INF368 – P4 – Part I

Exercise 1: Function Approximation

In class, we have seen that we can use linear function both for feature engineering ϕ(s) and for function approximation f(·; w).

Task 1*: What is the difference between using a linear function for feature engineering and function approximation?

Linear Function for Feature Engineering (ϕ(s)):

In feature engineering, the goal is to transform the raw input state (s) into a new set of features (ϕ(s)) that better represent the underlying characteristics of the problem. When using a linear function for feature engineering, each feature is a linear combination of the original input features. For example, if the original input features are [x1, x2, x3], the linearly engineered features might be [1, x1, x2, x3, x1*x2, x2*x3]. Linear feature engineering aims to capture linear relationships between the input features and the target variable. It provides a simple and interpretable way to represent the data.

Linear Function for Function Approximation (f(·; w)):

In function approximation, the goal is to approximate an unknown target function (f) that maps input features to output values. When using a linear function for function approximation, the function f(·; w) takes the form of a linear combination of the input features weighted by parameters (w). Mathematically, it can be expressed as f(x) = w1*x1 + w2*x2 + ... + wn*xn. Linear function approximation aims to model the relationship between the input features and the unknown target function using a linear function. The parameters (w) are learned from the data through techniques such as gradient descent.

Key Differences:

Purpose:

Feature engineering with linear functions focuses on transforming the input features into a new feature space where linear relationships might be better captured.

Function approximation with linear functions focuses on approximating the target function itself using a linear model.

Representation:

Linear feature engineering creates new features that are linear combinations of the original input features.

Linear function approximation directly models the target function as a linear combination of the input features.

Flexibility:

Linear feature engineering allows for the creation of non-linear relationships between the input features by including interactions and transformations.

Linear function approximation is inherently limited to linear relationships between the input features and the target variable.

Complexity:

Linear feature engineering can lead to higher-dimensional feature spaces, increasing the complexity of the model.

Linear function approximation maintains simplicity due to its linear nature, which can be advantageous in terms of interpretation and computational efficiency.

In summary, while both linear feature engineering and linear function approximation involve linear functions, they serve different purposes and operate at different stages of the machine learning pipeline. Feature engineering focuses on transforming the input features, whereas function approximation aims to model the target function itself.

Task 3 *: Consider the policy improvement theorem that we discussed in the context of dynamic programming. Why do the guarantees of this theorem fail to apply in case of function approximation? What step of the proof does not hold anymore?

The Policy Improvement Theorem in dynamic programming provides a guarantee that if a new policy has a higher value than the current policy for all states (or at least as good as the current policy for some states), then the new policy is guaranteed to be as good as or better than the current policy. This is guaranteed due to the use of the argmax here:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(a,s) \leq \max_{a \in A} q_\pi(a,s) = q_\pi(\pi'(s), s)$$

$$v_\pi(s) \leq q_\pi(\pi'(s), s)$$

Which also yields:

$$q_\pi(\pi'(s), s) \leq R + \gamma q_\pi(\pi'(s'), s')$$

However, with function approximation there is no such guarantee of this inequality. This is due to multiple factors:

Approximation Errors: The approximation errors introduced by function approximation mean that the value estimates provided by V_π (the value function) may not accurately represent the true values. Consequently, the assumption $v_\pi(s) \leq q_\pi(\pi'(s), s))$ for all states s may not hold true.

Generalization Errors: Function approximation introduces generalization errors because the approximated value function is only an estimate of the true value function. This means that the value estimates for states not explicitly visited during training may be inaccurate. Consequently, a seemingly improved policy based on the approximate value function may not actually be better in practice.

Non-Monotonic Policy Improvements: Even if π' has higher value than π in some states, it may not lead to a globally better policy due to the complexities introduced by function approximation. The nature of policy improvements isn't always constantly growing and can thus violate the assumptions made in the proof of the Policy Improvement Theorem.

Function Approximation with Bootstrapping and/or Off-policy learning: In many cases of function approximation, we use bootstrapping methods and/or off-policy learning (which leads to the deadly triad which could hinder convergence, as stated in the lectures). In such cases, the value estimates are updated based on other value estimates, introducing additional complexities and potential instabilities in the learning process. This can further deviate the learned policy from the optimal policy.

Task 4 *: Suppose you have been training the Mars Rover 2.0 robot using the tabular TD(λ) algorithm and relying on off-policy data gathered by the Mars Rover 1.0. A colleague suggests that to increase the performance of the Mars Rover 2.0 you should use a neural network to approximate its action-value function. What sort of challenge would you expect to face if you were to follow the suggestion of your colleague?

Introducing function approximation, such as using a neural network to approximate the action-value function, to off-policy learning scenarios like the one described with the Mars Rover 2.0 robot trained on data from the Mars Rover 1.0, presents several challenges. Here's how it affects the learning process:

Function Approximation Errors: Neural networks are powerful function approximators, but they introduce approximation errors. When the action-value function is approximated using a neural network, it may not perfectly represent the true action-values. This can lead to suboptimal or even divergent behavior, especially when combined with off-policy learning where the data might not be perfectly aligned with the approximation.

Generalization Challenges: Neural networks generalize patterns from the data they are trained on. When trained on off-policy data from the Mars Rover 1.0, the neural network may struggle to generalize to the different conditions and environments encountered by the Mars Rover 2.0. This can lead to poor performance or instability in learning.

Catastrophic Forgetting: Neural networks can suffer from catastrophic forgetting, where new information learned during training causes the network to forget previously learned knowledge. In the context of off-policy learning, if the data distribution between the Mars Rover 1.0 and 2.0 is significantly different, training the neural network solely on data from the 1.0 may result in forgetting important knowledge relevant to the 2.0's environment.

Data Distribution Mismatch: Off-policy learning relies on data gathered from a different policy than the one being updated. If the data distribution from the Mars Rover 1.0 is different from the policy being learned for the Mars Rover 2.0, this can lead to issues known as distribution shift or covariate shift. The neural network may struggle to learn from the off-policy data effectively, leading to degraded performance.

Exploration-Exploitation Tradeoff: Off-policy learning algorithms like TD(λ) rely on balancing exploration and exploitation. Introducing function approximation may impact this balance, as neural networks may be prone to exploit known patterns in the data rather than exploring new states and actions. This can lead to suboptimal policies and hinder the ability of the Mars Rover 2.0 to discover new strategies.

In summary, introducing function approximation, particularly using neural networks, to off-policy learning scenarios like training the Mars Rover 2.0 on data from the Mars Rover 1.0 presents significant challenges related to approximation errors, generalization, catastrophic forgetting, data distribution mismatch, and the exploration-exploitation tradeoff. This was expected considering this was already touched upon on the previous task with the deadly triad.

Exercise 2 – Planning

Task 1: What would be vπ(uni) and vπ(home) computed using MC?

$$V_\pi(uni) = \frac{1+1+0+1+1+1+1+0}{8} = \frac{6}{8} = 0.75$$

$$V_\pi(home) = 0$$

Task 2: What would be vπ(uni) and vπ(home) computed using TD?

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Where α = 1 and γ = 1.

We get the results:

$$V_\pi(uni) = 0$$

$$V_\pi(home) = 1$$

Task 3: Are the state values computed by MC and TD identical? If not, can you explain why not?

Using Monte Carlo (MC) methods, I determine state values by averaging out the rewards that follow visits to that state, waiting until the end of each episode to make the calculation. This technique depends entirely on observed outcomes, which means it has high variance due to the fluctuating nature of returns but is unbiased as it doesn't presume the value of the next state.

For instance, using MC, I've calculated *Vπ*(*uni*) as 0.75, which is the mean of the rewards obtained from being at "uni" over 8 episodes. Since there are no subsequent rewards from "home," MC doesn't provide a value for *Vπ*(*home*).

In contrast, Temporal Difference (TD) methods updates the value estimates progressively, using the rewards seen along with the projected values of future states—this is the essence of bootstrapping. This method introduces bias because it depends on existing estimates, which may not be accurate, especially early on. However, it typically has lower variance than MC because it smooths out the updates by incorporating these estimates.

Given the learning rate α=1 and the discount factor γ=1, the TD approach yields a $V\pi(uni)$ of 0. This is because the observed rewards of 0, coupled with an initial state value of 0, lead to no increment. Meanwhile, $V\pi(home)$ stays at 1, which is its initial setting, because the single transition involving "home" doesn't contribute any reward that would influence its value.

The difference between MC and TD estimations, as observed in the values for $V\pi(uni)$ and $V\pi(home)$, arises from their distinct operational frameworks: MC calculates based on real rewards and is unaffected by predictions, making it unbiased but with higher variance. TD, on the other hand, adjusts estimates incrementally and is influenced by predicted future values, which introduces bias but reduces variance, leading to smoother convergence in the learning process.

Task 4: What quantities do you need to have a full learned model ˆM of the game?

To have a fully learned model M^ of the game from the provided data, the needed quantities are:

1. **Transition Function $T(s,a,s')$:** Maps state-action pairs to their resulting next states. In deterministic environments like the example, it directly maps each state-action pair to a specific next state. For stochastic environments, it would specify probabilities for landing in different next states given s and a.
2. **Reward Function $R(s,a,s')$:** Assigns immediate rewards for transitioning from state s to s' due to action a. It maps state-action-next state triplets to numerical rewards. This could be a simple lookup for deterministic cases or capture expected rewards/distributions for stochastic settings.

Task 5: Recall the definition of the transition function T .

The transition function T is a core component of Markov Decision Processes (MDPs) describes the environment's dynamics. It defines how the environment transitions from one state to another based on the agent's actions. The nature of T varies depending on whether the environment is deterministic or stochastic:

- **Deterministic Transition Function**: In deterministic environments, the transition function $T(s,a)$ maps each state-action pair (s,a) directly to a single subsequent state s'. For every action taken by the agent in a state, the outcome is predictable and always results in the same next state: $T:S\times A\rightarrow S$

- **Stochastic Transition Function**: In stochastic environments, the outcome of an action in each state can lead to multiple possible next states, each with a specific probability. The transition function $T(s,a,s')$ then represents the probability of transitioning to state s' when action a is taken in state s. The probabilities for all possible next states from any state-action pair sum to 1: $T:S\times A\times S\rightarrow[0,1]$ $\sum s'\in S$ $T(s,a,s')=1$

The transition function is essential for understanding and modeling the behavior of the environment, allowing agents to anticipate the results of their actions and plan their strategies to maximize rewards over time.

Task 6: Infer T from the trajectories using a table

To infer the transition function T from the provided trajectories, we'll construct a table representing the probabilities of transitioning from one state to another given an action.

The given trajectories are as follows:

- T1 to T7: Actions taken in the state "uni" lead directly to a terminal state (not explicitly named, so we'll call it "end") with the action "take exam".
- T8: Starts in "home", takes the action "travel" to reach "uni", then "take exam" to reach the terminal state "end".

It is a deterministic environment so we can directly map state-action pairs to their outcomes without needing to calculate probabilities. However, for completeness, we'll note that the probabilities are 1 (since each action deterministically leads to a specific next state).

Here's the inferred transition function T represented in a table format:

Current state	Action	Next state	Probability
Home	Travel	Uni	1
uni	Take exam	End	1

Task 7: Recall the definition of the reward function R.

The reward function R in the context of Markov Decision Processes (MDPs) quantifies the immediate reward an agent receives after transitioning from one state to another due to its action. It's a critical component that guides the agent's learning by providing feedback on the consequences of its actions.

Deterministic Reward Function, For deterministic environments, R(s, a, s') assigns a specific numerical reward to each transition from state s to state s' as a result of action a. The function directly maps a state-action-next state triplet to a reward value, reflecting the immediate benefit or cost of the action: R: S *A *S -> R

Stochastic Reward Function, In stochastic environments, the reward for a given state-action-next state triplet may vary. The reward function might define expected rewards or a distribution of possible rewards for each transition, accommodating the variability in outcomes.

The reward function essentially evaluates the desirability of transitions, encouraging the agent to seek actions leading to higher rewards. It plays a pivotal role in shaping the agent's policy—its strategy for selecting actions to maximize cumulative rewards over time.

Task 8: Infer R from the trajectories using a table.

To infer the reward function R from the provided trajectories, we construct a table that maps each state-action-next state triplet to the observed reward. The provided trajectories indicate the rewards received for specific actions in given states:

- The rewards are an average value for the ('uni', 'take_exam') state-action pair and a reward for the ('home', 'travel') state-action pair.

Here's the inferred reward function R represented in a table format:

Current state	Action	Reward
Uni	Take Exam	0.75
Home	Travel	0

Task 9: How does the reward function R differ from the value function vπ(·).

The reward function R and the value function $V\pi(.)$ each serves a distinct role in understanding and navigating an environment:

Reward Function R:

- R maps state-action-next state triplets to numerical rewards, specifying the immediate reward for transitions due to actions.
- It evaluates the immediate outcome of an action, reflecting short-term gains or losses, and varies based on deterministic or stochastic settings.

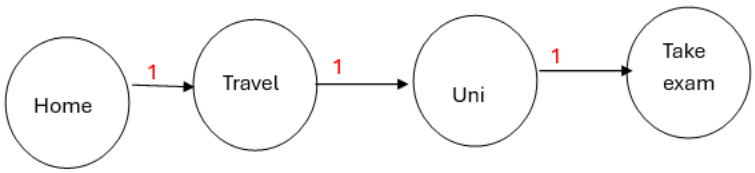
Value Function Vπ(.):

- Vπ(s) calculates the expected long-term return from state s under policy π, including all future rewards.
- It aids in choosing paths that maximize cumulative rewards, adjusting for policy and discounting future rewards to reflect their present value.

Key Differences:

- Scope: R focuses on immediate rewards for specific transitions, while $V\pi$ evaluates the expected total reward from a state over time, following a particular policy.
- Dependence: R is a property of the environment and is independent of the agent's policy, whereas $V\pi$ is inherently tied to the chosen policy, reflecting its performance.
- Calculation: R is determined by the environment's dynamics and is known a priori or learned through interaction. $V\pi$, on the other hand, is computed based on R and the transition probabilities, requiring algorithms like Monte Carlo methods or Temporal Difference learning to estimate.

Task 10: Draw the graph for the MDP ˆM you have learned



Task 11: Using the model ˆM that you have inferred, what would be vπ(uni) and vπ(home) computed using DP?

Dynamic programming can be used to compute $V_\pi(uni)$ and $V_\pi(home)$. For this MDP M^, I am using the Bellman equation for policy evaluation to iteratively calculate the value of each state.

The Bellman equation for policy evaluation is given by:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma v_\pi(s')]$$

For $V_\pi(uni)$:

$$V_\pi(uni) = \frac{1+1+0+1+1+1+1+0}{8} = \frac{6}{8} = 0.75$$

$$V_\pi(uni) = 0.75 + 1 * V_\pi(uni)$$

Since the value from “uni” to “uni” would not change because the only transition from uni leads back to uni, $V_\pi(uni)$ would converge to the expected reward of the "take_exam" action, which is 0.75.

$V_\pi(home)$:

For "home", given that it always transitions to "uni" with a reward of 0 and we take the updated value of "uni":

$V_\pi(home) = 0 + 1 * V_\pi(uni)$ = 0.75

The state values for "uni" and "home" computed using DP with the given model:

$V_\pi(uni) = 0.75$

$V_\pi(home)$= 0.75

Task 12 (*): How do the state-values you have learned with DP compare to the ones learned by MC and TD? Are they identical or not? If not, can you explain why not?

The state values learned with Dynamic Programming (DP), Monte Carlo (MC), and Temporal Difference (TD) can differ due to the nature of each method:

- **Dynamic Programming (DP)**: The value of $V_\pi(uni)$ and $V_\pi(home)$ calculated using DP, given the information from the trajectories, and assuming $\gamma = 1$, is 0.75 for both states. This value is obtained from the expected reward for the "take_exam" action at "uni", and the fact that "home" leads directly to "uni" with a reward of 0.
- **Monte Carlo (MC)**: Estimates the value of states by averaging the returns (total accumulated rewards) from episodes. The MC estimate for $V_\pi(uni)$ was given as 0.75, which matches the DP value in this case because the average reward for "take_exam" matches the expected reward used in DP. In addition, $V_\pi(home)$ was 0.
- **Temporal Difference (TD)**: The TD method led to a value of $V_\pi(uni)$=0 and $V_\pi(home)$ =1 when using α=1 and γ=1, which is different from the values obtained through DP and MC.

The values are not identical across methods for several reasons:

- **Initialization**: TD and MC can be sensitive to the initial value estimates, whereas DP assumes knowledge of the true model and is not sensitive to initial values.
- **Sampling vs. Full Knowledge**: MC and TD rely on sampled data from episodes, which can introduce variance, while DP uses the full model to compute values, which is more stable but requires complete knowledge of the environment.
- **Update Mechanism**: MC waits until the end of an episode to update values, TD updates values at each step, and DP iteratively applies the Bellman equations to convergence.

Task 13: Compute again vπ(uni) and vπ(home) using MC from real and simulated data

MC Estimation for $V_\pi(uni)$:

$$V_\pi(uni) = \frac{1+1+0+1+1+1+1+1+1+1+1+0+1+1+1+0}{16} = 0.8125$$

MC Estimation for Vπ(home):

$$V_\pi(home) = \frac{0+0+1+0}{4} = 0.25$$

Task 14: Compute again vπ(uni) and vπ(home) using TD from real and simulated data

TD Estimation for $V_\pi(uni)$:

$$V_\pi(uni) = \frac{1+1+0+1+1+1+1+0+1+1+1+1+0+1+1+0}{16} = 0.75$$

TD Estimation for Vπ(home):

$$V_\pi(home) = \frac{0+1+1+0}{4} = 0.50$$

Task 15 *: Did the values computed by MC and TD change? If so, how did they change and why?The values computed by Monte Carlo (MC) and Temporal Difference (TD) learning methods did change with the addition of new simulated data.

Monte Carlo (MC) Method

- MC methods estimate the value function by averaging the returns following visits to a state over complete episodes. When I integrated simulated data, which may have different reward structures or state transitions compared to real data, the average returns for certain states changed. This is because MC's estimations are directly influenced by the actual outcomes observed in the episodes, and adding simulated data introduces new outcomes that weren't present in the real data alone.
- The MC method is subject to high variance in its estimates, especially when the number of episodes is limited. By adding simulated data, I effectively increased the sample size but also introduced variance from the simulated environment, which could differ in dynamics from the real environment. This variance affects the precision of the value estimates.

Temporal Difference (TD) Learning

- TD methods update value estimates based on existing estimates and the observed rewards from single steps, rather than waiting for an episode to conclude. This process, known as bootstrapping, allows TD to adjust to new data more quickly.

When I combined real and simulated data, the TD method immediately began incorporating the simulated transitions and rewards into its updates. If the simulated environment did not perfectly match the real environment, these updates could reflect inaccuracies, causing the TD estimates to diverge from those based on real data alone.

- TD methods introduce a bias due to bootstrapping, where the current estimates influence future estimates. This bias can lead to faster convergence but can also make the method sensitive to the quality of the simulated data. If the simulated data introduces systematic errors in transitions or rewards, TD's value estimates can be biased toward these inaccuracies.

The observed changes in value computations when integrating real and simulated data highlight a key RL principle: the accuracy of value function estimates is deeply influenced by the data's fidelity to the actual environment dynamics. Simulated data, while invaluable for enriching the learning experience and increasing sample diversity, must accurately reflect the real-world dynamics to avoid misleading the learning algorithms.

Moreover, this experience underscores the importance of balancing the exploration of new states and actions (potentially facilitated by simulated data) with the exploitation of known rewarding paths. Both MC and TD methods must navigate this balance, and the inclusion of simulated data directly impacts their ability to do so effectively, depending on the fidelity of that data to real-world dynamics

Exercise 3 – Monte Carlo Tree Search

Task 1 – (*) Suppose you are training an agent to drive a self-driving car. A colleague, who has recently read about Monte Carlo Tree Search, suggests that the implementation of MCTS in your problem could improve the quality of the driving. What do you think of this suggestion? What considerations would you need to make to decide whether the use of MCTS is justified or not?

Monte Carlo Tree Search (MCTS) Explained:

MCTS is a heuristic search algorithm that combines tree search with random sampling. It simulates multiple trajectories (rollouts) of possible actions from the current state of the

environment to estimate the value of each action. Here's a simplified explanation of the steps involved:

1. **Selection**: Starting from the root node (current state), it selects nodes in the tree according to a selection policy, usually based on some exploration-exploitation trade-off like the Upper Confidence Bound (UCB) algorithm.
2. **Expansion**: Once a leaf node is reached (i.e., a node representing a state that hasn't been explored fully), the algorithm expands the tree by adding child nodes corresponding to possible actions from that state.
3. **Simulation/Rollout**: From each newly added node, it performs a rollout simulation. This involves playing out a random or heuristic policy until a terminal state is reached, gathering rewards along the way.
4. **Backpropagation**: After reaching a terminal state in the simulation, the results (rewards) are backpropagated up the tree to update the value estimates for each action and state visited.
5. **Selection of Action**: Finally, after a certain number of iterations or computational budget, MCTS selects the action with the highest estimated value according to some criteria (e.g., highest average reward).

Applying MCTS to Self-Driving Cars:

Advantages:

1. **Exploration**: MCTS inherently explores the action space by simulating various possible trajectories, which can be beneficial in complex environments where the optimal action may not be immediately apparent.
2. **Adaptability**: It can adapt to different driving scenarios and learn from experience, potentially improving over time as it accumulates more data and refines its estimates.
3. **Uncertain Environments**: In environments with uncertainty (e.g., unpredictable traffic patterns, changing road conditions), MCTS can provide robust decision-making by considering a range of possibilities.

Considerations:

1. **Computational Complexity**: MCTS can be computationally intensive, especially in real-time applications like self-driving cars where decisions need to be made quickly. Implementing it efficiently is crucial to ensure real-time performance.

2. **Modeling the Environment**: The effectiveness of MCTS depends on how well the environment can be modeled and how accurately rewards can be estimated. In a complex and dynamic environment like driving, modeling errors can significantly impact performance.
3. **Alternative Approaches**: It's essential to compare the potential benefits of using MCTS against other reinforcement learning techniques or traditional control methods. Depending on the specific requirements and constraints of the self-driving system, simpler algorithms might be more suitable.

In conclusion, while MCTS offers promising advantages for decision-making in uncertain environments like self-driving cars, its practical feasibility and effectiveness depend on various factors such as computational resources, environment modeling, and comparison with alternative methods. Conducting experiments and simulations to evaluate its performance in the specific context of self-driving could help determine whether its implementation is justified.