Designing Reliable Navigation Behaviors for Autonomous Agents in Partially Observable Grid-world Environments

Andrew R. Buck, Derek T. Anderson, and James M. Keller
Dept. of Electrical Engineering and Computer Science
University of Missouri
Columbia, MO, USA
Email: {buckar, andersondt, kellerj}@missouri.edu

Cindy Bethel and Audrey Aldridge
Dept. of Computer Science and Engineering
Mississippi State University
Starkville, MS, USA
Email: cbethel@cse.msstate.edu, ala214@msstate.edu

***Abstract*—Deciding where to go is one of the primary challenges in designing an agent that can explore an unknown environment. Grid-worlds provide a flexible framework for representing different variations of this problem, allowing for various types of goals and constraints. Typically, agents move one cell at a time, gathering new information at each time step. However, recomputing a new action after each step can lead to unintended behaviors, such as indecision and forgetting about previous goals. To mitigate this, we define a set of persistent feature layers that can be used by either a linear weighted policy or a neural network approach to identify potential destination locations. The outputs of these policies are processed using knowledge of the environment to ensure that objectives are met in a timely and effective manner. We demonstrate how to train and evaluate a U-Net model in a custom grid-world environment and provide guidance and suggestions for how to use this approach to build complex agent behaviors.**

I. Introduction

Grid-world environments have often served as benchmark problems for designing intelligent behaviors. From simple navigation tasks to games of great complexity, grid-worlds are adaptable to many different domains. The specifics of each grid-world problem are defined on a case by case basis, but in general, they consist of one or more agents placed on a grid of cells that can move about and possibly interact with the environment. There is usually a goal for each problem instance, and a common task is to develop a policy for each agent that will optimize for the stated goal.

Some examples of grid-world environments are the Minigrid library [1] and the BabyAI platform [2], which have been used to develop and test reinforcement learning (RL) algorithms. One characteristic of these environments is that they are typically abstract, not necessarily representing real environments or observation models, but rather serving to study the sequential decision-making problem. Often, these environments are used to visually represent a simplified and discrete state space, and to gain insight into the inner workings of an algorithm using a problem that is intuitive and accessible.

Although grid-worlds are often used as toy or benchmark problems, their general expressiveness makes them ideally suited to represent certain types of real-world tasks as well. Robotic navigation tasks can be studied using grid-worlds where the agent may be a ground robot or an Unmanned Aerial Vehicle (UAV). Occupancy maps [3] can be seen as a type of grid-world environment where the state of each cell (Free or Occupied) is initially unknown. Other representations include using fuzzy sets to represent possibilistic uncertainty maps [4] and 3D voxel representations such as OctoMap [5] and UFOMap [6].

A common task in a grid-world environment is to decide where to go next. In the more abstract grid-worlds, this is often implemented on a step-by-step basis, with the agent deciding on a movement action (up, down, left, or right) at each time step. In some of the more realistic environments, movement is specified using waypoints and path planning. In an RL setting, this corresponds to the action space of the environment. A small action space with only four possible actions leads to scenarios that require many steps, and many opportunities to demonstrate "unintelligent" behavior. Rewards in this context tend to be sparse, and it is common to see RL algorithms take a long time to train if the size of the environment grows large. Alternatively, using waypoints as actions can lead to very large action spaces and requires some intermediate method to translate high-level commands into low-level step-by-step movements. One possible strategy employed by Li et al. [7] is to use dedicated mapping, decision, and planning modules such that once an RL algorithm has picked a destination, the task of planning a route and navigating there is managed by other optimized planners, such as A*.

Herein, we design a custom grid-world environment to study how agent movement can be defined in terms of step-by-step actions while still adhering to an overall goal that may change with the discovery of new information. To accomplish this, we compute and maintain a set of persistent feature layers that store the agent observations as a type of "mental map". This is then used to determine where the agent should go in order to accomplish a specific task. We first introduce a linear weighted policy that aggregates user-supplied preferences. Using this as a baseline, we then train a U-Net model [8] to learn how to map the features of the environment and the agent's

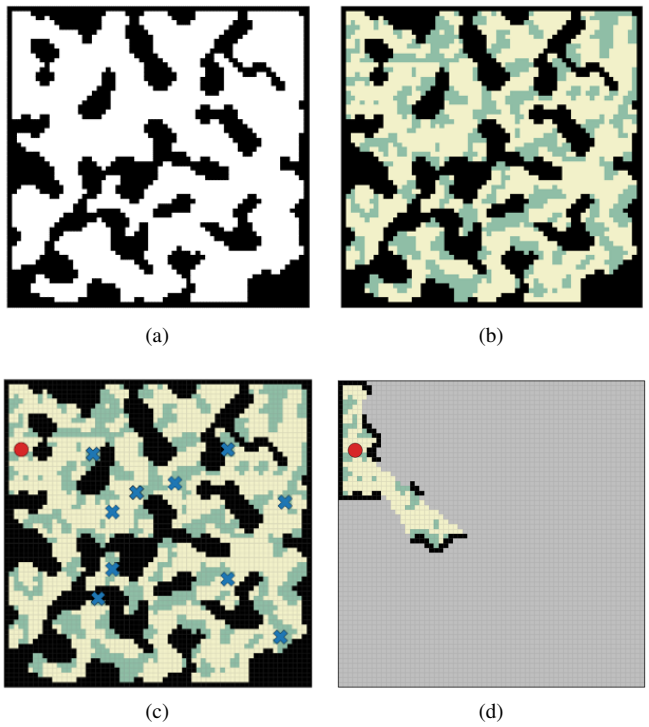


Fig. 1. Grid-world environment creation. (a) Grid cells are defined as either FREE (white) or WALL (black) using cellular automata rules and region dilation to create a single fully-connected cave-like map. (b) Terrain types such as MEADOW (tan) and FOREST (green) can be added to the free cells using additional cellular automata rules. (c) The agent (red circle) and targets (blue crosses) are placed in the map randomly. (d) The agent receives an observation based only on the parts of the map that are visible from the agent's location. Unobserved grid cells are masked out in gray. (Agent and target symbols are enlarged for clarity.)

mental map into high-level waypoint actions. These are then processed and applied as low-level movement actions.

The problem setting we consider is the target detection problem, which can be considered to be a variation of the traveling salesman problem or the search and rescue problem. We use procedurally generated environments similar to those in [9] and extend that work with a more generalized solution algorithm. The resulting control policy could be applied in synthetic 3D scenes using the voxel grid mapping approach of [10]. The U-Net policy we propose is adaptable and can be fine-tuned for specific problem types using an RL method.

The remainder of this paper is as follows. Section II describes the grid-world environment used in this work. Section III introduces the linear weighted policy for solving the target detection problem. Section IV describes the neural network policy and explains the training and evaluation methods. Finally, Section V gives conclusions and plans for future work.

II. GRID-WORLD ENVIRONMENT

A. Environment Description

The grid-world environment we use in this work consists of a square grid of 64×64 cells, each of which can represent one environmental state. The two most basic states are FREE and WALL, which indicate traversable cells and obstacles respectively. In some scenarios, a FREE cell can be further classified with a terrain type of MEADOW or FOREST. This naming convention serves to contextualize the problem as one of exploring a relatively large outdoor space, but could easily be adapted to represent other types of terrain.

An episode of the "game" consists of a single agent exploring the grid-world environment in search of some goal. (Multi-agent versions of the game are planned for future work.) The agent is located in one of the FREE environment cells and can move in four directions: up, down, left, or right. Each time step, the agent selects one of these four actions and moves to an adjacent cell. There is no uncertainty in the outcome of a movement action, and attempts to move into a WALL result in the agent staying in place.

Targets are placed throughout the environment and serve as potential goals for the agent. One target is designated as the true target and the rest are considered to be false alarms. An agent moving into the grid cell of the true target ends the episode, whereas an agent moving into the cell of a false alarm clears the target from the map, but does not end the episode. Targets may not always be detected by the agent and can require multiple observations to identify. This is meant to simulate the way in which an object detection algorithm may fail to recognize a true target, or mis-classify a false alarm. The probability of detection for true targets and false alarms is a function of the distance to the agent and the terrain type of the grid cell the target is placed in.

Rewards are provided according to the episode specification and can be changed to elicit different behaviors. Herein, we count the number of steps it takes the agent to find the true target by giving a reward of -1 for each step until episode termination. In more complex versions of the problem, rewards could be used to influence the types of terrain that the agent prefers to travel in, or the relative importance of clearing false alarms. For example, the agent could receive a reward of -1 for each step that ends in MEADOW, a reward of -10 for each step that ends in FOREST, a reward of $+100$ for clearing a false alarm, and a reward of $+1000$ for finding the true target.

B. Environment Generation

Each episode of the game consists of a randomly generated instance of the environment, based on a set of user-defined parameters. The environment creation process follows that of [9], and consists of several steps, starting with a cellular automata phase to define the FREE and WALL cells. During this phase, the cellular evolution rules are alternately applied with region dilation until a single fully-connected FREE region is created, such as in Fig. 1a. Some post-processing clean up ensures that the outer border states are WALL and diagonal gaps are filled in. The result is a cave-like map with both winding passages and large open areas that is well-suited for exploration problems.

Following the creation of the cave walls, some environment instances continue to define the MEADOW and FOREST terrain types by applying another round of cellular automata rules as

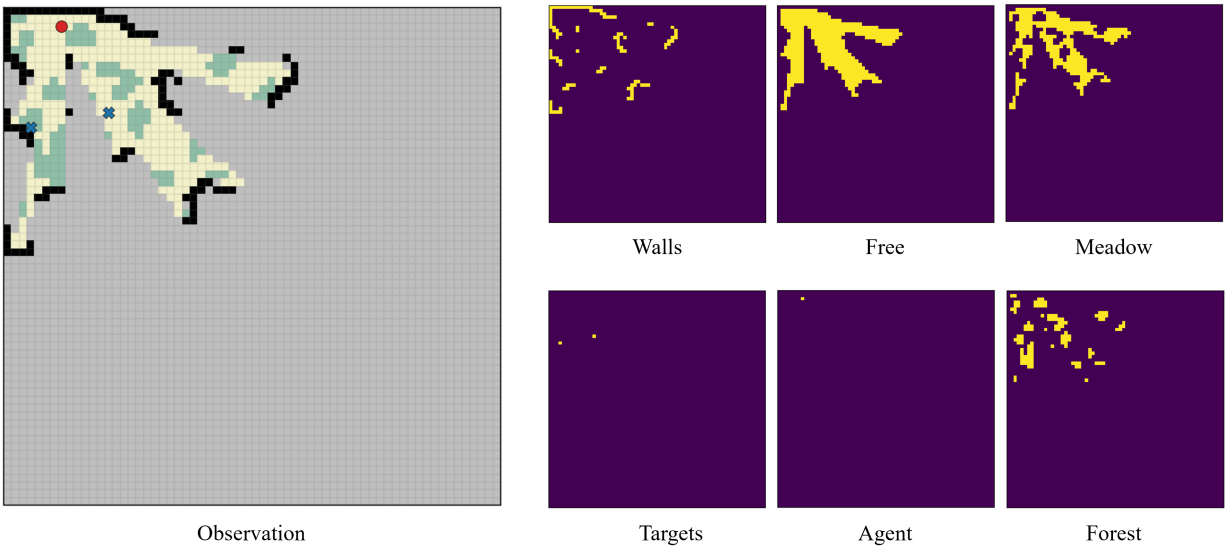


Fig. 2. An observation provided by the grid-world environment. The rendered observation (intended for human viewing) is shown on the left and the six binary feature maps used for computation are shown on the right.

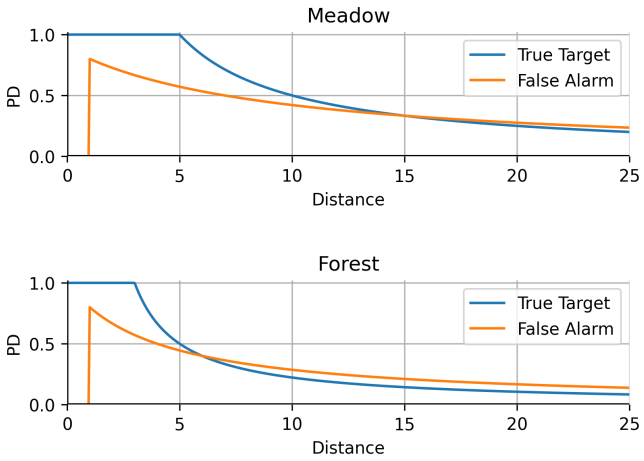


Fig. 3. Probability of detecting true targets and false alarms in both MEADOW and FOREST terrain, given the agent's distance from the target.

shown in Fig. 1b. In this case, the existing walls are used as boundaries, and the regions are not required to be connected.

Lastly, the environment is populated with the agent and 10 targets. The rules governing their placement can vary for each scenario, but typical methods include tabu-disk sampling, in which all sampled locations have a minimum separation, and random walk sampling, where targets are more likely to be clustered together in a group. Fig. 1c shows an example placement of an agent and targets.

C. Observations

The state of the environment is represented by the collective states of each grid cell, the position of the agent, and the locations of the remaining targets. However, the full state is not directly available to the agent. Instead, the agent receives an observation of the environment based on its current location. Any grid cell that is visible by line-of-sight from the agent's position without passing through any WALL cells is considered visible. The properties of all visible cells (including the terrain type, agent location, and any visible targets) are included as part of the observation (Fig. 1d).

The environment provides observations in the form of binary feature masks, indicating the presence or absence of a specific property. The typical feature layers are *Walls*, *Free*, *Targets*, *Agent*, *Meadow*, and *Forest* as shown in Fig. 2. These feature maps represent the instantaneous state of the environment and do not explicitly contain any historical information.

Targets are checked at the time of observation to determine if they are detected or not. We use a probabilistic model based on the agent's distance to the target and the terrain type of the grid cell containing the target. The intent is for true targets and false alarms to be harder to detect and distinguish at far distances, and also easier to find in MEADOW than in FOREST. For a true target t_{tt}, we define the probability of detection at a given distance d as,

$$p(t_{tt} \mid d) = \begin{cases} 1, & d < a \\ \frac{1}{b(d-a)+1}, & d \geq a \end{cases}, \tag{1}$$

where we set $a = 5$ and $b = 0.2$ for MEADOW, and $a = 3$ and $b = 0.5$ for FOREST. For a false alarm target t_{fa}, we use

$$p(t_{fa} \mid d) = \begin{cases} 0, & d < 1 \\ \frac{0.8}{b(d-1)+1}, & d \geq 1 \end{cases}, \tag{2}$$

where we set $b = 0.1$ for MEADOW, and $b = 0.2$ for FOREST. Plots of these functions are shown in Fig. 3.

III. A LINEAR WEIGHTED POLICY

One of the simplest ways to approach "solving" this game is to define an explicit policy for how to select actions given

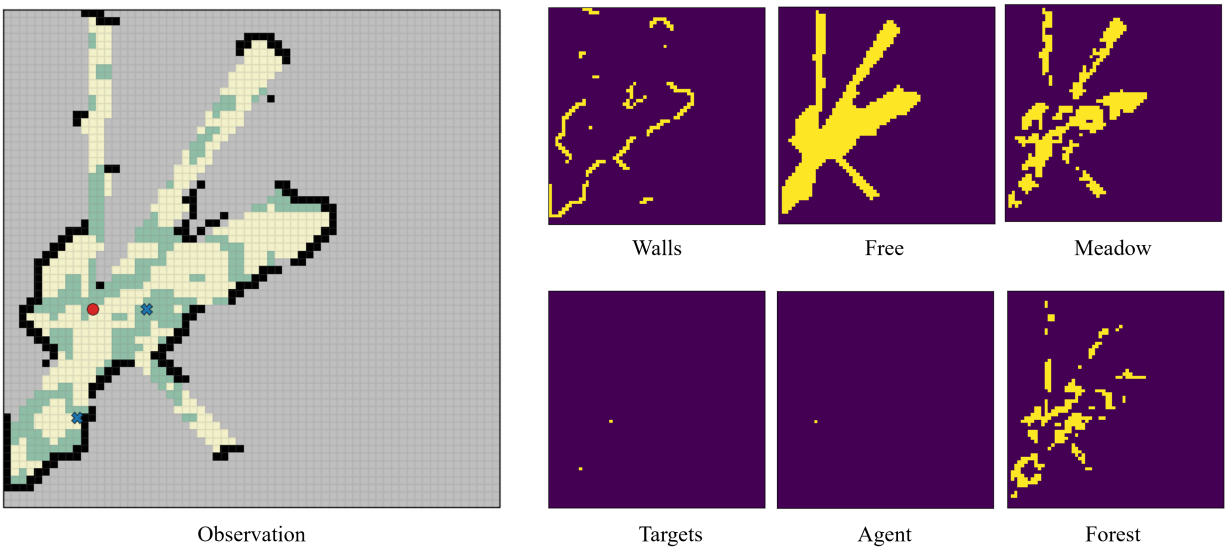


Fig. 4. An observation provided by the grid-world environment 60 steps after the starting observation in Fig. 2. The rendered observation (intended for human viewing) is shown on the left and the six binary feature maps used for computation are shown on the right.

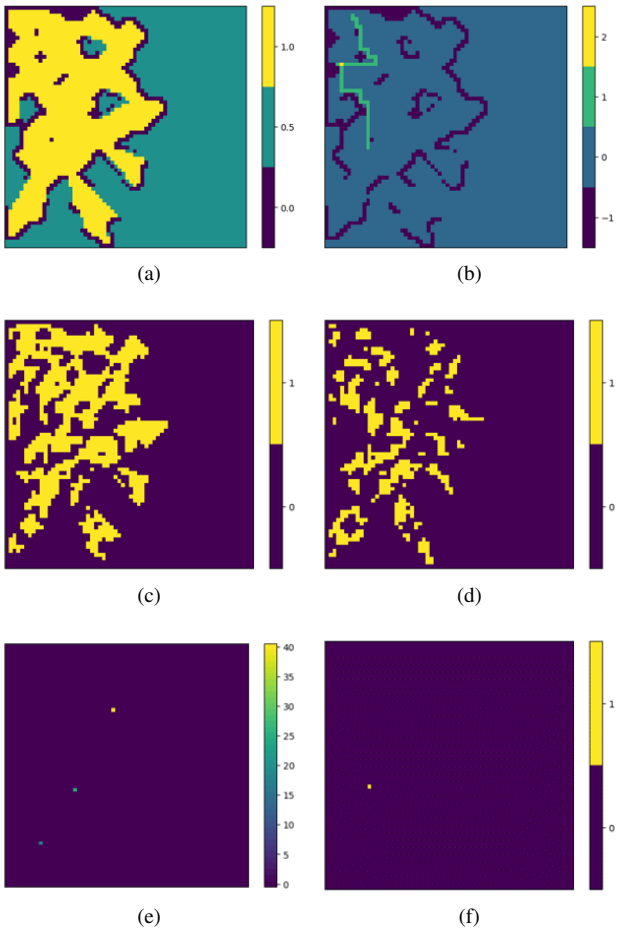


Fig. 5. Persistent feature layers that are updated after each step and used as input for the neural network policy. (a) Occupancy map, M_O. (b) Path history, M_P. (c) Meadow, M_{t1}. (d) Forest, M_{t2}. (e) Target history, M_T. (f) Agent position, M_A.

any observation. Fundamentally, this is an ill-posed problem in the sense that the best strategy depends on the previous observations. Consider the observation in Fig. 4, received 60 steps after the observation in Fig. 2. Without any form of history, the agent has no way of remembering parts of the map after it has left. As the agent explores the environment, it acquires new information and will build an internal "mental map" of its surroundings. This can be represented as a set of new feature layers that persist between actions and store the history of what the agent has observed (Fig. 5). From these features, we can compute a set of value maps that indicate where the agent should go. Following this approach leads to a rigidly-defined policy with a small set of user-defined parameters. We describe this method in the following sections.

A. *Persistent Feature Layers*

The first persistent feature layer we discuss is the occupancy map, M_O, shown in Fig. 5a. Each cell in M_O starts as UNOBSERVED with a value of 0.5. The cells are set to either 1 when a cell is observed as FREE or 0 when a cell is observed as WALL. We assume that the environment is static, although this approach could be used to model dynamic environments as well, using a value between 0 and 1 to represent the likelihood that a cell is occupied. The occupancy map is used for path planning to represent where the agent can move and what parts of the map remain unexplored. As a post-processing step, we fill in unreachable areas of the environment as WALL.

The next persistent feature layer is the target history layer, M_T, shown in Fig. 5e. This layer starts with each cell set to 0. Each time a target is observed in a grid cell, the value is incremented by 1. The target history layer keeps track of where and how often targets have been observed.

The next two persistent feature layers are the terrain history maps, M_{t1} and M_{t2}, shown in Fig. 5c and 5d. These binary

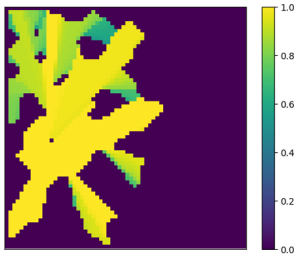
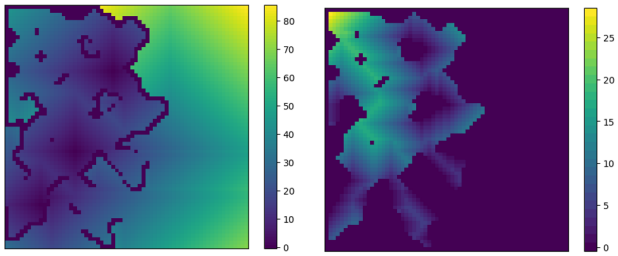


Fig. 6. Time since last observation layer, M_R, scaled to the unit interval and maintained as an additional persistent feature layer.



(a) (b)

Fig. 7. Distance maps computed from the observation in Fig. 4 and persistent feature layers in Fig. 5. (a) Target distance map, D_T. (b) Unobserved distance map, D_U.

maps save the terrain types, MEADOW and FOREST, for each observed FREE cell. The value of a cell, 0 or 1, indicates the presence of a given terrain type.

Finally, the path history feature layer, M_P saves the number of times each grid cell has been visited by the agent (Fig. 5b). The layer starts with each cell set to 0 and is incremented by 1 each time the agent visits a cell. Though not strictly necessary to include, we set all WALL cells to -1 when observed to aid in the visual interpretability of this feature layer.

One additional persistent feature layer that can be used is the time since a grid cell was last observed, shown in Fig. 6. This layer, M_R, is initialized to all 0, and it is set to 1 for all FREE cells at the time of observation. All other cells are discounted by a factor $\gamma_r = 0.99$, eventually approaching 0 if they have not been recently observed. This feature layer is useful for knowing how up-to-date the mental map is and for tasks involving persistent monitoring.

B. Computed Feature Layers

Two of the most useful metrics for deciding which action to take are the distance to the nearest target (Fig. 7a) and the distance to the nearest unobserved grid cell (Fig. 7b). By moving to a grid cell that is closer to a target, the agent can *exploit* knowledge that it has learned, whereas by moving closer to an unobserved grid cell, the agent *explores* new areas of the environment. Together, these metrics allow for a balance of exploration and exploitation that ultimately leads to solving any instance of the game.

The distance maps are computed by performing a breadth-first search from a set of starting locations, utilizing only the grid cells that have been observed to be not WALL (i.e. the cells for which $M_O > 0$). For the target distance map, D_T, any grid cell that has been observed to have a target ($M_T > 0$) is set as a starting location. The value of D_T for any cell then contains the shortest possible distance to one of these target cells. Note that the algorithm can search both FREE and UNOBSERVED cells and may therefore assign a smaller distance than the true value if a hidden WALL exists. For the unobserved distance map, D_U, all UNOBSERVED cells ($M_O = 0.5$) are used as the starting locations, and the distances stored in D_U are the shortest distances to the nearest unobserved cell.

In the present work, the distance maps represent the number of steps required to reach a goal, and therefore all steps have a uniform cost of 1. In a true multi-objective setting (left for future work), the distance maps could be a function of various environment features, such as terrain type, distance from a wall, visibility, or other user-defined criteria.

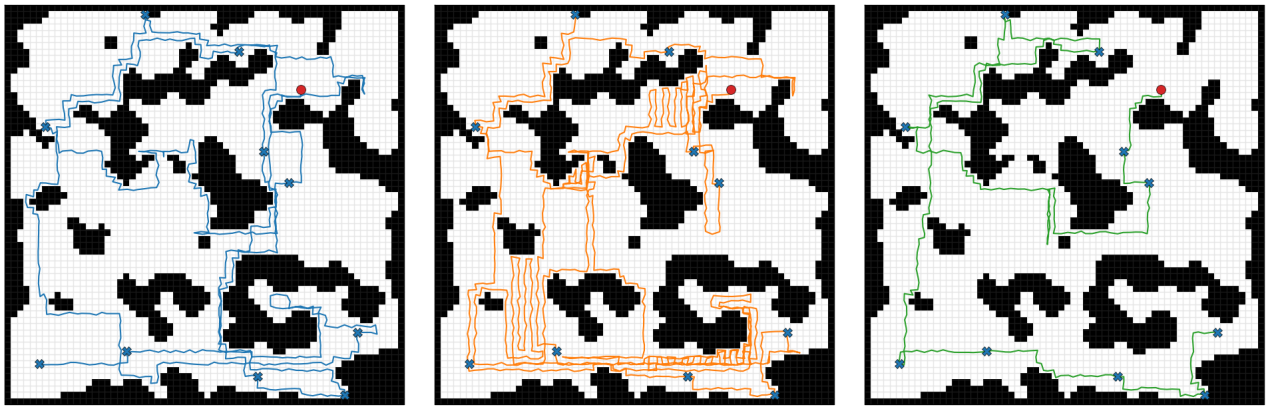
C. Policy Weights

A typical policy for solving this grid-world environment starts by using incoming observations to update and maintain the persistent feature layers of the "mental map". After new information is acquired each step, the distance maps are recomputed and used to decide the direction to move. Perhaps the most straightforward policy is the greedy policy, π_{G}. With this policy, we define a combined distance map D that represents the minimum remaining distance to some goal location. For the greedy policy, D is defined as either the distance to the nearest known target, D_T if a target has been observed but not yet cleared, or the distance to the nearest unobserved grid cell, D_U, otherwise. The neighboring grid cells of the current agent position are then checked in D. The direction of the cell with the smallest value that is not WALL is chosen as the action. This policy defaults to an exploitation mode that always moves towards any discovered targets. If there are no known target locations, the policy reverts to an exploration mode and moves towards the nearest unobserved grid cell in an attempt to discover new information.

While the greedy policy is intuitive and performs reasonably well on most environment instances, it can be improved. In some cases, the best strategy would be to finish exploring an area before moving across the entire map to clear a target, only to have to return and finish exploring later. Clearly, there should be some balance between exploration and exploitation. The weighted policy, π_{W}, uses a combination of both D_T and D_U to define D. We define three weights, w_t, w_u, and w_p, representing the relative importance of the target feature, the unobserved feature, and the path history feature. The path history is included to discourage revisiting locations and to prevent the agent from getting stuck in local minima. The combined distance map D is then defined as

$$D = w_t D_T + w_u D_U + w_p M_P. \tag{3}$$

As with the greedy method, the agent selects the action that moves to a cell with a smaller value in D.



(a) (b) (c)

Fig. 8. A grid-world environment solved with three different weighted policies. In each case, the agent (red circle) must visit all 10 targets (blue crosses). The agent's observations are limited by line-of-sight as it explores the initially unknown environment. (a) Exploration ($\theta = -1$). Total steps: 644. (b) Balanced ($\theta = 0$). Total steps: 1005. (c) Exploitation ($\theta = 1$) Total steps: 322.

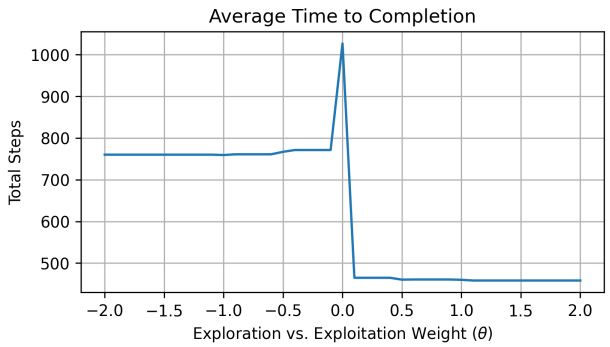


Fig. 9. Total number of steps required to clear 10 targets in a grid-world, averaged over 100 instances and plotted for varying values of θ, indicating the degree of exploration vs. exploitation.

The weighted policy provides a way to parameterize the behavior of the agent. The three weights can be seen as a way to specify the importance of the different features. Additional features could be added to this approach to create more directed behavior.

D. Parameter Sweep Experiment

To demonstrate the effect of the parameters on the weighted policy, we consider a simplified grid-world problem environment. In this experiment, we ignore terrain and only consider FREE and WALL cells. Targets are also assumed to always be detected so long as they are visible by line-of-sight. We place 10 targets in the environment with the tabu-disk sampling method and require the agent to clear all 10 targets. This is effectively a variation of the traveling salesman problem where the distances are unknown a priori.

We assign the weights using a single parameter θ, where

$$w_t = \begin{cases} 1+\theta, & \theta \geq 0 \\ 1, & \theta < 0 \end{cases}, \tag{4}$$

$$w_u = \begin{cases} 1, & \theta \geq 0 \\ 1-\theta, & \theta < 0 \end{cases}, \tag{5}$$

and

$$w_p = 1. \tag{6}$$

This gives more weight to exploitation when $\theta > 0$, and more weight to exploration when $\theta < 0$.

An example environment is shown in Fig. 8, where three different settings of θ are used for the weighted policy. In each of these cases, ties in action selection are chosen deterministically for reproducability. When $\theta = -1$ (Fig. 8a), the agent takes 644 steps and prioritizes exploring the environment over clearing targets. When $\theta = 0$ (Fig. 8b), the objectives are balanced, but this leads to indecision and frequent pacing back and forth until the path history weight pushes the agent either toward a target or some unobserved region, resulting in a total path length of 1005 steps. Finally, when $\theta = 1$ (Fig. 8c), the agent prioritizes clearing targets and has the shortest path length of 322 steps. This is not the shortest possible route, however, since limited visibility causes the agent to make a detour when it discovers a distant target, only to turn around and retrace its route when it then discovers a closer one.

In general, we find that with few exceptions, shorter paths can be obtained by favoring exploitation over exploration. This is confirmed by repeating this experiment for 100 randomly generated environments with values of θ ranging from -2 to 2. The average number of steps required to complete each map is shown in Fig. 9. A noticeable peak can be observed for $\theta = 0$, suggesting that when the objectives are balanced, the agent is more likely to be indecisive and get stuck in a local minimum, requiring the accumulation of path history to push it out. The flat response as θ grows large or small indicates that the weighted policy begins to act as pure exploration or exploitation with little mixing of the two strategies. In order to elicit more complex behavior, a more challenging version of the problem is required.

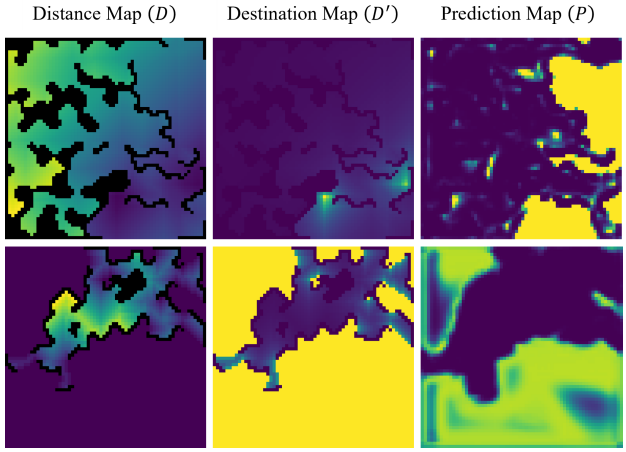


Fig. 10. Destination maps predicted by the U-Net after 1000 training steps.

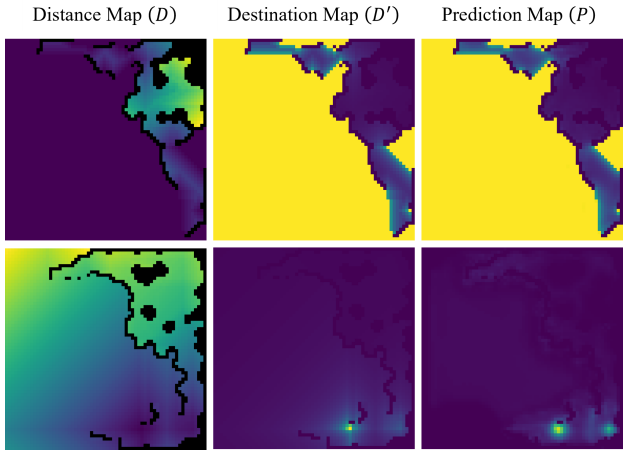


Fig. 11. Destination maps predicted by the U-Net after 500,000 training steps.

IV. A Neural Network Policy

One of the shortcomings of the linear weighted policy is that it requires manually defined feature layers and preference weights. Learning the weights for a given problem specification may be possible, but as demonstrated in the previous section, optimal values may be trivially easy to find. Still, the level of control provided by the linear weighted policy and the explainability aspect are appealing qualities. In this section, we define a neural network policy, π_{NN}, that extends the linear weighted policy and provides the framework for learning advanced behaviors.

A. U-Net Training

The U-Net architecture [8] was originally proposed to perform image segmentation, taking an image input and producing a classification map with a label for each pixel. This method of using an encoder and decoder pair to map image features to a latent space and then reconstruct an image has found broad success, and we utilize the Segmentation Models library [11] implementation of a U-Net in our work. We use a ResNet-34 encoder [12] with a 6-channel 64×64 image input corresponding to the 6 persistent feature layers shown in Fig. 5. The output is a single channel 64×64 image with a sigmoid activation function.

We train this model to learn where to go next in the environment given the current state of the agent's mental map. The combined distance map, D, from the linear weighted policy is defined such that the most desirable locations have the smallest values, and the agent moves in the direction of its lowest valued neighbor cell. Here, we scale and invert the distance map to be in the range $[0, 1]$ such that the most desirable locations are assigned higher values. Assuming that the smallest possible value in D is 0 for any cell that is not Wall, the scaled destination map, D', is defined for a grid cell c as,

$$D'(c) = \begin{cases} 0, & c \text{ is Wall} \\ \frac{1}{D(c)+1}, & \text{otherwise} \end{cases}. \tag{7}$$

To train the model, the linear weighted policy is applied continuously in randomly generated environments of the kind described in Section II with $\theta = 1$. When the episode is complete, a new instance is generated and training continues. At each step, the combined distance map, D, is transformed into the scaled destination map, D', which is used as the target output of the model. The predicted map, P, is generated from the persistent feature layers of the current observation, and the RMSE loss is computed over all cells that are not Wall. Some example outputs of the model after 1000 training steps are shown in Fig. 10, and after 500,000 training steps in Fig. 11. After this amount of training, the model was generally able to reproduce any destination map provided to it.

B. From Prediction to Action

The destination map predicted by the model cannot be used to select agent actions directly, as it remains too imprecise and unreliable. As with the balanced objectives from the linear weighted policy, situations can arise where the agent becomes stuck and unable to proceed. To remedy this, we build and maintain a value map, V, that represents the most up-to-date assessment of where the agent should go. This provides a flexible solution that can be adapted for various types of agent movement, including step-by-step actions in the grid-world environment and the waypoint-based control method often used on real robot or UAV hardware. Fig. 12 shows an overview of the steps taken to turn the model prediction into a usable value map for selection agent actions.

The value map, V, is initialized to zero at the beginning of an episode and is updated after every new observation. These observations are passed to the model to produce the predicted destination map, P, for the current step (Fig. 12a). The grid cells with the highest value in the predicted destination map are selected as the potential destinations, C. One of these cells will be chosen as the current destination, c_d, and added to the value map. The current destination can be thought of as the next waypoint that the agent would like to visit, given all of the information up to the current time step. In practice, this

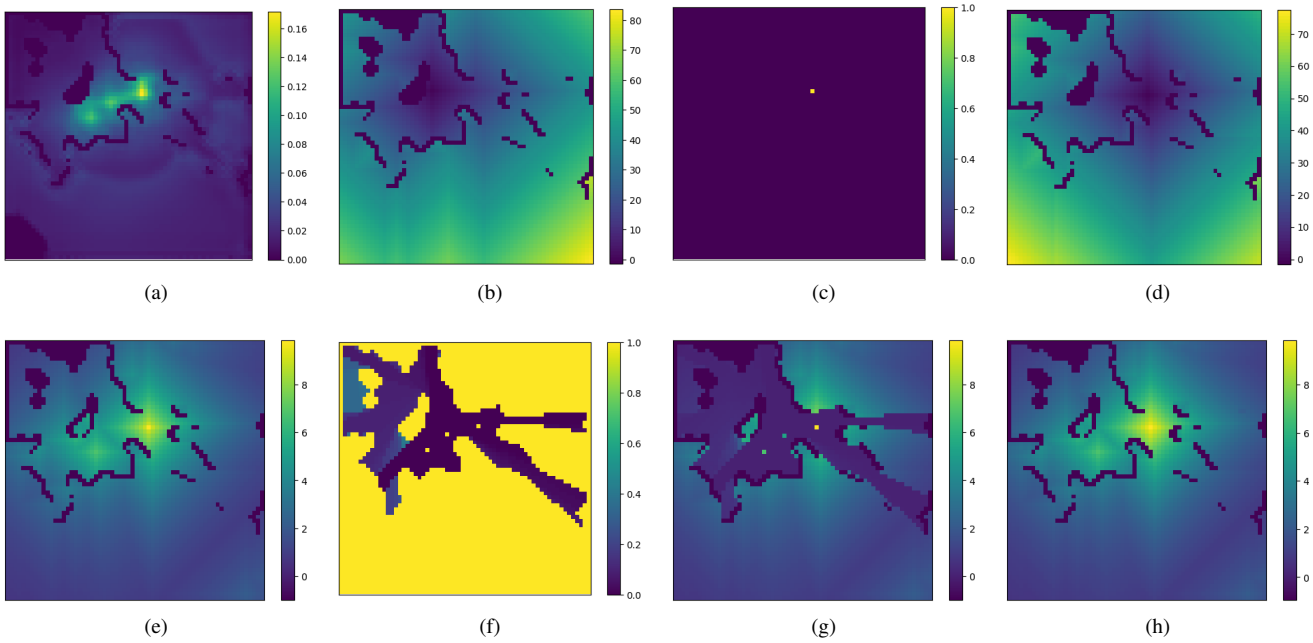


Fig. 12. Steps in the procedure for updating the persistent value map from the model prediction. (a) Predicted destination map, P. (b) Distance to the agent. (c) Probability of selecting a grid cell as the current destination. (d) Distance from the current destination, D_P. (e) The initial persistent value map, V. (f) Reduction factor, γ, based on the time since a cell was last observed. (g) Temporary value map, V'. (h) Updated value map, V'', with a smooth gradient.

location can change drastically between steps, so it is not used as a goal directly.

To give more priority to nearby locations, the value of each potential destination $c_i \in C$ is scaled by the inverse distance to the agent, d_i (Fig. 12b), and the unnormalized probability of selecting c_i as the current destination is given as

$$\tilde{p}(c_i) = \frac{P(c_i)}{d_i + 1}. \tag{8}$$

These values are normalized and the current destination, c_d, is selected with probability

$$p(c_d) = \frac{\tilde{p}(c_d)}{\sum_{c \in C} \tilde{p}(c)}. \tag{9}$$

Fig. 12c shows the probability of selecting each cell for the given example, which in this case is limited to a single grid cell, but could be multiple locations if the prediction map becomes saturated.

A new distance map, D_P (Fig. 12d), is computed from the sampled destination location such that the value of each grid cell is the minimum possible number of steps required to reach c_d. The first step of updating the value map (Fig. 12e) is to multiply the values in V by a discount factor $\gamma = 0.99$ and then increment by the inverse distance of D_P. Finally, the value map is reduced in areas that have been recently observed, as measured by the "time since last observation" layer, M_R (Fig. 6). This is to prevent the agent from assigning too much value to nearby locations and getting stuck in a local optimum. The reduction factor for a grid cell c is defined as

$$\lambda(c) = \begin{cases} 1, & M_T > 0 \\ 1 - M_R(c), & \text{otherwise} \end{cases}. \tag{10}$$

Fig. 12f shows the reduction factor for each cell in the given example. This effectively eliminates any value in the directly observed area except for targets, forcing the agent to move to new areas unless there is an obvious target that can be cleared. The temporary value map V' (Fig. 12g) for a grid cell c is defined as

$$V'(c) = \lambda(c)\left(\gamma V(c) + \frac{1}{D_P(c) + 1}\right). \tag{11}$$

Finally, new the value map, V'' (Fig. 12h), is computed by propagating the values in V' back to the agent's location. V'' is initially set to be equal to V' and then iteratively updated. For each cell c, the updated value is determined by the maximum value of its neighbors, $N(c)$ multiplied by a discount factor $\mu = 0.95$.

$$V''(c) \leftarrow \mu \cdot \max_{n \in N(c)} V''(n). \tag{12}$$

The updates continue until the values in V'' have settled or a maximum number of iterations have occurred. The persistent value map is then updated as $V \leftarrow V''$. The result of this procedure is a smooth gradient in the local region around the agent, so that the previous method of selecting an action to move to a neighbor grid cell with a better (larger) value in V can be applied. If the control method were waypoint-based, the next waypoint could be selected as the cell with the maximum value in V, only changing when the value of a different cell exceeds the current value by some threshold.

The entire process of updating the value map and selecting an action after each observation is summarized by the pseudocode in Algorithm 1.

Algorithm 1 Value Map Update and Action Selection

Initialize V with all 0 values
for each new observation **do**
  Get predicted destination map P from the U-Net
  Identify the maximum cells C in P
  Sample the current destination c_d from C
  Compute the distance map D_P from c_d
  Determine the reduction factor λ from M_R and M_T
  Compute V' (Equation 11)
  Propagate values in V' to get V'' (Equation 12)
  Set $V \leftarrow V''$
  Select the action that moves to the neighboring grid cell with the largest value in V
end for

C. Comparing to the Linear Weighted Policy

The neural network policy provides an alternative way to solve grid-world problems without manually specifying feature weights. The U-Net in our model was trained to replicate the behavior of the linear weighted policy with $\theta = 1$. As such, it could be expected to perform comparably to that approach. To verify this, we evaluate the neural network policy and the linear weighted policy with 100 grid-world instances, using $\theta \in \{-1, 0, 1\}$ for the linear wieghted policy. The results of this experiment are shown in Fig. 13. The episodes are sorted based on the average number of steps required across all policies.

Across the 100 episodes, the average number of steps was 229 for the neural network policy, and it was 231 for the linear weighted policy with $\theta = 1$. For $\theta = -1$, the average number of steps was 465, and for $\theta = 0$ it was 459. On average, the neural network policy performed about as well as the linear weighted policy it was based on, and significantly better than the other variations.

In certain episodes, the exploration-based policy performed better than the exploitation-based policy, and the neural network policy was best. Fig. 14 shows one such example case. In this environment, the exploitation policy (Fig. 14b) explored the entire map without detecting the true target (orange cross). Once there are no more UNOBSERVED cells, the agent has no sense of where to go and is guided only by the path history, M_P, until it eventually finds the target. The exploration policy (Fig. 14a) and the neural network policy (Fig. 14c) both manage to find the target more quickly. Fig. 15 shows an image sequence of the path history, U-Net prediction, and value map of the neural network policy as it solves the problem.

V. CONCLUSIONS AND FUTURE WORK

In this work, we presented a linear weighted policy and a neural network policy for solving the target detection problem in partially observable grid-worlds. Both methods were able to solve general instances of the problem and could be adapted for use in more realistic settings. The linear weighted policy made use of manually defined feature layers, which provide a way to integrate user preferences into a common

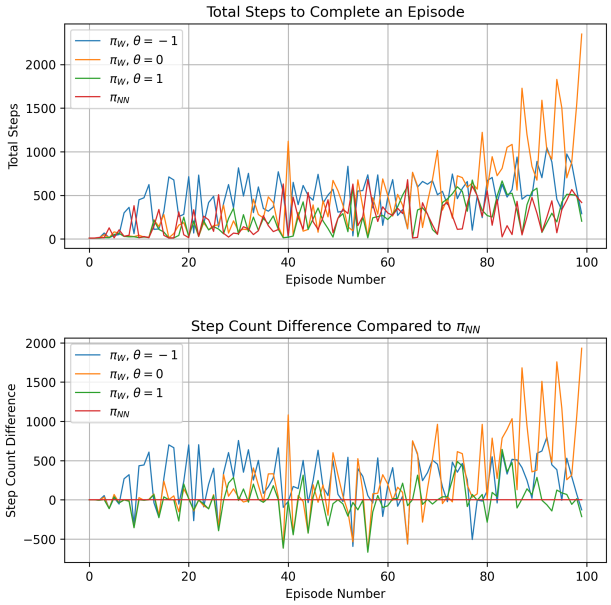


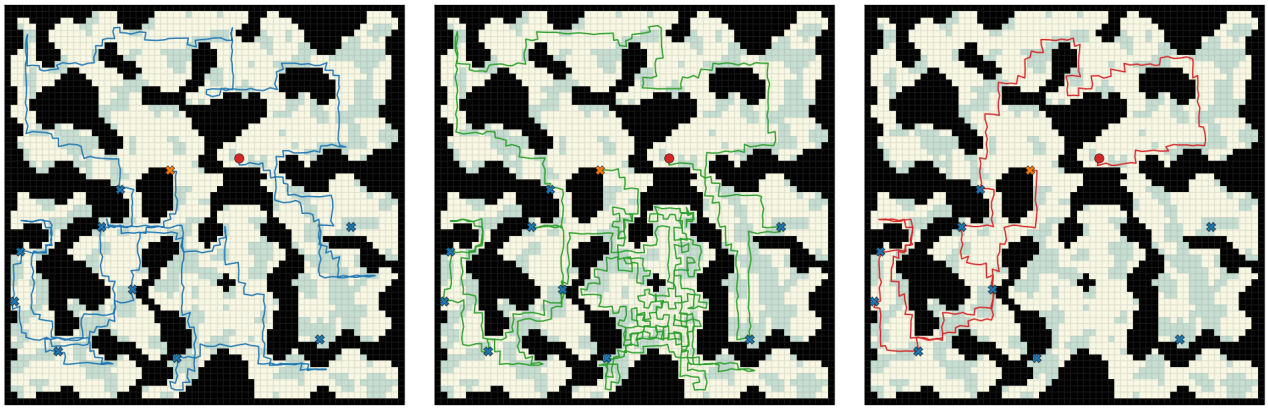
Fig. 13. Total steps required to complete 100 different grid-world environments for the linear weighted policy, π_{W}, with $\theta \in -1, 0, 1$, and the neural network policy, π_{NN}. The top graph shows the raw step counts, and the bottom graph shows the relative step counts compared to the neural network policy.

multi-objective framework. We demonstrated this idea with a straightforward parameter to balance exploration and exploitation and found exploitation to be generally more beneficial in this problem setting. The neural network policy used a U-Net model to predict a destination map of where to go, and after post-processing, it was able to match the performance of the linear weighted approach.

These policies show that grid-world problems involving the search for a target in an unknown environment can be generally solved using hand-crafted or learned feature maps. The problem setting in these experiments make use of the terrain type only as a way to influence the detection of a target, but more complex problems could also use terrain or other environment features to direct agent behavior by specifying different rewards.

The U-Net model is capable of learning how to map environment features to desired actions. One clear direction for future work is to extend this approach with an RL framework to further improve the neural network policy, possibly incorporating recurrent or attention-based architectures. For this approach to be more successful than a hand-crafted approach, a more complex problem setting will likely be required that cannot be easily solved by the linear combination of feature maps. In simpler environments, there is less to be gained from the adoption of a neural network policy.

Finally, this work could be adapted to multi-agent settings with a formal multi-objective framework. There are several ways these models could incorporate human teaming and specific user preferences. Ultimately, we would like to see this work applied towards more realistic scenarios involving 3D scenes and physical agents such as ground robots or UAVs.



(a) (b) (c)

Fig. 14. A case where the exploration policy performs better than the exploitation policy, and both are outperformed by the neural network policy. The agent (red circle) must detect and clear the one true target (orange cross) and distinguish it from the false alarms (blue crosses). (a) Exploration policy (578 steps). (b) Exploitation policy (880 steps). (c) Neural netwok policy (282 steps).

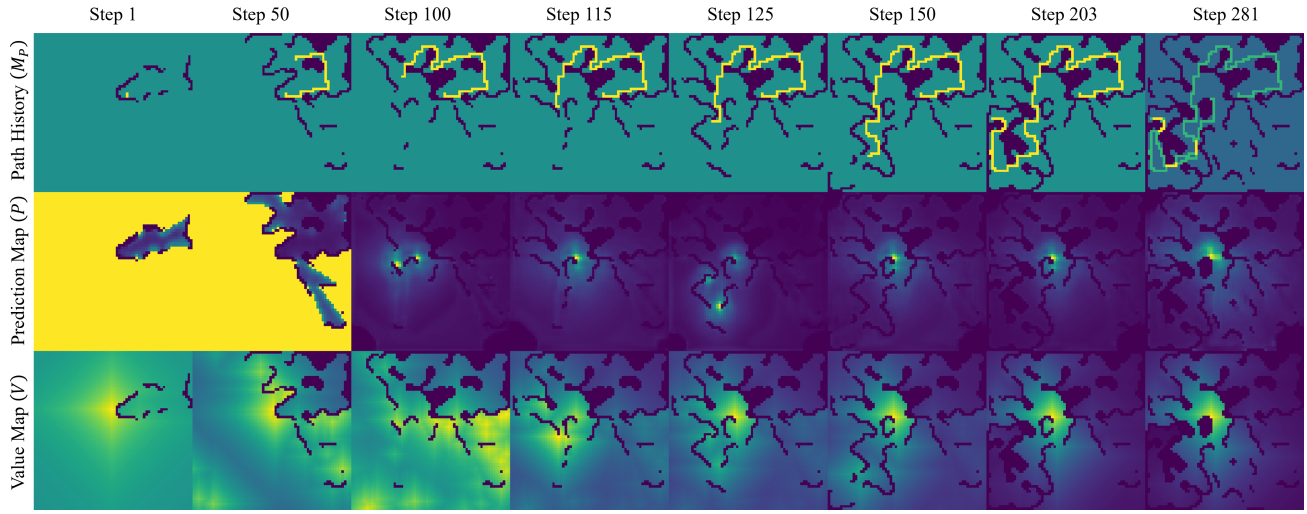


Fig. 15. Image sequence of the path history, U-Net prediction, and value map of the neural network policy as the agent solves the environment in Fig. 14.

REFERENCES

[1] M. Chevalier-Boisvert, B. Dai, M. Towers, R. de Lazcano, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry, "Minigrid & Miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks," *CoRR*, vol. abs/2306.13831, 2023.

[2] M. Chevalier-Boisvert, D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen, and Y. Bengio, "BabyAI: First steps towards grounded language learning with a human in the loop," in *International Conference on Learning Representations*, 2019.

[3] S. Thrun, "Learning Occupancy Grid Maps with Forward Sensor Models," *Autonomous Robots*, vol. 15, no. 2, pp. 111–127, Sep. 2003.

[4] G. Oriolo, G. Ulivi, and M. Vendittelli, "Real-time map building and navigation for autonomous robots in unknown environments," *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 28, no. 3, pp. 316–333, Jun. 1998.

[5] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, Apr. 2013.

[6] D. Duberg and P. Jensfelt, "UFOMap: An Efficient Probabilistic 3D Mapping Framework That Embraces the Unknown," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6411–6418, Oct. 2020.

[7] H. Li, Q. Zhang, and D. Zhao, "Deep Reinforcement Learning-Based Automatic Exploration for Navigation in Unknown Environment," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 6, pp. 2064–2076, Jun. 2020.

[8] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.

[9] A. R. Buck and J. M. Keller, "A myopic Monte Carlo strategy for the partially observable travelling salesman problem," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, Jul. 2016, pp. 632–639.

[10] A. Buck, R. Camaioni, B. Alvey, D. T. Anderson, J. M. Keller, R. H. Luke, III, and G. Scott, "Unreal engine-based photorealistic aerial data generation and unit testing of artificial intelligence algorithms," in *Geospatial Informatics XII*, vol. 12099. SPIE, May 2022, pp. 59–73.

[11] P. Iakubovskii, "Segmentation models pytorch," https://github.com/qubvel/segmentation_models.pytorch, 2019.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778.