A ADDITIONAL EXPERIMENTAL SETTINGS AND RESULTS

A.1 A SIMPLE POLICY EVALUATION PROBLEM.

We present a simple policy evaluation experiment to show the influence of the model's representational capacity on the converged performance. We consider the following permutation-invariant (PI) models: ❶ Deep Set, ❷ DA-MLP (apply data augmentation to a permutation-sensitive MLP model), ❸ DPN (apply DPN to the same MLP model), ❹ HPN (apply HPN to the same MLP model) and ❺ Attention (use self-attention layers and a pooling function to achieve PI).

The experimental settings are as follows. There are 2 agents in total. Each agent i only has one dimension of feature x_i. For the convenience of analyzing, we set that each x_i is an integer and $x_i \in \{1, 2, ..., 30\}$, i.e., each agent i only has 30 different features. Thus the size of the joint state space ($[x_1, x_2]$) after concatenating is $30 * 30 = 900$. To make the policy evaluation task permutation-invariant, we simply set the target value Y of each state $[x_1, x_2]$ as $x_1 * x_2$.

In section 4.3, when introducing HPN, we asked a question that whether we can provide an infinite number of candidate weight matrices of DPN. In this simple task, we can directly construct a separate weight matrix for each feature x_i. Since x_i is discrete (which is enumerable), we can explicitly maintain a parameter table and exactly record a different embedding weight for each different feature x_i. We denote this direct method as ❻ DPN(∞), which means 'DPN with infinite weights'. Our HPN uses a hypernetwok to approximately achieve 'infinite weight' by generating a different weight matrix for each different input x_i. We use the Mean Square Error (MSE) as the loss function to train these different models. The code is also attached in the supplementary material, which is named '*Synthetic_Policy_Evaluation.py*'.

The comparison of the learning curves and the converged MSE losses of these different PI models are shown in Fig.11 and Table 1 respectively. We conclude that DPN(∞) > HPN > Attention > DPN > DeepSet > DA-MLP, where '>' means 'performs better than', which indicates that increasing the representational capacity of the model can help to achieve much less MSE loss.

Table 1: The comparison of the converged MSE losses of these different PI models.

	DA-MLP	Deep Set	DPN	Attention	HPN	DPN(∞)
MSE	1495.55	1401.23	119.54	49.12	6.34	0.37

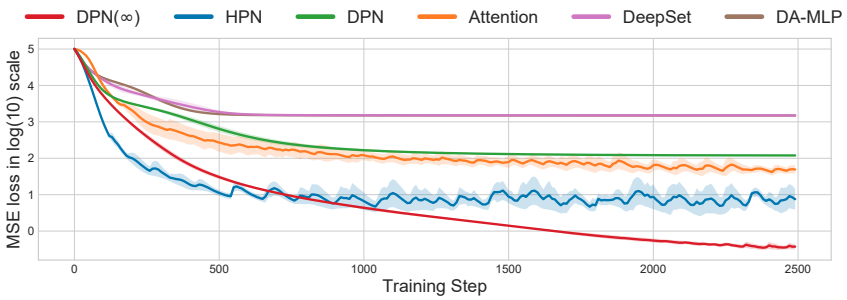


Figure 11: The comparison of the learning curves of different PI models.

If we keep each $x_i \in \{1, 2, ..., 30\}$ and simply increase the agent number, the changes of the original state space by simple concatenation and the reduced state space by using PI representations are shown in Table 2 below. We see that by using PI representations, the state space can be significantly reduced.

Table 2: Changes of the state space size with the increase of the agent number.

agent number	2	3	4	5
simple concatenation	900	27000	810000	24300000
PI representation (percent)	465 (0.52)	4960 (0.18)	40920 (0.05)	278256 (0.01)

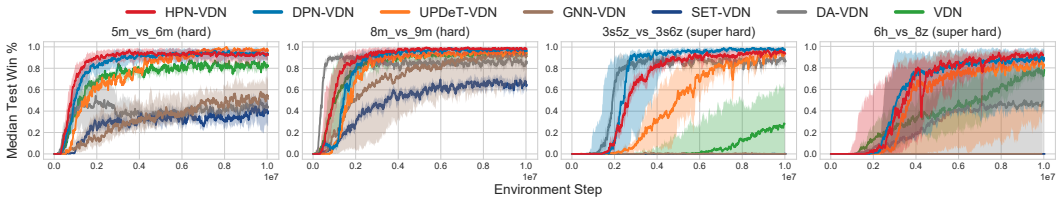


Figure 12: Comparisons of VDN-based methods considering the PI and PE properties.

A.2 Comparison with VDN-based PI and PE baselines

The learning curves of the PI/PE baselines equipped with VDN are shown in Fig.12, which demonstrate that: HPN $\geq$ DPN $\geq$ UPDeT $>$ VDN $>$ GNN $\approx$ SET $\approx$ DA[7]. Specifically, (1) HPN-VDN and DPN-VDN achieve the best win rates; (2) Since UPDeT uses a shared token embedding layer followed by multi-head self-attention layers to process all components of the input sets, the PI and PE properties are implicitly taken into consideration. The results of UPDeT-VDN also validate that incorporating PI and PE into the model design could reduce the observation space and improve the converged performance in most scenarios. (3) GNN-VDN achieves slightly better performance than SET-VDN. Although permutation-invariant is maintained, GNN-VDN and SET-VDN perform worse than vanilla QMIX, (especially in *3s5z_vs_3s6z* and *6h_vs_8z*, the win rates are approximate 0%). This confirms that the use of a shared embedding layer $\phi(x_i)$ for each component x_i limits the representational capacities and restricts the final performance. (4) DA-VDN significantly improves the learning speed and performance of vanilla VDN in *3s5z_vs_3s6z* by data augmentation and much more times of parameter updating. However, the learning process is unstable, which collapses in all other scenarios due to the perturbation of the input features, which validates that it is hard to train a permutation-sensitive function (e.g., MLP) to output the same value when taking different orders of features as inputs.

A.3 Ablation Studies.

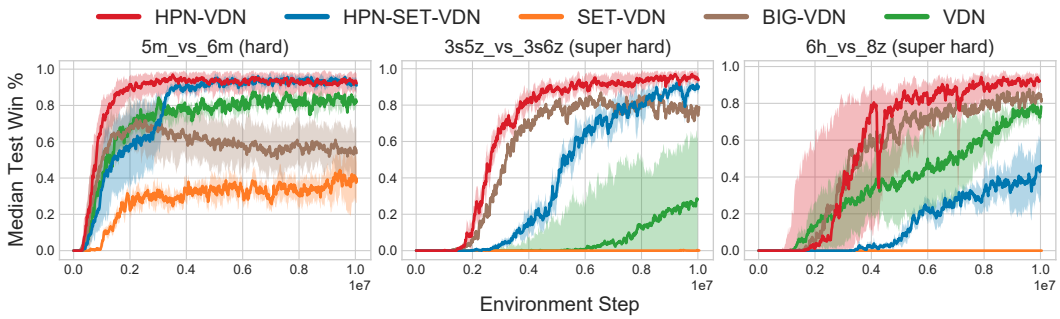


Figure 13: Ablation studies. All methods are equipped with VDN.

A.3.1 Enlarging the Network Size.

We also enlarge the agent network of vanilla VDN (denoted as BIG-VDN) such that the number of parameters is larger than our HPN-VDN. The detailed numbers of parameters are shown in Table 3. The results are shown in Fig.13. We see that simply increasing the parameter number cannot always guarantee better performance. For example, in 5m_vs_6m, the win rate of BIG-VDN is worse than the vanilla VDN. In 3s5z_vs_3s6z and 6h_vs_8z, BIG-VDN does achieve better performance, but the performance of BIG-VDN is still worse than our HPN-VDN in all scenarios.

A.3.2 Importance of the PE output layer and the capacity of the PI input layer

To validate the importance of the permutation-equivariant output layer, we also add the hypernetwork-based output layer of HPN to SET-VDN (denoted as HPN-SET-VDN). The results are shown in Fig.13. We see that incorporating an APE output layer could significantly boost the

[7]We use the binary comparison operators here to indicate the performance order of these algorithms.

Table 3: The parameter numbers of the individual Q-networks in BIG-VDN, BIG-QMIX and our HPN-VDN, HPN-QMIX.

Parameter Size	BIG-VDN, BIG-QMIX	HPN-VDN, HPN-QMIX
5m_vs_6m	109.964K	72.647K
3s_vs_5z	108.555K	81.031K
8m_vs_9m	114.959K	72.839K
corridor	127.646K	76.999K
3s5z_vs_3s6z	121.487K	98.375K
6h_vs_8z	113.55K	76.935K

performance of SET-VDN, and that the converged performance of HPN-SET-VDN is superior to the vanilla VDN in 5m_vs_6m and 3s5z_vs_3s6z.

However, due to the limited representational capacity of the shared embedding layer of Deep Set, the performance of HPN-SET-VDN is still worse than our HPN-VDN, especially in 6h_vs_8z. Note that the only difference between HPN-VDN and HPN-SET-VDN is the input layer, e.g., using hypernetwork-based customized embeddings or a simply shared one. The results validate the importance of improving the representational capacity of the permutation-invariant input layer.

A.4 Applying HPN to QPLEX and MAPPO

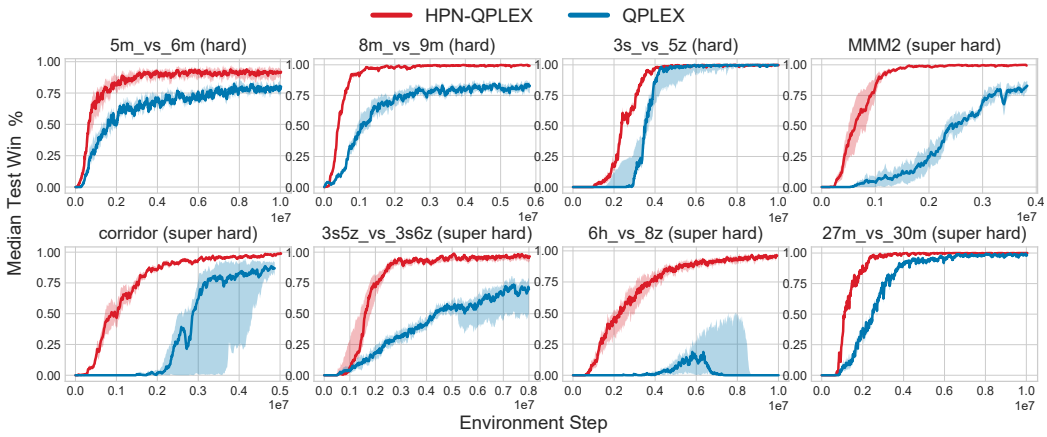


Figure 14: The learning curves of HPN-QPLEX compared with vanilla QPLEX in the hard and super hard scenarios of SMAC.

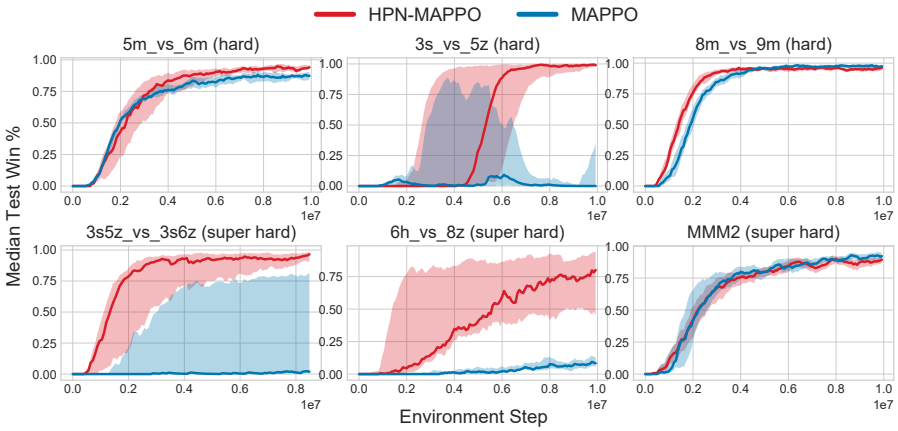


Figure 15: The learning curves of HPN-MAPPO compared with the vanilla MAPPO in the hard and super hard scenarios of SMAC.

To demonstrate that our methods can be easily integrated into many types of MARL algorithms and boost their performance, we also apply HPN to a typical credit-assignment method QPLEX (Wang et al., 2020a) (denoted as HPN-QPLEX) and a policy-based MARL algorithm MAPPO (Yu et al., 2021) (denoted as HPN-MAPPO). The results are shown in Fig.14 and Fig.15. We see that HPN significantly improves the performance of QPLEX and MAPPO, which validates that our method can be easily combined with existing MARL algorithms and improves their performance (especially for super hard scenarios).

A.5 Applying HPN to Deep Coordination Graph.

Recently, Deep Coordination Graph (DCG) (Böhmer et al., 2020) scales traditional coordination graph based MARL methods to large state-action spaces, shows its ability to solve the relative overgeneralization problem, and obtains competitive results on StarCraft II micromanagement tasks. Further, based on DCG, (Wang et al., 2021) proposes an improved version, named Context-Aware SparsE Coordination graphs (CASEC). CASEC learns a sparse and adaptive coordination graph (Wang et al., 2021), which can largely reduce the communication overhead and improve the performance. Besides, CASEC incorporates action representations into the utility and payoff functions to reduce the estimation errors and alleviate the learning instability issue.

Both DCG and CASEC inject the permutation invariance inductive bias into the design of the pairwise payoff function $q_{ij}(a_i, a_j|o_i, o_j)$. They achieve permutation invariance by permuting the input order of $[o_i, o_j]$ and taking the average of both. To show the generality of our method, we also apply HPN to the utility function and payoff function of CASEC and show the performance in Fig.16. The codes of CASEC and HPN-CASEC are also added in the supplementary materials (See *code/src/config/algs/casec.yaml* and *code/src/config/algs/hpn_casec.yaml*).

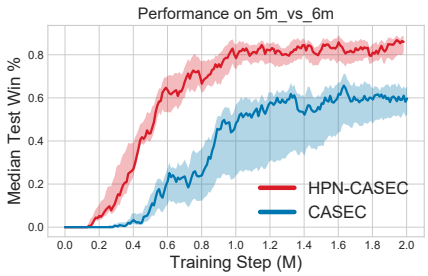


Figure 16: The learning curves of HPN-CASEC and CASEC in 5m_vs_6m.

In Fig.16, we compare HPN-CASEC with the vanilla CASEC in 5m_vs_6m. Results show that HPN can significantly improve the performance of CASEC, which validate that HPN is very easy to implement and can be easily integrated into many existing MARL approaches.

A.6 Multiagent Particle Environment

We evaluate the proposed DPN and HPN on the classical Multiagent Particle Environment (MPE) (Lowe et al., 2017) tasks, where the actions only consist of movement actions. Therefore, only the permutation invariance property is needed. We follow the experimental settings of PIC (permutation-invariant Critic for MADDPG, which utilizes GNN to achieve PI, i.e., GNNMADDPG) (Liu et al., 2020) and apply our DPN and HPN to the centralized critic Q-function of MADDPG (Lowe et al., 2017). Each component x_i represents the concatenation of agent i's observation and action. The input set X_j contains all agents' observation-actions. We implement the code based on the official PIC. The baselines we considered are PIC (Liu et al., 2020), DA-MADDPG (Ye et al., 2020) and MADDPG (Lowe et al., 2017). The tasks we consider are as follows:

- **Cooperative navigation**: n agents move cooperatively to cover L landmarks in the environment. The reward encourages the agents to get close to landmarks. An agent observes its location and velocity, and the relative location of the landmarks and other agents.

- **Cooperative predator-prey**: n slower predators work together to chase M fast-moving prey. The predators get a positive reward when colliding with prey. Preys are environment controlled. A predator observes its location and velocity, the relative location of the L landmarks and other predators and the relative location and velocity of the prey.

The learning curves of different methods in the cooperative navigation task (the agent number $n = 6$) and the cooperative predator-prey task (the agent number $n = 3$) are given in Fig.9. Besides, We further test HPN on two more cooperative navigation tasks with 100 and 200 agents respectively. The learning curves are shown in Figure 17. The results show that HPN-MADDPG can significantly improve the performance of vanilla MADDPG and achieves superior sample efficiency and converged performance than PIC. All experiments are repeated for five runs with different random seeds. We see that our HPN-MADDPG outperforms the PIC, DA-MADDPG and MADDPG baselines in these two tasks, which validates the superiority of our permutation-invariant designs.

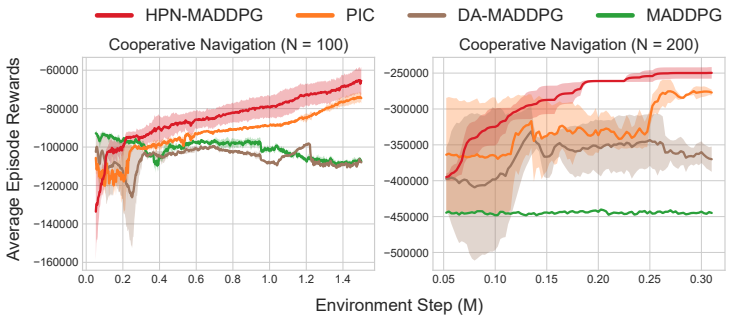


Figure 17: Comparisons of HPN-MADDPG against PIC, DA-MADDPG and MADDPG in cooperative navigation with 100 and 200 agents.

A.7 Google Research Football

We evaluate HPN in two Google Research Football (GRF) academic scenarios: 3_vs_1_with_keeper and counterattack_hard. In these tasks, we control the left team players except for the goalkeeper. The right team players are controlled by the built-in rule-based bots. The agents need to coordinate their positions to organize attacks and only scoring leads to rewards. The observations are factorizable and are composed of five parts: ball information, left team, right team, controlled player information and match state. Detailed feature lists are shown in Table 4. Each agent has 19 discrete actions, including moving, sliding, shooting and passing. Following the settings of CDS (Chenghao et al., 2021), we also make a reasonable change to the two half-court offensive scenarios: we will lose if our players or the ball returns to our half-court. All methods are tested with this modification. The final reward is +100 when our team wins, -1 when our player or the ball returns to our half-court, and 0 otherwise.

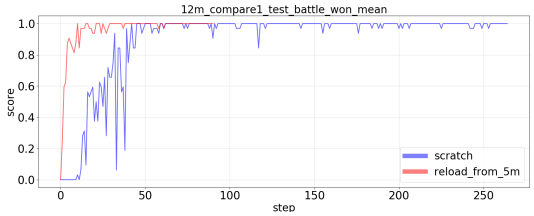
We apply HPN to QMIX and compare it with the SOTA CDS-QMIX (Chenghao et al., 2021). In detail, when applying HPN to QMIX, both the PI actions, e.g., moving, sliding and shooting, and the PE actions, e.g., long_pass, high_pass and short_pass are considered. For each player, since the targets of these passing actions directly correspond to its teammates, we apply the PE output layer to generate the Q-values of these passing actions, where the hypernetwork takes each ally player's features as input and generates the weight matrices for the passing actions. Besides, in the official GRF environment, as we cannot directly control which teammates the current player passes the ball to, we take a max pooling over all ally-related Q-values to get the final Q-values for the three passing actions. We show the average win rate across 5 seeds in Fig.10. HPN can significantly boost the performance of QMIX and our HPN-QMIX outperforms the SOTA method CDS by a large margin in these two scenarios.

A.8 Generalization: Can HPN generalize to a new task with a different number of agents?

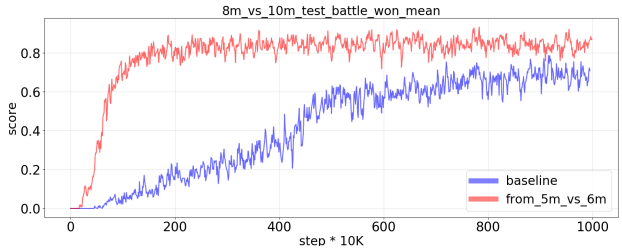
Apart from achieving PI and PE, another benefit of HPN is that it can naturally handle variable numbers of inputs and outputs. Therefore, as also stated in the conclusion section, HPN can be potentially used to design more efficient multitask learning and transfer learning algorithms. For

Observation	Player	Absolute position
		Absolute speed
	Left team	Relative position
		Relative speed
	Right team	Relative position
		Relative speed
	Ball	Absolute position
		Belong to (team ID)
State	Left team	Absolute position
		Absolute speed
		Tired factor
		Player type
	Right team	Absolute position
		Absolute speed
		Tired factor
		Player type
	Ball	Absolute position
		Absolute speed
		Absolute rotate speed
		Belong to (team ID)
		Belong to (player ID)

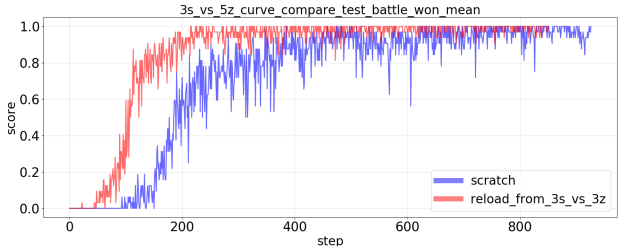
Table 4: The feature composition of the observation and the state in Google Research Football



(a) Transfer learning results on 12m. Red: reload the learned policy in 5m to 12m and then continuously train the policy. Blue: learn from scratch.



(b) Transfer learning results on 8m_vs_10m. Red: reload the learned policy in 5m_vs_6m to 8m_vs_10m and then continuously train the policy. Blue: learn from scratch.



(c) Transfer learning results on 3s_vs_5z. Red: reload the learned policy in 3s_vs_3z to 3s_vs_5z and then continuously train the policy. Blue: learn from scratch.

Figure 18: Transferring the learned HPN-VDN policy in one task to a new task with a different number of agents.

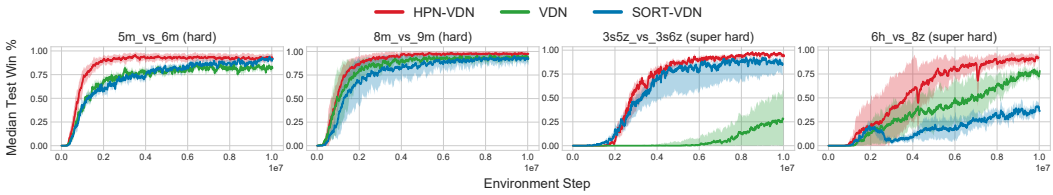
example, we can directly transfer the learned HPN policy in one task to new tasks with different numbers of agents and improve the learning efficiency in the new tasks. Transfer learning results of 5m $\rightarrow$ 12m, 5m_vs_6m $\rightarrow$ 8m_vs_10m, 3s_vs_3z $\rightarrow$ 3s_vs_5z are shown in Fig.18. We see that the previously trained HPN policies can serve as better initialization policies for new tasks.

A.9 TO ACHIEVE PI AND PE, WHAT IF WE JUST SORT THE ENTITIES ACCORDING TO DISTANCE FROM THE FOCAL AGENT?

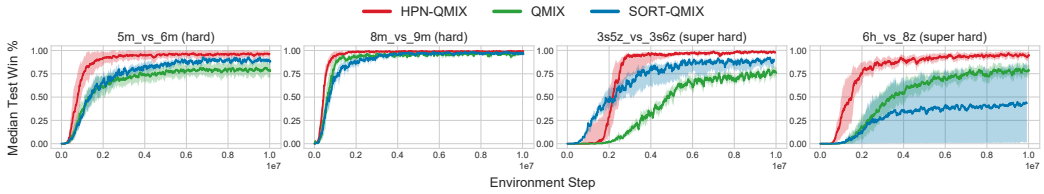
(1) When we first started working on this project, we have also considered a similar baseline: we sort the entities according to (type, distance), i.e., according to their types first and then the relative distances if two entities' types are same. But we found that this solution do not always work well. Here, we provide the learning curves of HPN, QMIX, VDN, SORT-QMIX, and SORT-VDN in 4 hard and super hard scenarios on SMAC in Figure 19. The results show that the sorting baseline can slightly improve the performance of vanilla QMIX/VDN in 5m_vs_6m and 3s5z_vs_3s6z. However, in 8m_vs_9m and 6h_vs_8z, it harms the performance.

(2) The reason is that each entity has many types of features, e.g., relative x, relative y, relative distance, entity type, health point, shield, etc. Relative distance is just one of them. Simply sorting the entities by their relative distances while ignoring the influences of the other features may not be appropriate. Besides, as different x and y can have the same distance and different distances can have the same order, the same $o_i[j]$ may be arranged at different positions and be multiplied by different 'weight matrices' (according to $z_i = \sum_{j=1}^{m} o_i[j]\mathcal{W}_{\text{in}}[j]$). Therefore, learning may become unstable if we frequently reorder the inputs by distance only.

(3) Thus, our target is **not only matching the observation and action belonging to the same entity but stabilizing the learning process by always assigning the same weight matrix $\mathcal{W}_{\text{in}}[j]$, i.e., a stable weight, to the same entity features** $o_i[j]$ no matter where $o_i[j]$ is arranged. In this paper, we propose DPN and HPN to achieve this.



(a) Comparisons of HPN-VDN, VDN, and a baseline that sorts the entities in observation according to their distances from the focal agent (denoted as SORT-VDN).



(b) Comparisons of HPN-QMIX, QMIX, and a baseline that sorts the entities in observation according to their distances from the focal agent (denoted as SORT-QMIX).

Figure 19: Comparisons of HPN with the sorting based baseline.

A.10 EVALUATE HPN ON SMAC-V2

SMAC-v2 makes three major changes to SMAC: randomising start positions, randomising unit types, and restricting the agent field-of-view and shooting range to a cone. These first two changes increase more randomness to challenge contemporary MARL algorithms. The third change makes features harder to infer and adds the challenge that agents must actively gather information (require more efficient exploration). Since our target is not to design more efficient exploration algorithms, we keep the field-of-view and attack of the agents a full circle as in SMAC.

- **Random Start Positions**: Random start positions come in two different types. First, there is the *surrounded* type, where the allied units are spawned in the middle of the map, and surrounded by enemy units. This challenges the allied units to overcome the enemies approach from multiple angles at once. Secondly, there are the *reflect_position* scenarios. These randomly select positions for the allied units, and then reflect their positions in the midpoint of the map to get the enemy spawn positions. Example figures are shown in Figure 20 below.

- **Random Unit Types**: Battles in SMACv2 do not always feature units of the same type each time, as they did in SMAC. Instead, units are spawned randomly according to certain pre-fixed probabilities. Units in StarCraft II are split up into different races. Units from different races cannot be on the same team. For each of the three races (Protoss, Terran, and Zerg), SMACv2 uses three unit types. Detailed generation probabilities are shown in Figure 21.

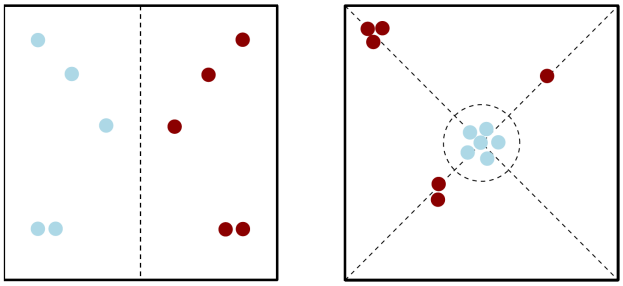


Figure 20: Examples of the two different types of start positions, opposite and surrounded. Allied units are shown in blue and enemy units in dark red.

Race	Unit	Generation Probability
Terran	Marine	0.45
	Marauder	0.45
	Medivac	0.1
Protoss	Stalker	0.45
	Zealot	0.45
	Colossus	0.1
Zerg	Zergling	0.45
	Hydralisk	0.45
	Baneling	0.1

Figure 21: Detailed generation probabilities of the three types of units for the three races (Protoss, Terran, and Zerg).

Our HPN can naturally handle the two types of new challenges. Thanks to the PI and PE properties, our HPN is more robust to the randomly changed start positions of the entities. Thanks to the entity-wise modeling and using hypernetwork to generate a customized *weight matrix* for each type of unit, HPN can handle the randomly generated unit types as well. The comparisons of HPN-VDN with VDN on three difficult scenarios across the three races (Protoss, Terran, and Zerg) are shown in Figure 22. Results show that our HPN significantly improves the sample efficiency and the converged test win rates of the baseline VDN.

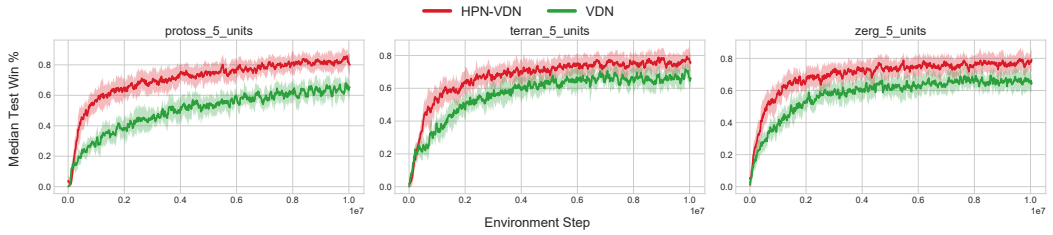


Figure 22: The learning curves of HPN-VDN and VDN in 3 difficult scenarios of SMAC-v2.

B TECHNICAL DETAILS

B.1 DECENTRALIZED PARTIALLY OBSERVABLE MDP

We model a fully cooperative multiagent task as a Dec-POMDP (Oliehoek & Amato, 2016), which is defined as a tuple $\langle \mathcal{N}, \mathcal{S}, \mathcal{O}, \mathcal{A}, P, R, Z, \gamma \rangle$. $\mathcal{N}$ is a set of n agents. $\mathcal{S}$ is the set of global states. $\mathcal{O} = \{\mathcal{O}_1, \dots \mathcal{O}_n\}$ denotes the observation space for n agents. $\mathcal{A} = \mathcal{A}_1 \times \dots \times \mathcal{A}_n$ is the joint action space, where $\mathcal{A}_i$ is the set of actions that agent i can take. At each step, each agent i receives a private observation $o_i \in \mathcal{O}_i$ according to the observation function $Z(s, \mathbf{a}) : \mathcal{S} \times \mathcal{A} \to \mathcal{O}$, and produces an action $a_i \in \mathcal{A}_i$ by a policy $\pi_i(a_i|o_i)$. All agents' individual actions constitute a joint action $\mathbf{a} = \langle a_1, ..., a_n \rangle \in \mathcal{A}$. Then the joint action $\mathbf{a}$ is executed and the environment transits to the next state s' according to the transition probability $P(s'|s, \mathbf{a})$. All agents receive a shared global reward according to the reward function $R(s, \mathbf{a})$. All individual policies constitute the joint policy $\boldsymbol{\pi} = \pi_1 \times \dots \times \pi_n$. The target is to find an optimal joint policy $\boldsymbol{\pi}$ which could maximize the expected return $R_t = \sum_{t=0}^{T} \gamma^t r\left(s^t, \mathbf{a}^t\right)$, where γ is a discount factor and T is the time horizon. The joint action-value function is defined as $Q_{\boldsymbol{\pi}}(s_t, \mathbf{a}_t) = \mathbb{E}_{\boldsymbol{\pi}, P}\left[R_t | s_t, \mathbf{a}_t\right]$. Each agent's individual action-value function is denoted as $Q_i(o_i, a_i)$.

B.2 IMPLEMENTATION DETAILS OF OUR APPROACH AND BASELINES

The key points of implementing the baselines and our methods are summarized here:

(1) **DPN and HPN**: The proposed two methods inherently support heterogeneous scenarios since the entity's 'type' information has been taken into each entity's features. And the sample efficiency can be further improved within homogeneous agents compared to fixedly-ordered representation. For MMM and MMM2, we implemented a permutation-equivariant 'rescue-action' module for the only Medivac agent, which uses similar prior knowledge to ASN and UPDeT, i.e., action semantics. To focus on the core idea of our methods, we omitted these details in the method section.

The objective to train the weight selection network of PDN. As stated in the last paragraph of Section 4, all parameters of DPN are trained end-to-end with backpropagation according to the RL loss function. The weight selection network and the other networks work cooperatively to minimize the overall RL loss function.

How PI is achieved of DPN. As described in Section 4.2, for each $o_i[j]$, the weight selection network outputs the probability of selecting each weight matrix. During the forward pass at training step t, given the parameter snapshot of the weight selection network, the output probability of selecting each weight matrix is fixed. For each $o_i[j]$, we select the weight matrix with the maximum probability. However, directly selecting the argmax index is not differentiable. To make the selection process trainable, we apply a Straight Through Estimator [7] to get the one-hot encoding of the argmax index. We denote it as $\hat{p}_{\text{in}}\left(o_i[j]\right)$. The weight matrix with the maximum probability can be acquired by $\hat{p}_{\text{in}}\left(o_i[j]\right) \cdot \mathcal{W}_{\text{in}}$. As selecting the weight matrix with the maximum probability is a deterministic process, according to Equation (5), PI is guaranteed.

Besides, to encourage more exploration at the beginning of training, we also add small gumbel noises (Jang et al., 2016) to the 'logits' within the epsilon anneal time. Within this interval, PI cannot be

Figure 23: Applying Data Augmentation (DA) to the SMAC benchmark.

strictly guaranteed. When the epsilon anneal schedule is over, PI will be strictly guaranteed. The Straight Through Estimator written in PyTorch is shown below:

```
def straight_through(y_soft, dim):
    # Straight Through Estimator.
    index = y_soft.max(dim, keepdim=True)[1]
    y_hard = th.zeros_like(y_soft).scatter_(dim, index, 1.0)
    ret = y_hard - y_soft.detach() + y_soft
    return ret
```

(2) **Data Augmentation (DA)** (Ye et al., 2020): we apply the core idea of Data Augmentation (Ye et al., 2020) to SMAC by randomly generating a number of permutation matrices to shuffle the 'observation', 'state', 'action' and 'available action mask' for each sample simultaneously to generate more training data. An illustration of the Data Augmentation process is shown in Fig.23. A noteworthy detail is that since the attack actions are permutation-equivariant to the enemies in the observation, the same permutation matrix M_2 that is utilized to permute o_i^{enemy} should also be applied to permute the 'attack action' and 'available action mask of attack action' as well. The code is implemented based on PyMARL2 [8] for fair comparison.

(3) **Deep Set** (Zaheer et al., 2017; Li et al., 2021): the only difference between SET-QMIX and the vanilla QMIX is that the vanilla QMIX uses a fully connected layer to process the fixedly-ordered concatenation of the m components in o_i while SET-QMIX uses a shared embedding layer $h_i = \phi(x_i)$ to separately process each component x_i in o_i first, and then aggregates all h_is by sum pooling. The code is also implemented based on PyMARL2 for fair comparison.

(4) **GNN**: Following PIC (Liu et al., 2020) and DyAN (Wang et al., 2020b), we apply GNN to the individual Q-network of QMIX (denoted as GNN-QMIX) to achieve permutation-invariant. The code is also implemented based on PyMARL2.

(5) **ASN** (Wang et al., 2019): we use the official code and adapt the code to PyMARL2 for fair comparison.

(6) **UPDeT** (Hu et al., 2021b): we use the official code [9] and adapt the code to PyMARL2 for fair comparison.

(7) **VDN and QMIX**: As mentioned in Section 3, vanilla VDN/QMIX uses fixedly-ordered entity-input and fixedly-ordered action-output (both are sorted by agent/enemy indices). Although VDN and QMIX do not explicitly consider the permutation invariance and permutation equivariance prop-

[8] https://github.com/hijkzzz/pymarl2
[9] https://github.com/hhhusiyi-monash/UPDeT

erties, they train a permutation-sensitive function to figure out the input-output relationships according to their fixed positions, which is implicit and inefficient.

All code of the baselines as well as our Networks is attached in the supplementary material.

C HYPERPARAMETER SETTINGS

For all MARL algorithms we use in SMAC (Samvelyan et al., 2019) (under the MIT License), we keep the hyperparameters the same as in PyMARL2 (Hu et al., 2021a) (under the Apache License v2.0). We list the detailed hyperparameter settings used in the paper below in Table 5 to help peers replicate our experiments more easily. Besides, the code of our methods as well as the baselines is attached in the supplementary material.

Table 5: Hyperparameter Settings of VDN-based or QMIX-based Methods.

Parameter Name	Value
Exploration-related	
action_selector	epsilon_greedy
epsilon_start	1.0
epsilon_finish	0.05
epsilon_anneal_time	100000 (500000 for *6h_vs_8z*)
Sampler-related	
runner	parallel
batch_size_run	8 (4 for *3s5z_vs_3s6z*)
buffer_size	5000
t_max	10050000
Agent-related	
mac	hpn_mac for HPN, dpn_mac for DPN, set_mac for Deep Set, updet_mac for UPDeT and n_mac for others
agent	hpn_rnn for HPN, dpn_rnn for DPN, set_rnn for Deep Set, updet_rnn for UPDeT and rnn for others
HPN_hidden_dim	64 (only for HPN)
HPN_layer_num	2 (only for HPN)
permutation_net_dim	64 (only for DPN)
Training-related	
softmax_tau	0.5 (only for DPN)
learner	nq_learner
mixer	qmix or vdn
mixing_embed_dim	32 (only for qmix-based)
hypernet_embed	64 (only for qmix-based)
lr	0.001
td_lambda	0.6 (0.3 for *6h_vs_8z*)
optimizer	adam
target_update_interval	200

D COMPUTING ENVIRONMENT

We conducted our experiments on an Intel(R) Xeon(R) Platinum 8171M CPU @ 2.60GHz processor based system. The system consists of 2 processors, each with 26 cores running at 2.60GHz (52 cores in total) with 32KB of L1, 1024 KB of L2, 40MB of unified L3 cache, and 250 GB of memory. Besides, we use 2 GeForce GTX 1080 Ti GPUs to facilitate the training procedure. The operating system is Ubuntu 16.04.

E LIMITATIONS

We currently follow the settings of (Qin et al., 2022; Wang et al., 2020b; Hu et al., 2021b; Long et al., 2019; Wang et al., 2019), where the configuration of the input-output relationships and the observation/state structures are set manually.

What if the observations are images or the structural information is not available?

The high-level idea of this paper is to leverage some formats of symmetries to reduce the size of the search space. Since typical MARL benchmarks represent observations as factorizable vectors (which can provide more direct and compact information than images), we currently focus on the permutation symmetries, i.e., PI and PE.

For image inputs, the rotational or reflectional symmetries are more prominent characteristics. Thus, we could leverage rotation invariance or rotation equivariance to design better MARL algorithms, which is also a novel research direction.

For vector inputs, when the structural information is unknown, a potential solution is:

- (1) Learning action representations using a forward model. We want to learn action representations that can reflect the effects of actions on the environment and other agents. The effect of an action can be measured by the induced reward and the change in the states.
- (2) Using all actions' representations as queries and using all entities' embedded features (potentially generated by HPN) as keys and values, we leverage the self-attention mechanism to generate the Q-values of each action. Since the self-attention computation is invariant to the input entities' order, PI and PE are achieved. And the input-output relationships may be learned implicitly by the self-attention mechanism.

Automatically detecting such structural information is interesting and we leave this as future works.