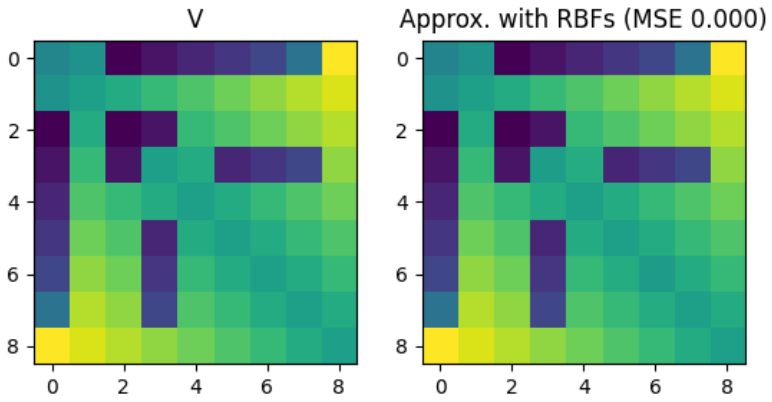

Figure 12: Learned Value Function

Coarse coding could work fine, but likely not as well as a tile based or RBF approach. Being circular instead of square in the grid based world would likely create difficulties, as several tiles would overlap within a single circle, and it could make it difficult to determine what the value of each tile should be. The features of the map itself are also not very circular, so coarse coding is not an obvious winning solution to determine our features.

Being a generalization of course coding and tile coding, RBF works well in the grid based world. Centers could be placed in each tile, as well as on tile borders, making it easy to model tile values and transitions between tiles, fostering good generalization between different states. The sigma hyperparamater can also be adjusted to make each RBF the size of around one tile. Using RBF, I managed to get my function approximation to converge essentially the ideal value function.

Aggregation would likely be a decent choice for this problem. As another tile based approach, it would likely be a good way to model features in the tile based world, and with enough tiles, it should allow us to model the values of each tile in the grid and patterns it needs to generalize between tiles.

5 Part 4

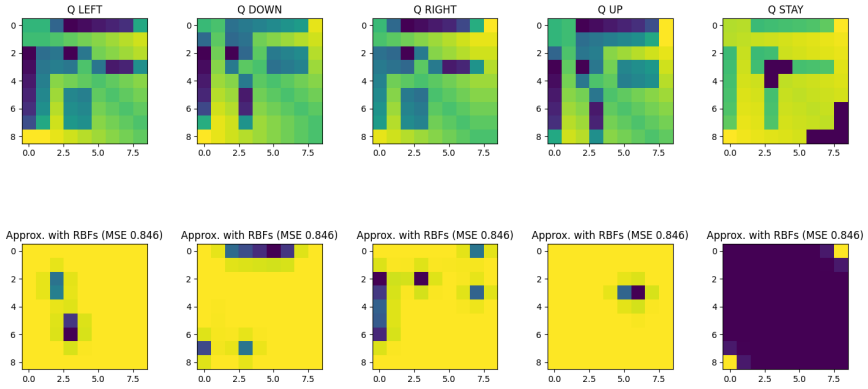

Figure 13: Q Function Heat Map

In order to learn the Q function instead of the V function, I changed my code to maintain separate weights for each of the actions. This means that inside my gradient descent loop, I also filtered to update the weights for each action only with states where that action was made.

The approximation learned was quite bad because we only have data for an agent following the optimal policy. As a result of not having data for other actions, even if we are in the goal state, take a sub optimal action, and remain in the goal state, the value of that action is still rated as very low, even though it should be high. Without samples of these other actions, it is difficult to propagate the true rewards and learn an accurate action-value function.

In addition to the features, the value of alpha is crucial in this scenario. Without a good learning rate, our action-value function estimate will be unable to move gradually in the optimal direction, and will not converge to a good function. Too large of alpha will cause our function to move too far in the direction of the gradient, overshooting the optimal weights and causing the gradient to skyrocket.

6 Part 5

Both SL regression and TD regression are trying to minimize the a version of the mean squared error, but in TD, we do not actually know what the value of y, or our true function, is. Instead, we use our current estimate of y, which is the reward we receive in the current state and the discounted return of the next state. This bootstrapping of values of one of the main differences between the two, and one of the difficulties of TD regression, as we are not always guaranteed to converge.

In order to minimize our loss between our true function y and guess $\hat{y}$, we can use gradient descent in both reinforcement learning and supervised learning. This allows us to shift the weights of our estimate function in the direction which reduces the MSE of our function. At each step of training, we take a small step in this direction to gradually minimize our loss, with the hope of being able to generalize to data which we are not training on.

As already mentioned, in RL we have the challenge of not actually knowing what our true value function is, so we do not have static targets to aim towards. Additionally, in RL we train online, so our dataset is constantly changing. Taken together, these facts mean that many traditional SL methods are not easy to apply to RL.

For TD, the difference in semi-gradient and gradient descent methods is if we also include the gradient of the value function in our TD target. With TD, the value function is used to bootstrap in the TD target, so we should include that use of value function for our next state in our calculation of the gradient for the value function; however, in semi gradient methods, we ignore this and just use the gradient of the value function for our current state. Semi gradient TD is not guaranteed to converge as well as gradient descent methods are, but often ends up learning faster, so we typically use semi gradient methods with bootstrapping instead.

If the actions were continuous instead of discrete, I would have to change my code to model actions using a set of features instead of discrete values. This would allow generalization between actions, based on samples that are trained

on in a similar way that we do it with states.