CMPUT 655
Assignment 6

September 29, 2024

Dikshant
1877098

---

1. **Function Approximation**

   • *What are the hyperparameters of each FA and how do they affect the shape of the function they can approximate?*

   - In RBFs the hyperparameter(s) is/are ... More/less ... will affect ..., while narrower/wider ... will affect ...

   - In tile/coarse coding the hyperparameter(s) is/are ...

   - In polynomials the hyperparameter(s) is/are ...

   - In state aggregation the hyperparameter(s) is/are ...

Ans:

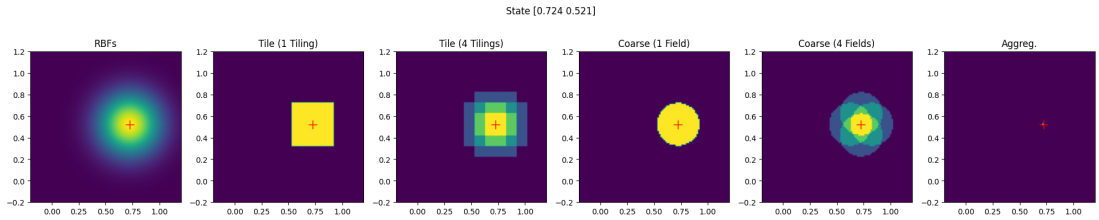


Figure 1: Function Approximation techniques

(a) **Radial Basis Functions (RBFs)**:

- *Hyperparameters*: Centers (c) and widths (σ)
- *Effect on Shape*:

• More centers can capture more localized features; fewer centers may lead to underfitting.

• Narrower widths create sharper peaks, allowing for precise approximations; wider widths result in smoother, more generalized functions.

(b) **Tile Coding / Coarse Coding**:

- *Hyperparameters*: Number of tilings and tile width
- *Effect on Shape*:

• More tilings enhance generalization but increase complexity; fewer tilings

may lose detail.
• Smaller tile widths provide finer resolution; larger widths can oversimplify the state space.

(c) **Polynomial Features**:
- *Hyperparameters*: Degree of the polynomial
- *Effect on Shape*:
• Higher degrees allow for more complex, non-linear relationships; too high can lead to overfitting.

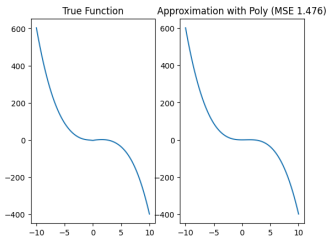
(d) **State Aggregation**:
- *Hyperparameters*: Number of states per aggregate - *Effect on Shape*:
• More states per aggregate smooths the function and reduces detail; fewer states can capture finer distinctions.

2. **Supervised Learning**

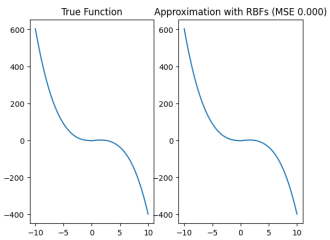
(a) With SL, (try to) train a linear approximation to fit the above function (y) using gradient descent.
• Start with all weights to 0.
• Feel free to use a better learning rate (maybe check the book for suggestions) and more iterations.
- Use all 5 FAs implemented above and submit your plots. Select the hyperparameters (degree, centers, sigmas, widths, offsets) to achieve the best results (lowest MSE).
- Assume the state is now 2-dimensional with s in [-10, 10] x [0, 1000], i.e., the 2nd dimension is in the range [0, 1000] while the 1st is still in [-10, 10].
• Also, assume that your implementation of RBFs, Tile Coding, and Coarse Coding allows to pass different widths/sigmas for every dimension of the state.
• How would you change the hyperparameters? Would you have more centers? Or wider sigmas/widths? Both? Justify your answer in one sentence.
- Note: we don't want you to achieve MSE 0. Just have a decent plot with each FA, or discuss if some FA is not suitable to fit the given function, and report your plots.

(b) Now repeat the experiment but fit the following function y.
• Submit your plots and discuss your results, paying attention to the non-smoothness of the new target function.
- How did you change your hyperparameters? Did you use more/less wider/narrower features?
- Consider the number of features. How would it change if your state would be 2-dimensional?

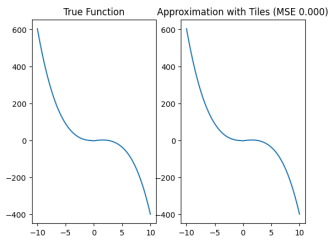
Ans:



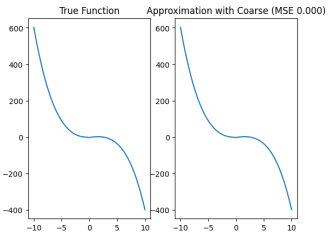
(a) Polynomial



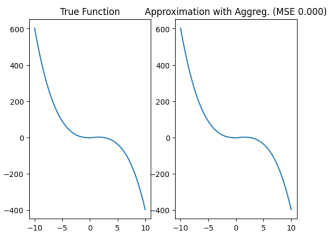
(b) RBFs



(c) Tile Coding



(d) Coarse Coding

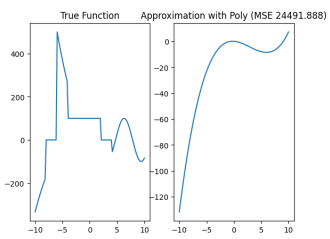


(e) Aggregation

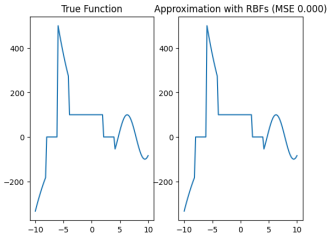
Figure 2: Supervised Learning on a smooth function

(a) **Hyperparameter changes**:

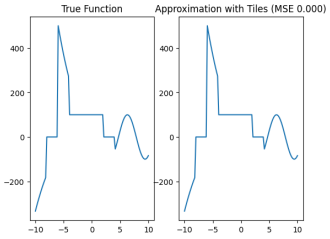
1. **RBFs**: We would increase the number of centers to adequately cover the state space and widen the sigmas to capture variations across both dimensions without overshooting the generalization.
2. **Tile Coding**: We would use more tilings to ensure coverage over the broader state space and opt for wider widths for the tiles to balance fine details and generalization.
3. **Coarse Coding**: Increasing the number of states per aggregate would smooth the approximation across the two-dimensional space while ensuring we don't lose important distinctions between states.
4. **Polynomials**: We might adjust the degree of the polynomial based on the function's complexity. A higher degree could capture intricate variations but may risk overfitting.
5. **State Aggregation**: In a 2D state space, we would consider increasing the number of aggregates to account for the additional dimension, which helps us to capture a more detailed representation of the function.



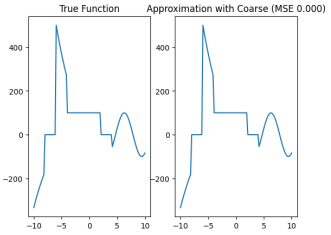
(a) Polynomial



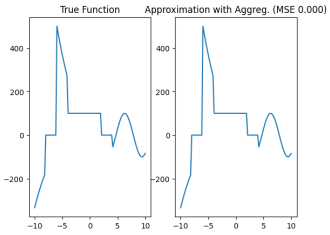
(b) RBFs



(c) Tile Coding



(d) Coarse Coding



(e) Aggregation

Figure 3: Supervised Learning on a noisy function

(b) **Non-Smooth function**:

• We can use narrower features for RBFs and Tile Coding to capture the abrupt changes in the function. For Coarse Coding, we can reduce the number of states per aggregate to allow for more sensitivity to the function's variations.

• The main adjustment involved increasing feature granularity (more centers and narrower widths) to better capture the non-smooth characteristics of the function, allowing for more accurate modeling of abrupt transitions.

• Apart from polynomial approximation, all other behaved pretty well. The polynomial approximation was unable to reduce the error due to its lower degree, a very high degree polynomial might be suitable for this task.

(c) **2-dimensional state**:

In a two-dimensional state space, the number of features is critical as their distribution greatly impacts performance. While more features can enhance approximations, there's a need to balance this against overfitting risks.

3. **Semi-gradient TD Prediction - Approximate V**

Consider the Gridworld depicted below. The dataset below contains episodes collected using the optimal policy, and the heatmap below shows its V-function.

- Consider the 5 FAs implemented above and discuss why each would be a good/bad choice. Discuss each in at most two sentences.

- The given data is a dictionary of (s, a, r, s', term, Q) arrays. Unlike the previous assignments, the state is the (x, y) coordinate of the agent on the grid.
- Run batch semi-gradient TD prediction with a FA of your choice (the one you think would work best) to learn an approximation of the V-function. Use gamma = 0.99. Increase the number of iterations, if you'd like.
- Plot your result of the true V-function against your approximation using the provided plotting function.

Ans: Approximating Value Function:

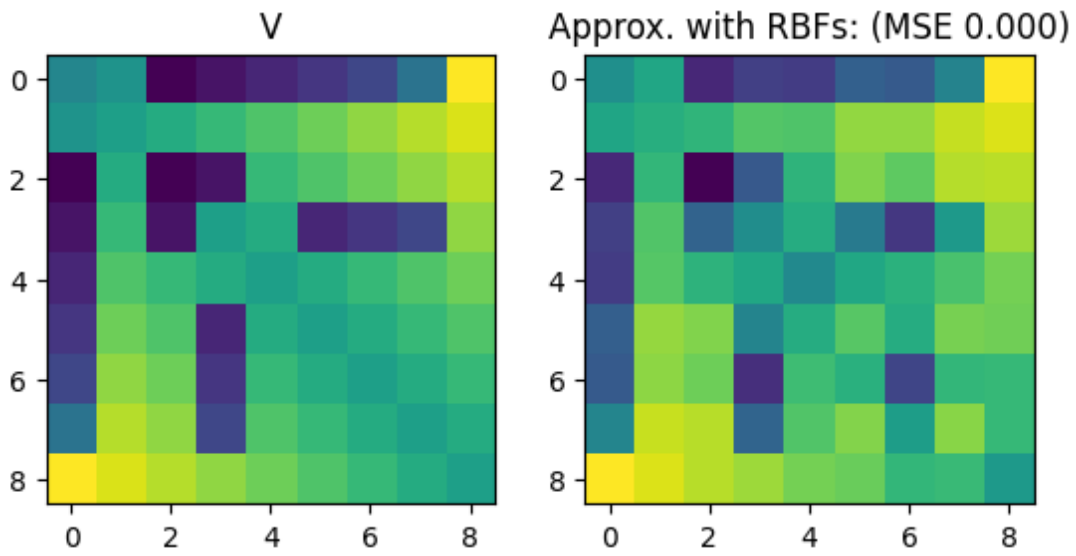


Figure 4: Value Function Approximation

- Radial Basis Functions (RBFs) are the best choice for approximating the V-function, as they can effectively capture local variations and adapt to the small blotchy patches in the value function plot.
- Tile Coding and Coarse Coding are relatively better options but struggle with the scattered bad states in the gridworld, making it challenging to generalize over fixed-size patches.
- Polynomial function approximation is the least suitable due to its inability to handle the evident breaks in the value function, leading to poor generalization.
FA Preference Order: *RBFs > Tile Coding > Coarse Coding > Aggregation > Polynomial*

4. **Semi-gradient TD Prediction - Approximate Q**

- Run TD again, but this time learn an approximation of the Q-function. How did you have to change your code?

- You'll notice that the approximation you have learned seem very wrong. Why? (hint: what is the given data missing? For example, is there any sample for LEFT/RIGHT/ UP/DOWN at goals?)

- You may still be able to learn a Q-function approximation that makes the agent act (almost) optimally. Beside your features and how they generalize, what other hyperparameter is crucial in this scenario?

Ans: Approximating State Action Value Function:

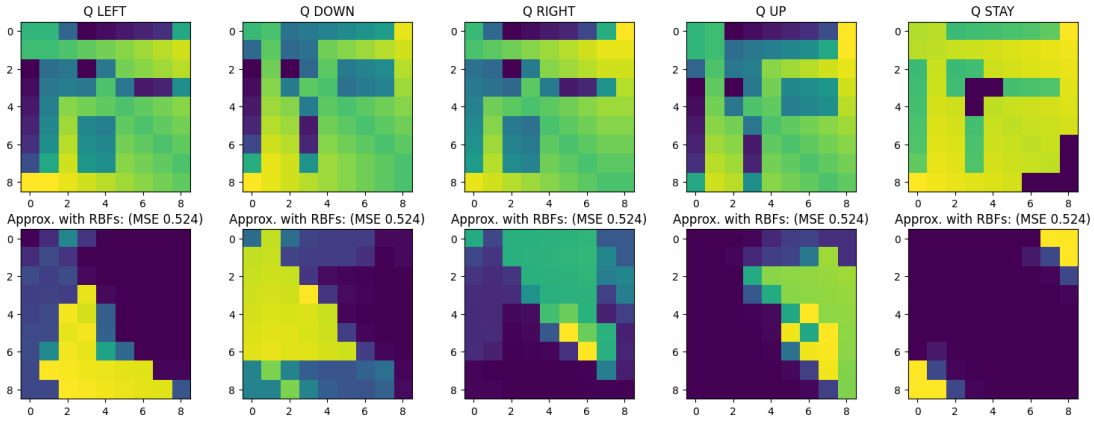


Figure 5: Value Function Approximation

- The learned Q-function approximation is often inaccurate because the dataset may lack sufficient samples for certain actions at critical states like goals. This lack of diversity in action sampling can lead to overgeneralization and an inability to accurately represent the Q-values for less frequently encountered actions, resulting in poor performance.learning.
- Also, the error between the leaned and the optimal Q-function was high because we lacked data for the transitions of non-optimal actions, which are crucial for accurate learning.
- This approach relied on the greedy Q-value of the next state for estimating the TD-target, which may not adequately capture the dynamics of non-optimal actions.
- The learning rate significantly influences convergence, and proper tuning is essential to minimize the mean squared error (MSE) of the Q-function.

5. **Discussion**

• *Discuss similarities and differences between SL regression and RL TD regression.*
- Discuss loss functions, techniques applicable to minimize it, and additional challenges of RL.
- What are the differences between "gradient descent" and "semi-gradient descent" for TD?
- Assume you'd have to learn the Q-function when actions are continuous. How would you change your code?

Ans: Discussion:
- **Similarities and Differences Between SL and RL TD Regression**:
The core objective of both is to reduce prediction errors, yet they approach this goal quite differently. SL regression often focuses on minimizing mean squared error between predicted and actual target values, while RL TD regression targets temporal difference error discounting the future states. The techniques for minimizing these errors also vary; SL typically uses gradient descent and regularization based on a static dataset, whereas RL adapts through methods like temporal difference learning and Q-learning, learning from interactions in an evolving environment. Moreover, RL presents distinct challenges, such as balancing the need to explore new actions against exploiting known rewards and managing the complexities of delayed rewards, adding layers of difficulty not encountered in traditional SL methods.
- **Gradient Descent vs. Semi-Gradient Descent for TD**: Gradient Descent updates parameters using true target values from the entire dataset, focusing on minimizing overall loss, while Semi-Gradient Descent updates are based on approximated future values, allowing for biased but practical learning from the current policy.
- **Learning Q-Function with Continuous Actions**: To learn the Q-function in continuous action spaces, we can utilize function approximators like neural networks to model Q-values, consider policy gradient methods for direct policy optimization, and adjust action selection strategies to enable exploration using techniques like softmax or Gaussian noise.

Code can be accessed on github from here: Function Approximation