A. GPL action-value computation

Here, we show for completeness that the expression for the learner's action value in Equation 9 is exactly the expectation introduced in Equation 4 when other agents policy is approximated by the agent model from Equation 8 and the joint action value factorizes as in Equation 5. The indices indicating the parameters of the action value function and the agent model are left out for brevity.

$$\begin{aligned}
\bar{Q}(s_t, a^i) &= \mathbb{E}_{a^{-i}\sim \boldsymbol{q}(.|s_t)}[Q(s_t, a)|a^i = a^i_t] \\
&= \sum_{a^{-i}\in A^{-i}} Q(s_t, a) q(a^{-i}|s_t, a^i) \\
&= \sum_{a^{-i}\in A^{-i}} (\sum_{a^j \in A_j} Q^j(a^j|s_t) \\
&\quad + \sum_{a^j\in A_j, a^k\in A_k} Q^{j,k}(a^j, a^k|s_t)) q(a^{-i}|s_t, a^i) \\
&= Q^i(a^i|s_t) \\
&\quad + \sum_{a^j\in A_j, j\neq i} \big(Q^j(a^j|s_t) + Q^{i,j}(a^i, a^j|s_t)\big) q(a^j|s_t) \\
&\quad + \sum_{a^j\in A_j, a^k\in A_k, j,k\neq i} Q^{j,k}(a^j, a^k|s_t) q(a^j|s_t) q(a^k|s_t),
\end{aligned} \tag{14}$$

where in the first step we used the definition of the expectation and in the second step we substituted in Equation 5 for the joint action value model. In the third step we substituted in the agent model from Equation 8 and marginalized out actions from the agent model q that are not part of the corresponding action value factor. Note that Equation 14 is valid regardless of the number of teammates in the environment.

B. Environment details

We provide additional details to reproduce the environments used in the experiments. Source code and installation instructions for these environments are included as part of the supplementary materials.

B.1. Wolfpack

In the Wolfpack environment, we train preys to avoid capture via the DQN algorithm (Mnih et al., 2015). Each prey receives an RGB image of the 17×25 grid centered around the its location. If the 17×25 image patch exceeds the boundaries of the grid world, we draw blue grid cells representing the grid world boundaries while also padding black colored grid cells outside the visualized boundaries to create the input image. The input image is subsequently used by a convolutional neural network to compute the action value estimates. We then train six convolutional neural networks using Independent DQN (Tampuu et al., 2017) to control two preys and four wolves. During training prey are given a penalty of -1 each time they are captured while the wolves follow the same reward structure used to train GPL agents in Wolfpack.

At the end of training value networks of the wolves are discarded and the value networks of the prey are used for our experiment. We finally provide Wolfpack as a single-agent reinforcement learning environment which can readily be used for open teamwork. An example image representing a state of the Wolfpack environment is provided in Figure 4a. In our Wolfpack implementation, other teammates and prey are treated as non-playable characters and models or heuristics controlling them are provided as part of the environment source code.

B.2. Level-based foraging

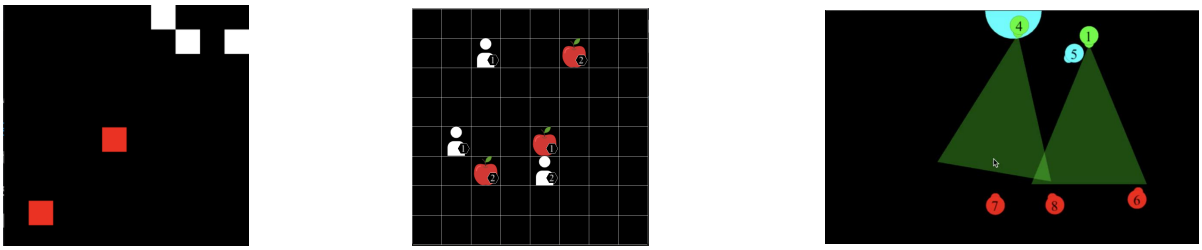
In the level-based foraging environment, levels of players and objects are sampled uniformly from the set $L = \{1, 2, 3\}$. The number of objects in the environment is set to three for each episode. Furthermore, initial locations of agents and objects are sampled uniformly from the available locations in the grid. An episode either terminates after 50 timesteps or after all objects have been collected. An image representing an example state of the level-based foraging game is provided in Figure 4b.

B.3. FortAttack

The FortAttack environment limits the agents to have a position between -0.8 and 0.8 for the horizontal axis coordinate, while the vertical axis coordinate is limited between -1 and 1. The fort is a semicircle centered on (0.0,1.0) with a radius of 0.3. When being initialized, the vertical axis coordinate of the attackers is initialized between -1 and -0.8 to make attackers start from a location that is far from the fort. On the other hand, defender's vertical axis coordinate is initialized between 0.8 and 1.

B.4. Teammate policies

We implement a diverse set of heuristics to control the teammates in Wolfpack and LBF. Further details of the heuristics used for both environments are provided in the following section. We provide empirical evidence showing that the set of heuristics is diverse and requires significant adaptation to achieve optimal performance by training an agent using the QL baseline against a specific type of teammate and evaluated the resulting policy against different types of teammates. We found that neither policies trained against a specific teammate nor a policy resulting from training against all possible types of teammates could reach the optimal performance against every teammate type.



(a) Wolfpack (b) Level-based foraging (c) FortAttack

Figure 4. An example screen capture of (a) Wolfpack, (b) level-based foraging, and (c) FortAttack. In Wolfpack, the white colored grid cells represent the players while the red ones represent the prey. With level-based foraging, the apple icons represent the location of objects to remove while the white icons represent the players that attempt to remove these objects. Levels of objects and players are visualized next to their respective icons. With FortAttack, The blue semicircle on the northern part of the 2D world is the fort. The blue, green, and red dots represents the learning agent, defenders, and attackers respectively. Agent 1 and 4 are visualized in the middle of executing the shooting action and the green triangles represents their shooting range.

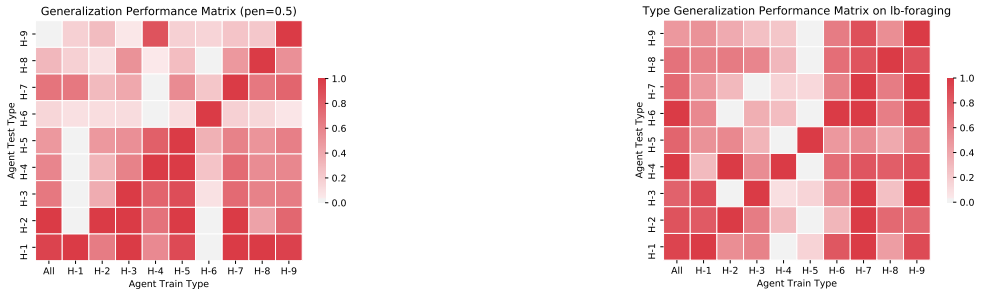
The results of this experiment for both environments are provided in Figure 5. As we have done in the team size generalization experiments, for each approach we periodically checkpoint the policies resulting from training and choose the checkpoint with the highest performance in training to be evaluated and reported in the heat matrix visualization. Figure 5 shows that even for policies trained against all types of agents, none of the resulting policies consistently achieves optimal performance for all teammate types.

On the other hand, for FortAttack we use the 5 pretrained policies provided by Deka & Sycara (2020). In the original work that proposed FortAttack, GNN-based networks were trained to create stochastic policies to control attacker and defenders. Deka & Sycara (2020) subsequently visually analyzed the resulting behaviour of the trained policies along different checkpoints and found different adopted by attackers and defenders during the training process. An example policy, which we eventually used as the different agent types for our experiments, was given for each type of strategy adopted by the attackers and defenders.

B.4.1. WOLFPACK

To create a diverse set of teammates for open ad hoc teamwork, we used the following mixture between heuristics proposed by Barrett et al. (2011) for the predator prey domain along with RL-based models to control teammates :

- **Random agent (H1):** The random agent chooses its action at any timestep by uniformly sampling the set of possible actions.
- **Greedy agent (H2):** The greedy agent chooses its action following the greedy predator heuristic provided in (Barrett et al., 2011). Intuitively, it sets the closest grid cell adjacent to the closest prey from its current location as its destination. It then chooses to move closer to the destination by moving along an axis for which it has the largest distance from the prey.
- **Greedy probabilistic agent (H3):** The greedy probabilistic agent chooses its action following the greedy probabilistic predator heuristic provided in (Barrett et al., 2011). The way it chooses its destination is the same with greedy agents. However, it randomly chooses one of the two available axis to move closer to the nearest prey. An agent's distance from the prey on each axis is provided as input to a boltzmann distribution to decide which axis should the agent move along.
- **Teammate aware agents (H4):** This agent follows the teammate aware predator heuristic from (Barrett et al., 2011). Intuitively, this heuristics assumes all teammates are using the same heuristic. It subsequently computes a hierarchy between agents based on their distance to their targeted preys. The hierarchy determines the sequence in which agents choose their actions. Agents must take into account the actions of agents higher up the hierarchy to avoid collision. An A^* planner is subsequently used to compute the action to reach the destination.
- **GNN-Based teammate aware agents (H5):** We train an RFM model with supervised learning to predict the actions taken by a group of teammate aware agents. This was done to avoid the possibly slow running time of the A^* planner in teammate aware agents. During interaction, it assumes that every agent is a teammate aware agent and passes their features along with prey locations as input to the network. Agent of this type subsequently uses the representation of its associated node as input to an MLP which has been trained to imitate the distribution over actions for teammate aware



(a) Wolfpack.

(b) Level-based foraging.

Figure 5. Generalization performance of a QL Agent trained to interact with a single teammate with a fixed type in both Wolfpack with a penalty of 0.5 (a) and level-based foraging (b). All experiments here are conducted using 4 seeds. The horizontal axis denotes the type of teammate encountered during training. The "All" versions are trained against a random teammate sampled uniformly over all possible types at the start of each episode. The vertical axis denotes the type of agent used to test the trained policies. Performance in each evaluation environment is scaled between zero and one by performing min-max scaling on each row of the heat matrix.

agents. Our agents subsequently samples the resulting distribution to decide their actions.

- **Graph DQN agents (H6):** We train an RFM-based controller trained by DQN to control a team of agents. It parses the state information following the input preprocessing method for GPL provided in Section C and provides it as input to an RFM. Node representations produced by the RFM are passed into an MLP to compute the action value function of the player associated to the node. Since this type only controls a single agent during interaction, only the action value associated to the controlled agent is used to take an action.

- **Greedy waiting agents (H7):** This heuristic is similar to greedy agents. However, agents are equipped with a waiting radius sampled randomly between three to five. The greedy heuristic is followed when either the Manhattan distance between the agent and closest prey is more than the waiting radius or when there is already another teammate inside the waiting radius of the closest prey. Otherwise, the agent will uniformly sample an action until the prey moves away or another teammate comes close to the prey.

- **Greedy probabilistic waiting agents (H8):** Similar to greedy waiting agents, agents of this type are equipped with a waiting radius sampled randomly between three to five. However, it is the greedy-probabilistic heuristic being followed when either the Manhattan distance between the agent and the targeted prey is more than the waiting radius or there is already another teammate inside the waiting radius.

- **Greedy team-aware waiting agents (H9):** Similar to greedy waiting agents, agents of this type are equipped with a waiting radius sampled randomly between three to five. However, it is the teammate aware heuristic being followed when either the Manhattan distance between the agent and the targeted prey is more than the waiting radius or there is already another teammate inside the waiting radius.

B.4.2. Level-based foraging

Similar to Wolfpack, we create a diverse set of teammate types for level-based foraging which requires agents to adapt their policies towards their teammates for achieving optimal performance. With level-based foraging, we use a mixture of heuristics (Albrecht & Ramamoorthy, 2013; Albrecht & Stone, 2017) and controllers trained using the A2C algorithm (Mnih et al., 2016) as our teammate policies. With the heuristic-based agents, their observations are limited to a square patch of grid cells with the agent's location being the center of the grid. The size of this observation square is uniformly sampled between 3×3, 5×5, or 7×7. Details of the different types of heuristics used in level-based foraging are provided below:

- **Heuristic H1:** This type of agent follows heuristic θ_j^{F2} proposed by Albrecht & Stone (2017) where agents under this heuristic follow the agent with the highest level if it observes another agent with a higher level than its own. If no agent has a higher level, it follows the farthest observable agent from their locations instead. The controlled agent then computes the object targeted by the leader agent if they follow heuristic H3 provided below and chooses an action that will get itself closer to the target object. If the agent is already next to the targeted object, it will choose to pick up the object. If the agent cannot follow the aforementioned rules due to not observing any objects in their observation square, it chooses the leader's position as its target instead. If no other teammates are observed, it uniformly samples an action from the set of possible actions instead.

- **Heuristic H2:** This type of agent follows heuristic θ_j^{F1} proposed by Albrecht & Stone (2017) where the controlled agent chooses a leader agent, assumes they follow certain heuristics to choose their targeted object, and gets itself closer to the object they think is targeted by the leader. Unlike H1, it chooses an observable agent with the farthest distance from itself as its leader. Furthermore, it assumes that the leader follows heuristic H4 provided below in choosing its target object. Otherwise, the way it chooses its actions when it is next to the targeted object is the same as in H1. Furthermore, its action selection method when there are no objects or teammate agents in its observation square follows that of H1.

- **Heuristic H3:** This type of agent follows heuristic θ_j^{L2} proposed by Albrecht & Stone (2017) where the controlled agent targets objects that have the highest level below its own level and gets closer to the object. If no objects in the set of visible objects are below its level, it chooses to target the item with the highest level instead. Otherwise, it uniformly samples actions from the set of possible actions when there are no objects observed in its observation square.

- **Heuristic H4:** This type of agent follows heuristic θ_j^{L1} proposed by Albrecht & Stone (2017) where the controlled agent targets the farthest visible object from its current location and gets closer to the object. When no objects are visible in its observation square, it samples actions uniformly from the set of possible actions.

- **A2C Agent (H5):** This type of agent is produced by independently training four agents together in level-based foraging using the A2C algorithm (Mnih et al., 2016). Among the four agents, we choose the one which has the highest performance compared to others as our A2C-based controller for this type. Also, agents of this type do not have observation squares but receive the whole state of the environment as input to their policy.

- **Heuristic H6:** This type of agent follows one of the heuristics proposed by Albrecht & Ramamoorthy (2013) for level-based foraging where agents always take actions that take them closer to the closest object in their observation square. If no objects exists, agents uniformly sample an action from the set of possible actions.

- **Heuristic H7:** This type of agent also follows one of the heuristics proposed by Albrecht & Ramamoorthy (2013). In choosing their actions, agents of this type go to the object inside the observation square which is closest to the center of all observed players. It follows heuristic H6 when no objects are observed in its observation square.

- **Heuristic H8:** This type of agent follows a heuristic proposed by Albrecht & Ramamoorthy (2013) where agents choose the closest object with the same level or lower than their own level as their target. If none such objects exists, the agent uniformly samples an action from its action space.

- **Heuristic H9:** This type of agent follows a heuristic proposed by Albrecht & Ramamoorthy (2013) where it scans its surrounding for observable target objects that has at most the same level as the sum of all observable agents' levels. It then computes the center of all agents' locations and chooses a target object with the least distance to the center of observable agents' location as its destination. If no possible target exists, it uniformly samples an action from its action space.

B.4.3. FORTATTACK

The behavior of attackers as well as defenders not under our control is determined by policies obtained in the experiments from Deka & Sycara (2020). The policies show distinct behavioral patterns summarized below. More information on how these policies were obtained can be found in Deka & Sycara (2020).

- **Guard Type 1 - Random guard:** For this agent, we randomly initialize a neural network and used it as the policy network for the guards.

- **Guard Type 2 - Flash laser:** Defenders position themselves in front of the fort and flash their lasers continually. This behavior is independent of the movement of the attackers.

- **Guard Type 3 - Spread out Flash laser:** Similar to type 1 but instead of positioning themselves in front of the fort the defenders spread out across the whole width of the environment.

- **Guard Type 4 - Smart spreading:** Defenders spread smartly across the defensive zone and only shoot to kill attackers.

- **Attacker Type 1 - Sneak:** Attackers spread out across the whole environment to maximize the likelihood of finding an open space in the defensive line.

- **Attacker Type 2 - Deceive:** Attackers split up their attack. That is if the majority of attackers come from the right, one attacker will try to sneak beyond the defenders from the other side.

- **Guard Type 6 - Ensemble-trained agents:** Guards of this type are trained by letting it interact against Attackers that are uniformly sampled from Attacker type 1 and type 2.

C. GPL Input preprocessing

As input to GPL, for each agent the observation is parsed into a set of vectors containing agent specific information, x, concatenated with shared state information, u, to create an input batch B. The agent specific information generally contains locations of the agents in both environments. In level-based foraging, information about an agent's level is also included in x. Other remaining information such as prey locations in Wolfpack or object location and level in level-based foraging are included in u. B is then passed to an LSTM along with the previous hidden states, θ, and cell states, c, of an LSTM to compute the embedding of each agent required by the models defined in GPL.

To deal with the changing batch size of B across time as a result of environment openness, the hidden and cell state are further processed inbetween timesteps. Assuming i_t and d_t correspond to the sets of added and removed agents at time t, f_{rem} removes the states associated to agents leaving the environment while f_{ins} inputs a zero vector for the states associated to agents joining the environment. We formally define this hidden and cell state processing following Equation 15 while additionally providing an example illustration of this processing method in Figure 6. Finally, since we assume that the hidden and cell states associated to all agents including the learning agent should be reset to **0** at the end of each episode, we define d_t as the set of all existing agents and i_t as the set of agents at the initial state of the following episode each time an episode ends.

$$Prep(\theta_t, c_t) = f_{ins}(f_{rem}(\theta_t, c_t, d_t), i_t) \tag{15}$$

D. GPL pseudocode

Before we describe the full GPL pseudocode, we first define important functions that we will use in the pseudocode. First, we denote the observation and hidden vector preprocessing method described in Appendix C as the **PREPROCESS** function. Furthermore, we denote the action-value and joint-action value computation through Equation (9) and (5) as the **MARGINALIZE** and **JOINTACTEVAL** functions respectively. Based on these functions, we define the **QV** function that preprocesses the input and computes the action-values for given joint-action value and agent networks. The computations in **QV** is provided in Algorithm 1.

Aside from these functions, we define **QJOINT** and **PTEAM**, which output is required to compute the loss functions, $L_{\beta,\delta}$ and $L_{\eta,\zeta}$, in Equation (11) and (10). **QJOINT** is a function that computes the predicted joint action value for an observed state and joint actions. On the other hand, **PTEAM** computes the joint teammate action probability at a state. Both **QJOINT** and **PTEAM** are further defined in Algorithm 2 and 3.

Algorithm 1 GPL Action Value Computation

Input: state s,
joint-action value model parameters $(\alpha_Q, \beta, \delta)$,
agent model parameters (α_q, η, ζ),
agent model LSTM hidden vectors $h_{t-1,q}$,
joint-action value model LSTM hidden vectors $h_{t-1,Q}$
function QV$(s, \alpha_Q, \alpha_q, \beta, \delta, \eta, \zeta, h_{t-1,Q}, h_{t-1,q})$
$\mathrm{B}, \theta_Q, c_Q \leftarrow$ **PREPROCESS**$(s, h_{t-1,Q})$
$\mathrm{B}, \theta_q, c_q \leftarrow$ **PREPROCESS**$(s, h_{t-1,q})$
$\theta'_Q, c'_Q \leftarrow \mathrm{LSTM}_{\alpha_Q}(B, \theta_Q, c_Q)$
$\theta'_q, c'_q \leftarrow \mathrm{LSTM}_{\alpha_q}(B, \theta_q, c_q)$
$\forall j, \bar{n}_j \leftarrow (RFM_\zeta(\theta'_q, c'_q))_j$
$\forall j, q_{\eta,\zeta}(.|s_t) \leftarrow \mathrm{Softmax}(\mathrm{MLP}_\eta(\bar{n}_j))$
$\forall j, a^j, Q^j_\beta(a^j|s_t) \leftarrow \mathrm{MLP}_\beta(\theta'^j_Q, \theta'^i_Q)(a^j)$
$\forall j, a^j, a^k,$

$$Q^{j,k}_\delta(a^j, a^k|s_t) \leftarrow \mathrm{MLP}_\delta(\theta'^j_Q, \theta'^k_Q, \theta'^i_Q)(a^j, a^k)$$

Compute $\bar{Q}(s, a^i)$ using Equation (9)
$\bar{Q}(s, .) \leftarrow$ **MARGINALIZE**(

$$q_{\eta,\zeta}(.|s_t), Q_\beta(.|s_t)), Q_\delta(., .|s_t)$$

)
return $\bar{Q}(s, .), (\theta'_Q, c'_Q), (\theta'_q, c'_q)$
end function

Algorithm 2 GPL Joint-Action Value Computation

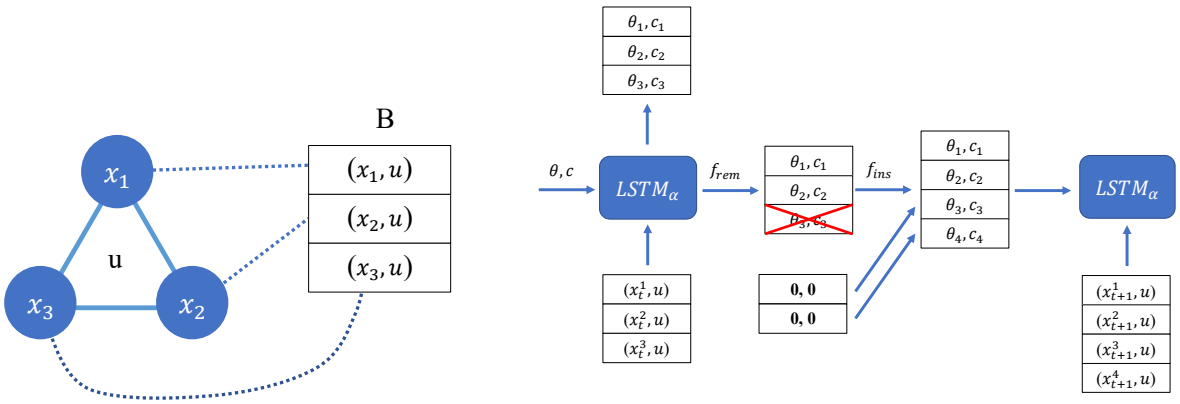
Input: state s, observed joint action a,
joint-action value model parameters $(\alpha_Q, \beta, \delta)$,
joint-action value model LSTM hidden vectors $h_{t-1,Q}$
function QJOINT$(s, a, \alpha_Q, \beta, \delta, h_{t-1,Q})$
$\mathrm{B}, \theta_Q, c_Q \leftarrow$ **PREPROCESS**$(s, h_{t-1,Q})$
$\theta'_Q, c'_Q \leftarrow \mathrm{LSTM}_{\alpha_Q}(B, \theta_Q, c_Q)$
$\forall j, a^j, Q^j_\beta(a^j|s_t) \leftarrow \mathrm{MLP}_\beta(\theta'^j_Q, \theta'^i_Q)(a^j)$
$\forall j, a^j, a^k,$

$$Q^{j,k}_\delta(a^j, a^k|s_t) \leftarrow \mathrm{MLP}_\delta(\theta'^j_Q, \theta'^k_Q, \theta'^i_Q)(a^j, a^k)$$

Compute $Q(s, a)$ using Equation (5)
$Q(s, a) \leftarrow$ **JOINTACTEVAL**(

$$a, q_{\eta,\zeta}(.|s_t), Q_\beta(.|s_t)), Q_\delta(., .|s_t)$$

)
return $Q(s, a)$
end function



(a) Observation preprocessing.

(b) LSTM hidden vector preprocessing.

Figure 6. The figure shows (a) the preprocessing of observation information into input for the GPL algorithm along with (b) the additional processing steps done to the agent embedding vectors to handle environment openness. Part (b) shows an example processing step where agent 3 is removed from the environment and two new agents joins the environment.

Algorithm 3 GPL Teammate Action Probability Computation

Input: state s, observed joint actions a,
agent model parameters (α_q, η, ζ),
agent model LSTM hidden vectors $h_{t-1,q}$
function PTEAM$(s, a, \alpha_q, \eta, \zeta, h_{t-1,q})$
$\mathrm{B}, \theta_q, c_q \leftarrow$ **PREPROCESS**$(s, h_{t-1,q})$
$\theta'_q, c'_q \leftarrow \mathrm{LSTM}_{\alpha_q}(B, \theta_q, c_q)$
$\forall j, \bar{n}_j \leftarrow (RFM_\zeta(\theta'_q, c'_q))_j$
$\forall j, q^j_{\eta,\zeta}(.|s) \leftarrow \mathrm{Softmax}(\mathrm{MLP}_\eta(\bar{n}_j))$
$q_{\eta,\zeta}(a^{-i}|s, a^i) \leftarrow \prod_{j \in -i} q^j_{\eta,\zeta}(a^j|s)$
return $q_{\eta,\zeta}(a^{-i}|s, a^i)$
end function

Using the functions we previously defined, we finally describe GPL's training algorithm. GPL collects experience from parallel environments through the modified Asynchronous Q-Learning framework (Mnih et al., 2016) where asynchronous data collection is replaced with a synchronous data collection instead. Despite this, it is relatively straightforward to modify the pseudocode to use an experience replay instead of a synchronous process for data collection. As in the case of existing deep value-based RL approaches, we also use a separate target network whose parameters are periodically copied from the joint action value model to compute the target values required for optimizing Equation 11. We finally optimize the model parameters in the pseudocode to optimize the loss function provided in Section 4.2 using gradient descent. GPL's training process is finally described in Algorithm 4.

E. Training Details

In this section we provide additional details about the training setup. First, we describe the way we add and remove agents in our open teamwork experiments. Secondly, we also provide details of the input padding method used by the baseline algorithms. Finally, we provide details of the architecture used in our experiments along with the hyperparameters used for training.

E.1. Environment openness

To induce environment openness in Wolfpack and LBF, we sample the duration for which an agent exists in the environment, along with the waiting duration required for an agent which is removed from the environment to get added to the environment. For Wolfpack, the active duration is sampled uniformly between 25 and 35 timesteps while the waiting duration is sampled uniformly between 15 and 25 timesteps. For level-based foraging, the active duration is sampled uniformly between 15 to 25 timesteps while the waiting duration is sampled uniformly between 10 to 20 timesteps.

In both environments, active duration is designed to be longer than waiting duration to create environments with large team sizes during interaction. On the other hand, both active and waiting duration for level-based foraging is less than its Wolfpack counterpart since the objects collected in level-based foraging remain stationary which causes the task to require less time to solve compared to Wolfpack. As a consequence, an episode might finish early and using larger active and waiting durations might cause agents to not be added or removed from the environment at all.

On the other hand, openness in FortAttack is solely induced

Algorithm 4 GPL Training

Input: Number of training steps T, time between updates t_{update}, time between target network updates t_{targ_update}.
Initialize the joint-action value model parameters, α_Q, β, δ.
Initialize the agent model parameters, α_q, η, ζ.
Create target joint-action value networks.

$$\alpha'_Q, \beta', \delta' \leftarrow \alpha_Q, \beta, \delta$$

$\theta_Q, c_Q, \theta_Q^{targ}, c_Q^{targ} \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}$
$\theta_q, c_q \leftarrow \mathbf{0}, \mathbf{0}$
$d\alpha_Q, d\alpha_q, d\beta, d\delta, d\eta, d\zeta \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}$
Observe s from environment
for $t = 1$ **to** T **do**
$h_Q, h_q, h_Q^{targ} \leftarrow (\theta_Q, c_Q), (\theta_q, c_q), (\theta_Q^{targ}, c_Q^{targ})$
$\bar{Q}(s, .), h'_Q, h'_q \leftarrow \mathbf{QV}(s, \alpha_Q, \alpha_q, \beta, \delta, \eta, \zeta, h_Q, h_q)$
Sample action according to the learning algorithm being used,

$$a_t^i \sim \begin{cases} \text{eps-greedy}(\epsilon, \bar{Q}(s, .)), & \text{if Q-Learning} \\ p_{\text{SPI}}(\bar{Q}(s, .), \tau) & \text{if SPI} \end{cases}$$

Execute a^i and observe a, r and s'.
Compute predicted joint-action value for a_t,

$$Q_{\beta,\delta}(s, a) \leftarrow \mathbf{QJOINT}(s, a, \alpha_Q, \beta, \delta, h_Q)$$

Compute action-value of next state using target network.

$$\bar{Q}'\left(s, a^i\right), h_Q^{targ}, \_ \leftarrow \mathbf{QV}(s', \alpha'_Q, \alpha_q, \beta', \delta', \eta, \zeta, h_Q^{targ}, h'_q)$$

Compute target value for updating the joint-action value model with,

$$y\left(r, s'\right) = r + \gamma \text{max}_{a^i} \bar{Q}'\left(s', a^i\right),$$

if Q-Learning is used, or

$$y\left(r, s'\right) = r + \gamma \textstyle\sum_{a^i} p_{\text{SPI}}(a^i|s') \bar{Q}'\left(s', a^i\right),$$

if using SPI.
Compute predicted action probabilities of teammates using the agent models,

$$q_{\eta,\zeta}(a^{-i}|s, a^i) \leftarrow \mathbf{PTEAM}(s, a, \alpha_q, \eta, \zeta, h_q)$$

Using $Q_{\beta,\delta}(s_t, a_t), y\left(r_t, s_{t+1}\right)$, and $q_{\eta,\zeta}(a^{-i}|s, a^i)$, compute $L_{\zeta,\eta}$ and $L_{\beta,\delta}$ with Equation (10) and (11).
Accumulate parameter gradients for updates

$$d\alpha_Q = d\alpha_Q + \nabla_{\alpha_Q} L_{\beta,\delta}, d\alpha_q = d\alpha_q + \nabla_{\alpha_q} L_{\eta,\zeta}$$
$$d\beta = d\beta + \nabla_\beta L_{\beta,\delta}, d\delta = d\delta + \nabla_\delta L_{\beta,\delta}$$
$$d\eta = d\eta + \nabla_\eta L_{\eta,\zeta}, d\zeta = d\zeta + \nabla_\zeta L_{\eta,\zeta}$$

if t mod $t_{\text{update}} = 0$ **then**
Update $\alpha_Q, \alpha_q, \beta, \delta, \eta, \zeta$ using gradient descent based on $d\alpha_Q, d\alpha_q, d\beta, d\delta, d\eta, d\zeta$.
$d\alpha_Q, d\alpha_q, d\beta, d\delta, d\eta, d\zeta \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}$
end if
if t mod $t_{\text{targ\_update}} = 0$ **then**
$\alpha'_Q, \beta', \delta' \leftarrow \alpha_Q, \beta, \delta$
end if
$(\theta_Q, c_Q), (\theta_q, c_q), s \leftarrow h'_Q, h'_q, s'$
end for

by agents getting destroyed or respawned. When an agent is destroyed, its distance to the agent that shot it down determines the waiting time required before it can reenter the environment. Specifically, we measure the distance between the destroyed agent and the shooter agent and linearly interpolate between 0 and 80 timesteps and round it to the closest integer to determine the waiting time. Agents are out for longer as they get closer to the shooter when they die.

Finally, for each environment, we uniformly sample the type of a teammate from the pool of possible teammate types when they are respawned. To provide different open processes for training and generalization, we vary the upper limit for team sizes in LBF, Wolfpack, and the attacking & defending teams in FortAttack. During training, the team size is limited to up to 3 agents. On the other hand, the upper limit is increased to 5 agents for each team in the generalization scenario.

E.2. Baseline input preprocessing

For our baseline approaches, in Section 5.2 we mentioned about padding the input vectors. We specifically use a placeholder value of -1 for features associated to inactive agents. Furthermore, since our generalization experiments can have up to five agents in the environment, these placeholders are added to the input until the length of the input vector corresponds to inputs of an environment with five agents. Adding placeholder values of -1 also applies to the predicted action probability concatenated to the input vectors for the QL-AM baseline.

We also need to ensure that a feature is not always assigned a placeholder value during training to prevent its associated model parameters from not being able to generalize when encountering input vectors which do not have placeholder values assigned to the feature. To handle this, we randomly assign teammates an integer index between one and four when they are added to the environment. We prevent different teammates from having the same index and use it to determine the location of their features in the input vector. This index remains the same while an agent is active in the environment. As a result, all features are assigned a non-placeholder value at some point during training.

Aside from concatenating the predicted action probabilities to the input vector, we tried the approach proposed by Tacchetti et al. (2019) which maps the output of the agent model into an RGB image which contains information about the probability of an agent being positioned at certain grid cells following the action it might execute at the current state. We use the same convolutional neural network architecture used in their work to produce a fixed-length embedding of the image. This embedding is subsequently concatenated to the input vector and passed as input to a deep RL approach. However, we decided not to use this approach as a baseline due to the following reasons:

- Our experiments indicate no improvement in performance in the 6.4 million steps we used to train our approaches. Due to the increased number of parameters introduced by the convolutional neural network, more training steps might be required and we have not found the number of steps that works for this approach yet.
- A fair comparison against GPL might be difficult since GPL does not receive RGB images as input.
- Actions in level-based foraging which lead to the same teammate location in the next state such as staying still and retrieving the object cannot be straightforwardly mapped into an RGB image following the mapping approach proposed by Tacchetti et al. (2019).

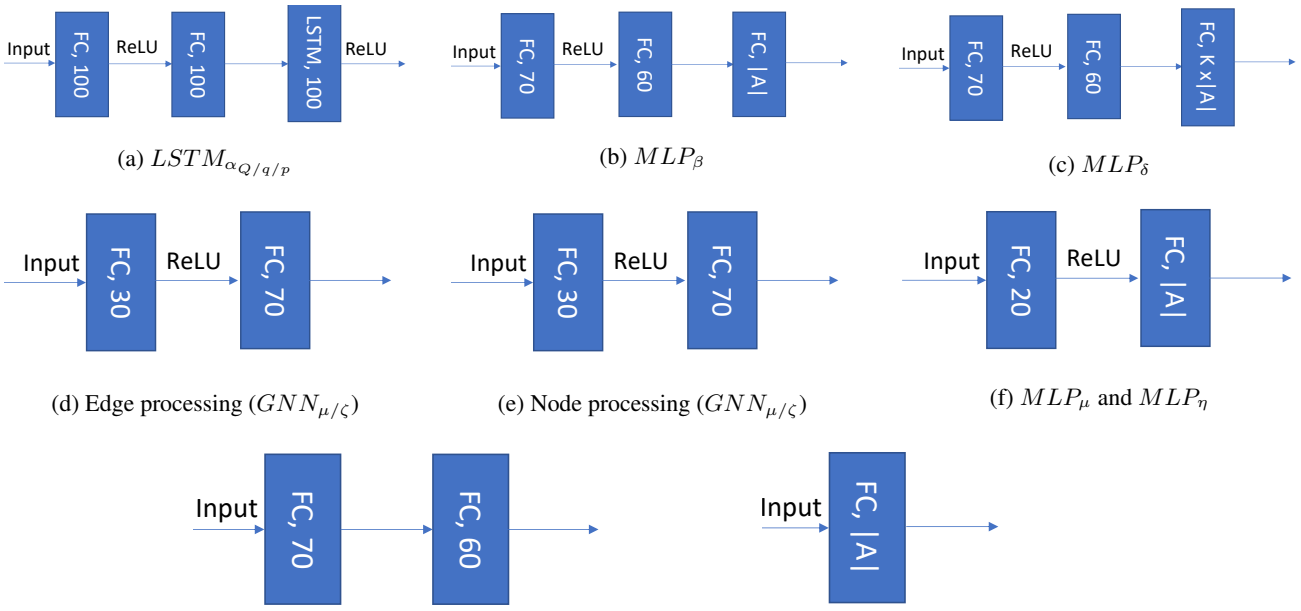
Nonetheless, we view the approach that concatenates predicted action probabilities to the input vector as an adequate representative of approaches that augment the input with teammate information and relies on a non-linear function approximation to learn a policy or value estimate for the agent.

E.3. Hyperparameters and network architecture

Details of the neural network architectures used by GPL in Wolfpack and LBF are provided in Figure 7. Before being processed by the LSTM, the type embedding network passes the input through two fully connected layers. Results from the embedding network are subsequently passed to the GPL component that has the type embedding as input. For the joint action value computation, the singular and pairwise utility computation utilizes an architecture provided in Figure 7b and Figure 7c. The agent and auxiliary agent models follows the architecture provided in Figure 7d, Figure 7e, and Figure 7f.

To allow a fair comparison between the baselines and GPL, the model architecture used by baselines in Wolfpack and LBF follows the architecture used by GPL. Specifically, baselines pass their input vectors to the architecture in Figure 7a and subsequently passes the output to an architecture following Figure 7b to compute the action values. For baselines that use agent and auxiliary agent models, the architecture of these models follows the same architecture provided by Figure 7d, Figure 7e, and Figure 7f.

For MADDPG and DGN, we use networks with similar sizes to those used in open ad hoc teamwork experiments. The only difference is we do not use LSTMs in the network since there is no need for type inference in the MARL approaches. As a result, the architecture used by DGN is simply the



(a) $LSTM_{\alpha_{Q/q/p}}$ (b) MLP_{β} (c) MLP_{δ}

(d) Edge processing ($GNN_{\mu/\zeta}$) (e) Node processing ($GNN_{\mu/\zeta}$) (f) MLP_{μ} and MLP_{η}

(g) Key, query, and value computation. (h) Action value computation.

Figure 7. **GPL and baseline architecture details for LBF and Wolfpack:** We provide details of the architecture used in GPL's type embedding network (a), singular utility computation (b), pairwise utility computation (c), edge embedding computation in the agent and auxiliary agent model (d), node embedding computation (e) in the agent and auxiliary agent model, the MLP used by the agent and auxiliary agent model to process the resulting GNN node embeddings (f), the size of the MLPs used for computing the key, query, and value for multihead attention in GNN-QL and GNN-QL-AM (g), and the final layer used in action value computation for GNN-QL & GNN-QL-AM. In all images, **FC** denotes a fully connected layer, **LSTM** denotes an LSTM layer, and the accompanying number denotes the size of the layer. Labels on the arrows indicate the non-linear functions used between the layers while no labels indicate no non-linear functions being applied to the resulting output vectors. With the baselines, we combine the architectures in (a) and (b) to compute the action values while the agent and auxiliary agent model used in some of the baselines follow (d), (e), and (f). With GNN-QL and GNN-QL-AM, the layer defined by (g) and (h) is subsequently used after (a) and (b) to compute the action values.

architecture used by GNN with the LSTM layers replaced with an MLP with the same output size. The decentralized policy for MADDPG is also similar with the value network architecture of QL with the LSTM replaced by an MLP with the same output size.

For training, we use the following hyperparameters for GPL and all proposed baselines:

- $K = 5$ for the low rank factorization of pairwise utility terms in GPL.
- 8 attention heads were used for DGN, GNN-QL and GNN-QL-AM.
- Data is collected from 16 parallel environments to collect a total experience of 6.4 million environment steps.
- Models are optimized using the Adam optimization algorithm with a learning rate of 2.5×10^{-4}.
- Models are updated every 4 steps on the parallel environment.
- Instead of updating the target networks by periodically copying the joint action value network, target networks are updated using a weighted average of the parameters of the joint action value network. Assuming that ϕ is a parameter of the joint action value network, we update the corresponding parameter in the target network $\phi^{'}$ by using $\phi^{'} \leftarrow (1-\alpha)\phi^{'} + \alpha\phi$, with α set to 10^{-3}.
- ϵ for the exploration policy is linearly annealed from 1 to 0.05 in the first 4.8 million environment steps and remains the same afterwards.
- We also use an attention weight regularization term, λ, of 0.03 for DGN and a temperature of 0.1 for MADDPG's gumbel softmax function.

These hyperparameters and network architectures are obtained by initially searching for a network and hyperparameter configuration which works for QL. After finding an architecture along with a set of hyperparameters which works best for QL, we train a similar sized architecture with the same hyperparameters for the rest of the baselines.

Despite our lack of hyperparameter tuning for the rest of the baselines, we still obtain better performance than QL in almost all cases.

For FortAttack, we initially started the training process with the same hyperparameter setup as LBF and Wolfpack. Due to QL not learning, we decided to increase the size of the network along with running the training process for more timesteps to take into account of the increased complexity of the environment compared to Wolfpack and LBF. Nonetheless, we did not find any success in training QL regardless of the different network sizes and hyperparameters we tried.

We subsequently focused on finding a network architecture along with hyperparameters that worked for GPL. A similar sized network with GPL along with GPL's training hyperparameters were used for training other baselines. This resulted in the following architecture and hyperparameters for FortAttack:

- $K = 6$ for the low rank factorization of pairwise utility terms in GPL.
- 8 attention heads were used for DGN, GNN-QL and GNN-QL-AM.
- Data was collected from 16 parallel environments to collect a total experience of 16 million environment steps.
- Models were optimized using the Adam optimization algorithm with a learning rate of 1.0×10^{-4}.
- Models were updated every 4 steps on the parallel environment.
- Instead of updating the target networks by periodically copying the joint action value network, target networks were updated using a weighted average of the parameters of the joint action value network. Assuming that ϕ is a parameter of the joint action value network, we updated the corresponding parameter in the target network $\phi^{'}$ by using $\phi^{'} \leftarrow (1-\alpha)\phi^{'} + \alpha\phi$, with α set to 10^{-3}.
- The exploration parameter ϵ was linearly annealed from 1 to 0.05 in the first 8 million environment steps and remains the same afterwards.
- 64 units were used in the fully connected and LSTM layer for $LSTM_{\alpha_{Q/q/p}}$. For comparison, in Wolfpack and LBF we used 100 units.
- 128 units were used for both fully connected layers in MLP_{β} and MLP_{δ}. For comparison, we used 70 and 60 units respectively in Wolfpack and LBF.
- 40 and 70 units were used respectively for the first and second layer in the edge and node processing network ($GNN_{\mu/\varsigma}$) of the agent model. For comparison, we used 30 and 70 units respectively for Wolfpack and LBF.
- We used 128 units in MLP_{μ} and MLP_{ν}. In comparison, we used 20 units for Wolfpack and LBF.
- Finally, we used 128 units for both fully connected layers involved in the key, query, and value computation under GNN-QL and GNN-QL-AM. For comparison, we used 70 and 60 units respectively for Wolfpack and LBF.

Finally the way these components were assembled into the architectures of the baselines was still the same between FortAttack and the other environments.

F. Additional MARL Training Results

In this section, we provide additional results from the MARL algorithms we used in our open ad hoc training when interacting against teammates that are jointly trained with itself.

F.1. MARL training performance against jointly trained teammates

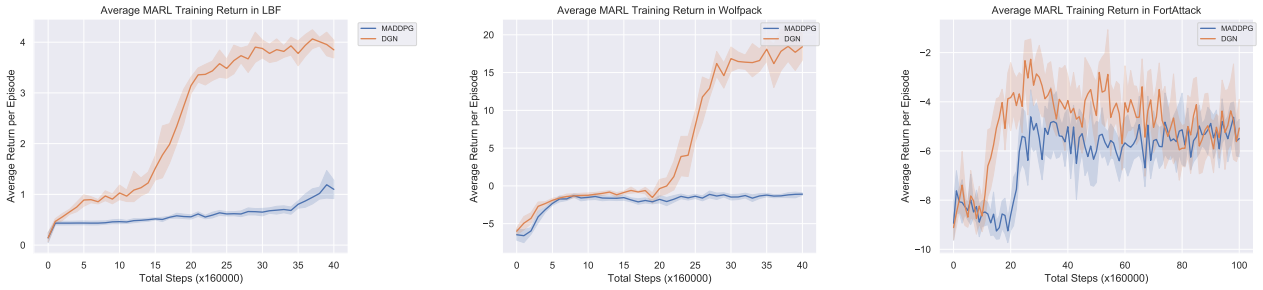
Following training that we did to create the MARL policies we evaluated in Section 5.5, we evaluate the MARL policies in an open process that is similar to the open process used in the training setup in Section 5.5. The only difference with the open process used in the training setup in Section 5.5 is that the sampled teammates are always agents that are jointly trained with the MARL agent. From the results provided in Figure 8, we see that the MARL baselines achieved better performances than when they interact against our ad hoc teamwork teammates.

F.2. MARL generalization performance against jointly trained teammates

In this section, we describe the performance of the MARL policies when the open process in Section F.1 is changed such that the upper limit on team size is five agents for LBF, Wolfpack, and attacker & defender teams in FortAttack. The results are provided in Table 3.

G. Team size & Fixed type generalization results

To further demonstrate that GPL outperforms the single-agent baselines in open ad hoc teamwork, we train GPL under different open processes than what we used in our



(a) Results in Level-based foraging (b) Results in Wolfpack (c) Results in FortAttack

Figure 8. **Open MARL results (training) against jointly trained agents:** Average and 95% confidence bounds of MADDPG and DGN returns during training (up to 3 agents in a team for LBF, Wolfpack, and attacker & defender teams in FortAttack). For each algorithm, training is done using eight different seeds and the resulting models are saved and evaluated every 160000 global steps. Compared to Figure 2, we see that the MARL baselines achieved better performance when interacting against jointly trained teammates than when they interacted with the unknown policies during open ad hoc teamwork.

Environment	MADDPG	DGN
LBF	0.64 ± 0.15	1.65 ± 0.25
Wolfpack	-1.15 ± 0.20	9.36 ± 2.46
FortAttack	0.03 ± 1.03	-8.13 ± 0.98

Table 3. **Open MARL results (generalization) against jointly trained agents:** Average and 95% confidence bounds of MADDPG and DGN returns during generalization (up to 5 agents in a team for LBF, Wolfpack, and attacker & defender teams in FortAttack). For each algorithm, training is done using eight different seeds and the resulting models are saved and evaluated every 160000 global steps. Among all saved policies, we choose the checkpoint that has the highest average performance during training (Section F.1) and report the mean and 95% confidence bounds of the performance of the policies at that checkpoint.

main experiment in Section 5.5. We exclude MARL baselines from these experiments since our main experiment shows that they consistently perform worse than our worst performing single-agent RL baselines. For the first experiment, we train agents in LBF, Wolfpack, and FortAttack under an open process where the learner's team consists of two agents during training.

The other experiment trains agents in FortAttack under two open processes where the types of teammates remains fixed during interaction. In the first open process, defender teammate type is fixed towards guard type 4 mentioned in Section B.4.3. By contrast, the second open process sets the defender types as guard type 6. The attackers in both open processes are configured to have attacker type 2.

Figure 9 shows the results for the fixed team size experiments. On the other hand, Figure 10 provides the resulting performance from training the learner against teammates of fixed types. The results further demonstrates GPL's superior performance to baselines for open ad hoc teamwork.

H. Wolfpack action value analysis

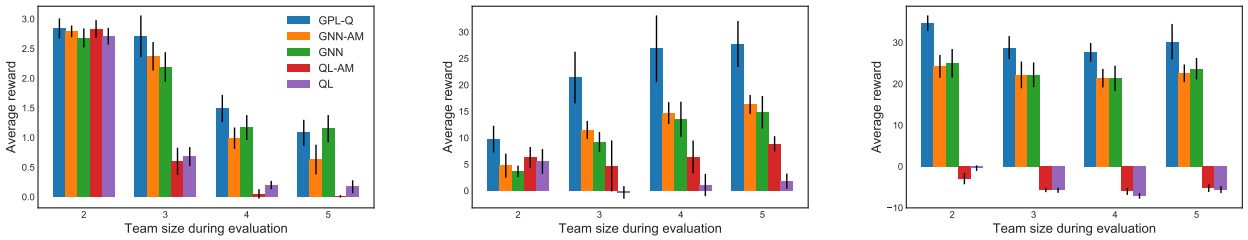
We provide an analysis of the individual and joint action values learned by GPL in Wolfpack. For this analysis, we still use $\bar{Q}_j$ as defined in Section 5.6. On the other hand, since the Wolfpack environment has no shooting actions, we replace $\bar{Q}_{j,k}$ with $C_{j,k}(a)$ that we define in Equation 16.

$$C_{j,k}(a) = Q_i^{j,k}(a^j, a^k|s) - N_{j,k}(a) \tag{16}$$

$$N_{j,k}(a) = \frac{\sum_{x,y \in A, x \neq a_j, y \neq a_k} Q_i^{j,k}(x, y|s)}{|A|^2 - 1} \tag{17}$$

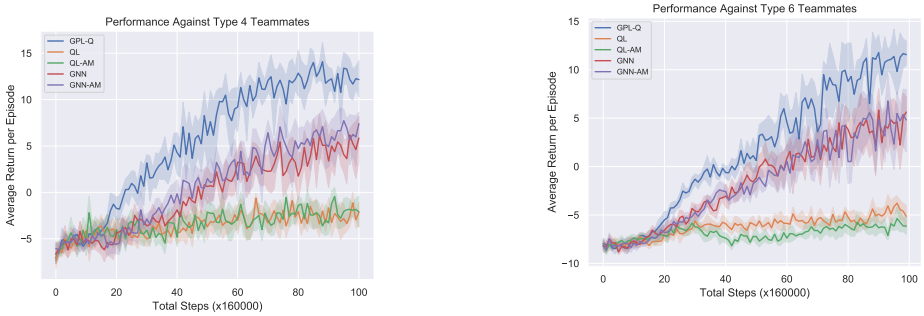
$C_{j,k}(a)$ denotes the difference between the pairwise action value associated to the action chosen by the agents compared to the average pairwise action value of other pairs of actions it could have taken. In our analysis, we only investigate $C_{j,k}(a)$ for observations that precede the capture of a prey by the learning agent. We further filter out values $C_{j,k}(a)$ for which j and k do not belong to the pack of wolves that successfully captured a prey together with the learning agent. By investigating this, we wanted to see whether GPL assigns a higher pairwise action value towards pairs of actions that lead to a joint prey capture.

The result of our analysis is provided in Figure 11. We find that GPL assigns higher individual action values for a teammate as they get closer towards the prey that the learning agent is hunting. This is a reasonable value assignment since teammates can better help the learning agent hunt as they get closer to the target prey. On the other hand, we also see that GPL progressively learns to increase the difference in pairwise action values between pairs of actions that lead to a capture and other alternative pairs of actions. Therefore, GPL helps the learning agent to understand the positive contribution resulting from pairs of actions leading towards a capture.



(a) Results in level-based foraging (b) Results in Wolfpack with 0.5 penalty (c) Results in FortAttack

Figure 9. **Team size generalization results:** Average rewards and 95% confidence bounds of GPL and baselines from team size generalization experiments collected over eight seeds. Agent was trained to interact in a team of two agents for LBF, Wolfpack, and FortAttack. With FortAttack, we used a setup where the number of attackers was always equal to the number of teammate defenders. Value networks are stored every 160000 steps and the performance of greedy policies from value networks stored at the checkpoint with the highest average performance during training were used to compute the average returns and their 95% confidence interval. This result shows that GPL-Q still outperforms other single-agent RL baselines, except for LBF where its performance is not significantly different from GNN and GNN-AM.



(a) Performance against Type 4 teammates in FortAttack. (b) Performance against type 6 teammates in FortAttack.

Figure 10. **Fixed type experiment results:** Average rewards and 95% confidence bounds of GPL-Q and baselines from fixed type experiments collected over eight seeds in FortAttack. Agent was trained to interact in a team of up to 3 agents with all teammates having a fixed type. Value networks are stored every 160000 steps and the performance of greedy policies from value networks stored at the checkpoint with the highest average performance during training were used to compute the average returns and their 95% confidence interval. The results illustrates the performance of the methods when trained against (a) type 4 teammates and (b) type 6 teammates.

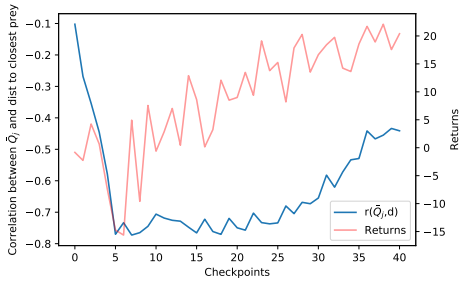
I. Baseline action-value analysis

In this section, we analyze the action-value estimates produced by the single-agent RL baselines used in our experiments. Despite recognizing the value of shooting, we show that the baselines do not learn the effects of other agents' action towards the learner as GPL does. This results in the significant performance gap between GPL and the baselines in our experiments.

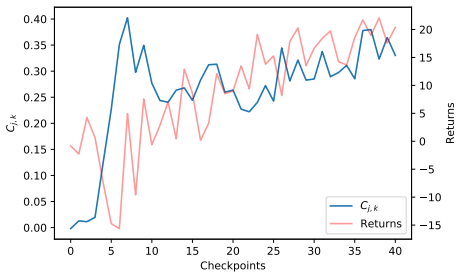
Following our analysis in Section 5.6, we limit the baselines' action-value analysis to the FortAttack environment. Our first analysis is done by comparing the difference between the action-value of shooting, $Q(s, \text{shoot})$, with the maximum action-value of all possible actions, $\max_a Q(s, a)$, at different states. When this difference is closer to zero, it means that the agent is more likely to choose to shoot at s. As provided by Figure 12, we subsequently compare this value between states where no attacker is in the learner's shooting range and states where at least one attacker is in the learner's shooting range.

As learning progresses, our results show that all algorithms results in the learner becoming increasingly aware of the value of shooting attackers. For GNN, the gap between the red and blue lines become increasingly large around the 50^{th} checkpoint, which coincides with when GNN and GNN-AM agents start to increase their performance. On the other hand, although there is still a significant difference between the values of the red and blue line, QL and QL-AM does not result in a large difference to these lines compared to GNN/GNN-AM. This inability to further highlight the value of shooting is then the reason why QL/QL-AM does not perform as good as GNN/GNN-AM.

We now show that despite learning the value of shooting attackers that are inside the learner's range, the single-agent



(a) Correlation between a teammate's $\bar{Q}_j$ with their distance to the closest prey from the learning agent.



(b) Average value of $C_{j,k}(a)$ for situations where agent j, k, and learning agent's actions leads to the capture of a prey.

Figure 11. **FortAttack individual and pairwise action value visualizations:** The visualizations show how various metrics evolve across different checkpoints and correlate with agent's learning performance. Data is obtained by executing the greedy policy entailed by the value network for 24000 timesteps. (11a) The Pearson correlation coefficient between a teammate's distance to the prey closest from the learning agent and $\bar{Q}_j$ is gathered and visualized. (11b) $C_{j,k}(a)$ is computed for observations preceding a capture of a prey by the learning agent. Analysis is further limited towards the edges associated to pairs of edges connecting agents that hunted the prey together with the learning agent.

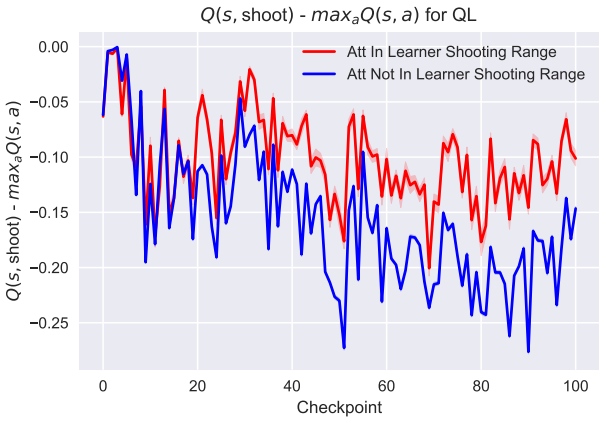
RL baselines do not learn concepts that are learned by GPL's pairwise utility model, MLP_δ. Since all baselines do not have a CG-based joint action-value model, it is difficult to measure $\bar{Q}_{j,k}$ as in Section 5.6. The closest metric to $\bar{Q}_{j,k}$ that is still obtainable from the value networks of the baselines is the state value, $V(s)$, defined as :

$$V(s) = \max_a Q(s, a).$$

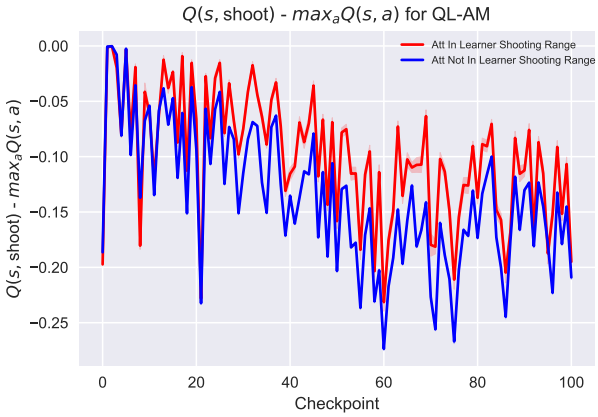
If we collect $V(s)$ in a reasonably large number of states and average it between states that share a specific trait, we can obtain a Monte Carlo estimate of the value assigned to that specific trait by the value network.

Following a similar data collection process for our analysis in Section 5.6, we measure $V(s)$ and contrast its average for states where there is at least an attacker in any defender's shooting range and states where there are no attackers in any defender's shooting range. By contrasting the green and blue line in Figure 13, we see that QL and QL-AM assign higher values to states where no attackers in any defender's shooting range. On the other hand, the blue and green line in plots associated to GNN and GNN-AM has similar values in most checkpoints. This demonstrates the baselines' inability to recognize the value of having an attacker in a defender's shooting range or, even worse, their preference towards states where there are no attackers in any defender's shooting range. Unlike the baselines, GPL learners do learn to assign higher utility values when an attacker is inside a defender's shooting range.

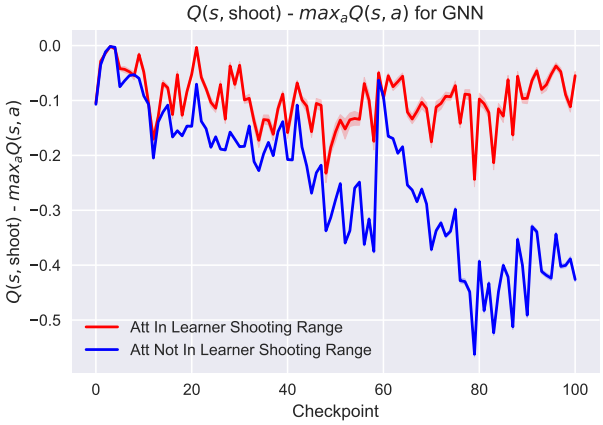
We have then showed that the single-agent RL baselines failed to learn the effects of other agents' action towards the learner as GPL. With GPL, learning these concepts became an important path to enable improved performance in FortAttack. The baselines' inability to learn these concepts eventually lead them to produce subpar performance compared to GPL.



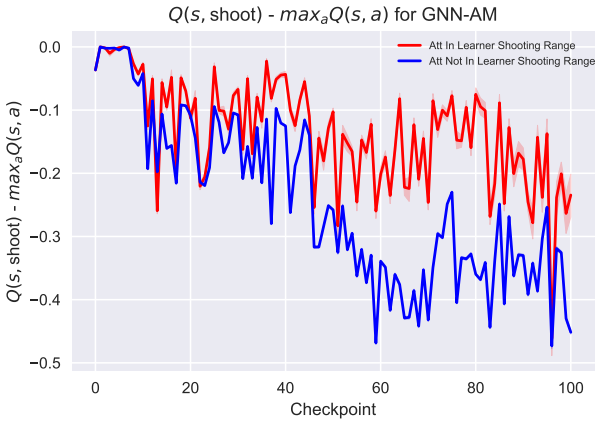
(a) $Q(s, \text{shoot}) - \max_a Q(s, a)$ for QL.



(b) $Q(s, \text{shoot}) - \max_a Q(s, a)$ for QL-AM.



(c) $Q(s, \text{shoot}) - \max_a Q(s, a)$ for GNN.

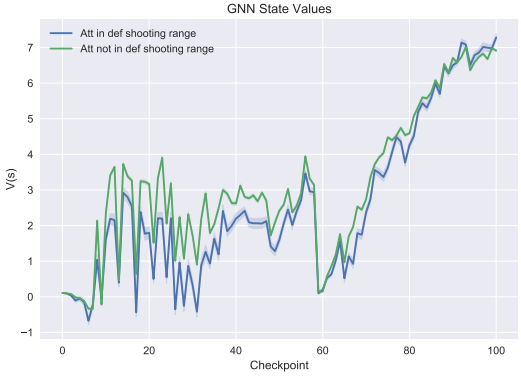


(d) $Q(s, \text{shoot}) - \max_a Q(s, a)$ for GNN-AM.

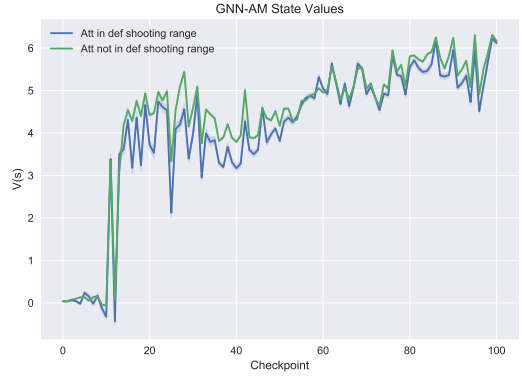
Figure 12. $\mathbf{Q(s,shoot) - max_a Q(s,a)}$ **for single-agent RL baselines:** This visualization compares the action-value of shooting and the estimated optimal action for (a) QL, (b) QL-AM, (c) GNN, and (d) GNN-AM. We obtain this figure by running a policy entailed by the value-networks that we saved during open ad hoc teamwork training done in Section 5.5. For each checkpoint, we gather data by running the policy for 480000 steps and recording the observed states. The observed states is subsequently organized based on whether there is at least an attacker is in the learner's shooting range or not. We finally measure $Q(s, \text{shoot}) - \max_a Q(s, a)$ at these states and visualize its mean and 95% confidence intervals. The blue line corresponds to states where there is no attacker in the learner's shooting range while the red line represents states where there is at least an attacker in the learner's shooting range. As training commences for all algorithms, this visualization shows that shooting becomes increasingly likely of becoming the optimal action whenever an attacker is inside the learner's shooting range.

(a) State values for QL.

(b) State values for QL-AM



(c) State values for GNN



(d) State values for GNN-AM

Figure 13. **State values for all single-agent RL baselines:** This visualization compares the state values for (a) QL, (b) QL-AM, (c) GNN, and (d) GNN-AM. This figure is obtained by running the policies that are stored during the open ad hoc teamwork training described in Section 5.5. Data collection is done by running these policies for 480000 steps and recording the observed states and predicted state-values throughout the process. To get a Monte Carlo estimate of the values of states when at least an attacker is in the defender's shooting range and when there are no attackers in any defender's shooting range, we average the recorded state values for these two situations. The blue line shows the average and 95% confidence bounds of $V(s)$ when at least an attacker is in a defender's shooting range. By contrast, the green line shows the average and 95% confidence bounds of $V(s)$ when no attacker is in any defender's shooting range. This figure shows that either the baselines assigns higher values to states where no attacker is in any defender's shooting range (QL and QL-AM) or the baselines most often fails to distinguish the value of both types of states (GNN and GNN-AM).