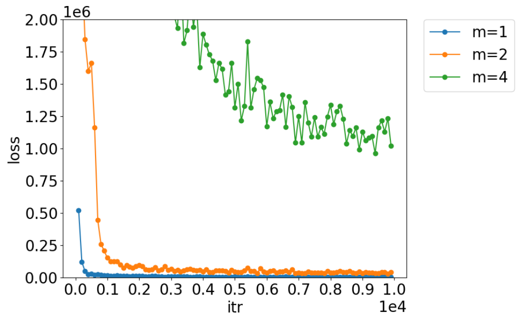
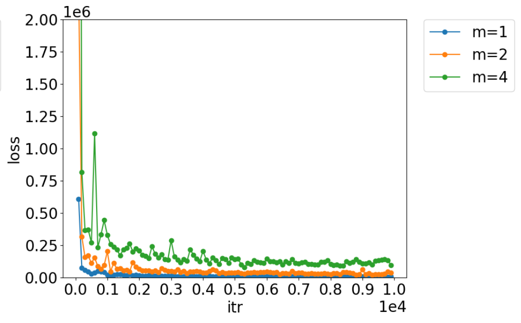
A Appendix

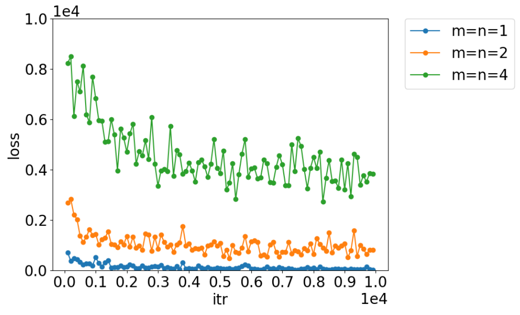
A.1 Using neural networks to approximate multi-resource coverage objectives



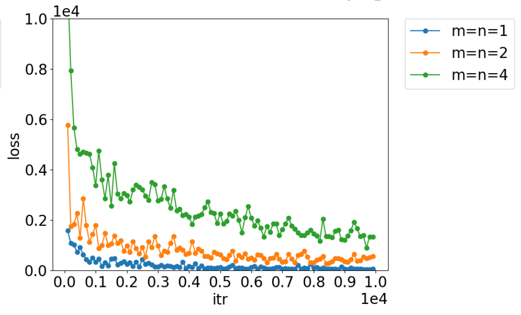
(a) Areal Surveillance with feedforward NN.



(b) Areal Surveillance with graph NN.



(c) Adversarial Coverage with feedforward NN.



(d) Adversarial Coverage with graph NN.

Figure 5: MSE for reward prediction with feedforward and graph neural networks.

We explore the efficacy of using neural networks to learn approximate differentiable models of the objectives. We trained neural networks, one per forest and per value of m (and n for two-agent games), to predict the reward of the defender (and attacker in case of two-agent game) on a single forest domain. The neural networks take as input the action u_D of the defender (and u_A also for two-agent game) and outputs a prediction for the reward $\hat{r}_{D,1p}$ ($\hat{r}_{D,2p}$ and $\hat{r}_{A,2p}$ for two-agent game). Please see section A.6 for network architectures and hyperparameters. Figure 5 shows the corresponding curves of mean square error (MSE) loss vs. training iterations for both games when the neural networks are strictly feedforward and also when we use graph neural networks (Battaglia et al., 2018), which treat each agent resource as a node in a graph and are much better at generalizing in combinatorial interaction settings. We observe that while graph neural networks are better than feedforward networks in approximating the coverage rewards, both of them still suffer from high absolute values of MSE even after 10000 training iterations. The MSE is much higher when a larger number of agent resources are used e.g., $m(=n) = 4$ cases (green lines) incur higher errors than $m(=n) = 1$ cases (blue lines) under all settings. This demonstrates that multi-resource coverage objectives are combinatorially hard to approximate even with graph neural networks, especially more so as the number of agents' resources and consequently combinatorial interaction between them increases.

A.2 Implicit boundary differentiation for gradient simplification

As mentioned in section 3.1, the term $\frac{\partial q_{Q\cap\delta S_i}}{\partial u_i}^T n_{q_{Q\cap\delta S_i}}$ from eq 2 can be simplified further using implicit differentiation of the boundary of S_i. In our example domains, the coverage boundaries induced by all resources (drones or lumberjacks) are circular. With the location of i-th drone as $u_i = \{p_i, h_i\}$ and for the j-th lumberjack as $u_j = p_j$, the boundaries are given as:

$$\delta S_i = \{q \mid ||q - p_i||_2 = h_i \tan\theta\} \quad \text{for drones, and}$$
$$\delta S_j = \{q \mid ||q - p_j||_2 = R_L\} \quad \text{for lumberjacks}$$

We illustrate the calculation of the $\frac{\partial q_{Q\cap\delta S_i}}{\partial u_i}^T n_{q_{Q\cap\delta S_i}}$ term for a drone below and the calculation follows similarly for lumberjacks. Any point $q \in Q \cap \delta S_i$ satisfies:

$$||q - p_i||_2 = h_i \tan\theta$$

Differentiating this boundary implicitly w.r.t. p_i and w.r.t. h_i gives:

$$\left(\frac{\partial q}{\partial p_i}^T - I_2\right)\frac{q-p_i}{||q-p_i||_2} = 0, \text{ and}$$
$$\frac{\partial q}{\partial h_i}^T \frac{q-p_i}{||q-p_i||_2} = \tan\theta.$$

Noting that the outward normal n_q at any point $q \in Q \cap \delta S_i$ is given by $\frac{q-p_i}{||q-p_i||_2}$, we now have:

$$\begin{aligned}\frac{\partial q}{\partial u_i}^T n_q &= \left\{\left(\frac{\partial q}{\partial p_i}^T n_q\right)^T, \frac{\partial q}{\partial h_i}^T n_q\right\}\\ &= \left\{\left(\frac{q-p_i}{||q-p_i||_2}\right)^T, \tan\theta\right\}\end{aligned}$$

A.3 Divide and conquer based shape discretizer

The python pseudo-code for the discretizer is shown below and makes use of a recursive geometric map-filling method which uses divide and conquer to efficiently compute the interior, exterior and boundary of any geometric shape stored in the *Shapely* geometric library format. Note that a minimal functional pseudo-code using *Numpy* has been presented here to facilitate understanding. Our actual code is more complex and allows working with PyTorch tensors on both CPU and GPU while also supporting batches of geometric objects. We also have other specialized versions (not shown here) which work faster for circular geometries.

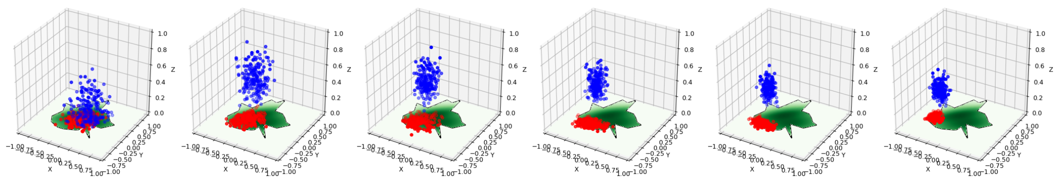
```
import numpy as np
from shapely.geometry import Polygon, Point


def get_g_map(geom, lims, deltas):
    ''' Computes the geometric maps from geometry.
    Args:
        geom: Shapely geometry object
        lims: Tuple (x_min, x_max, y_min, y_max) for generated
            geometric map
        deltas: Discretization bin size; tuple (delX, delY)

    Returns:
        g_map: numpy.ndarray of shape (nbinsX, nbinsY, 3)
        containing (interior, boundary, exterior) indicator of
        geometry in the third dimension.
    '''
    x_min, x_max, y_min, y_max = lims
    delX, delY = deltas
    nbinsX = round((x_max - x_min) / delX)
    nbinsY = round((y_max - y_min) / delY)

    g_map = np.zeros((nbinsX, nbinsY, 3)) # (int, bound, ext)
    fill(geom, g_map, 0, nbinsX, 0, nbinsY, lims, deltas)
    return g_map


def fill(geom, g_map, i1, i2, j1, j2, lims, deltas):
```

```
''' Fills g_map of shape (nbinsX, nbinsY, 3) with 1s at
    appropriate locations to indicate interior, exterior and
    boundary of the shape geom. This method makes recursive
    calls to itself and fills up the g_map tensor in-place.

Args:
    geom: A shapely.geometry object, e.g. Polygon
    g_map: A numpy.ndarray of shape (nbinsX, nbinsY, 3)
    i1: left x-coord of recursive rectangle to check against
    i2: right x-coord of recursive rectangle to check against
    j1: bottom y-coord of recursive rectangle to check against
    j2: top y-coord of recursive rectangle to check against
    lims: Tuple (x_min, x_max, y_min, y_max) for generated
        geometric map
    deltas: Discretization bin size; tuple (delX, delY)
'''
x_min, x_max, y_min, y_max = lims
delX, delY = deltas

box = Polygon([(x_min + i1*delX, y_min + j1*delY), \
               (x_min + i2*delX, y_min + j1*delY), \
               (x_min + i2*delX, y_min + j2*delY), \
               (x_min + i1*delX, y_min + j2*delY)])

if box.disjoint(geom):
    g_map[i1:i2, j1:j2, 2] = 1.0
elif box.within(geom):
    g_map[i1:i2, j1:j2, 0] = 1.0
else: # box.intersects(geom)
    if (i2 - i1 <= 1) and (j2 - j1 <= 1):
        g_map[i1:i2, j1:j2, 1] = 1
    elif (i2 - i1 <= 1) and (j2 - j1 > 1):
        j_mid = (j1 + j2) // 2
        fill(geom, g_map, i1, i2, j1, j_mid, lims, deltas)
        fill(geom, g_map, i1, i2, j_mid, j2, lims, deltas)
    elif (i2 - i1 > 1) and (j2 - j1 <= 1):
        i_mid = (i1 + i2) // 2
        fill(geom, g_map, i1, i_mid, j1, j2, lims, deltas)
        fill(geom, g_map, i_mid, i2, j1, j2, lims, deltas)
    else: # (i2 - i1 > 1) and (j2 - j1 > 1):
        i_mid = (i1 + i2) // 2
        j_mid = (j1 + j2) // 2
        fill(geom, g_map, i1, i_mid, j1, j_mid, lims, deltas)
        fill(geom, g_map, i_mid, i2, j1, j_mid, lims, deltas)
        fill(geom, g_map, i1, i_mid, j_mid, j2, lims, deltas)
        fill(geom, g_map, i_mid, i2, j_mid, j2, lims, deltas)
```

A.4 Mitigating local optima in best responses

During our preliminary experiments, we observed that learning to optimize resource locations or mixed strategies using purely gradient-based optimization can easily get stuck in local minima. While multiple re-runs in single-agent games can generate a reasonably good local minimum, in multi-agent games where the loss functions of agents are non-stationary due to changes in the other agents' mixed strategies, this leads to agents getting stuck in very sub-optimal local best responses. DeepFP maintains stochastic best responses to partially alleviate this issue, but doesn't completely mitigate it (for an example, see Figure 6).

While computing a global best response at every iteration of DeepFP can be costly (often infeasible), in practice it suffices to have a discontinuous exploration technique available in the best response



(a) Iter 0 (b) Iter 200 (c) Iter 400 (d) Iter 800 (e) Iter 1300 (f) Iter 1900

Figure 6: A sample sequence of iterations for DeepFP with $m = n = 1$ to demonstrate the attacker's best responses getting stuck in non-stationary local minima generated due to eventual adaptation by the defender; The drone (blue dots sampled from the defender's stochastic best response) eventually drives the lumberjack (red dots) into a corner from where it cannot cross over to other parts of the forest, because gradient-based optimization cannot jump over walls of high loss values.

update step. Hence, we propose a simple population-based approach wherein we maintain a set of K deterministic best responses $br_p^k(\sigma_{-p})$, for $p \in \{D, A\}$ and $\forall k \in [K]$. During the best response optimization step for agent p, we optimize the K best responses independently and play the one which exploits agent $-p$ the most. After the optimization step, the top $\frac{K}{2}$ best responses are retained while the bottom half are discarded and freshly initialized with random placements for the next iteration. This allows retention and further refinement of the current best responses over subsequent iterations, while discarding and replacing the ones stuck due to the opponent exploiting them. Since best responses get re-ranked every iteration, neither agent can excessively exploit a best response and cause the opponent to get stuck, because the opponent just switches to a different best response from its population in subsequent iterations.

A.5 Choosing population size K

Finally since the number of population members K is an important hyperparameter for our proposed approach, we show the effect on defender's exploitability by increasing K in Table 3. As expected, the exploitability decreases when using larger population sizes due to better exploration and finding more optimal (local) best responses while running DeepFP. Increasing K also reduces the variance of our metrics considerably. However using large population sizes also directly increases the computational burden and hence we have used $K = 4$ in all our experiments as a reasonable trade-off between achieving better metrics and having manageable run-times.

Table 3: Exploitability of defender for $m = n = 2$ averaged across forest instances with increasing population size K.

Variant	$\epsilon_D(\sigma_D)$
brnet	399.9488 ± 57.7006
pop1	348.9498 ± 98.4338
pop2	189.8122 ± 73.6444
pop4	141.0912 ± 13.8966
pop6	$\mathbf{127.9152 \pm 12.8323}$

A.6 Hyperparameters and model architectures

A.6.1 Learning differentiable reward models

While learning differentiable reward models with neural networks, we trained all networks for $10,000$ iterations with the Adam optimizer having learning rate 0.01 and a batch size of 64. The network architectures used are shown in Table 4.

A.6.2 DeepFP

For DeepFP, we run a total of 1000 outer fictitious play iterations and 100 inner optimization iterations to update best responses using the Adam optimizer with learning rate 0.001 and batch size 16. The network architecture for best response nets in *brnet* variant are shown in Table 5.

Table 4: Network architectures for reward models

Game	Net type	Structure
Areal Surveillance	*nn*	$\mathbb{R}^{m\times 3} \xrightarrow{fc,relu} \mathbb{R}^{128} \xrightarrow{fc,relu} \mathbb{R}^{512} \xrightarrow{fc,relu} \mathbb{R}^{128} \xrightarrow{fc,relu} \mathbb{R}^{1}$
Areal Surveillance	*gnn*	$\mathbb{R}^{m\times 3}, \_, \_ \xrightarrow[3\to32]{node\_enc} \mathbb{R}^{32}, \_, \_ \xrightarrow[64\to16]{edge\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \_$ $\xrightarrow[48\to32]{node\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \_ \xrightarrow[48\to16]{glob\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16}$ $\xrightarrow[96\to16]{edge\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16} \xrightarrow[64\to32]{node\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16}$ $\xrightarrow[64\to1]{glob\_net} \mathbb{R}^{1}$
Adversarial Coverage	*nn*	$\begin{tikzcd} \mathbb{R}^{m\times 3} \arrow[r, "{fc,relu}"] & \mathbb{R}^{128} \arrow[d, "cat"] & {} \arrow[r, "{fc,relu}"] & \mathbb{R}^{128} \arrow[r, "fc"] & \mathbb{R}^{1} \\ & \mathbb{R}^{256} \arrow[r, "{fc,relu}"] & \mathbb{R}^{512} \arrow[u] \arrow[d] & & \\ \mathbb{R}^{n\times 2} \arrow[r, "{fc,relu}"] & \mathbb{R}^{128} \arrow[u, "cat"'] & {} \arrow[r, "{fc,relu}"] & \mathbb{R}^{128} \arrow[r, "fc"] & \mathbb{R}^{1} \end{tikzcd}$
Adversarial Coverage	*gnn*	$\mathbb{R}^{(m+n)\times 3}, \_, \_ \xrightarrow[3\to64]{node\_enc} \mathbb{R}^{64}, \_, \_ \xrightarrow[128\to32]{edge\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \_$ $\xrightarrow[96\to64]{node\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \_ \xrightarrow[96\to32]{glob\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32}$ $\xrightarrow[192\to32]{edge\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32} \xrightarrow[128\to64]{node\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32}$ $\xrightarrow[128\to2]{glob\_net} \mathbb{R}^{2}$

Table 5: Network architectures for DeepFP *brnet* best responses

Net type	Structure
Defender's *brnet*	$\begin{tikzcd} & {} \arrow[r, "{fc,tanh}"] & \mathbb{R}^{m\times 2} \\ \mathbb{R}^{32} \arrow[r, "{fc,relu}"] & \mathbb{R}^{256} \arrow[u] \arrow[d] & \\ & {} \arrow[r, "{fc,relu}"] & \mathbb{R}^{m\times 1} \end{tikzcd}$
Attacker's *brnet*	$\mathbb{R}^{32} \xrightarrow{fc,relu} \mathbb{R}^{256} \xrightarrow{fc,tanh} \mathbb{R}^{n\times 2}$