A. Network Architecture

With the complex observation and action structures, the design of the policy network architecture is critical, in order to extract the right information from the raw observation and issue the proper action. The overall architecture of the policy network of SCC is shown in Fig. 4, which was designed based on the performance in supervised learning. With the same input and output interfaces provided by the game engine, the overall policy structure of SCC is similar to that of AlphaStar, while there are some key components that are different, which will be discussed in detail below.

A.1. Minimap Size

As part of the observation, the minimap consists of 6 planes of spatial information, i.e., height, visibility, player relative, alerts, pathable and buildable, which is directly provided by the raw interface and its size can be specified. AlphaStar uses minimaps of size 128×128, as it is the only part of spatial input, it also consists most of the data size. We reduce the minimap size from 128×128 to 64×64, which reduces the per sample data size by around 49.5%. We compared the overall performance of the two settings in supervised learning, and observed that the two different minimap sizes lead to almost identical training and evaluation results. As a result, we finally adopt minimap of size 64×64, for smaller storage and higher computational efficiency.

A.2. Group Transformer

Among all parts of the observations, the set of units is most informative, with each one consists of the unit type, owner, position and many other fields. The whole set of units can be divided into three groups, namely my units, enemy units and neutral units. In AlphaStar, all three groups of units are put together and is then processed by transformer. In our point of view, the three groups of units are naturally separate and processing them separately allows a more comprehensive expression. It will also facilitate the processing later, for example, the selected units head can only select units from my units.

For each group of units, one multi-head self attention block is applied with the unit feature vectors in the same group as the queries, keys and values, and two multi-head cross attention block are applied with the queries being the unit feature vectors in the same group and that of the other two unit groups being the keys and values (Vaswani et al., 2017). For each of the multi-head attention blocks, there are two heads each of size 32, and the outputs of the three multi-head attention blocks are concatenated together. The whole process is repeated three times to yield the final encoded unit features for the three groups of units, which will act as the attention keys being used in the selected units and the target unit heads, and also be reduced into a single vector being fed into the LSTM.

A.3. Attention-based Pooling

To reduce the extracted unit feature vectors of each group into a single vector, a simple average-pooling is applied in AlphaStar. In a similar scenario, the max-pooling is utilized in OpenAI five. Both operators are non-trainable which potentially limits their expressive capability. Inspired by the attention-based multiple instance learning (Ilse et al., 2018), we propose the trainable attention-based pooling based on the multi-head attention. More specifically, the extracted unit feature vectors are treated as the keys and values in a multi-head attention, and multiple trainable weight vectors are created as the queries. The outputs of the multi-head attention are then flattened to yield the reduced vector, with the dimension being defined by the number of query weight vectors, and also the number of head and head size of the multi-head attention. It is observed that this trainable reduce operator gives better performance in supervised learning.

A.4. Conditional Structures

With the observations encoded into vector representations, concatenated and processed by a residual LSTM block, the action is then decided based on the LSTM output. To manage the structured, combinatorial action space, AlphaStar uses an auto-regressive action structure in which each subsequent action head conditions on all previous ones, via an additive auto-regressive embedding going through all heads. During the design of SCC, we found that it is critical for all other heads to condition on the selected action, which defines what to do, for example, moving, attacking, or training a unit, and for the two heads that decide the target unit or position, it is also helpful to condition on the selected units. However, for all other heads, it is not necessary to condition on each other, thus the order of these heads, say the skip frames and queued heads, also does not matter. In addition, in view of the limited expressive capability of the additive operator, we adopt the structure of concatenation followed by a fully connected layer, to provide full flexibility for the network to learn a better conditional relationship.

Due to the importance of the selected action for some action heads, we propose to condition on it further via the conditioned concat-attention. To be specific, the concat-attention is applied in the target unit head to output the probability distribution from which the target unit is sampled. In the original concat-attention (Luong et al., 2015), the attention score is computed as follows:

$$\mathrm{score}(\mathbf{q}, \mathbf{u}_i) = \mathbf{v}^T \tanh(\mathbf{W}[\mathbf{q}; \mathbf{u}_i]),$$

where $\mathbf{q}$ is the query, $\mathbf{u}_i$ is the encoded unit feature vector serving as the key, $\mathbf{W}$ and $\mathbf{v}$ are the weights to be trained.

In the proposed conditioned concat-attention, we replace the single weight vector $\mathbf{v}$ by the embedding of the sampled selected action. Since there are different embeddings for different selected actions, the conditioned concat-attention provides the capability of defining different function mappings for different selected actions. It makes sense intuitively, since different selected actions may need totally different criteria for selecting the target, for example, the repair action may want to target damaged alliance units and the attack action may want to target nearby enemy units. The same conditioned concat-attention is also applied inside the pointer network in the selected units head, in place of the original simple dot product attention, to also further condition on the selected action.

B. Training Platform

We developed a highly scalable training system based on the actor-learner architecture. The diagram of the training process for one agent is shown in Fig. 5. In our system, there are a number of Samplers, each of which continuously runs a single SC2 environment and collects training data. When a rollout of training data is generated, it will be sent to a Trainer over the network and saved into a local buffer. When there are sufficient training data in the local buffer, Trainer starts to execute a training step, during which it repeatedly samples a batch of training data, computes the gradient, and executes MPI all-reduce operation. When the training data has been utilized some number of times, this training step is terminated and the local buffer is cleared. After that, Trainers distribute updated network parameters to Predictors through network. Predictor provides batched inference service on GPU for Samplers to make efficient use of resource. For each agent, we run about 1000 (AlphaStar 16,000) concurrent SC2 matches, and can collect a sample batch of total size 144,000 and perform a training step in about 180 seconds. So about 800 (AlphaStar 50,000) agent steps can be processed per second on average.

To support the whole league training, we also build a league system as shown in Fig. 6, which consists of four components, i.e., Storage, Predictor, Scheduler and Evaluator. We introduce each component in the following.

- Storage: we use a MySQL DB service to store league information, like agent-id, agent-type, model-path and so on. The evaluation results of win rates between agents are also saved into the storage.
- Predictor: we use a single cluster of Predictors to provide inference service for all agents in the league, and the cluster is shared by training agents. Not only GPUs, CPUs are also used as Predictors.
- Scheduler: Scheduler maintains the predictors and provides naming service, which receives agent-id and returns an available Predictor (allocate one if not exists). As the number of agents in league keeps growing, there is no guarantee for each agent with at least one GPU Predictor, in that case the CPU Predictors are used instead. The distribution of requests over agents also changes in the training process, Scheduler is also responsible to auto scale the Predictor number for agents according to the request amount.
- Evaluator: an Evaluator is needed to get the win rates between agents in league. The evaluation results are saved into Storage and will be used to calculate the matchmaking distribution.

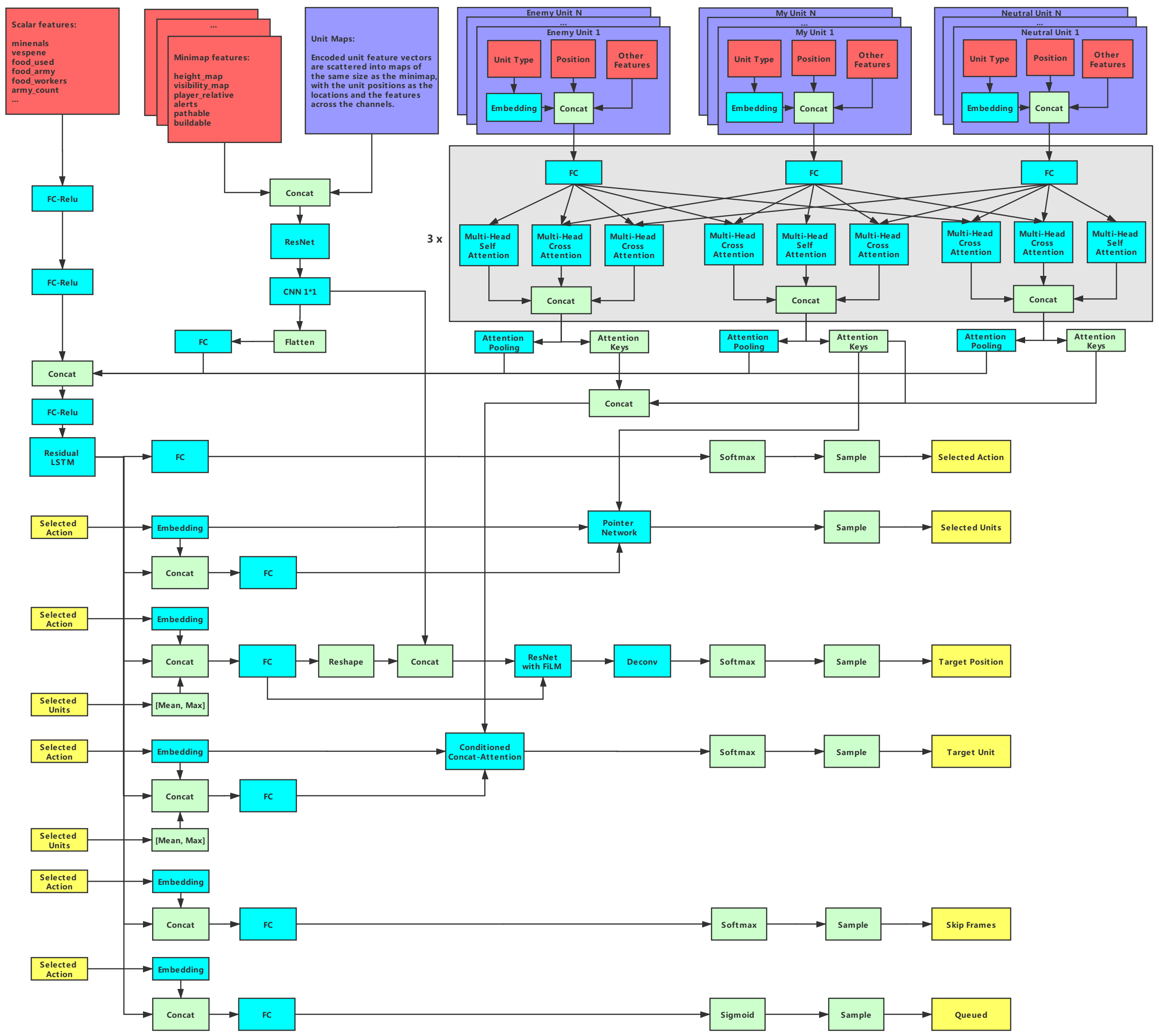

Figure 4. Overview of the policy network architecture of SCC.

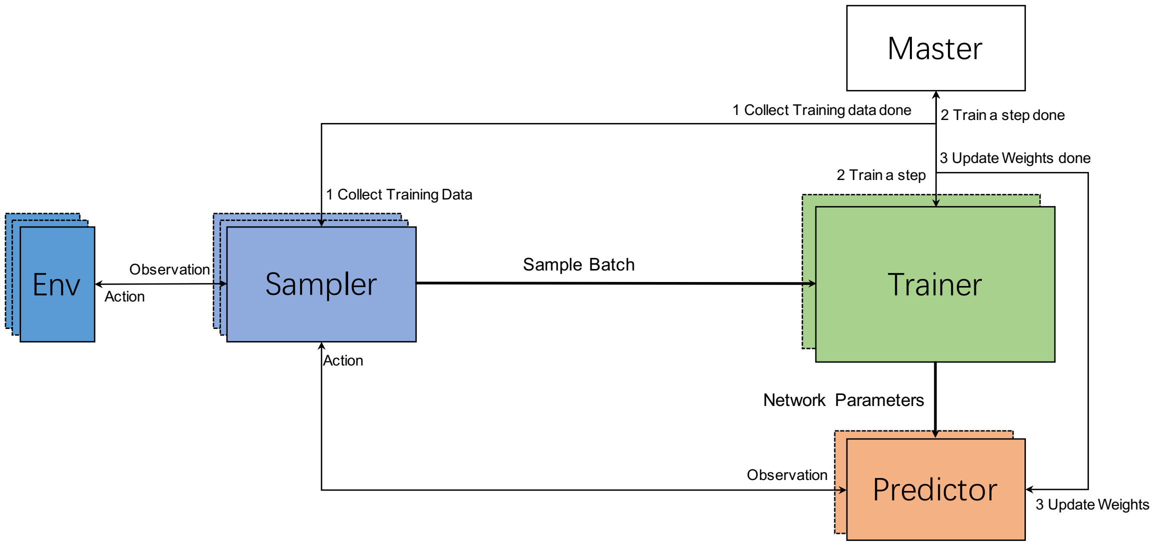

Figure 5. Diagram of the training process.

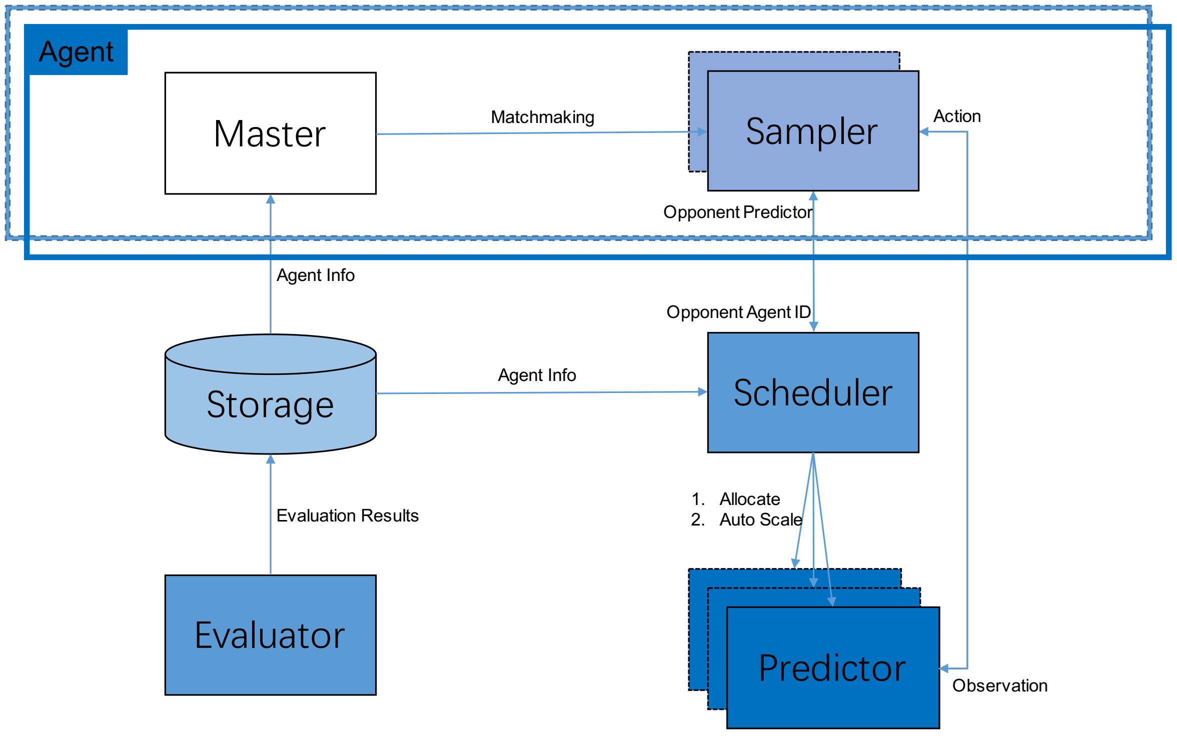

Figure 6. Diagram of the league training.