ROB 534: Sequential Decision Making in Robotics, Winter 2025
HW #2: Optimization, Deep Learning, and Informative Path Planning
Due: 2/18/25 (midnight)

Questions

1. Performance Guarantees (20 pts)

a. You are given the problem of placing a limited number of sensors (up to K) to maximize a submodular objective function [1]. **What is the performance guarantee relative to optimal for the greedy solution (i.e., selecting the best location for sensor one, placing the sensor, selecting the best location for the second sensor given the first is already placed, etc.)? Explain what this means in your own words (1-2 sentences).**

b. Assume you must generate a trajectory (with path constraints) to maximize a submodular objective function [2]. The vehicle can only take K samples along the trajectory. You devise the following algorithm: greedily select the locations the maximize F(P)/C(P) and connect them using an approximation to TSP. Assume that the optimal solution maximizes the total quantity F(P)/C(P), where F(P) is the submodular function evaluation for path P, and C(P) is the time cost of executing trajectory P. **What, if any, is the performance guarantee for this algorithm? Can you give an example where this approach performs poorly?**

Programming Assignment (Do not start at last minute!) (60 points)

The zip file "Homework2.zip" contains helper functions and data for this homework. Download it in addition to this word document. **You should use Python for this programming assignment to interface with Pytorch.** The 'readme' file in the zip package provides more information on installing and running the code.

Your robot starts at (0,0) and must navigate to a specific corner of the map. The goal depends on which world you occupy. The worlds are constructed from the MNIST dataset, a hand drawn number dataset consisting of numbers 0-9. Each world is a 28x28 grid.
If the MNIST digit is:

- 0-2: the goal is at (0,27)
- 3-5: the goal is at (27,27)
- 6-9: the goal is at (27,0)

The robot can travel one space at a time and can observe the value of the spaces it has visited and spaces one step away. It costs -1 reward to move and -400 reward if you travel to the wrong goal. Reaching the correct goal provides +100 reward, at which point the mission ends. You will design algorithms to minimize your cost to reach the correct goal. We have provided two trained neural networks to help. One network provides an estimate of what the world looks like given what you have currently observed. The other network takes this estimate and provides an estimate of what digit the world belongs to.

Step 1 (30 points): This assignment deals with a discrete version of the problem. The robot can only select waypoints that are on the grid (e.g., (1.5,1.5) is not a valid waypoint). Keep in mind that subsequent waypoints should always be distance 1 or 0 (if remaining stationary) from each other.

i. Implement a greedy solver for the discrete problem that can only move in the 4 cardinal directions (N,W,S,E). The robot should look one step ahead and move to the location that gains the maximal information based on the neural network prediction (you should determine how to calculate the maximal information). Hint: consider the values of the pixels in the prediction image.

The robot must (at a point of your choosing) stop greedily gathering information and move to a corner to terminate the mission (if correct) or incur the appropriate penalty (if incorrect). If an incorrect corner is chosen, the robot should move to another corner (or continue gathering information) until it reaches the correct corner. Hint: consider the softmax output values from the network as part of this decision.

How did you calculate the information quality from the neural network prediction? How do you decide when to move to a corner and stop moving around greedily? Show two example trajectories on different digits. Provide your average reward over 10 trials using your greedy algorithm.

Step 2 (30 points):

i. Develop and implement your own algorithm with the goal of outperforming the greedy solver. You should use a more sophisticated informative path planning technique (e.g. sampling-based solver, linear programming, or branch and bound). **Describe your algorithm formally. Report your average reward over 10 trials (1 trial per digit). Also report the average time it took to find this solution. You should terminate your algorithm after 15 minutes of run time (i.e. solutions that take more than 15 minutes to find are considered invalid).**

Discussion (20 points): Address the following questions.
1) When designing your algorithm in Step 2, what design criteria did you use to determine what to implement? Compare and contrast how your algorithm meets these design criteria relative to other algorithms taught in this course (use at least two additional examples from the course lectures or reading group discussions).
2) The neural network provides you with a prediction of the correct digit, but not an uncertainty on that prediction. If you had access to an uncertainty estimate, how might you use this to modify your algorithm from Step 2(ii)? Describe qualitatively in words; you do not need to specify the new algorithm formally.
3) Consider the case where you have multiple robots communicating with each other and performing cooperative information gathering in this scenario. What additional design considerations would there be in this case? Describe qualitatively how you would modify your algorithm to coordinate multiple robots to efficiently gather information.

References

[1] Andreas Krause, Carlos Guestrin, "Submodularity and its Applications in Optimized Information Gathering," In ACM Transactions on Intelligent Systems and Technology, vol. 2, no. 4, 2011.

[2] Singh, Amarjeet, et al. “Efficient planning of informative paths for multiple robots,” IJCAI, vol. 7, 2007.

ROB 534 SDM HW2 - Chelse VanAtter

Questions:

1. Performance Guarantees (20 pts)
   a. Performance guarantee relative to optimal for greedy solution
      The performance guarantee relative to optimal for the greedy solution is at least (1 - 1/e) which is approximately 63% of the optimal solution. This means that even though it selects sensor placement one at a time without considering the placement of future sensors (makes locally optimal decisions), this approach is still effective and near-optimal with a provable worst-case bound.

   b. Performance guarantee for this algorithm
      Using greedy selection for sampling points and selecting waypoints for maximizing F(P)/C(P) ensures at least a (1 - 1/e) performance guarantee. However, implementing the TSP approximation for trajectory construction does not preserve this guarantee and may degrade performance, especially if the TSP heuristic is suboptimal so there is no performance guarantee as it depends on the effectiveness of the TSP solution.

      Example of poor performance:
      If the highest F(P)/C(P) values are in locations that are isolated and require long travel distances with long detours or extensive backtracking, the chosen trajectory may be inefficient, spend too much time traveling, and collect a fewer and suboptimal number of samples than an optimal selection.

Programming Assignment:

Step 1:

- How did you calculate the information quality from the neural network prediction?

  I calculated the information quality by using the predicted map generated by the World Estimating Network. The greedy navigation algorithm calculates the information gain by taking the absolute value of the difference of the estimated world's value at (x,y) and 0 which is a reference certainty value. If the absolute value is higher that means there is more uncertainty there and exploring there will provide more information.

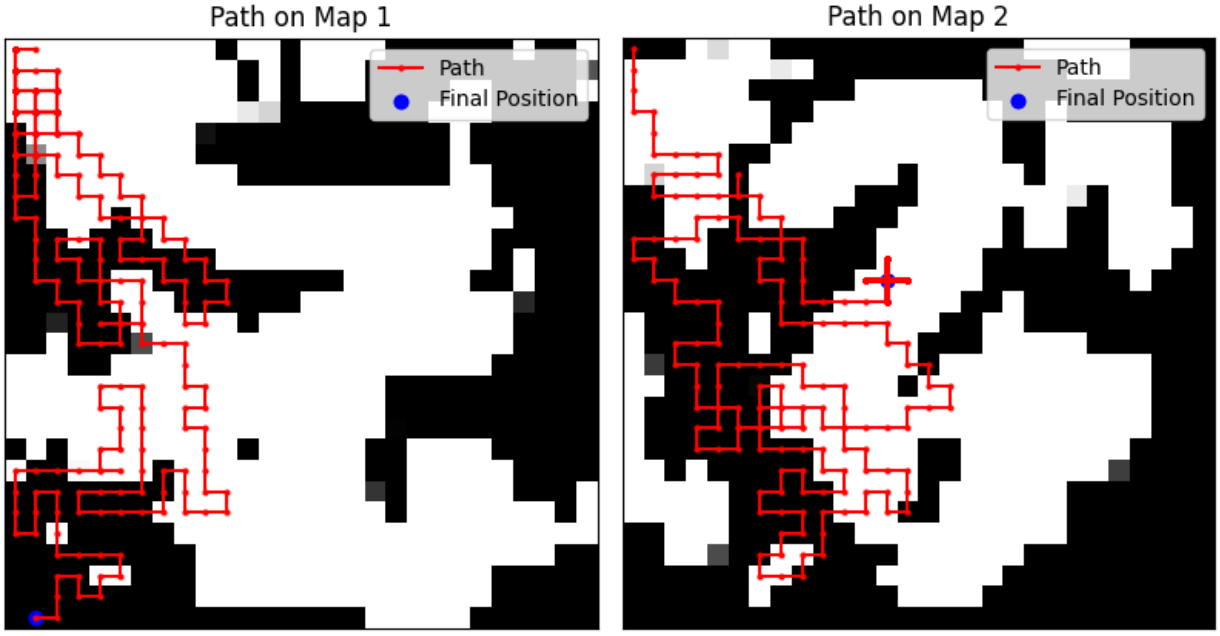
- How do you decide when to move to a corner and stop moving around greedily?

  The robot initially explores unexplored areas with high information gain and uncertainty until the map is covered sufficiently enough. If multiple moves are

possible, it selects the move with the most uncertainty/information gain. If there are no available unexplored tiles left, it chooses a random valid move.

After the map is explored significantly and the predicted digit is the same for multiple steps and exceeds the set threshold, it switches to goal-directed movement where it moves towards the goal corner. It moves in the direction of the goal by using sign differences and if the direct move is not valid, it picks a random move.

- Two example trajectories on different digits



- Reward Scores for 10 trials:
  [-154, -1000, -1399, -1399, -258, -1000, -1798, -1399, -3394, -38]

- Average reward over 10 trials using greedy algorithm
  -1183.9

Step 2:

- Formal description of algorithm
  The algorithm uses a modified A* algorithm with an information-driven heuristic to balance exploration and goal-directed movement. The heuristic combines goal distance and information gain to reward moves that reveal unexplored areas while ensuring progress towards the goal. Using A* instead of the greedy approach allows the robot to plan more globally efficient paths instead of making local, short-sighted decisions and reducing oscillation and revisiting locations.

A*-Based Path Planning with Information Gain Heuristic Algorithm

Input: q_0 (initial position), x_{goal} (goal corner), exploredMap (current knowledge of the environment)
Output: Path to goal

procedure AStarBasedNavigator(q_0, x_{goal}, exploredMap)
    Initialize open list O and closed list C
    Initialize cost dictionary g(q) = infinity for all nodes, set $g(q_0) = 0$
    Initialize heuristic dictionary h(q) = EuclideanDistance(q, x_{goal})
    Push q_0 into priority queue O with priority $f(q_0) = g(q_0) + h(q_0)$
    While O is not empty **do**
        $q_{current} \leftarrow$ O.pop() (node with lowest f-cost)
        if $q_{current} = x_{goal}$ **then**
            return reconstructPath($q_{current}$)
        C.add($q_{current}$)
        for $q_{neighbor}$ in getValidMoves($q_{current}$,exploredMap) **do**
            if $q_{neighbor} \in C$ **then continue**
            tentativeCost = $g(q_{current})$ + moveCost($q_{current}$,$q_{neighbor}$)
            infoGain = calculateInfoGain(exploredMap, $q_{neighbor}$)
            $h(q_{neighbor})$ = EuclideanDistance($q_{neighbor}$,x_{goal}) - α * infoGain
            $f(q_{neighbor}) = tentativeCost + h(q_{neighbor})$
            if $q_{neighbor} \notin O$ or tentativeCost <$g(q_{neighbor})$ **then**
                $g(q_{neighbor}) \leftarrow$ tentativeCost
                store $q_{current}$ as parent of $q_{neighbor}$
                O.push($q_{neighbor}$,$f(q_{neighbor})$)
    return failure (no valid path found)

Subprocedures:

- getValidMoves(q, exploredMap) → Returns valid neighboring moves for the robot.
- calculateInfoGain(exploredMap, q) → Computes how much new information would be gained by moving to q.
- moveCost(q_1, q_2) → Computes movement cost (e.g., penalizing revisits).
- reconstructPath(q_goal) → Traces back through parent pointers to get the full path.

(Generated this formal algorithm description by referencing the final report example on Canvas and with the assistance of ChatGPT)

- Reward Scores for 10 trials:
  [-946, 74, 74, 74, -472, 74, -500, -499, -444, -938]

- Average reward over 10 trials
  -350.3

- Time to find solutions:
  Total: 11.12 seconds (average time per solution is 1.11 seconds)

Python code is attached at end of this document

Discussion Questions:

1. When designing your algorithm in Step 2, what design criteria did you use to determine what to implement? Compare and contrast how your algorithm meets these design criteria relative to other algorithms taught in this course (use at least two additional examples from the course lectures or reading group discussions).

   The design criteria were:
   - Optimality: prioritizing reaching the goal efficiently using globally efficient paths instead of local greedy choices
   - Exploration vs exploitation balance: exploring new areas when necessary but also moving towards the goal once enough information is gained
   - Avoiding loops and oscillation: minimizing redundant movements and not revisiting the same locations
   - Robustness to uncertainty: handling imperfect information from neural networks while still making reasonable decisions
   - Computational efficiency and speed (because I'm extremely impatient): operating within reasonable time constraints and being scalable to different maps

Comparisons to other algorithms:

- Modified A* vs Dijkstra's Algorithm
  - Dijkstra's algorithm explores all possible paths uniformly, guaranteeing the shortest path but at a higher computational cost
  - This algorithm uses a heuristic function to focus on most promising paths making it more computationally efficient at finding goal-directed solutions
- Modified A* vs RRT
  - RRT generates random samples to expand a tree towards the goal, but it is better for continuous domains with complex constraints
  - This algorithm is more structured and efficient in grid-based navigation, ensuring a predictable and direct route to the goal.

2. The neural network provides you with a prediction of the correct digit, but not an uncertainty on that prediction. If you had access to an uncertainty estimate, how might you use this to modify your algorithm from Step 2(ii)? Describe qualitatively in words; you do not need to specify the new algorithm formally.

If I had access to an uncertainty estimate I could modify the algorithm to improve the decision-making process by using adaptive exploration vs exploitation so that when uncertainty is high, the algorithm could prioritize exploration more to gather more information before deciding on a goal which could prevent premature decisions based on weak predictions. Conversely, when uncertainty is low, the algorithm could prioritize moving towards the goal which would reduce unnecessary detours. I could also implement a weighted heuristic that adjusts based on uncertainty so if a prediction is uncertain it could prioritize paths with higher information gain but if the prediction is confident it could prioritize moving towards the goal. Instead of committing to a single goal location, the algorithm could maintain a probability distribution over a set of possible goal locations. Incorporating these uncertainty estimates would lead to more informed decision-making which would reduce errors due to premature predictions and prevent unnecessary exploration resulting in faster, more efficient navigation.

3. Consider the case where you have multiple robots communicating with each other and performing cooperative information gathering in this scenario. What additional design considerations would there be in this case? Describe qualitatively how you would modify your algorithm to coordinate multiple robots to efficiently gather information.

   Additional design considerations/modifications:

   - Avoiding redundant exploration:
     - Design consideration: each robot should explore different regions to maximize coverage and prevent revisiting the same areas.
     - Modification: the robots should have a shared map where each robot updates the areas it explored in real time. A task allocation system could dynamically assign unexplored regions to minimize overlapping exploration
   - Efficient communication and data sharing:
     - Design consideration: robots should be able to exchange information effectively without significant delays or bandwidth issues
     - Modification: use a message-passing system to update each robot on explored areas, classification confidence, and goal updates when necessary avoiding sending messages too frequently
   - Coordinated goal selection and adaptation:
     - Design consideration: robots should coordinate goal selection based on overall uncertainty instead of each robot heading to its own predicted goal

    - Modification: if multiple digits are classified with high confidence, robots can split up accordingly and if uncertainty is high they can cooperatively gather more information before committing to a final goal.
- Dynamic role assignment:
    - Design consideration: some robots could focus on information gathering while others can focus on path optimization and navigating to the goal
    - Modification: implement a system where robots dynamically switch between exploration, classification refinement, and goal navigation roles based on current needs.
- Conflict resolution for movement
    - Design consideration: robots should not block other robots paths or collide with or get in the way of other robots
    - Modification: use multi-agent path planning such as prioritized A* or cooperative A* to ensure collision avoidance. Robots can also use leader-follower strategies so some take primary navigation roles while others assist.

Step1.py

```python
__author__ = 'Chelse VanAtter'

import gzip
import numpy as np
from PIL import Image
from matplotlib import pyplot as plt
import random
from random import randint
from RobotClass import Robot
from GameClass import Game
from RandomNavigator import RandomNavigator
from networkFolder.functionList import Map, WorldEstimatingNetwork, DigitClassificationNetwork

__author__ = 'Chelse VanAtter'

import gzip
import numpy as np
from PIL import Image
from matplotlib import pyplot as plt
import random
from random import randint
from RobotClass import Robot
from GameClass import Game
from RandomNavigator import RandomNavigator
from networkFolder.functionList import Map, WorldEstimatingNetwork, DigitClassificationNetwork

class GreedyNavigator:
    def __init__(self):
        self.uNet = None  # Neural network for map estimation
        self.classNet = None  # Neural network for digit classification
        self.prediction = None  # Stores last digit prediction
        self.path_history = []  # Track robot's movement

    def setNetworks(self, uNet, classNet):
        # Assign the neural networks after object creation
        self.uNet = uNet
        self.classNet = classNet

    def getAction(self, robot, exploredMap):
        if not self.uNet or not self.classNet:
            raise ValueError("Both neural networks must be initialized before calling getAction.")

        mask = (exploredMap != 128).astype(int)  # Identifies explored areas
        estimated_world = self.uNet.runNetwork(exploredMap, mask)  # Predict missing areas
        digit_probs = self.classNet.runNetwork(estimated_world)
        self.prediction = digit_probs.argmax()
        #print(f"Digit identified: {self.prediction}")
```

```
48
49          x, y = robot.getLoc()
50          self.path_history.append((x, y))  # Log movement
51
52          possible_moves = {'left': (x - 1, y), 'right': (x + 1, y), 'up': (x, y - 1), 'down':
    (x, y + 1)}
53
54          move_options = []
55
56          for move, (nx, ny) in possible_moves.items():
57              if robot.checkValidMove(move):
58                  if exploredMap[nx, ny] == 128:  # Unvisited location
59                      info_gain = self.calculateInfoGain(estimated_world, nx, ny)
60                      move_score = random.uniform(0.5, 0.7) * info_gain
61                      move_options.append((move, move_score))
62
63          if move_options:
64              move_options.sort(key=lambda x: x[1], reverse=True)
65              best_choice = move_options[0][0] if random.random() < 0.8 else
    random.choice(move_options[:min(3, len(move_options))])[0]
66          else:
67              best_choice = random.choice(list(possible_moves.keys()))  # If no good move, pick a
    random one
68
69          if best_choice in possible_moves and robot.checkValidMove(best_choice):
70              new_x, new_y = possible_moves[best_choice]
71              exploredMap[new_x, new_y] = 0
72
73          return best_choice
74
75      def calculateInfoGain(self, estimated_world, x, y):
76          """ Calculate information gain at a location """
77          return np.abs(estimated_world[x, y] - 0)
78
79
80  # Simulation Setup
81  map_data = Map()
82  all_scores = []
83
84  for i in range(10):
85      print(f"Running Simulation for Map Number: {i + 1}")
86      robot = Robot(0, 0)
87      navigator = GreedyNavigator()
88
89      uNet = WorldEstimatingNetwork()
90      classNet = DigitClassificationNetwork()
91      navigator.setNetworks(uNet, classNet)
92
93      game = Game(map_data.map, map_data.number, navigator, robot)
94      last_prediction = None
95      consistent_count = 0
```

```
96
97      for step in range(1000):
98          reached_goal = game.tick()
99
100         if navigator.prediction == last_prediction:
101             consistent_count += 1
102         else:
103             consistent_count = 0
104
105         if step %100 == 0:
106             print(f"Step {step}: Robot at {robot.getLoc()}, Score: {game.getScore()},
    Prediction: {navigator.prediction}")
107
108         last_prediction = navigator.prediction
109
110         if reached_goal:
111             print(f"Goal reached in {game.getIteration()} steps!")
112             break
113
114     print(f"Final Score: {game.score}")
115     all_scores.append(game.score)
116
117     estimated_mask = (game.exploredMap != 128).astype(int)
118     estimated_world = uNet.runNetwork(game.exploredMap, estimated_mask)
119
120     color_map = np.stack((estimated_world,) * 3, axis=-1)
121     fig, ax = plt.subplots(figsize=(5, 5))
122     ax.imshow(color_map, cmap='gray')
123
124     if navigator.path_history:
125         x_vals, y_vals = zip(*navigator.path_history)
126         ax.plot(y_vals, x_vals, color='red', linewidth=1.5, marker='o', markersize=2,
    label="Path")
127         ax.scatter([y_vals[-1]], [x_vals[-1]], color='blue', label="Final Position", s=40)
128
129     ax.set_xticks([])
130     ax.set_yticks([])
131     plt.title(f"Path on Map {i + 1}")
132     plt.legend()
133     plt.show()
134
135     predicted_digit = classNet.runNetwork(estimated_world).argmax()
136     print(f"Final Predicted Digit: {predicted_digit}")
137
138     map_data.getNewMap()
139     print(f"Simulation for Map Number {i + 1} Complete\n")
140
141 print(f"All Scores: {all_scores}")
142 print(f"Average Score: {sum(all_scores) / len(all_scores)}")
143
```

Step2.py

```python
import numpy as np
import random
import time
import heapq
from RobotClass import Robot
from GameClass import Game
from networkFolder.functionList import Map, WorldEstimatingNetwork, DigitClassificationNetwork

class AStarNavigator:
    def __init__(self):
        self.uNet = None  # Neural network for map estimation
        self.classNet = None  # Neural network for digit classification
        self.prediction = None  # Stores last digit prediction
        self.visited = set()

    def setNetworks(self, uNet, classNet):
        self.uNet = uNet
        self.classNet = classNet

    def heuristic(self, x, y, goal):
        return abs(x - goal[0]) + abs(y - goal[1])  # Manhattan distance heuristic

    def getValidMoves(self, robot, x, y, exploredMap):
        moves = {'left': (x - 1, y), 'right': (x + 1, y), 'up': (x, y - 1), 'down': (x, y + 1)}
        valid_moves = []

        for move, (nx, ny) in moves.items():
            if robot.checkValidMove(move) and (nx, ny) not in self.visited:
                valid_moves.append((move, (nx, ny)))

        return valid_moves

    def getAction(self, robot, exploredMap):
        x, y = robot.getLoc()
        self.visited.add((x, y))

        estimated_world = self.uNet.runNetwork(exploredMap, (exploredMap != 128).astype(int))
        digit_probs = self.classNet.runNetwork(estimated_world)
        self.prediction = digit_probs.argmax()
        goal = self.getGoal(self.prediction)

        priority_queue = []
        for move, (nx, ny) in self.getValidMoves(robot, x, y, exploredMap):
            score = self.heuristic(nx, ny, goal) + random.uniform(0.1, 0.3)
            heapq.heappush(priority_queue, (score, move))

        if priority_queue:
            return heapq.heappop(priority_queue)[1]
```

```
        else:
            return random.choice(['left', 'right', 'up', 'down'])  # Fallback move

    def getGoal(self, prediction):
        if prediction in [0, 1, 2]:
            return (0, 27)
        elif prediction in [3, 4, 5]:
            return (27, 27)
        else:
            return (27, 0)

map_data = Map()
all_scores = []
times = []

for i in range(10):
    print(f"Running Simulation for Map Number: {i + 1}")
    robot = Robot(0, 0)
    navigator = AStarNavigator()

    uNet = WorldEstimatingNetwork()
    classNet = DigitClassificationNetwork()
    navigator.setNetworks(uNet, classNet)

    game = Game(map_data.map, map_data.number, navigator, robot)

    start_time = time.time()
    time_limit = 15 * 60  # 15 minutes in seconds

    for step in range(1000):
        if time.time() - start_time > time_limit:
            print("Terminating simulation: exceeded 15-minute limit.")
            break

        reached_goal = game.tick()
        if reached_goal:
            break

    elapsed_time = time.time() - start_time
    times.append(elapsed_time)

    print(f"Final Score: {game.score}, Time Taken: {elapsed_time:.2f} seconds")
    all_scores.append(game.score)
    map_data.getNewMap()

print(f"All Scores: {all_scores}")
print(f"Average Score: {sum(all_scores) / len(all_scores):.2f}")
print(f"Total Time Taken: {sum(times):.2f} seconds")
print(f"Average Time Per Solution: {sum(times) / len(times):.2f} seconds")

```