Table 4. Dataset Nutrition Label for the StarCraftImage dataset.

StarCraftImage			
Motivation	Existing multi-agent spatial reasoning datasets have focused on hyper-realism, leading to large image/video sizes which are expensive to prototype on. Therefore, the ML community lacks an easy-to-use, yet complex behaviored, dataset for prototyping new spatial reasoning algorithms. To fill this gap, we construct a StarCraftImage, a dataset based on StarCraft II replays that has intelligent human behaviors, while being as easy to use as MNIST and CIFAR10 and still enabling complex spatial reasoning tasks.		
Collection Process	We construct this dataset based on 60K StarCraft II replays (taken from Replay Pack 3.16.1 - Pack 1), and use PySC2 to create 3.6 million images which summarize a window of 255 game states from the replays. We additionally collect all relevant metadata such as game outcome and player races. Each window is represented as an image and processed with standard computer vision tools.		
Possible Uses	Since our dataset contains diverse, interesting, and intelligent (even adversarial) behaviors, and thus, it should provide a relevant simple benchmark for rapid prototyping of multi-agent spatial reasoning tasks and algorithms before trying on realistic data. For example, we map common spatial reasoning tasks to event prediction (i.e., game outcome and StarCraft race prediction), target identification (i.e., determining unit type from only knowing unit locations), and missing data imputation using both corruption models and the fog-of-war from the game engine.		
Availability	The StarCraftImage dataset is available for download at https://starcraftdata.davidinouye.com/ and available under a permissive CC BY 4.0 license. The code to recreate (or extend) the dataset extraction and processing, as well as code to load the data, rerun benchmarks, and recreate demos is maintained on GitHub with an MIT license.		
Metadata	Each window in the StarCraftImage dataset is paired with relevant metadata collected during the extraction process. The metadata for each window consists of 52 entries which belong to one of three main subgroups: dynamic metadata (which is information that is specific to each window), static metadata (which is information that is specific to each replay, but does not change across windows within a replay), and computed metadata (which is information added by a user e.g., a label for that window for a classification task). For details please seen "Metadata Description" subsection in the Appendix.		
Dataset Composition	The dataset comprises of three main formats: StarCraftHyper, StarCraftCIFAR10, and StarCraftMNIST		
	StarCraftHyper	StarCraftCIFAR10	StarCraftMNIST
Description	A 340x64x64 px hyperspectral image which has the most information of the three representations. Specifically, there is a channel for each unit type as well as creep and visibility, all collected for each player. Additionally there are channels which contain map info such as map height, placement grid, and unit pathing grids. Additionally, each image is paired with corresponding metadata.	A 32x32 RGB image dataset which was synthesized from the StarCraftHyper dataset such that the Player1 information is embedded into the Blue channel, Player2 to the Red channel, and Neutral information to the Green channel. Each RGB image comes with a corresponding label from one of 10 classes. This was done so it exactly matches the format of the CIFAR10 dataset.	A 28x28 Grayscale image dataset which was synthesized from the StarCraftCIFAR10 dataset such that the Player1 information is pushed into the range of [0.55, 1] px values, Player2 to the [0.0, 0.45] px values, and Neutral information to the range of [0.48, 0.52], and overlaid with decreasing precedence of P1, P2, N. Each grayscale image comes with a corresponding label from one of 10 classes. This was done so it exactly matches the format of the MNIST dataset.
How to access	After downloading the dataset, one can use the StracraftImage dataset class from the code repo. E.g., for accessing training data: sc2_train = StarCraftImage(data_root, subdir, train=True)	This can be used similar to CIFAR10 via unpickling the compressed StarCraftCIFAR10.tar.gz file. Or, the whole StarCraftImage dataset can be loaded in RGB format using the StarCraftCIFAR10 python class.	This can be used similar to MNIST via uncompressing the StarCraftMNIST.gz file. Or, the whole StarCraftImage dataset can be loaded in grayscale format using the StarCraftMNIST python class.
Data structure	If in a sparse format: sparse matrix with shape 340x64x64 If in a dense format: dense matrix with shape Nx64x64 where N is the max # of units at one location in a batch	Tuple with (3 x 32 x 32 numpy matrix (aka, RGB image), integer label between 0-9) -- matching the exact format of CIFAR10	Tuple with (28 x 28 numpy matrix (aka, grayscale image), integer label between 0-9) -- matching the exact format of MNIST
Number of Samples	3,607,787 images	50K training images, 10K test images	60K training images, 10K test images
Recommended Data Splits	Depends on task, see Metadata Description, Splitting dataset in corresponding paper.	Already split along into 10 balanced classes based off of (map_name, is_from_begining_of_game). See "Splitting dataset to k Classes" in paper for details.	Already split along into 10 balanced classes based off of (map_name, is_from_begining_of_game). See "Splitting dataset to k Classes" in paper for details.
Noise levels	No noise, but noise can be added as a preprocessing step (see "Simulated Data Corruption Models") in paper. However, if multiple units of the same type cross the same location within a window, only the most recent crossing will be recorded.	In addition to the StarCraftHyper noise, the unit type ID information is removed from this representation.	In addition to the StarCraftHyper noise, the unit type ID information is removed from this representation.
Contains Confidential/ Person Identifiable Data?	None	None	None

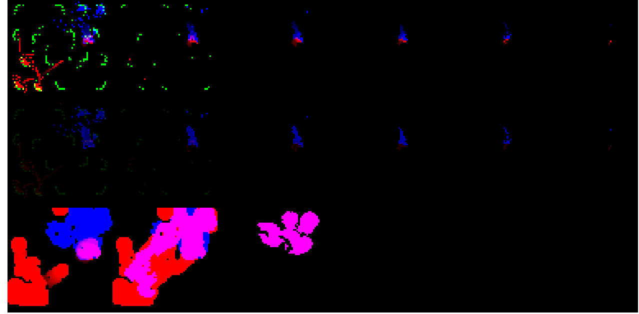
A. Dataset Availability, Licensing, and Management

The StarCraftImage dataset is available for download at https://starcraftdata.davidinouye.com/ which contains the full extracted data from the 3.6 million windows, the metadata for all windows, the StarCraftMNIST train/test datasets, and the StarCraftCIFAR10 train/test datasets. The code to recreate (or extend) the dataset extraction and processing, as well as code to load the data, rerun benchmarks, and recreate demos, can be found at the previous link, and is maintained on GitHub. Instructions for loading and using the dataset can be found in the README in the dataset as well as in the code repository. The dataset has been openly published under a permissive CC BY 4.0 license, and the code has been openly published on GitHub with an MIT license. The authors bear all responsibility in case of violation of rights and confirm the CC license for the provided datasets.

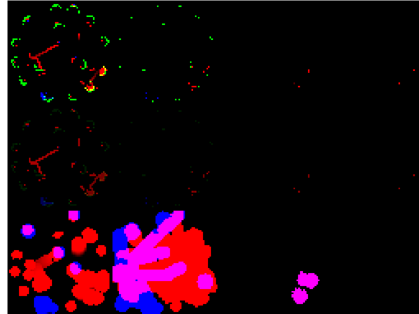
B. Direct Loading of Window Data

While we encourage using the corresponding PyTorch dataset classes that we have developed (one class for each representation) to load in StarCraftImage data, one can also directly access the data by loading in the relevant `.png` file and metadata row for each window. To assist with this direct data access, we now describe the data structure used to store the image data for each window (i.e., how to correspond each `.png` to the hyperspectral format H discussed in subsection 2.2).

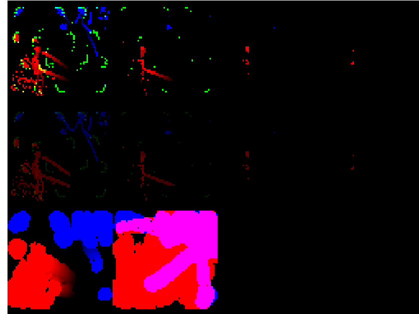
As a reminder, the bag-of-units representation collapses the channel, axis of our hyperspectral image H into k ID matrices and k timestamp matrices of size $(64, 64)$, where the ID matrix contains the PID of the units present at each (x, y) coordinate, the timestamp matrices contain the corresponding timestamp that the unit was last seen, and k is the max number of units present at one (x, y) location in H, seen in the top right of Fig. 2. We can further compress this bag-of-units representation by stacking the bag-of-units for player 1, player 2, and neutral to match the RGB structure of a `.png` image, where the red channel corresponds to player 2, the blue channel to player 1, and the green channel corresponds to the neutral units (e.g., mineral deposits). To fit the structure of a RGB image, we can tile the bag-of-units into rows where the first RGB row corresponds to the timestamp matrices and the second row corresponds to the PID matrices. Finally, we add a third row to record the map state information for each player, specifically, the map state row contains: [RGB 'is_visible', RGB 'is_seen', RGB 'creep']. This leaves us with a RGB `.png` image with height $3 * 64$ and width $k * 64$. Examples of this can be seen in Fig. 7.



(a) Max number of units overlapped is 6.



(b) Max number of units overlapped is 4.



(c) Max number of units overlapped is 4.

Figure 7. Three examples of the dense bag-of-units `.png` show how the hyperspectral image data for a window is stored in a simple `.png` file. The hyperspectral information is represented by tiling 64 x 64 RGB images. The first row is the unit timestamps (0-255), the second row is the unit ids (0-255), and the third row contains map state information (`is_visible`, `is_seen`, and `creep`). The blue channel encodes player 1, the red channel encodes player 2, and the green channel encodes neutral elements. We note how the width of the image varies, as it is determined by how many overlapping units at the same location there are in that window. For example, the top example had 6 units overlapping at one location, so it has a width of six 64x64 images whereas the other two only had a max of 4 units overlapping at one location.

C. Broader Impact

We introduced this spatial reasoning dataset to allow for quick prototyping of complex multi-agent spatial reasoning ML models and easy benchmarking to compare models (similar to the use cases of MNIST and CIFAR10). While our dataset contains complex dynamics that are based on

real human actions, it is still a simulation-based dataset, and thus methods tested on this dataset should be further tested in real-world cases before a real-world deployment. Additionally, since this dataset is a general spatial reasoning dataset that can either be directly applied or easily adapted to real-world cases, there is an opportunity for this dataset to be used for tasks that have a negative societal impact (e.g., unauthorized surveillance/tracking). We do not condone the usage of this dataset for the development of harmful models for such negative tasks. Furthermore, since our replays are created by humans and have personal metadata like the actions per minute (APM) and match-making rating (MMR) for each player, this could possibly be used to uniquely attribute a replay to a player. However, this likely is only possible for extreme APM, MMR values (e.g., the top MMR value), and even then, APM is match-specific and a player's MMR updates with each match. Finally, all replays were freely uploaded in an open-source manner and (to the best of our knowledge) contain no personally identifying information (e.g., name of the uploader, upload IP address, etc.).

D. Metadata Description and Suggested Class-wise Splits

In this section we discuss the metadata collected alongside the image data for each window in StarCraftImage.

D.1. Metadata Description

For each window in StarCraftImage, we also collected relevant match/window metadata, which can be seen in Fig. 8. Each entry belongs to one of three main subgroups: dynamic metadata (which is information that is specific to each window), static metadata (which is information that is specific to each replay but does not change across windows within a replay), and computed metadata (which is information added by a user e.g., a label for that window for a classification task). Namely, the dynamic metadata contains a vector of tabular features for both player1 and player2 such as resource counts for each player. Specfically, these tabular features correspond to: `['player_id', 'minerals', 'vespene', 'food_used', 'food_cap', 'food_army', 'food_workers', 'idle_worker_count', 'army_count', 'warp_gate_count', 'larva_count']`. Additionally, the dynamic metadata contains: `date_time_str` which is a string representing the date that window was added to the dataset, `frame_idx` which is the frame index within a replay which corresponds to the last frame included in a window (e.g., if a window's `dynamic.frame_idx`=1000 then that window summarizes frames 745 to 1000 of the given replay). The dynamic `window_percent` corresponds to how far into a match that window takes place, represented as a fraction. For the `static` metadata, this is broken into `game_info` (which corresponds to information that is mostly match specific such as map information) and `replay_info` (which replays to information about the replay file and the players contained in the file). In the `game_info`, the race information is encoded following the PySC2 convention where Terran = 1, Zerg = 2, Protoss = 3, and Random = 4. The player-level information can be found in the `replay_info.player_stats` section where APM corresponds to the player's Actions Per Minute for that match and the player's MMR is the player's Match Making Rating (which can be thought of as a skill-level determined by Blizzard, where higher is more skilled). We include these metadata to give more details about each window, but most importantly to allow a user to split the StarCraftImage dataset along these features for a specific task. For example, if one is developing a model which should generalize to new environments, a user can split this dataset on the `static.game_info.map_name` feature, and use windows from five of the seven maps for training/validation and test on windows from the remaining two maps. To aid in determining filtering methods, histograms for numeric entries within the metadata can be seen in Fig. 9.

D.2. Splitting dataset to k Classes

When working with global-spatial reasoning tasks (e.g., whole-image classification), the question of how to split this dataset into k classes arises. Thus, we suggest some possible ways to split the dataset along with simple benchmark accuracy values for comparing the difficulty of the splits for two of the most common classification schemes ML: binary classification (k=2) and 10-way classification (k=10). For all splitting experiments we mention running below, we use the same ConvNet architecture of two convolutional layers with max-pooling in-between, three fully connected layers, all with ReLU activations, and train for 20 epochs using SGD with a learning rate of 0.001 and momentum of 0.9.

For binary classification, the obvious choice is to perform match outcome prediction (i.e., "did Player 1 win?"), as mentioned in the main body of this work. While an important task, this can be difficult even for human experts watching a StarCraftII E-sports event as well as difficult for an AI to solve/ For example, the best model in [38] can only achieve 65% outcome prediction training accuracy for frames taken 15 minutes into a game, and when tested on the grayscale StarCraftMNIST and RGB StarCraftCIFAR10 datasets split on the match outcome variable (which include windows throughout all points in the game rather than just mid-to-end game), we report 57.9% and 59.4% test accuracy, respectively, on the same task. A binary prediction task that is more easily interpreted is the task of predicting if a window comes from the first half or second half of a replay ("is `dynamic.window_percent` ¡ 0.5?"). This is also somewhat easier to solve (we report a testing ac-

Metadata for Window: 3,607,787	
dynamic.date_time_format	%Y-%m-%d_%H-%M-%S
dynamic.date_time_str	2023-03-16_19-06-53
dynamic.frame_idx	14855
dynamic.num_windows	58
dynamic.timestamp	1679008013
dynamic.window_idx	57
dynamic.window_percent	0.982758621
static.extracted_image_size	[64, 64]
static.game_info.map_name	Odyssey LE
static.game_info.mod_names	['Mods/Core.SC2Mod', 'Mods/Liberty.SC2Mod', 'Mods/Swarm.SC2Mod', 'Mods/Void.SC2Mod', 'Battle.net/Cache/f1/9f/f19f56b...8ea3a5.s2ma']
static.game_info.options.raw	TRUE
static.game_info.options.score	TRUE
static.game_info.player_info.player_1.race_actual	3
static.game_info.player_info.player_1.race_requested	3
static.game_info.player_info.player_2.race_actual	1
static.game_info.player_info.player_2.race_requested	1
static.game_info.start_raw.map_size.x	168
static.game_info.start_raw.map_size.y	184
static.game_info.start_raw.pathing_grid.bits_per_pixel	8
static.game_info.start_raw.pathing_grid.size.x	168
static.game_info.start_raw.pathing_grid.size.y	184
static.game_info.start_raw.placement_grid.bits_per_pixel	8
static.game_info.start_raw.placement_grid.size.x	168
static.game_info.start_raw.placement_grid.size.y	184
static.game_info.start_raw.playable_area.p0.x	8
static.game_info.start_raw.playable_area.p0.y	8
static.game_info.start_raw.playable_area.p1.x	160
static.game_info.start_raw.playable_area.p1.y	164
static.game_info.start_raw.start_locations.x	143.5
static.game_info.start_raw.start_locations.y	24.5
static.game_info.start_raw.terrain_height.bits_per_pixel	8
static.game_info.start_raw.terrain_height.size.x	168
static.game_info.start_raw.terrain_height.size.y	184
static.num_frames_per_window	255
static.replay_info.base_build	55958
static.replay_info.data_build	55958
static.replay_info.data_version	5BD7C31B44525DAB46E64C4602A81DC2
static.replay_info.game_duration_loops	14855
static.replay_info.game_duration_seconds	663.2159424
static.replay_info.game_fps_calculated	22.39843624
static.replay_info.game_version	3.16.1.55958
static.replay_info.player_stats.player_1.apm	242
static.replay_info.player_stats.player_1.mmr	3192
static.replay_info.player_stats.player_1.result	Loss
static.replay_info.player_stats.player_1.result_int	2
static.replay_info.player_stats.player_2.apm	103
static.replay_info.player_stats.player_2.mmr	3120
static.replay_info.player_stats.player_2.result	Win
static.replay_info.player_stats.player_2.result_int	1
static.replay_name	378bfb2ce94.....e9451dc93.SC2Replay
computed.target_id	9
computed.target_label	('Odyssey LE', 'End')

Figure 8. An example of the metadata collected for each window of a replay. Descriptions of the key, value pairs are given in Appendix D.

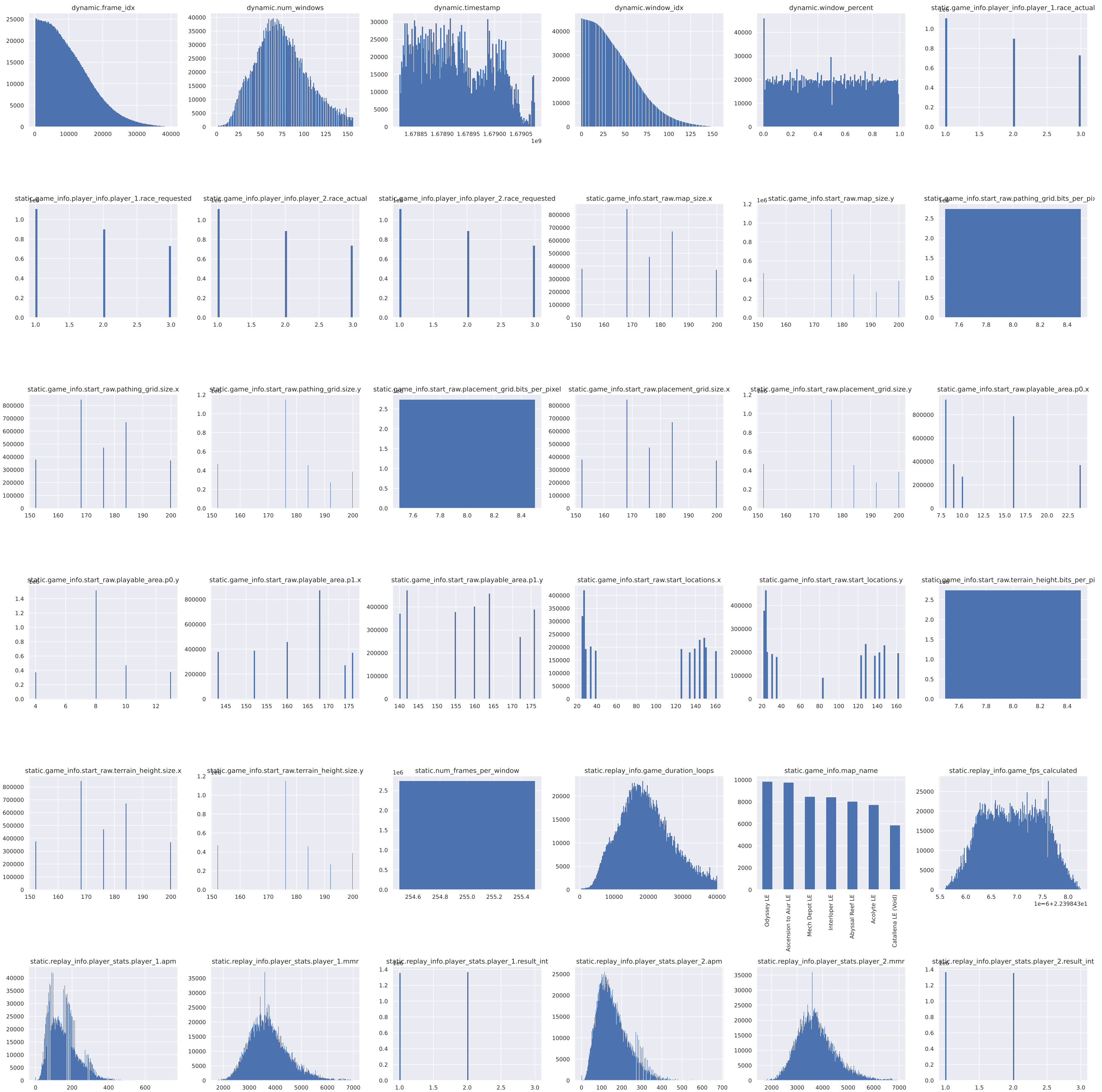


Figure 9. Histograms of the numerical values entries in the metadata (best viewed zoom in). To keep the x-axis interpretable, we excluded any outliers which have a game_duration_loop $>$ 40,000 and any replays which hold a negative MMR value for either player (which is likely a result of a bug in the `.SC2Replay` file). Any histograms which are fully rectangular are static values at the center point of the bin (e.g., `static.num_frames_per_window`=255 for all windows).

curacy of 74.3% and 76.9% for grayscale StarCraftMNIST and RGB StarCraftCIFAR10 datasets with this split, respectively), while still requiring a model to learn environmental dynamics to solve.

Splitting the StarCraftImage dataset for 10-way classification (e.g., StarCraftMNIST and StarCraftCIFAR10) is more difficult since there are no natural way to split the dataset 10 ways. The splitting method which most closely aligns with spatial reasoning problems is likely splitting via player race information + match outcome prediction, as this requires learning battlefield strategy/dynamics (for outcome prediction) and understanding of unit information

(for player race prediction). Thus, this is the 10-way splitting method suggested in subsection 2.4 in the main body of this work. However, from a purely ML perspective, this is an extremely difficult classification problem; which is supported by our testing accuracy of 26.4% and 28% for StarCraftMNIST and StarCraftCIFAR10 datasets created with the split detailed in subsection 2.4. Thus, for purposes with a stronger abstract ML focus, we suggest a 10-way split that combines the "is_beginning_or_end" binary variable from above with a prediction of the map_name. This task requires the model to learn environment information for the map_name and battlefield dynamics for the beginning/end prediction and is more solvable than a split requiring match outcome prediction. Specifically, since there are 7 maps in total, we suggest subsampling to only 5 maps, then further splitting each of these 5 map groups into "beginning " and "end" groups (based on whether or not the window takes place in the beginning 50% of the match), to get 10 classes. Examples from such a split can be seen in figure Fig. 13, and in our experiments, we received a testing accuracy of 77.2% and 77.9% for StarCraftMNIST and StarCraftCIFAR10 datasets, respectively. Heuristically, we have found this 10-way split to be a good balance between problem realism and difficulty/human interpretability, and thus we will be using it for the following task demonstrations.

E. Benchmark Evaluations On Multi-Agent Spatial Reasoning Tasks

In this section, we report benchmark results on 4 benchmark multi-agent spatial reasoning tasks, which incorporate training 60 U-Net-based [35] models. Unlike the benchmark results in the main paper, (e.g., Table 1), the results seen here and throughout the rest of the appendix are trained on a smaller StarCraftImage dataset (specifically, these results are generated from a random 1.8 million window subset, i.e. a random 50% subset of the main dataset). This was done to allow for faster model training, thus allowing us to add more models beyond the three ResNet models seen in the main paper (specifically, 60 models were trained on this smaller dataset).

The four benchmark tasks consist of two tasks on unit type identification and two tasks for unit tracking (next hyperspectral window prediction). Both task sets consist of first training and evaluating on "clean" (unaltered) data as well as a second task of training on data which is first passed through a simulation of a noisy sensor network. This simulation consists of 50 sensors with a radius of 5.5 pixels with different sensor placement methodologies (e.g., grid, random, quasi-random, and diagonal barrier) and communication failures during sensor fusion. For reference, grid-based placements are commonly used in environmental monitoring data sets where they are optimally placing sensors to cover the environment space [28]. Random and Quasi-random deployments are typical when sensors cannot be placed optimally (e.g., when they are dropped out of planes/helicopters). The barrier placements come from the well-studied barrier coverage problem, which is commonly used for border surveillance, road monitoring, etc. For further examples studying these different coverage types see [10] which provides a taxonomy for different coverage protocols including the ones mentioned above. For a study on different failure types, including the ones seen here, see [45]. Examples of the different placement types can be seen in Fig. 10.

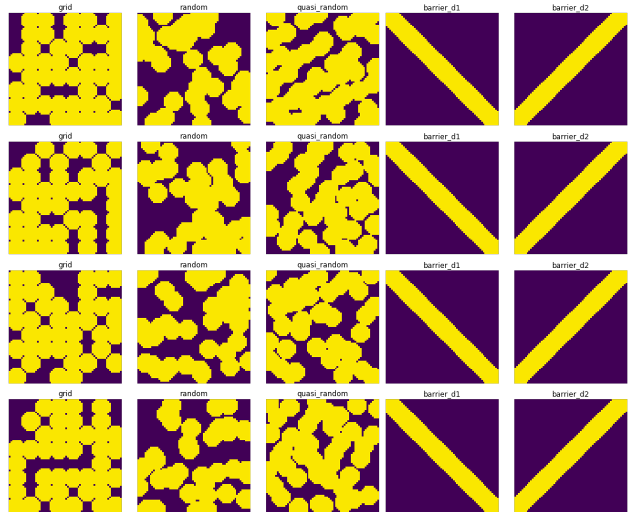


Figure 10. Example masks for sensor network simulations containing 50 sensors with five different placement strategies [grid, random, quasi-random, barrier-d1, barrier-d2] (where yellow is a location that is visible). Each sensor has a radius of 5.5. pixels and a 20% chance of being dropped during the sensor fusion process (e.g., due to a communication failure).

E.1. Unit Type Identification

As discussed in section 3, the goal of this task is to train a model to take a 64 x 64 RGB image (similar to the format of the StarCraftCIFAR10 images) as input and to output a 64 x 64 matrix corresponding to the unit ids for each location. This problem is analogous to fine-grained multi-object detection, where given raw images (e.g., satellite images), our goal is to predict what kind of unit is present at each location (if there is a unit present at all). For example, for a given window, if there is a non-zero value at location, (Red, i, j), then we know an enemy unit passed through location (i, j) – our goal now is to figure out that unit's type (e.g., ZERG_QUEEN, PROTOSS_ORACLE, etc.).

We used the unaltered RGB images as input and synthesized the 64 x 64 unit id label matrix from the StarCraftHyper dataset. In cases where multiple units were present at

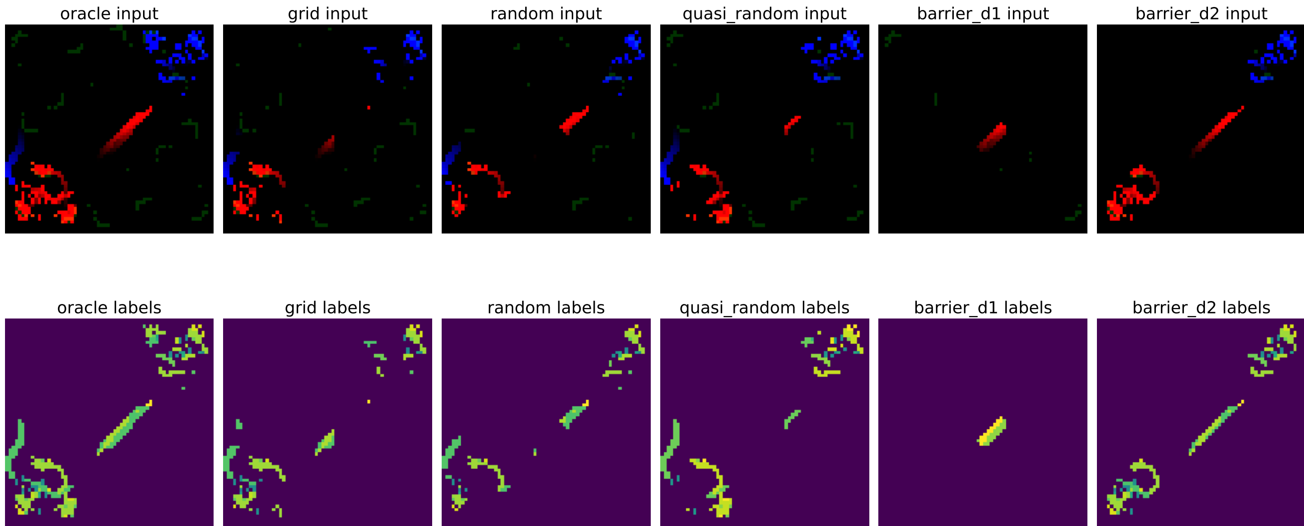


Figure 11. Example input-output pairs for the sample from the Unit Type Identification task with the clean representation as well as representations which are prepossessed by corruption simulations via faulty sensing networks five different placement strategies [grid, random, quasi-random, barrier-d1, barrier-d2] (where yellow is a location that is visible). Note that during training, the labels are also masked to simulate training on noisily labeled data.

the same location (which is possible since a window covers a span of 255 in-game frames/seconds), we set the id for that location to that of the most recent unit. We used the FastAI library [17] to train six U-Net models [35] with backbones: ResNet18, ResNet34 [15], Squeezenet1_0, Squeezenet1_1 [19], XResNet18, XResNet34 [16], on this dataset for 10 epochs with cross-entropy loss, a batch size of (512 for ResNet*, 256 for Squeezenet*, and 512 for XResNet*), and default FastAI configuration settings. We trained each of the above models on the clean (i.e., noiseless) dataset as well as on all five sensor placement variations where both the input images and output label matrices were masked by the generated sensor masks (see Fig. 11 for examples), yielding 36 models in total. During testing, we tested all models on held-out *clean* data, which simulates the situation where one has noisy training data but wants to evaluate an algorithm with respect to clean ground truth data for final evaluation. We report the Cross-Entropy error, Unit Accuracy (was the unit type correctly predicted), and the averaged Dice coefficient for all models in Table 6. As expected, there is significant performance derogation across models when moving from the clean data to the noisy data, and this is most evident in the diagonal barrier placements. As seen by the unit accuracy metric, this is a hard problem (there are 340 possible unit ids for each location), which we hope future work will be able to innovate upon.

E.2. Next Hyperspectral Window Prediction

Here our goal is to use the StarCraftHyper dataset to train a model on the common spatial reasoning task of object tracking. For this, we frame this task as: a given replay, we want to take the kth hyperspectral window as input (with shape 340 x 64 x 64) and have a model forecast how all units will move to their locations in the $k+1$ hyperspectral window. Specifically, the model must output the *difference* (i.e., movement) between the two hyperspectral windows ($ground_truth = (H_{k+1} - H_k)$ and has shape (340, 64, 64)).

To do this, we use the FastAI library [17] and the SMP library [18] to train four U-Net [35] models with backbones: ResNet18, ResNet34, ResNet50 [15], and ResNext50_32x4d [43] on both clean versions of the dataset and five noisy versions of the dataset matching the five sensor network simulations (see Fig. 12 for examples). Due to the large size of the samples, we use a batch size of 20 across all models, and to accelerate the training process we randomly subsample the overall training data to 60K window pairs. We train all models for 10 epochs using the Mean-Squared Error loss and otherwise default FastAI configuration settings. We then test our models on 10K held out *clean* window pairs, and report the MSE loss. Additionally, we bin the $ground_truth$ test data into $[-1, 0, 1]$ where location (u_{id}, i, j) is -1 if a specific unit type *left* location

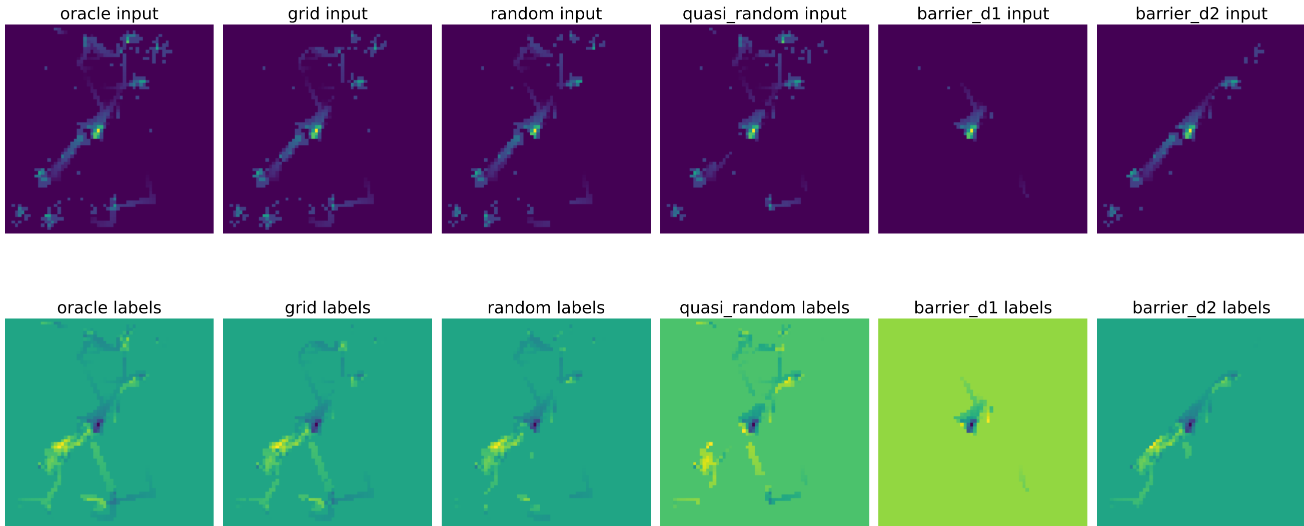


Figure 12. Example input output pairs of the same sample from the Next Hyperspectral Window prediction task with the clean representation as well as representations which are prepossessed by corruption simulations via faulty sensing networks with five different placement strategies [grid, random, quasi-random, barrier-d1, barrier-d2] (where yellow is a location that is visible). Note that during training, the labels are also masked to simulate training on noisily labeled data.

(i, j) from window k to window $k + 1$ (i.e. the unit moved away from that spot), 0 means no movement happened, and 1 means unit u_{id} moved *into* location (i, j). With this, we report the False Positive Rate (% of times a model predicts movement happens when it doesn't), True Positive Rate (+) (% of the time a model correctly predicts $+1$, a unit moved into a location), and the True Positive Rate (-). The results for this can be seen in Table 7, where while the MSE is lowest for the models trained on the clean data, these models also tend to have a higher false positive rate.

F. Additional Demonstrations of ML tasks on the StarCraftImage Dataset

Predicting match outcome As mentioned previously, this is a very difficult task which can be hard even for human experts. [38] performed the match outcome prediction task on features extracted from PySC2, and report only 65% outcome prediction training accuracy for frames taken as far as 15 minutes into a game. For this task, we use two datasets: one which has the grayscale window formats of StarCraftMNIST and the other with the RGB window formats of StarCraftCIFAR10. These datasets are constructed such that the train/test splits for the positive class (Player1IsWinner) and negative class (Player1IsNotWinner) are evenly balanced with (60k, 10k) and (50k, 10k) train, test examples for the grayscale and RGB datasets, respectively. We found the performance of our ConvNet model (mentioned in the previous section) to be similar to that of [38], where we see only 57.9% test accuracy on the grayscale window dataset and 59.4% test accuracy on the RGB window dataset.

10 class classification Here we use the map_name + is_beginning_or_end 10-way class split mentioned in subsection D.2 and seen in Fig. 13. We apply our ConvNet model to the StarCraftMNIST and StarCraftCIFAR10 datasets. After training for 20 epochs, we received a testing accuracy of 77.2% and 77.9% for StarCraftMNIST and StarCraftCIFAR10, respectively.

Imputing occluded objects For this task, we simulate occluded objects via randomly masking out a circle of pixels with a radius 5px, which can be seen to approximate cloud coverage or a fused image with missing data (see left of Fig. 15 for examples). For this task, our goal is to impute the missing information and to do this we implement a VAE model [22] to denoise the inpainted image. Our VAEImputer encoder and decoder both consist of six convolutional layers (with a 3x3 kernel, stride of 2, and padding of 1), each with batch norm and leakyReLu activations and with one fully connected layer in-between. For the encoder, the convolutional layers consist of $[3, 32, 64, 128, 128, 512]$ channels, the decoder con-

volutional layers consist of $[512, 128, 128, 64, 32, 3]$ channels, and the fully connected layer takes in the 512 channels and projects this to the 64 dimensional μ and σ latent parameter space. We train the model for 100 epochs on StarCraftMNIST (which consists of the same 60k training images as in the 10-class map-based split), with Adam optimizer [21] with a learning rate of 5e-3 and $\beta = (0.9, 0.999)$. The results can be seen in Fig. 15, where the model does well at imputing the missing data, at the expense of blurring the image due to artifacting from the VAE.

Unit type identification In this task we simulate the case where a model is only given a raw image showing the presence of a unit, but not what type of unit it is. This can be mapped to a colorization task by first taking an RGB sample from StarCraftCIFAR10 (where each channel corresponds to a specific player's units) and averaging along the color dimension to get a grayscale image that has no owner information. Then, to recover the owner information (i.e. whether a unit belongs to Player 1, Player 2, or Neutral), we colorize the image by predicting which channel each unit should belong to. To do this, we use a ResNet-101 model [15], which has been adapted to have an output dimensionality of 32x32x3 (the number of pixels in a StarCraftCIFAR10 image). We train this model on the StarCraftCIFAR10 dataset for 100 epochs using the Adam optimizer [21] with a learning rate of 5e-3 and $\beta = (0.9, 0.999)$. The results can be seen in Fig. 15, where the model correctly identifies between neutral and non-neutral units, but has trouble determining whether a unit belongs to Player 1 or Player 2 due to both players having random starting corners of the map.

Table 5. A per-window frequency table for the non-neutral units across all windows in the StarCraftImage dataset, where Avg. Per Win. corresponds to the average number of times that unit is present per window, Perc. is the number of times that unit appeared divided by the total unit appearances, and Cum Perc. is the cumulative percentage up to that row. Note, this analysis was performed on the 30k replay subset but should be quite similar to the 60k frequencies.

Unit Name	Avg. Per Win.	Perc.	Cum Perc.	Unit Name	Avg. Per Win.	Perc.	Cum Perc.	Unit Name	Avg. Per Win.	Perc.	Cum Perc.
TERRAN_SCV	29.4	12.4%	12.4%	PROTOSS_CARRIER	0.7	0.3%	92.3%	PROTOSS_DARKSHRINE	0.1	0.0%	99.5%
ZERG_DRONE	21.5	9.1%	21.5%	TERRAN_RAVEN	0.7	0.3%	92.6%	PROTOSS_FLEETBEACON	0.1	0.0%	99.5%
PROTOSS_PROBE	19.7	8.3%	29.8%	PROTOSS_GATEWAY	0.6	0.3%	92.9%	TERRAN_THORAP	0.1	0.0%	99.6%
ZERG_ZERGLING	16.0	6.7%	36.5%	ZERG_SPAWNINGPOOL	0.6	0.2%	93.1%	TERRAN_FUSIONCORE	0.1	0.0%	99.6%
TERRAN_MARINE	14.1	5.9%	42.5%	PROTOSS_WARPPRISM	0.6	0.2%	93.4%	TERRAN_VIKINGASSAULT	0.1	0.0%	99.6%
ZERG_OVERLORD	9.0	3.8%	46.2%	TERRAN_HELLIONTANK	0.6	0.2%	93.6%	ZERG_LOCUSTMPFLYING	0.1	0.0%	99.6%
TERRAN_MEDIVAC	5.7	2.4%	48.7%	TERRAN_COMMANDCENTER	0.6	0.2%	93.9%	ZERG_CHANGELING	0.1	0.0%	99.7%
ZERG_ROACH	5.1	2.2%	50.8%	ZERG_EVOLUTIONCHAMBER	0.6	0.2%	94.1%	TERRAN_STARPORTFLYING	0.1	0.0%	99.7%
ZERG_CREEPTUMORBURROWED	5.0	2.1%	52.9%	TERRAN_FACTORYTECHLAB	0.6	0.2%	94.4%	TERRAN_REACTOR	0.1	0.0%	99.7%
ZERG_HYDRALISK	4.6	1.9%	54.9%	TERRAN_THOR	0.5	0.2%	94.6%	ZERG_LOCUSTMP	0.0	0.0%	99.7%
PROTOSS_STALKER	4.4	1.9%	56.7%	PROTOSS_COLOSSUS	0.5	0.2%	94.8%	ZERG_ROACHBURROWED	0.0	0.0%	99.7%
TERRAN_MARAUDER	4.3	1.8%	58.6%	PROTOSS_HIGHTEMPLAR	0.5	0.2%	95.0%	ZERG_ULTRALISKCAVERN	0.0	0.0%	99.8%
PROTOSS_PYLON	4.1	1.7%	60.3%	PROTOSS_CYBERNETICSCORE	0.5	0.2%	95.2%	PROTOSS_ORACLESTASISTRAP	0.0	0.0%	99.8%
ZERG_LARVA	3.8	1.6%	61.9%	PROTOSS_DARKTEMPLAR	0.5	0.2%	95.4%	TERRAN_GHOSTACADEMY	0.0	0.0%	99.8%
TERRAN_SUPPLYDEPOT	3.4	1.4%	63.3%	ZERG_SPINECRAWLER	0.4	0.2%	95.6%	TERRAN_TECHLAB	0.0	0.0%	99.8%
ZERG_QUEEN	3.3	1.4%	64.7%	TERRAN_WIDOWMINEBURROW	0.4	0.2%	95.7%	ZERG_SPINECRAWLERUPROOTEI	0.0	0.0%	99.8%
PROTOSS_ZEALOT	2.9	1.2%	65.9%	TERRAN_BATTLECRUISER	0.4	0.2%	95.9%	ZERG_LURKERDENMP	0.0	0.0%	99.8%
TERRAN_SIEGETANK	2.8	1.2%	67.1%	PROTOSS_INTERCEPTOR	0.4	0.2%	96.1%	PROTOSS_WARPPRISMPHASIN(	0.0	0.0%	99.8%
TERRAN_REFINERY	2.7	1.2%	68.3%	TERRAN_STARPORTREACTOR	0.4	0.2%	96.2%	PROTOSS_PYLONOVERCHARGEI	0.0	0.0%	99.9%
ZERG_MUTALISK	2.7	1.1%	69.4%	PROTOSS_STARGATE	0.4	0.2%	96.4%	ZERG_OVERLORDCOCOON	0.0	0.0%	99.9%
ZERG_EXTRACTOR	2.3	1.0%	70.3%	PROTOSS_FORGE	0.4	0.2%	96.6%	ZERG_CHANGELINGZEALOT	0.0	0.0%	99.9%
TERRAN_BARRACKS	2.3	1.0%	71.3%	TERRAN_ARMORY	0.4	0.2%	96.7%	ZERG_RAVAGERCOCOON	0.0	0.0%	99.9%
ZERG_BANELING	2.3	0.9%	72.2%	PROTOSS_ROBOTICSFACILITY	0.4	0.1%	96.9%	ZERG_CHANGELINGZERGLINGWI	0.0	0.0%	99.9%
TERRAN_VIKINGFIGHTER	2.2	0.9%	73.2%	ZERG_BANELINGNEST	0.3	0.1%	97.0%	ZERG_SPORECRAWLERUPROOTE	0.0	0.0%	99.9%
PROTOSS_ADEPT	2.2	0.9%	74.1%	TERRAN_STARPORTTECHLAB	0.3	0.1%	97.1%	ZERG_GREATERSPIRE	0.0	0.0%	99.9%
ZERG_EGG	2.0	0.8%	74.9%	ZERG_SWARMHOSTMP	0.3	0.1%	97.2%	ZERG_CHANGELINGMARINESHIE	0.0	0.0%	99.9%
PROTOSS_ASSIMILATOR	1.9	0.8%	75.7%	PROTOSS_ADEPTPHASESHIFT	0.3	0.1%	97.3%	TERRAN_AUTOTURRET	0.0	0.0%	99.9%
TERRAN_SUPPLYDEPOTLOWERE	1.9	0.8%	76.5%	PROTOSS_TWILIGHTCOUNCIL	0.3	0.1%	97.4%	ZERG_NYDUSNETWORK	0.0	0.0%	99.9%
ZERG_OVERSEER	1.9	0.8%	77.3%	ZERG_LAIR	0.3	0.1%	97.6%	ZERG_ZERGLINGBURROWED	0.0	0.0%	99.9%
TERRAN_REAPER	1.8	0.8%	78.1%	ZERG_LURKERMP	0.3	0.1%	97.7%	ZERG_LURKERMPEGG	0.0	0.0%	100.0%
TERRAN_MISSILETURRET	1.7	0.7%	78.8%	ZERG_ROACHWARREN	0.2	0.1%	97.8%	TERRAN_KD8CHARGE	0.0	0.0%	100.0%
TERRAN_HELLION	1.7	0.7%	79.5%	TERRAN_FACTORYREACTOR	0.2	0.1%	97.9%	ZERG_CHANGELINGMARINE	0.0	0.0%	100.0%
ZERG_HATCHERY	1.6	0.7%	80.2%	TERRAN_ORBITALCOMMANDFL'	0.2	0.1%	97.9%	ZERG_SWARMHOSTBURROWED	0.0	0.0%	100.0%
PROTOSS_WARPGATE	1.6	0.7%	80.9%	ZERG_CREEPTUMOR	0.2	0.1%	98.0%	PROTOSS_DISRUPTORPHASED	0.0	0.0%	100.0%
TERRAN_MULE	1.5	0.6%	81.5%	ZERG_BROODLING	0.2	0.1%	98.1%	ZERG_BROODLORDCOCOON	0.0	0.0%	100.0%
PROTOSS_IMMORTAL	1.5	0.6%	82.2%	ZERG_BANELINGCOCOON	0.2	0.1%	98.2%	ZERG_DRONEBURROWED	0.0	0.0%	100.0%
PROTOSS_OBSERVER	1.4	0.6%	82.8%	TERRAN_BUNKER	0.2	0.1%	98.3%	ZERG_NYDUSCANAL	0.0	0.0%	100.0%
TERRAN_ORBITALCOMMAND	1.3	0.6%	83.3%	PROTOSS_TEMPEST	0.2	0.1%	98.3%	ZERG_BANELINGBURROWED	0.0	0.0%	100.0%
PROTOSS_NEXUS	1.3	0.5%	83.9%	TERRAN_GHOST	0.2	0.1%	98.4%	ZERG_CHANGELINGZERGLING	0.0	0.0%	100.0%
TERRAN_CYCLONE	1.2	0.5%	84.4%	ZERG_BROODLORD	0.2	0.1%	98.5%	ZERG_INFESTORTERRAN	0.0	0.0%	100.0%
TERRAN_LIBERATOR	1.2	0.5%	84.9%	TERRAN_PLANETARYFORTRESS	0.2	0.1%	98.6%	ZERG_TRANSPORTOVERLORDCC	0.0	0.0%	100.0%
TERRAN_WIDOWMINE	1.2	0.5%	85.4%	TERRAN_BARRACKSFLYING	0.2	0.1%	98.6%	TERRAN_POINTDEFENSEDRONE	0.0	0.0%	100.0%
PROTOSS_PHOENIX	1.1	0.5%	85.9%	ZERG_SPIRE	0.2	0.1%	98.7%	ZERG_HYDRALISKBURROWED	0.0	0.0%	100.0%
PROTOSS_ORACLE	1.1	0.5%	86.3%	ZERG_INFESTATIONPIT	0.1	0.1%	98.8%	ZERG_INFESTEDTERRANSEGG	0.0	0.0%	100.0%
ZERG_CORRUPTOR	1.1	0.5%	86.8%	PROTOSS_DISRUPTOR	0.1	0.1%	98.8%	TERRAN_NUKE	0.0	0.0%	100.0%
PROTOSS_MOTHERSHIPCORE	1.0	0.4%	87.2%	TERRAN_COMMANDCENTERFLY	0.1	0.1%	98.9%	player_(Unknown)	0.0	0.0%	100.0%
PROTOSS_PHOTONCANNON	1.0	0.4%	87.7%	ZERG_INFESTOR	0.1	0.1%	98.9%	ZERG_QUEENBURROWED	0.0	0.0%	100.0%
ZERG_RAVAGER	1.0	0.4%	88.1%	ZERG_HYDRALISKDEN	0.1	0.1%	99.0%	ZERG_PARASITICBOMBDUMMY	0.0	0.0%	100.0%
TERRAN_BARRACKSREACTOR	1.0	0.4%	88.5%	ZERG_VIPER	0.1	0.1%	99.0%	PROTOSS_SHIELDBATTERY	0.0	0.0%	100.0%
TERRAN_FACTORY	1.0	0.4%	88.9%	TERRAN_SENSORTOWER	0.1	0.0%	99.1%				
TERRAN_STARPORT	0.8	0.4%	89.2%	ZERG_INFESTORBURROWED	0.1	0.0%	99.1%				
ZERG_SPORECRAWLER	0.8	0.4%	89.6%	TERRAN_LIBERATORAG	0.1	0.0%	99.2%				
PROTOSS_SENTRY	0.8	0.3%	89.9%	ZERG_OVERLORDTRANSPORT	0.1	0.0%	99.2%				
PROTOSS_VOIDRAY	0.8	0.3%	90.2%	ZERG_CREEPTUMORQUEEN	0.1	0.0%	99.3%				
TERRAN_BANSHEE	0.8	0.3%	90.6%	PROTOSS_MOTHERSHIP	0.1	0.0%	99.3%				
ZERG_ULTRALISK	0.7	0.3%	90.9%	PROTOSS_TEMPLARARCHIVE	0.1	0.0%	99.3%				
PROTOSS_ARCHON	0.7	0.3%	91.2%	ZERG_HIVE	0.1	0.0%	99.4%				
TERRAN_BARRACKSTECHLAB	0.7	0.3%	91.5%	ZERG_LURKERMPBURROWED	0.1	0.0%	99.4%				
TERRAN_ENGINEERINGBAY	0.7	0.3%	91.8%	PROTOSS_ROBOTICSBAY	0.1	0.0%	99.4%				
TERRAN_SIEGETANKSIEGED	0.7	0.3%	92.1%	TERRAN_FACTORYFLYING	0.1	0.0%	99.5%				

Table 6. Results for the Unit Identification Benchmarks.

	Cross Entropy						Unit Accuracy (ignoring non-units)						Multi-class Dice					
Model \ Placement	oracle	grid	random	quasi	diag1	diag2	oracle	grid	random	quasi	diag1	diag2	oracle	grid	random	quasi	diag1	diag2
unet_resnet18	0.087	0.255	0.355	0.257	1.155	0.948	56.0%	37.8%	28.0%	35.2%	9.3%	17.7%	0.172	0.119	0.094	0.111	0.050	0.074
unet_resnet34	0.078	0.257	0.370	0.262	1.087	0.852	55.2%	34.1%	27.9%	33.9%	9.3%	17.8%	0.159	0.095	0.097	0.103	0.051	0.075
unet_squeezenet1_0	0.161	0.291	0.380	0.227	1.170	1.308	49.3%	30.0%	22.6%	32.5%	8.8%	16.7%	0.126	0.081	0.062	0.103	0.045	0.068
unet_squeezenet1_1	0.086	0.288	0.315	0.254	1.060	1.106	49.3%	29.1%	24.6%	29.9%	8.3%	15.2%	0.136	0.078	0.078	0.086	0.042	0.058
unet_xresnet18	0.076	0.261	0.316	0.269	1.318	1.030	56.3%	37.3%	25.0%	35.8%	9.5%	18.1%	0.169	0.111	0.073	0.115	0.051	0.080
unet_xresnet34	0.083	0.244	0.331	0.235	1.256	0.984	56.5%	38.7%	27.8%	32.8%	9.6%	18.2%	0.173	0.125	0.085	0.094	0.052	0.079
Average over models	*0.095*	*0.266*	*0.344*	*0.251*	*1.175*	*1.038*	*53.8%*	*34.5%*	*26.0%*	*33.4%*	*9.1%*	*17.3%*	*0.156*	*0.102*	*0.081*	*0.102*	*0.048*	*0.072*

Table 7. Results for the Next Hyperspectral Window Prediction Benchmarks.

	MSE						FPR (nonzero is "positive", zero is "negative")					
Model \ Placement	oracle	grid	random	quasi	diag1	diag2	oracle	grid	random	quasi	diag1	diag2
unet_resnet18	3.85	3.84	3.91	3.88	4.12	4.03	15.0%	17.1%	15.4%	15.9%	9.7%	12.8%
unet_resnet34	3.85	3.86	3.92	3.89	4.12	4.04	16.2%	13.1%	15.0%	15.7%	9.6%	11.9%
unet_resnet50	3.71	3.84	3.91	3.87	4.14	4.06	16.6%	15.6%	15.8%	17.0%	16.6%	16.5%
unet_resnext50_32x4d	3.83	3.87	3.93	3.85	4.12	4.05	16.5%	15.5%	15.9%	17.1%	8.1%	11.4%
Average over models	*3.81*	*3.85*	*3.92*	*3.87*	*4.13*	*4.05*	*16.1%*	*15.3%*	*15.5%*	*16.4%*	*11.0%*	*13.1%*

	TPR(+) (positive diff is "positive")						TPR(-) (negative diff is "positive")					
Model \ Placement	oracle	grid	random	quasi	diag1	diag2	oracle	grid	random	quasi	diag1	diag2
unet_resnet18	60.6%	68.9%	65.8%	62.3%	43.3%	54.0%	55.6%	46.7%	40.8%	47.8%	27.8%	31.5%
unet_resnet34	59.7%	57.4%	63.4%	60.0%	45.8%	52.3%	58.3%	47.5%	42.6%	44.1%	31.4%	33.2%
unet_resnet50	62.3%	67.2%	65.3%	68.0%	45.6%	51.3%	58.0%	46.9%	42.6%	46.4%	29.7%	30.6%
unet_resnext50_32x4d	60.4%	65.5%	69.0%	65.3%	43.5%	52.3%	58.6%	46.0%	39.6%	52.7%	17.3%	23.3%
Average over models	*60.8%*	*64.8%*	*65.9%*	*63.9%*	*44.6%*	*52.5%*	*57.6%*	*46.8%*	*41.4%*	*47.7%*	*26.5%*	*29.7%*

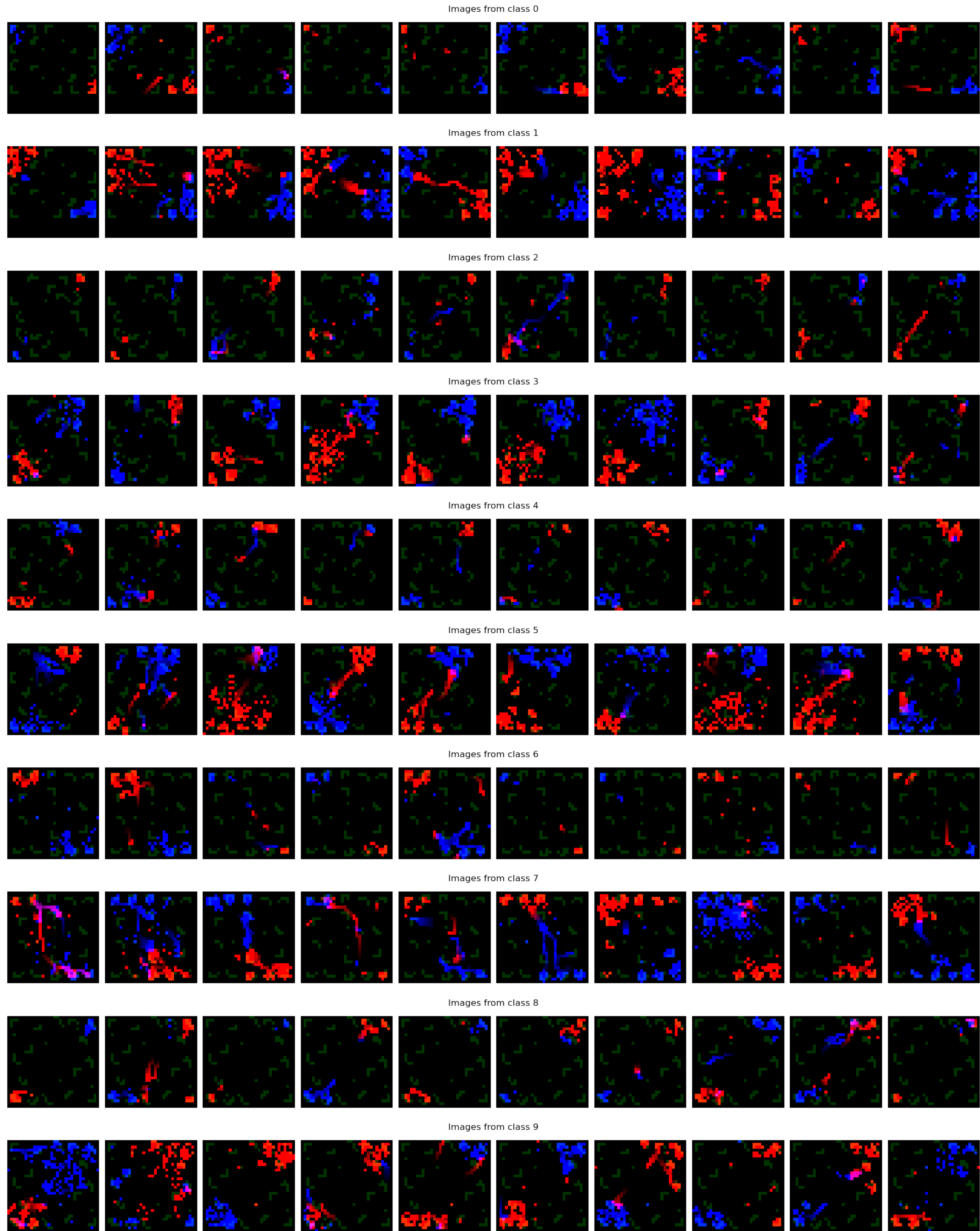


Figure 13. 10 random samples from each class (where each row is its own class), from the map_name + is_begining_or_end 10-way class split. The class label to variable information mapping is as follows: Class_0=('Acolyte LE', 'Beginning'), Class_1=('Acolyte LE', 'End'), Class_2=('Abyssal Reef LE', 'Beginning'), Class_3=('Abyssal Reef LE', 'End'), Class_4=('Ascension to Aiur LE', 'Beginning'), Class_5=('Ascension to Aiur LE', 'End'), Class_6=('Mech Depot LE', 'Beginning'), Class_7=('Mech Depot LE', 'End'), Class_8=('Odyssey LE', 'Beginning'), Class_9=('Odyssey LE', 'End')).

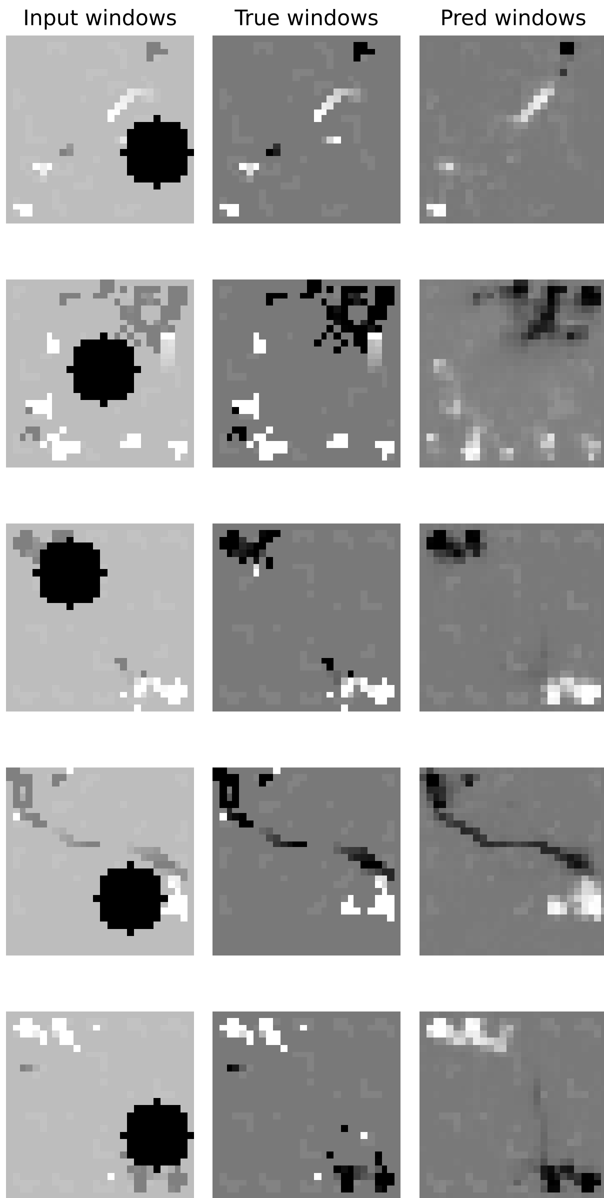


Figure 14. Five examples from the VAEImputer model trained to denoise an imputed corrupted aerial image (where in this case an occlusion with a 5px radius has been simulated). As can be seen. Note, the difference in colorization between the input windows and true windows is simply due to plotting renormalization due to the occlusion.

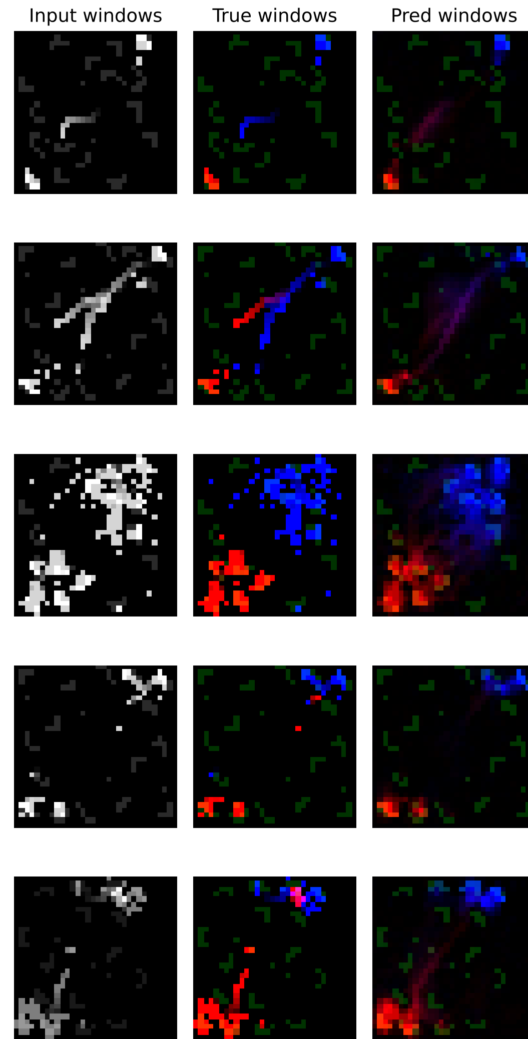


Figure 15. Five examples from the ResNet-101 model [15] trained to identify the owner of each unit in a window, where this task is akin to an image colorization task where each owner (Player1, Neutral, Player2) is placed on a separate channel. As can be seen in the examples here, it is difficult at times for the model to determine the difference between Player 1 and Player 2 due to both players having random starting corners of the map.