Supplementary Materials for "Multi-Task Multi-Agent Reinforcement Learning with Interaction and Task Representations"

APPENDIX A
DETAILS OF THE ALGORITHM

To handle the varying dimensions of state inputs in different multi-agent tasks, we use the multi-head attention mechanism in the selfish utility function, task representation function and state encoder to extract population-invariant representations. Additionally, we present a mask $\mathcal{M}_{o,t}$ to make these representations satisfy the partially observable property.

Similarly, we also use the multi-head attention mechanism in the coordinated incremental utility network to learn fixed-length representations of actions selected by each agent's preceding agents. To guarantee that each agent can only observe the actions selected by its preceding agents, we introduce a sequence mask $\mathcal{M}_e$ to customize eligible action representations for each agent respectively during the centralized training stage, and accordingly accelerate the learning process.

Owing to the fact that multiple multi-agent tasks may have different participating agents, state spaces, observation spaces and other elements, the experiences from these tasks also have different dimensions and we need to establish multiple replay buffers to store them respectively, which may be prohibitively expensive. To avoid this, we align the dimensions of experiences of all tasks (including both training and testing tasks) by padding zero vectors based on the maximum number of entities (including both agents and non-agents). Also we introduce a scenario mask $\mathcal{M}_s$ to indicate which entity is padded in different tasks. Accordingly we can shape an universal replay buffer for all tasks (including both training and testing tasks) to enable flexible batch data processing. What's more, note that the pre-definition of the maximum number of entities doesn't introduce any limitation and our algorithm still can adapt to other tasks that are not included in training and testing tasks.

In this section, we give a detailed description of the partially observable mask $\mathcal{M}_{o,t}$, the sequence mask $\mathcal{M}_e$, and the scenario mask $\mathcal{M}_s$ for more clarity, based on the assumption that the maximum number of entities is m (n for agents and $(m-n)$ for non-agents) for all training and testing tasks.

A. Partially Observable Mask

In the formation of Factored Dec-POMDP (Section III-A), we specifically define the local observation of each agent i as $o_t^i = \{s_t^{e^j} | \mu_t(i, e^j) = 1, \forall e^j \in \mathcal{E}\}$, where $\mu_t(i, e^j)$ is a binary observable mask which indicates whether agent i can observe entity e^j at current time step t and $\mu_t(i, i)$ always equals 1. Based on $\mu_t(i, e^j)$, we introduce the partially observable mask $\mathcal{M}_{o,t}$, which is described as a $n \times m$ matrix below:

$$\begin{bmatrix} \mu_t(1,e^1) & \mu_t(1,e^2) & \mu_t(1,e^3) & \cdots & \mu_t(1,e^m) \\ \mu_t(2,e^1) & \mu_t(2,e^2) & \mu_t(2,e^3) & \cdots & \mu_t(2,e^m) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mu_t(i,e^1) & \mu_t(i,e^2) & \mu_t(i,e^3) & \cdots & \mu_t(i,e^m) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mu_t(n,e^1) & \mu_t(n,e^2) & \mu_t(n,e^3) & \cdots & \mu_t(n,e^m) \end{bmatrix},$$

where each component $\mu_t(i, e^j)$ indicates that whether agent i can observe entity e^j at current time step t. Based on the mask $\mathcal{M}_{o,t}$, we set the scores of unobservable entities to $-\inf$ for each agent in the multi-head attention mechanism. Then the weights outputted by the softmax operator layer approximate 0 and thus the derived representations of each agent satisfy the partially observable property. In the experiments, we include the partially observable masks into transitions and store them into the replay buffer for more flexible data processing.

B. Scenario Mask

Furthermore, we introduce a scenario mask $\mathcal{M}_s$ that indicates which entities really exist in each task and which ones are padded. Based on it, we can remove the information of these padded entities in all vectors (*e.g.,* the hidden info outputted by GRU, the values outputted by utility functions, the representations in the state encoder, the partially observable representations in both selfish utility function and task representation encoder, the representations of actions in coordinated incremental utility function, etc). Specifically, the scenario mask $\mathcal{M}_s$ is defined by an one-dimensional array below:

$$\left[\delta(e^1), \delta(e^2), \delta(e^3), \ldots, \delta(e^m)\right],$$

where each element $\delta(e^j)$ lies in $\{0, 1\}$ and indicates whether entity e^j is padded or not. Based on this mask, we set scores of these padded entities to -inf in all multi-head attention mechanisms and further set the corresponding components of them in other representations (such as the utility values, the hidden info outputted by GRU and so on) to 0 directly. Additionally, the scenario mask keeps fixed during each episode and is reset only when we initialize tasks. Also we include the scenario masks into experiences and store them into the replay buffer.

C. Sequence Mask

As done in the decoder of vanilla Transformer, we also present a sequence mask $\mathcal{M}_e$ into the coordinated incremental

utility network to guarantee that each agent can only observe the actions selected by its preceding agents during training. Specifically, the sequence mask $\mathcal{M}_e$ is a special $n \times n$ lower triangular matrix where all non-zero elements are equal 1:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 1 & \cdots & 1 \end{bmatrix}.$$

Based on the sequence mask, for each agent, we retain the score of action representations of its preceding agents in the multi-head attention module of the coordinated incremental utility function network and set the others to -inf. Accordingly, each agent's final action representations outputted by the multi-head attention module only contain actions selected by its preceding agents, which satisfies the requirement.

D. Transformation in the Attention Mechanism

In this work, we use the multi-head attention mechanism to extract population-invariant representations. Here we give a comprehensive description about the transformation in the attention mechanism for more clarity. As shown in Figure 1, given the state inputs $s = (s^{e^1}, s^{e^2}, \ldots, s^{e^m})$ whose length is equal to the maximum number of entities m, we first reshape it to the vector $\boldsymbol{S}$ whose shape is (m, d), where d represents the dimension of each entity's local state. In Section III-A, we assume the same local state spaces for all entities. Thus we use a shared hidden layer (dubbed as the function F) to encode the local states of all entities:

$$F(\boldsymbol{S}; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{S}\boldsymbol{W} + \boldsymbol{b}, \boldsymbol{S} \in \mathbb{R}^{m \times d}, \boldsymbol{W} \in \mathbb{R}^{d \times h}, \boldsymbol{b} \in \mathbb{R}^h, \tag{1}$$

where h is the units of the hidden layer.

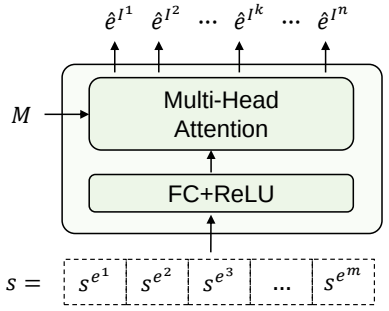

Fig. 1. A general illustration of the multi-head attention mechanism.

With the function F, we derive the representation $\boldsymbol{X} = F(\boldsymbol{S}; \boldsymbol{W}, \boldsymbol{b})$ whose shape is (m, h) and the rows of it represent the entities. Now we specify the operation that defines an attention head. Given the addition inputs $J \subseteq \mathbb{Z}^{[1,m]}$, a set of indices that select which rows of $\boldsymbol{X}$ are used to compute queries such that $\boldsymbol{X}_J \in \mathbb{R}^{|J| \times h}$, and $\mathcal{M}$, a mask (*e.g.,* the partially observable mask $\mathcal{M}_{o,t}$, the scenario mask $\mathcal{M}_s$ and the sequence mask $\mathcal{M}_e$) specifying the information of which entities each query entity can include into its own contexts. We refer to the attention mechanism as the operator ATTEN and show the transformations in it below:

$$\begin{aligned} &\mathrm{ATTEN}(J, \boldsymbol{X}, \mathcal{M}; \boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V) \\ &= \mathrm{softmax}(\mathrm{mask}(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{h^a}}, \mathcal{M}))\boldsymbol{V} \in \mathbb{R}^{|J| \times h^a}, \end{aligned} \tag{2}$$

where:

$$\begin{aligned} &\boldsymbol{Q} = \boldsymbol{X}_J \boldsymbol{W}^Q, \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^K, \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^V, \mathcal{M} \in \{0,1\}^{|J| \times m}, \\ &\boldsymbol{W}^Q \in \mathbb{R}^{h, h^a}, \boldsymbol{W}^K \in \mathbb{R}^{h, h^a}, \boldsymbol{W}^V \in \mathbb{R}^{h, h^a}, \end{aligned} \tag{3}$$

where h^a represents the hidden units of current attention head.

In Equation (2), the operation $\mathrm{mask}(\boldsymbol{A}, \mathcal{M})$ takes two equal sized matrices (the score matrix whose shape is $(|J| \times m)$ and the mask matrix whose shape is also $(|J| \times m)$) and fills the entries of $\boldsymbol{A}$ with -inf in the indices where $\mathcal{M}$ is equal to 0. After the $\mathrm{softmax}$, the weights of these entries become zero, thus preventing the query entities from attending to specific entities. $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$ are all learnable parameters of the query, key and value layers.

After receiving the vector outputted by each attention head, we further define the multi-head attention mechanism MHA as the parallel computation of n^a attention heads below:

$$\begin{aligned} &\mathrm{MHA}(J, \boldsymbol{X}, \mathcal{M}) \\ &= \mathrm{concat}_{b \in (1,2,\ldots,n^a)}(\mathrm{ATTEN}(J, \boldsymbol{X}, \mathcal{M}; \boldsymbol{W}_b^Q, \boldsymbol{W}_b^K, \boldsymbol{W}_b^V)), \end{aligned} \tag{4}$$

where the final outputted vector is the concatenation of outputs of n^a attention heads.

Please note that here we use the fixed number of entities m and the ones of agents n only for clarity and the attention mechanism above can be applicable to multiple multi-agent tasks with different m and n. Owing to the assumption that all entities have the same local state spaces, the shared hidden layer F and the attention head ATTEN are universal to different multi-agent tasks to encode the local states of entities in them. Thus the final algorithm is also universal to multiple tasks that have different number of entities and agents.

E. Attention Mechanisms in Selfish Utility Function and Task Representation Encoder

As an example, as shown in Figure 2, to make the learned representations $(\hat{e}^{I^1}, \hat{e}^{I^2}, \ldots, \hat{e}^{I^n})$ of all agents in the selfish utility function network satisfy the partially observable property, we replace the mask $\mathcal{M}$ with the partially observable mask $\mathcal{M}_{o,t}$, and define the index inputs J as the agent set $\mathcal{N}$. With the agent set $\mathcal{N}$, the computed queries $Q = \boldsymbol{X}_{\mathcal{N}}\boldsymbol{W}^Q \in \mathbb{R}^{n \times h^a}$, which only contain the agents. Then we calculate the score matrix with $\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{h^a}} \in \mathbb{R}^{n,m}$ and further set the entries in it to -inf where $\mathcal{M}_{o,t}$ equals to 0. As a result, for each row (query agent), the weights of corresponding invisible entities outputted by the $\mathrm{softmax}$ are equal to 0 such that each query agent I^k only incorporates information of the entities that lie in its observable field into its local representations $\hat{e}^{I^k}$, which follow the partially observable setting. Note that only one attention layer is permitted in this process to avoid that the information of unseen entities is propagated through multiple rounds.

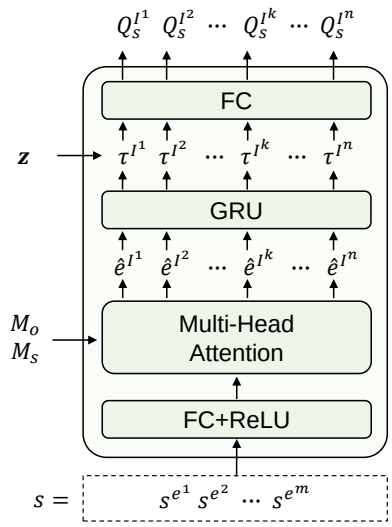

Fig. 2. Structure of the selfish utility function.

In addition, we also introduce the scenario mask $\mathcal{M}_s$ to remove the information of padded entities for flexible batch data processing. In detail, the original $\mathcal{M}_s$ is an one-dimensional array whose length is m, which indicates which entities are padded. Similarly, we also use the agent set $\mathcal{N}$ as the index inputs J and further select the components $\mathcal{M}_{s,\mathcal{N}}$ whose length is equal to n. Then we expand its dimension to make it have the same size as some vectors (*e.g.,* the hidden info outputted by GRU, the final utility values). Finally we set the entries in these vectors to 0 where the expanded $\mathcal{M}_{s,\mathcal{N}}$ is equal to zero.

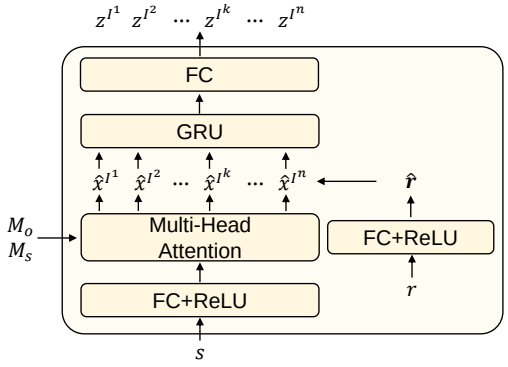

Fig. 3. Structure of the task representation encoder.

Moreover, as shown in Figure 3, we use these two masks in the task representation encoder to extract partially observable representations $\hat{x}^{I^1}, \hat{x}^{I^2}, \ldots, \hat{x}^{I^n}$, which is consistent with the operation above.

F. Attention Mechanism in Coordinated Incremental Utility Function

Figure 4 shows structure of the coordinated incremental utility function, where the inputs of each agent I^k are the actions $a^{<I^k} = (a^{I^1}, a^{I^2}, \ldots, a^{I^{k-1}})$ selected by its preceding agents. When conducting training, we have access to all agents' actions. Specifically, we begin with an arbitrary indicator a^{begin} and initialize the action inputs of all agents as $\boldsymbol{a} = (a^{\text{begin}}, a^{I^1}, a^{I^2}, \ldots, a^{I^{n-1}})$, whose length is equal to the number of agents n. Owing to the assumption that all agents have the same local action spaces, we can further reshape it to the vector $\boldsymbol{A}$ whose shape is (n, d^a), where d^a represents the dimension of each agent's action. Following the operations in the attention mechanism stated above, we replace $\boldsymbol{S}$ with the action vector $\boldsymbol{A}$ and set the index inputs J to the agent set $\mathcal{N}$. Also we replace the mask $\mathcal{M}$ with the sequence mask $\mathcal{M}_e$. The remaining operations are the same as the vanilla ones of the attention mechanism.

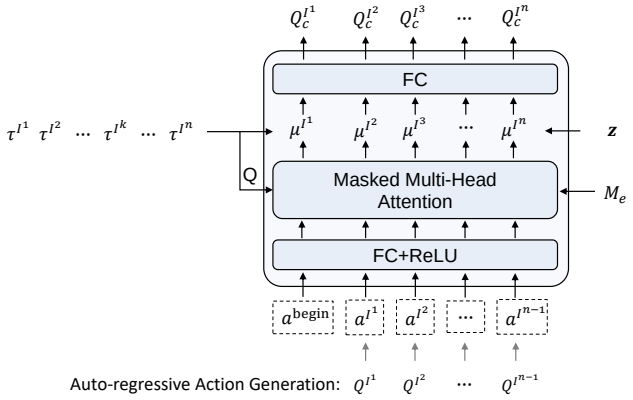

Fig. 4. Structure of the coordinated incremental utility function.

During centralized training, we have access to the selected actions of all agents at each time step, such that we can shape the action inputs $\boldsymbol{a}$ and the vector $\boldsymbol{A}$ directly. With the sequence mask $\mathcal{M}_e$, for each agent I^k, we retain the scores of actions selected by its preceding agents and set others to -$\inf$. Accordingly, in Figure 4, the learned action representations μ^{I^k} of each agent I^k only contain the information of the actions selected by its preceding agents, which satisfies the proposed paradigm of sequential action selections.

During decentralized execution, all agents select their actions sequentially following a pre-defined decision order. Here we initialize $\boldsymbol{A}$ as a zero vector whose shape is (n, d^a) and insert a^{begin} into the first row of it. Then we feed this initialization into the coordinated incremental utility function to calculate $Q_c^{I^1}$ for the first agent I^1. And agent I^1 selects its action a^{I^1} based on its final utility value $Q^{I^1} = Q_s^{I^1} + Q_c^{I^1}$. Then we insert the action a^{I^1} into the second row of $\boldsymbol{A}$ and calculate $Q_c^{I^2}$ for the second agent I^2 based on it. And agent I^2 selects its action a^{I^2} according to the final utility value $Q^{I^2} = Q_s^{I^2} + Q_c^{I^2}$. We further insert the action a^{I^2} into the third row of $\boldsymbol{A}$. Such loop continues until the last agent I^n selects its action a^{I^n} based on the actions selected by its preceding agents.

G. Attention Mechanism in the Mixing Network

To coordinate the action selections of all agents efficiently, we use a mixing network similar to the one in QMIX to calculate the joint value function Q^{total} based on the utility functions $(Q^{I^1}, Q^{I^2}, \ldots, Q^{I^n})$ of all agents. Such mixing network

taking all agents' utility functions as inputs requires that the parameters of the mixing layers should depend on the number of agents present, while incorporating the state information. As shown in Figure 5, we again use the multi-head attention mechanism to extract representations $\hat{\boldsymbol{s}} = (\hat{s}^{I^1}, \hat{s}^{I^2}, \ldots, \hat{s}^{I^n})$ for present agents, whose shape is (n, h) and h is the hidden dimension of the multi-head attention mechanism. We achieve this by setting the index inputs J to the agent set $\mathcal{N}$ and also remove the information of padded entities for each query agent with the scenario mask $\mathcal{M}_s$.

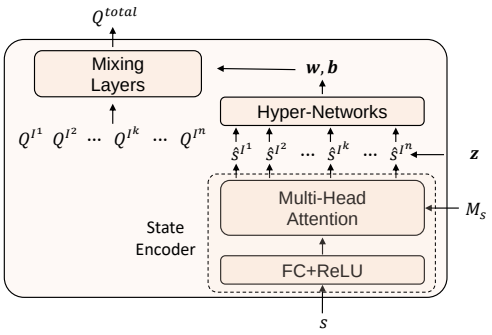

Fig. 5. Structure of the mixing network.

After extracting the latent representations $\hat{\boldsymbol{s}}$, we concatenate them with the learned task representations $\boldsymbol{z}$ of all agents. The shape of this concatenation is $(n, h + h^z)$, where h^z represents dimension of the task representation. Based on this concatenated inputs $[\hat{\boldsymbol{s}}, \boldsymbol{z}]$, we then use four hyper-networks to generate the weights $\boldsymbol{W}_1 \in \mathbb{R}^{+(n \times h^m)}, \boldsymbol{w}_2 \in \mathbb{R}^{+(h^m)}$ and the biases $\boldsymbol{b}_1 \in \mathbb{R}^{h^m}, b_2 \in \mathbb{R}^1$ of two-layer mixing layers, where h^m represents the hidden units of the mixing layers. For the weight $\boldsymbol{w}_2 \in \mathbb{R}^{+(h^m)}$ and the bias $\boldsymbol{b}_1 \in \mathbb{R}^{h^m}$, the shapes of original outputs of the corresponding hyper-networks are (n, h^m) and we average them across the n sized dimensions, which makes their shapes change from (n, h^m) to (h^m). For the bias b_2, the shape of original outputs of its hyper-network is $(n, 1)$ and we also average them to make their shape be equal to 1. And we use the absolute value function to make the weights positive to satisfy the monotonicity condition. This procedure enables the dynamic generation of mixing networks whose input size varies with the number of agents present. Then based on all agents' utility functions $\boldsymbol{q} = (Q^{I^1}, Q^{I^2}, \ldots, Q^{I^n})$, the joint value function $Q^{total}(\boldsymbol{\tau}, \boldsymbol{a}, \boldsymbol{z}, s)$ is calculated as:

$$Q^{total}(\boldsymbol{\tau}, \boldsymbol{a}, \boldsymbol{z}, s) = \mathrm{ELU}((\boldsymbol{q}^\top \boldsymbol{W}_1) + \boldsymbol{b}_1^\top)\boldsymbol{w}_2 + b_2. \quad (5)$$

APPENDIX B
DETAILS OF THE EXPERIMENTS

To validate the effectiveness of our algorithm, we compare it against several baselines under both multi-task and zero-shot settings. In addition, we conduct ablation studies and further visualize the learned task representations of multiple tasks. Here we describe the settings of these experiments for more clarity.

A. Experimental Setting

For both MPE and SMAC benchmarks, we run 8 parallel threads and equip them with specific tasks respectively. Under the multi-task setting, at the beginning of each episode, we sample from the training tasks and reset the environment according to the sampled tasks. All experiments under this setting are carried out with 5 different random seeds. We set the max number of training steps T_{max} to 5050000 for the task sets in MPE benchmark and 10000000 for the ones in SMAC benchmark. During training, we evaluate the learned policies per T_{eval} time steps and roll out n episodes to calculate the average episodic rewards (or the average win rates). We set T_{eval} to 10000 for the task sets of MPE benchmark and 50000 for the ones of SMAC. And we set n to 32 for the task sets of MPE and 160 for the ones of SMAC.

For the zero-shot setting, we also run 8 parallel threads and sample from the testing tasks for each of them at the beginning of each episode. Then we load the trained policies (5 trained models correspond to 5 random seeds) in the multi-task experiments respectively and roll out k episodes to evaluate the zero-shot performance of these trained policies. Specifically, we set k to 32 for Cooperative Navigation Series, 64 for Predator and Prey Series and Predator and Prey Series Pro, and 160 for all task sets (3-8m, 3-8csz, 3-8MMM and 3-8sz) of SMAC benchmark. For the task sets of MPE benchmark, after finishing rolling out k episodes for all 5 trained policies respectively, we calculate the average episodic reward of these k episodes and further calculate the mean and the standard error of the average episodic rewards of 5 trained policies. As for the task sets of SMAC benchmark, we calculate the average win rates, as well as the mean and the standard error of the average win rates of all trained policies.

The ablation studies follow the same settings as the experiments above. For the visualization of the learned representations of multiple tasks, we choose Cooperative Navigation Series as the testbed and load the trained policies to collect agents' trajectories in all tasks (including both training and testing tasks). Specifically, we use the trained policies of all agents to roll out 16 episodes in the tasks (2,2),(3,3),(4,4),(5,5) and (6,6) respectively. Then we use the corresponding trained task representation encoder to infer the task representations for all agents. Finally we visualize them with t-SNE.

B. Implementation

The whole structure of RIT is shown in the content of our paper. Figure 1 (a1) represents the selfish utility network, which is composed by one hidden layer (of 64 units and RELU activation), one multi-head attention module (of b units, c heads and RELU activation), one GRU module (of 64 units) and one linear layer that outputs the selfish utility value. The coordinated incremental utility network is shown in Figure 1 (a2) and it consists of one hidden layer (of 64 units and RELU activation), one masked multi-head attention module (of b units, c heads and RELU activation) and one linear layer that outputs the coordinated incremental utility value.

Figure 1 (b) is the task representation encoder and it also contains one hidden layer (of 64 units and RELU activation),

one multi-head attention module (of b units, c heads and RELU activation), one GRU module (of 64 units) and one linear layer that outputs task representations. Also it use one extra hidden layer (of 64 units and RELU activation) to encode the rewards, which serve as the inputs of the GRU module together with the action-observation representations to encode the augmented action-observation-reward histories.

Figure 1 (c) represents the structure of the mixing network. In the mixing network, we first encode the global states with the state encoder, which includes one hidden layer (of 64 units and RELU activation) and one multi-head attention module (of b units, c heads and RELU activation). After deriving the state representations, we use several hyper-networks (achieved by several fully connected networks) to generate the weights and biases of the mixing layers. Finally we use these mixing layers to calculate the joint value functions based on all agents' utility value functions. We set b, c to $64, 1$ for the task sets of MPE benchmark and $64, 2$ for the ones of SMAC benchmark.

For the baseline UPDeT-MIX and UPDeT-VDN, the utility function of each agent is implemented by a transformer (only contains the encoder of the vanilla transformer). Specifically, it contains one hidden layer (of 64 units and RELU activation), a encoder blocks and one linear layer that outputs the utility values. Each encoder block is composed with one multi-head attention module (of b units, c heads and RELU activation), one linear layer and a MLP with one hidden layers (of 64 units and RELU activation). Also it use extra LayerNorm layers and the residual operations. In addition, UPDeT-MIX uses the same mixing network as RIT to deal with the global states with varying dimensions in multiple multi-agent tasks. We set a to 1 for all experiments and follow the same hyper-parameters b, c of the multi-head attention module as RIT.

For the baseline REFIL, we achieve it based on the vanilla code released alongside the paper. In REFIL, we use the same hyper-parameters of the multi-head attention module as RIT to ensure fair comparisons.

For the baseline QMIX-ATTN and VDN-ATTN, they have the same utility function network for all agents, which contains one hidden layer (of 64 units and RELU activation), one multi-head attention module (of b units, c heads and RELU activation), one GRU module (of 64 units) and one linear layer that outputs the utility values. We use the same b, c of the multi-head attention module as RIT. In addition, QMIX_ATTN uses the same mixing network as RIT and UPDeT_MIX to deal with multiple multi-agent tasks.

For the baseline ROMA, we achieve it based on the vanilla code released by its authors. Specifically, the utility function network of each agent in ROMA is composed with one hidden layer (of 64 units and RELU activation), one multi-head attention module (of b units, c heads and RELU activation), one GRU module (of 64 units) and one linear layer that outputs the utility value. Note that the parameters of the final linear layer are generated by two extra linear layers that condition on the inferred roles. To learn these roles, the role encoder contains one hidden layer (of 64 units and RELU activation), one multi-head attention module (of b units, c heads and RELU activation) and a MLP with one hidden layer (of 16 units and LeakyRELU activation) that outputs the learned roles. Except for these components, there are some other auxiliary function networks used to optimize the learned roles in the utility function network of each agent. The setting of hyper-parameters b, c of ROMA is the same as RIT and the other components follow the same setting as the vanilla code of ROMA. In addition, we also equip ROMA with the same mixing network as RIT to encode the global states with varying dimensions and deal with multiple multi-agent tasks.

For the baseline RIT-VDN, its follows the same structure as RIT. What's different is that RIT-VDN directly adds the utility value functions of all agents to calculate the joint value function.

We achieve all algorithms above based on the open-source codebase PyMARL2, which enables fair comparison.

TABLE I
HYPER-PARAMETERS OF RIT IN THE MPE DOMAIN.

Hyper-parameters	Value	Hyper-parameters	Value
epsilon_start ϵ_{max}	1.0	lr	0.001
epsilon_finish ϵ_{min}	0.05	mixing_embed_dim	32
epsilon_anneal_time $T_{epsilon}$	50000	hypernet_embed_dim	64
n_rollout_threads	8	optimizer	Adam
buffer_size N_{max}	5000	attention_dim	64
batch_size N_{batch}	64	n_heads	1
env_representation_dim	20	hidden_dim	64
T_{max}	5050000	target_update_interval	200

TABLE II
HYPER-PARAMETERS OF SCVD IN THE SMAC DOMAIN.

Hyper-parameters	Value	Hyper-parameters	Value
epsilon_start ϵ_{max}	1.0	lr	0.001
epsilon_finish ϵ_{min}	0.05	mixing_embed_dim	32
epsilon_anneal_time $T_{epsilon}$	500000	hypernet_embed_dim	64
n_rollout_threads	8	optimizer	Adam
buffer_size N_{max}	5000	attention_dim	64
batch_size N_{batch}	32	n_heads	2
env_representation_dim	20	hidden_dim	64
T_{max}	10000000	target_update_interval	200

C. Hyper-parameters

Table I and Table II show the hyper-parameters of our algorithm in MPE and SMAC domains respectively. Other baseline algorithms follow the same setting except for the env_representation_dim. We achieve all algorithms based on the open-source codebase PyMARL2, which enables fair comparison of multiple algorithms.

We run all experiments with five different random seeds by default and plot the mean/std in all figures. The experiments in the MPE domain are performed on a server with two 12-core Intel(R) Xeon(R) CPU E5-2650 v4 CPUs, 256GB RAM and 8 NVIDIA GTX1080Ti GPUs. And the experiments in the SMAC domain are performed on a server with two 22-core Intel(R) Xeon(R) Gold 6238 CPUs, 512GB RAM and 8 NVIDIA GTX2080Ti GPUs.