8. Appendix

8.1. Discretised continuous control

8.1.1. Hyperparameters

For our experiments in discretised continous control, we use a standard DQN trainer (Mnih et al., 2015) with the following parameters.

Parameter	Value
batch size	128
replay buffer size	10000
target update interval	200
ϵ initial	1.0
ϵ final	0.1
ϵ decay	25000 env steps
ℓ lead-in	25000 env steps
ℓ growth	25000 env steps
env steps per model udpate	4
Adam learning rate	5e-4
Adam ϵ	1e-4

For GAS experiments, we keep the mixing coefficient $\alpha = 0$ for 25000 environment steps, and then increase it linearly by 1 every 25000 steps until reaching the maximum value. We use $\gamma = 0.998$ for our Acrobot experiments, but reduce it to $\gamma = 0.99$ for Mountain Car to prevent diverging Q-values.

Our model consists of fully-connected ReLU layers, with 128 hidden units for the first and 64 hidden units for all subsequent layers. Two layers are applied as an encoder. Then, for each ℓ one layer is applied on the current embedding to produce a new embedding, and an evaluation layer on that embedding produces the Q-values for that level.

8.2. StarCraft micromanagement scenarios

8.2.1. Scenarios and learned strategies

We explore five Starcraft micromanagement scenarios: 50 hydralisks vs 50 hydralisks, 80 marines vs 80 marines, 80 marines vs 85 marines, 60 marines vs 65 marines, 95 zerglings vs 50 marines. In these scenarios, our model controls the first set of units, and the opponent controls the second set.

The opponent is a scripted opponent that holds its location until an opposing unit is within range to attack. Then, the opponent will engage in an "attack-closest" behavior, as described in Usunier et al. (2016), where each unit individually targets the closest unit to it. Having the opponent remain stationary until engaged makes this a more difficult problem – the agent must find its opponent, and attack into a defensive position, which requires good positions prior to engagement.

As mentioned in section 6.2, all of our scenarios require control of a much larger number of units than previous work. The 50 hydralisks and 80v80 marines scenarios are both imbalanced as a result of attacking into a defensive position. The optimal strategy for 80 marines vs 85 marines and 60 vs 65 marines requires slightly more sophisticated unit positioning, and the 95 zerglings vs 50 marines scenario requires the most precise positioning. The agent can use the enemy's initial stationary positioning to its advantage by slightly surrounding the opponent in a concave, ensuring that the outermost units are in its attack range, but far enough away to be out of range of the center-most enemy units. Ideally, the timing of the groups in all scenarios should be coordinated such that all units get in range of the opponent at roughly the same point in time. Figure 5 shows how our model is able to exhibit this level of unit control.

8.2.2. Features

We use a standard features for the units and map, given by TorchcraftAI [1]

For each of the units, the following features are extracted:

- Current x, y positions.
- Current x, y velocities.
- Current hitpoints
- Armor and damage values
- Armor and damage types
- Range versus both ground and air units
- Current weapon cooldown
- A few boolean flags on some miscellaneous unit attributes

Approximate normalization for each feature keep its value approximately between 0-1.

For the map, the following features are extracted for each tile in the map:

- a one-hot encoding of tile's the ground height (4 channels)
- boolean representing or not the given tile is walkable
- boolean representing or not the given tile is buildable
- and boolean representing or not the given tile is covered by fog of war.

The features form a $HxWx7$ tensor, where our map has height H and width W.

[1] https://github.com/TorchCraft/TorchCraftAI

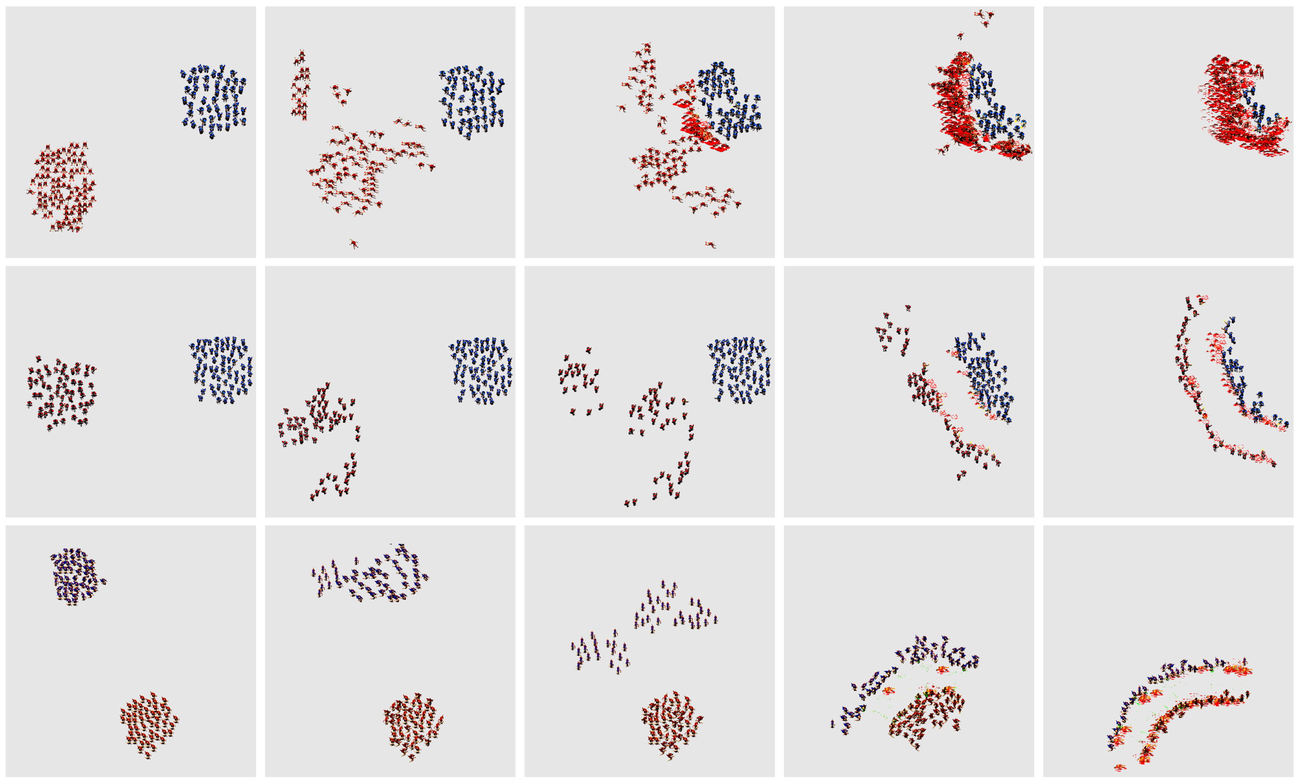

Figure 5: Final learned policies of StarCraft micromanagement unit control with growing action spaces. Scenarios shown from left to right at time 0, 3, 5, 10, 15 seconds. Top to bottom the scenarios are: 60 marines vs 65 marines, 50 hydralisks vs 50 hydralisks, 95 zerglings vs 50 marines. In these examples, the opponent is always on the right, and the agent controlled by model trained with GAS is on the left.

8.2.3. Environment details

We use a frame-skip of 25, approximately 1 second of real time, allowing for reasonably fine-grained control but without making the exploration and credit assignment problems too challenging.

We calculate at every timestep the difference in total health points (HP) and number of units for the enemy from the last step, normalised by the total starting HP and unit count. As a reward function, we use the normalised damage dealt, plus 4 times the normalised units killed, plus an additional reward of 8 for winning the scenario by killing all enemy units. This reward function is designed such that the agent gets some reward for doing damage and killing units, but the reward from doing damage will never be greater than from winning the scenario. Ties and timeouts are considered losses.

8.3. Experimental details

8.3.1. Model

As described in Section 6.2.2 a custom model architecture is used for Starcraft micromanagement. Each unit's feature vector is embedded to size 128 in step 2 of Figure 3. The grid where the unit features and map features are scattered onto is the size of the Starcraft map of the scenario in walk-tiles downsampled by a factor of 8. After being embedded, the unit features for ally and enemy units are concatenated with the downsampled map features and sent into a ResNet encoder with four residual blocks (stride 7 padding 3). The output is an embedding of size 64.

The decoder uses a mean pooling over the embedding cells as described in Section 6.2.2. Each evaluator is a 2-layer MLP with 64 hidden units and 17 outputs, one for each action. All layers are separated with ReLU nonlinearities.

8.3.2. Training hyperparameters

We use 64 parallel actors to collect data in a short queue from which batches are removed when they are consumed by the learner. We use batches of 32 6-step segments for each update.

For the Q-learning experiments, we used the Adam optimizer with a learning rate of 2.5×10^{-4} and $\epsilon = 1 \times 10^{-4}$. For the MM baseline experiments, we use a learning rate of 1×10^{-4}, entropy loss coefficient of 8×10^{-3} and value loss coefficient 0.5. The learning rates and entropy loss coefficient were tuned by random search, training with $\mathcal{A}_0$ from scratch on the 80 marines vs 80 marines scenario with 10 configurations sampled from $\texttt{log\_uniform}(-5, -3)$ for the learning rate and $\texttt{log\_uniform}(-3, -1)$ for the entropy loss coefficient.

For Q-learning, we use an ϵ-greedy exploration strategy , decaying ϵ linearly from 1.0 to 0.1 over the first 10000 model updates. We also use a target network that copies the behaviour model's parameters every 200 model updates.

We also use a linear schedule to grow the action-space. There is a lead in of 5000 model updates, during which the action-space is held constant at $\mathcal{A}_0$, to prevent the action space from growing when ϵ or the policy entropy is too high. The action-space is then grown linearly at a rate of 10000 model updates per level of restriction, so that after 10000 updates, we act entirely at $\mathcal{A}_1$ and after 20000, entirely at $\mathcal{A}_2$.