Target Entity	Prompt	True Positive Rate	True Negative Rate
Spider	"Is there a spider in this image?"	22.27%	100.00%
	"Spiders in Minecraft are black. Is there a spider in this image?"	73.42%	94.54%
	"Spiders in Minecraft are black and have red eyes and long, thin legs. Is there a spider in this image?"	50.50%	99.85%
Cow	"Is there a cow in this image?"	71.00%	45.41%
	"Cows in Minecraft are black and white. Is there a cow in this image?"	98.22%	2.00%
	"Cows in Minecraft are black and white and have four legs. Is there a cow in this image?"	96.67%	7.35%
Sheep	"Is there a sheep in this image?"	88.00%	59.83%
	"Sheep in Minecraft are white. Is there a sheep in this image?"	100.00%	0.00%
	"Sheep in Minecraft are white and have four legs. Is there a sheep in this image?"	100.00%	0.00%

Table 5: InstructBLIP's performance at decoding text indicating that it detected the presence of a target entity when given different prompts. We use this as a proxy metric for prompt engineering for RL, allowing us to determine which prompt to use for PR2L.

A Prompt Evaluation for RL in Minecraft

We discuss how to evaluate prompts to use with PR2L, by showcasing an example for a Minecraft task. We start by noting that the presence and relative location of the entity of interest for each task (i.e., spiders, sheep, or cows) are good features for the policy to have. To evaluate if a prompt elicits these features from the VLM, we collect a small dataset of videos in which each Minecraft entity of interest is on the left, right, middle, or not on screen for the entirety of the clip. Each video is collected by a human player screen recording visual observations from Minecraft of the entity from different angles for around 30 seconds at 30 frames per second (with the exception of the video where the entity is not present, which is a minute long).

We propose prompts that target each of the two features we labeled. First, we evaluate prompts that ask "Is there a(n) [entity] in this image?" As the answers to these questions are just yes/no, we see how well the VLM can directly generate the correct answer for each frame in the collected videos. The VLM should answer "yes" for frames in the three videos where the target entity is on the left, right, or middle of the screen and "no" for the final video. Second, we evaluate if our prompts can extract the entity's relative position (left, right, or middle) in the videos where it is present. We note that the prompts we tried could not extract this feature in the decoded text (e.g., asking "Is the [entity] on the left, right, or middle of the screen?" will always cause the VLM to decode the same text). Thus, we try to see if this feature can be extracted from the decoded texts' representations. We measure this by fitting a three-category linear classifier of the entity's position given the *token-wise mean* of the decoded tokens' final embeddings. This is an unsophisticated and unexpressive classifier, i.e., we do not have to worry about the model potentially memorizing the data, which means that good classification performance corresponds to an easy extractability of said feature.

We evaluate three types of prompts per task entity for the first feature: one simply asking if the entity is present in the image (e.g., "Is there a spider in this image?") and two others adding varying amounts of auxiliary information about visual characteristics of the entity (e.g., "Spiders in Minecraft are black. Is there a spider in this image?" and "Spiders in Minecraft are black and have red eyes and long, thin legs. Is there a spider in this image?"). We present evaluations of all such prompts in Table 5. We find that the VLM benefits greatly from auxiliary information for the spider case only, likely because spiders in Minecraft are the most dissimilar to the ones present in natural images of real spiders, whereas cows and sheep are still comparatively similar, especially in terms of scale and color. However, adding too much auxiliary information degrades

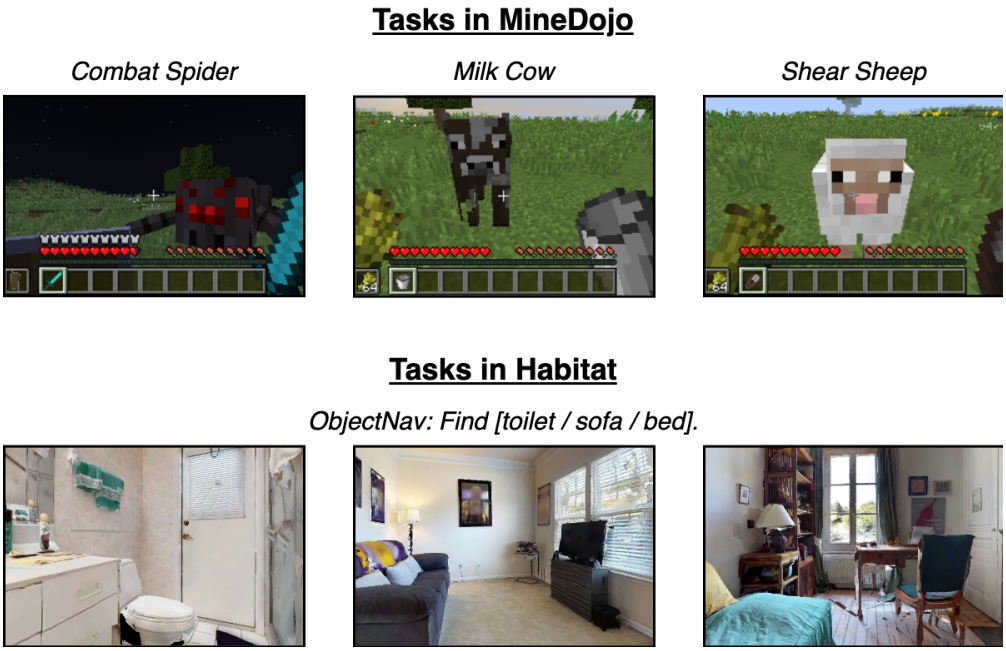

Figure 3: Example tasks, observations, and task-relevant prompts from MineDojo and Habitat.

performance, perhaps because the input prompt becomes too long, and therefore is out-of-distribution for the types of prompts that the VLM was pre-trained on. This same argument may explain why auxiliary information degrades performance for the other two target entities as well, causing them to almost always answer that said entities are present, even when they are not. Once more, considering that these targets exhibit a higher degree of visual resemblance to to their real counterparts compared to Minecraft spiders, it is reasonable to infer that the VLM would not benefit from auxiliary information. Furthermore, taking into account that the auxiliary information we gave is more common-sense than the information given for the spider, it could imply that the prompts are also more likely to be out-of-distribution (given that "sheep are white" is so obvious that people would not bother expressing it in language), causing the systematic performance degradation.

For the probing evaluation, we find that all three prompts reach similar final linear classifiabilities for each of their target entities, as shown in Figure 4. While this does not aid in choosing one prompt over another, it does confirm that the VLM's decoded embeddings for each prompt still contains this valuable and granular position information about the target entity, *even though the input prompt did not ask for it.*

B MineDojo Details

B.1 Environment Details

Spaces. The observation space for the Minecraft tasks consists of the following:

1. **RGB:** Egocentric RGB images from the agent. (160, 256, 3)-size tensor of integers $\in \{0, 1, ..., 255\}$.
2. **Position:** Cartesian coordinates of agent in world frame. 3-element vector of floats.
3. **Pitch, Yaw:** Orientation of agent in world frame in degrees. Note that we limit the pitch to 15° above the horizon to 75° below for *combat spider*, which makes learning easier (as the agent otherwise often spends a significant amount of time looking straight up or down). Two 1-element vectors of floats.

4. **Previous Action:** The previous action taken by the agent. Set to no operation at the start of each episode. One-hot vector of size $|\mathcal{A}| = 53$ for *combat spider* and 89 otherwise (see below).

This differs from the simplified observation space used in Fan et al. (2022) in that we do not use any nearby voxel label information and impose pitch limits for *combat spider*. This observation space is the same for all Minecraft experiments.

The action space is discrete, consisting of 53 or 89 different actions:

1. **Turn:** Change the yaw and pitch of the agent. The yaw and pitch can be changed up to $\pm 90°$ in multiples of $15°$. As they can both be changed at the same time, there are $9 \times 9 = 81$ total different turning actions. The turning action where the yaw and pitch changes are both $0°$ is the no operation action. Note that, since we impose pitch limits for the spider task, we also limit the change in pitch to $\pm 30°$, meaning there are only 45 turning actions in that case.
2. **Move:** Move forward, backward, left, right, jump up, or jump forward for 6 actions total.
3. **Attack:** Swing the held item at whatever is targeted at the center of the agent's view.
4. **Use Item:** Use the held item on whatever is targeted at the center of the agent's view. This is used to milk cows or shear sheep (with an empty bucket or shears respectively). If holding a sword and shield, this action will block attacks with the latter.

This non-*combat spider* action space is the same as the simplified one in Fan et al. (2022). All experiments for a given task share the same action space.

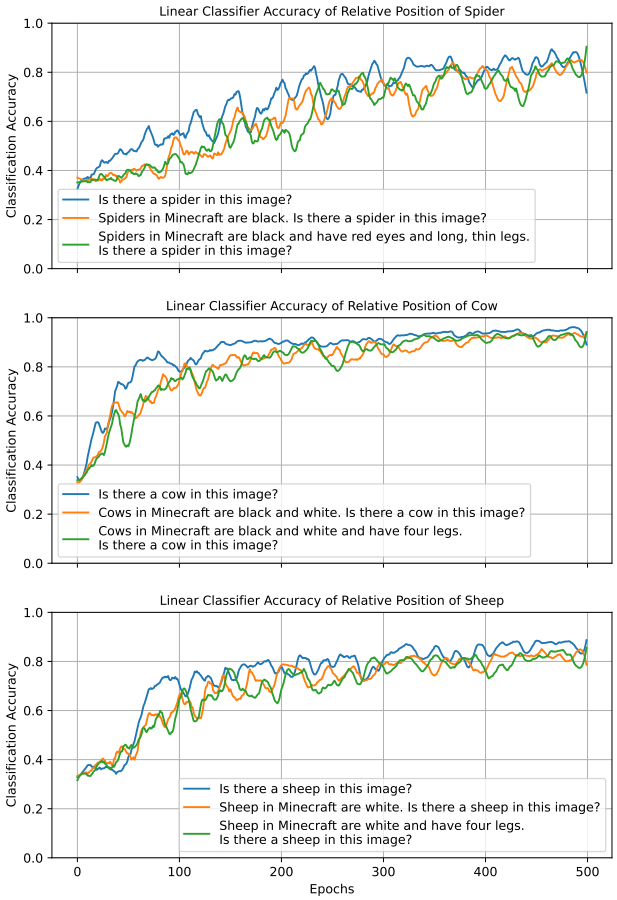

Figure 4: We train a linear classifier to predict the relative position of the target entity (left/right/middle) based on the average VLM embeddings decoded in response to each associated candidate prompt. We find that all three candidate prompts per task elicit embeddings that are similarly highly conducive to this classification scheme.

World specifications. MineDojo implements a fast reset functionality that we use. Instead of generating an entirely new world for each episode, fast reset simply respawns the player and all specified entities in the same world instance, but with fully restored items, health points, and other relevant task quantities. This lowers the time overhead of resets significantly, but also means that some changes to the world (like block destruction) are persistent. However, as breaking blocks generally takes multiple time steps of taking the same action (and does not directly lead to any reward), the agent empirically does not break many blocks aside from tall grass (which is destroyed with a single strike from any held item). We keep all reset parameters (like the agent respawn radius, how far away entities can spawn from the agent, etc) at their default values provided by MineDojo.

We stage all tasks in the same area of the same programmatically-generated world: namely, a sunflower plains biome in the world with seed 123. This is the default location for the implementation of the spider

Hyperparameter	Task					
	Combat Spider	*Milk Cow*	*Shear Sheep*	*Combat Zombie*	*Combat Enderman*	*Combat Pigman*
Total Train Steps	150000			100000		
Rollout Steps				2048		
Action Entropy Coefficient				5e-3		
Value Function Coefficient				0.5		
Max LR	5e-5	1e-4	1e-4	5e-5	1e-4	5e-5
Min LR	5e-6	1e-4	1e-4	5e-6	1e-4	5e-6
Batch Size				64		
Update Epochs				10		
γ				0.99		
GAE λ				0.95		
Clip Range				0.2		
Max Gradient Norm				0.5		
Normalize Advantage				True		

Table 6: PPO hyperparameters for Minecraft tasks, shared by the baselines, our method, and ablations.

combat task in MineDojo. We choose this specific world/location as it represents a prototypical Minecraft scene with relatively easily-traversable terrain (thus making learning faster and easier).

Additional task details and reward functions. We provide additional notes about our Minecraft tasks.

Combat spider: Upon detecting the agent, the spider approaches and attacks; if the agent's health is depleted, then the episode terminates in failure. The agent receives +1 reward for striking any entity and +10 for defeating the spider. We also include several distractor animals (a cow, pig, chicken, and sheep) that passively wander the task space; the agent can reward game by striking these animals, making credit assignment of success rewards and the overall task harder.

Milk cow: The agent also holds wheat in its off hand, which causes the cow to approach the agent when detected and sufficiently nearby. For each episode, we track the minimum visually-observed distance between the agent and the cow at each time step. The agent receives $+0.1|\Delta d_{\text{min}}|$ reward for decreasing this minimum distance (where $\Delta d_{\text{min}} \leq 0$ is the change in this minimum distance at a given time step) and +10 for successfully milking the cow.

Shear sheep: As with *milk cow*, the agent holds wheat in its off hand to cause the sheep to approach it. The reward function also has the same structure as that task, albeit with different coefficients: $+|\Delta d_{\text{min}}|$ for decreasing the minimum distance to the sheep and +10 for shearing it.

Combat zombie: Same as *combat spider*, but the enemy is a zombie. We increase the episode length to 1000, as the zombie has more health points than the spider.

Combat enderman: Same as *combat spider*, but the enemy is an Enderman. As with combat zombie, we increase the episode length to 1000. Note that Endermen are non-hostile (until directly looked at for sufficiently long or attacked) and have significantly more health points than other enemies. We thus enchant the agent's sword to deal more damage and decrease the initial spawn distance of the enderman from the agent.

Combat pigman: Same as *combat spider*, but the enemy is a hostile zombie pigman. As with combat zombie, we increase the episode length to 1000.

B.2 Policy and Training Details

For our actual RL algorithm, we use the Stable-Baselines3 (version 2.0.0) implementation of clipping-based PPO (Raffin et al., 2021), with hyperparameters presented in Table 6. Many of these parameters are the same as the ones presented by Fan et al. (2022). For the spider trials, we use a cosine learning rate schedule:

$$\text{LR}(\text{current train step}) = \text{Min LR} + (\text{Max LR} - \text{Min LR}) \left(\frac{1 + \cos\left(\pi \frac{\text{current train step}}{\text{total train steps}}\right)}{2} \right) \tag{1}$$

Policy Transformer Hyperparameters	
Transformer Token Size	512 / 128
Transformer Feedforward Dim	512 / 128
Transformer Number Heads	2
Transformer Number Decoder Layers	1
Transformer Number Encoder Layers	1
Transformer Output Dim	128
Transformer Dropout	0.1
Transformer Nonlinearity	ReLU
Policy MLP Hyperparameters	
Number Hidden Layers	1
Hidden Layer Size	128
Activation Function	tanh
VLM Generation Hyperparameters	
Max Tokens Generated	6
Min Tokens Generated	6
Decoding Scheme	Greedy

Table 7: All policy hyperparameters for all Minecraft tasks. Smaller token sizes and feedforward dimensions are used for *combat [zombie/enderman/pigman]*.

We also present the policy and VLM hyperparameters in Table 7. The hyperparameters and architecture of the MLP part of the policy are primarily defined by the default values and structure defined by the Stable-Baselines3 `ActorCriticPolicy` class. Note that the no generation ablation, VLM image encoder baseline, and MineCLIP trials do not generate text with the VLM, and so all do not use the associated process's hyperparameters. The MineCLIP trials also do not use a Transformer layer in the policy, due to not receiving token sequence embeddings. It instead just uses a MLP, but with two hidden layers (to supplement the lowered policy expressivity due to the lack of a Transformer layer).

Additionally, InstructBLIP's token embeddings are larger than ViT-g/14's (used in the VLM image encoder baseline), and so may carry more information. However, the VLM does not receive any privileged information over the image encoder *from the task environment* – any additional information in the VLM's representations is therefore purely from the model's prompted internal knowledge. Still, to ensure consistent policy expressivity, we include a learned linear layer projecting all representations for this baseline and our approach to the same size (512 dimensions) so that the rest of the policy is the same for both.

Minecraft training runs were run on 16 A5000 GPUs (to accommodate the 16 seeds).

C Habitat ObjectNav Details

C.1 Environment Details

The spaces and agent/task specifications are largely the same as the defaults provided by Habitat, as specified in the HM3D ObjectNav configuration file (Savva et al., 2019).

Spaces. The observation space for Habitat consists of the following:

1. **RGB:** Egocentric RGB images from the agent. (480, 640, 3)-size tensor of integers $\in \{0, 1, ..., 255\}$. By default, agents also receive depth images, but we remove them to ensure that state representations are grounded primarily in visual observations.
2. **Position:** Horizontal Cartesian coordinates of agent. 2-element vector of floats.
3. **Compass:** Yaw of the agent. Single floats.

4. **Previous Action:** The previous action taken by the agent. Set to no operation at the start of each episode. One-hot vector of size $|\mathcal{A}| = 4$.
5. **Object Goal:** Which object the agent is aiming to find. One-hot vector of size 3.

The action space is the standard Habitat-Lab action space, though we remove the pitch-changing actions, leaving only four:

1. **Turn:** Turn left or right, changing the yaw by 30°.
2. **Move Forward:** Move forward a fixed amount or until the agent collides with something.
3. **Stop:** Ends the episode, indicating that the agent believes it has found the goal object.

All observations, actions, and associated dynamics are deterministic.

World specifications. In ObjectNav, an agent is spawned in a household environment and must find and navigate to an instance of a specified target object in as efficient a path as possible. Doing so effectively requires a common-sense understanding of where household objects are often found and the structure of standard homes.

Habitat provides a standardized train-validation split, consisting of 80 household scenes for training (from which one can run online RL or collect data for offline RL or BC) and 20 novel scenes for validation, thereby testing policies' generalization capabilities. These scenes come from the Habitat-Matterport 3D v1 dataset (Ramakrishnan et al., 2021).

C.2 Policy and Training Details

In line with previous work (Ramrakhya et al., 2023; Yadav et al., 2023b; Majumdar et al., 2023), we train our policies with behavior cloning (BC) on the Habitat-Web human demonstration dataset of 77k trajectories (12M steps) (Ramrakhya et al., 2022). We adopt many of the same design choices provided by said prior works, but with a few critical differences:

1. Due to compute limitations, we were unable to train on the full dataset (as those original works used 512 parallel environments to roll out demo trajectories and collect data). Instead, we used a subset of the dataset, built by dividing the dataset by both target object and scene, then sampling every tenth demo. This would ensure that our training data still contained examples from every training scene + target object combination that existed. In total, our subsampled dataset contains approximately 1.1M steps over 7550 trajectories.
2. We adopt the same optimizer, scheduler, and associated hyperparameters as Majumdar et al. (2023), but find a learning rate of $1e-4$ to be more effective than their $1e-3$.
3. Rather than sampling partial trajectory rollouts from 512 parallel environments as done by Majumdar et al. (2023), our batches contain full trajectories, though with the same total number of transitions per batch as in that work. This means that our batches potentially contain less diverse data (due to observations from fewer different total scenes being present), but allow us to compute up-to-date full trajectory hidden states for the RNN portion of our policy. We use gradient accumulation to achieve this, once again due to compute limitations.
4. While Majumdar et al. (2023) trains for 24k gradient steps (observing approximately 400M transitions.), we find using only approximately a tenth of that (40 epochs through our smaller dataset, so around 40M transitions) to reach peak performance for our policy. The scheduler still assumes the full training run will last for 400M transitions, so our LR decays at the same rate as with VC-1. Furthermore, for fairness, we leave our VC-1 baseline policies (trained on our subsampled datasets) training beyond 40 epochs, and report their validation performance at both 40 and 120 epochs (when its performance saturates).

5. For policies that receive visual observations as a sequence of tokens (PR2L, VC-1 with patch embeddings), we apply 2D average pooling with kernel sizes of 4×4 to reduce down to 16 tokens. Then, we pass those tokens through a learned Transformer layer, instead of the learned compression layer used by Majumdar et al. (2023). We do this to ensure that policy performance differences are due to representation quality, not architecture.
6. We employ inflection upweighting during training, as done by Ramrakhya et al. (2023); Yadav et al. (2023b); Majumdar et al. (2023). However, we also categorically upweight the cross entropy loss of stopping and turning by 1.5 (due to them being uncommon but important), as we observe this increases learning speed for all policies.
7. We do not employ any image augmentation or loss regularization to prevent overfitting. However, we find our policy exhibits strong generalization performance in unseen validation scenes nonetheless.

For PR2L-specific design choices:

1. Our chosen VLM is the Prismatic VLM (Karamcheti et al., 2024) with Dino+SigLIP as a vision backbone and Llama2-7B-pure as the language backbone. We use the 224px version, which maps images to 256 visual tokens (which, as described above, get compressed into 16 via pooling).
2. To reduce the size of VLM representations for PR2L, we embed one observation (sampled uniformly at random) from each trajectory in our subsampled dataset with our VLM, then compute all resulting tokens' principle component vectors. We then use said vectors to lower all tokens' dimensionality down from 4096 to 1024 (i.e., corresponding approximately to their first 1024 principle components).
3. Like with the Minecraft experiments, we take the VLM's last two layers' embeddings and treat them as our promptable representations. However, unlike with Minecraft, we stack each VLM token's two embeddings (forming new embeddings of size 2048), rather than concatenate all of them.
4. For generating text in response to our task-relevant prompt, we use sample-based decoding with fixed random seed prior to the decoding with temperature 0.4 and $32 - 48$ new tokens generated.
5. The learned Transformer layer of our policy is the same as the one used in the Minecraft experiments, but with token embedding sizes of 1024.

All Habitat training was done on an A100 GPU server. Generation of data and evaluations were done on 16 A5000 GPUs for parallelization.

D Simplified Habitat Offline RL Experiments

While our primary Habitat experiments use behavior cloning to stay consistent with past works, we also run offline RL experiments on a simplified version of ObjectNav to better explore how VLM representations aid action learning. We discuss the details of said setting now.

D.1 Environment Details

We pick 32 reconstructed 3D home environments with at least one instance of each of the three target objects (toilet, bed, and sofa) and an annotator quality score of at least 4 out of 5. We choose to remove *plants* and *televisions* from the goal object set due to finding numerous unlabeled instances of them. Additionally, we remove chairs, as they are significantly more common than other goal objects and thus usually can be found in much shorter episodes. This simplified problem formulation enables us to remove many of the "tricks" that aid ObjectNav, such as using omnidirectional views or policies with history; our agent makes action decisions purely based on its current visual observation and pose, allowing us to do "vanilla" RL to better isolate the effect of PR2L.

To generate data, we use Habitat's built-in greedy shortest geodesic path follower. Imitating such demonstrations allows policies to learn unintuitively emergent and performant navigation behaviors (Ehsani et al.,

2023) at scale. For each defined starting location in our considered households, we autonomously collect data by using the path follower to navigate to each reachable instance of the corresponding goal object. This yields high quality, near-optimal data. We then supplement our dataset by generating lower-quality data. Specifically, for each computed near-optimal path from a starting location to a goal object instance, we choose to inject action noise partway through the trajectory (uniformly at random from $0 - 90\%$ of the way through). At that point, all subsequent actions have a $0 - 50\%$ probability (again chosen uniformly at random) of being a random action other than the one specified by the path follower. To ensure that paths are sufficiently long, we choose to make the probability of choosing the stop action 10% and the other two movement actions 45%. In total, we collect 107518 observations over 2364 trajectories.

Reward functions. The ObjectNav challenge evaluates agents based on the average "success weighted by path length" (SPL) metric (Yadav et al., 2023a): if an agent succeeds at taking the *stop* action while close to an instance of the goal object, it gets $SPL(p, l) = \frac{l}{\max(l,p)}$ points, where l is the actual shortest path from the starting point to an instance of the goal object and p is the length of the path that the agent actually took during that particular episode. If the agent stops while not close to the target object, the SPL is 0. Thus, taking the most efficient path to the nearest goal object and stopping yields a maximum SPL of 1.

We use this to design our reward function. Specifically, when the agent stops, it receives a reward of $+10SPL(p, l)$. Additionally, we add a shaping reward of the change in geodesic distance to the nearest goal object instance each time the agent moves (where lowering that distance yields a positive reward).

D.2 Policy and Training Details

For our offline RL experiments in Habitat, we use Conservative Q-Learning (CQL) on top of the Stable-Baslines3 Contrib codebase's implementation of Quantile Regression DQN (QR-DQN) Kumar et al. (2020); Dabney et al. (2017). We choose to multiply the QR-DQN component of the CQL loss by 0.2. Using the notation proposed by Kumar et al. (2020), this is equivalent to $\alpha = 5$, which said work also uses. Other hyperparameters are $\tau = 1$, $\gamma = 0.99$, fixed learning rate of $1e - 4$, 100 epochs, and 50 quantiles (no exploration hyperparameters are specified, since we do not generate any new online data).

The policy architecture used for Habitat experiments are the same as those used for PPO, though the final network outputs quantile Q-values for each action (rather than just a distribution over actions). The action with the highest mean quantile value is chosen at evaluation time.

During training, we shuffle the data and load full offline trajectories until the buffer has at least $32 \times 1024 = 32768$ transitions or all trajectories have been loaded once that epoch. We then uniformly sample and train on batches of size 512 transitions from the buffer until each transition has been trained on once in expectation (e.g., $\sim \frac{\text{number of transitions in the buffer}}{512}$ batches). Each batch is used for 8 gradient steps before the next is sampled. We choose this data loading scheme to fit the training infrastructure provided by Stable-Baselines3 while not using up too much memory at once.

D.3 Experiments and Results

Our primary comparison is once again between our promptable representations and general-purpose non-promptable ones. We thus repeat the baseline described previously for Minecraft in Section 4.1, training a single agent for all three ObjectNav tasks using both PR2L and the VLM image encoder representations. We empirically note that longer visual embedding sequences tend to perform better in Habitat. To control for this, we opt to use InstructBLIP's Q-Former unprompted embeddings instead of the ViT embeddings directly (which are much longer than PR2L's embedding sequences). As InstructBLIP uses the former representations to extract visual features to be projected into language embedding space, this serves to close the gap in embedding sequence length between our two conditions while still providing us with general visual features that the VLM processes via prompting. In this case, we use the same InstructBLIP model as the Minecraft experiments and choose "What room is this?" as our task-relevant prompt.

We report evaluation success rates and average returns for the simplified Habitat ObjectNav setting in Figure 5. PR2L achieves nearly double the average success rate of the baseline (60.4% vs. 35.2%), supporting the hypothesis that PR2L works especially well when exploration is not needed. Lastly, in Appendix H.2, we

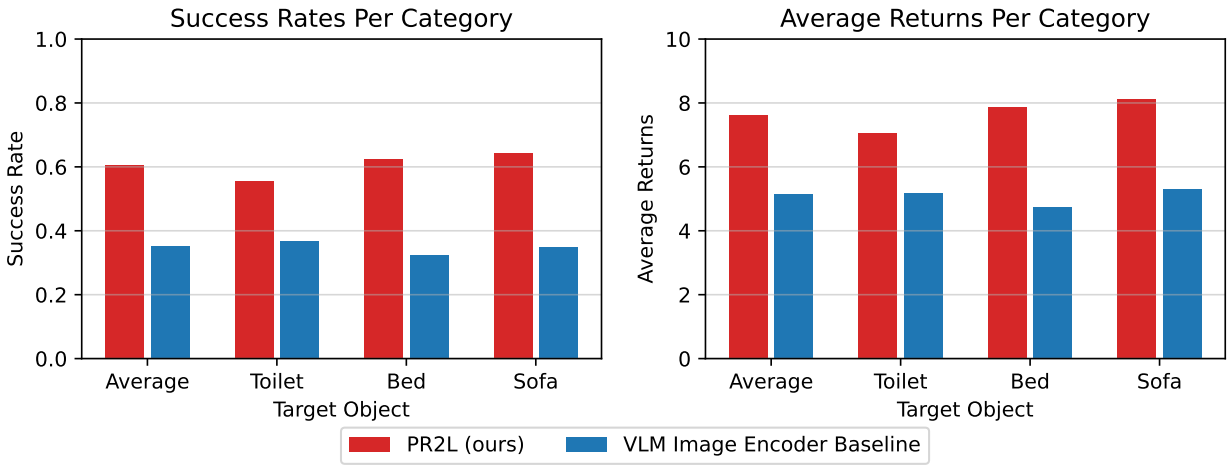

Figure 5: **Offline RL performance of PR2L and baselines in Habitat ObjectNav**. Plots show final evaluation success rates and average returns per target object and overall. PR2L outperforms the baseline in all cases.

find that PR2L causes the VLM to produce highly structured representations that correlate with an expert policy's value function: high-value states are typically labeled by the VLM as being from a room where one would expect to find the target object.

E Extended Discussion of Tasks and Results

E.1 Notes on Task-specific Systems

We designed experiments to specifically investigate the use of VLM embeddings as task-specific promptable representations for downstream sensorimotor policy learning. As such, we compare with other works that propose or evaluate either learning from scratch or from pre-trained representations, but *not* to systems in Minecraft and Habitat that require domain-specific engineered systems beyond just policy learning (such as Luo et al. (2022); Zhu et al. (2022)) or which target learning or producing higher-level plans or abstractions (such as Wang et al. (2023b)).

Such comparisons are not made as these works either aim to investigate other problems in control or are aiming to develop highly specialized and task-specific systems (whereas we present a general approach for policy learning). For instance, Voyager shows how an LLM can reason about and compose high-level hand-crafted control primitives (Wang et al., 2023a). Voyager's ability to complete harder tasks comes from its access to powerful hand-crafted high-level primitives that extensively leverage oracle information, which are composed into skills by GPT-4 (which does not handle any low-level control). Said hand-coded control primitives used in Voyager are very advanced and do much of the heavy-lifting. In particular, Voyager gives GPT-4 access to a dedicated `killMob(<entity name>)` control primitive function. This function calls a separate `bot.pvp.attack(<entity>)` (hand-written) function, which calls a hard-coded oracle pathfinder, aiming controller, and attack function to repeatedly approach and attack the specified entity until it is defeated. Thus, for Voyager, the skill for hunting sheep simply fills in the powerful `killMob()` primitive function with "sheep" as the target, abstracting away all low-level control via the oracle hand-written controllers.

Vitally, unlike PR2L, Voyager does not investigate how to use (V)LMs to learn these primitives. It thus cannot be applied to settings that lack such primitives (e.g., because oracle path planners are not available, like in Habitat). This makes PR2L complementary: we directly learn a policy to link observations to low-level actions (turning, moving, attacking, etc) via RL with no oracle information, while Voyager aims to compose pre-existing primitives into skills via LLMs.

Task	PR2L (Ours)	VLM Image Encoder	Ablations			
			No Prompt	No Generation	Change Aux. Text	Oracle Detector
Combat Spider	**97.6 ± 14.9**	51.2 ± 9.3	72.6 ± 14.2	66.6 ± 11.8	80.1 ± 12.6	58.0 ± 13.4
Milk Cow	**223.4 ± 35.4**	95.2 ± 18.7	116.6 ± 25.9	160.2 ± 23.6	80.5 ± 17.8	178.4 ± 42.5
Shear Sheep	**37.0 ± 4.4**	23.0 ± 3.6	23.8 ± 3.2	26.1 ± 4.5	27.8 ± 4.6	27.4 ± 9.3

Table 8: Minecraft ablations, VLM image encoder baseline, and our full approach. All achieve worse performance than PR2L. Values are final IQM success counts and intervals are the standard error.

E.2 Notes on Dreamer v3

We note that PR2L just proposes to use VLMs as a source of task-specific representations for RL tasks; it does not prescribe which learning algorithm to use. Therefore, in principle, one could use Dreamer in conjunction with PR2L and gain benefits from both the VLM representation and the choice of a strong model-based RL algorithm. However, while we leave this to future works, our Minecraft comparison (c) measures how well the approach does on our Minecraft tasks (as the original paper focuses more on the component subtasks involved in the *find diamond* task, all of which do not involve interacting with moving entities).

We find that Dreamer v3 is unable to learn our six tasks given the same number of environment interactions that PR2L+PPO was trained on. We hypothesize that this is due to its visual reconstruction-based world model not being suited for tasks requiring interaction with partially-observable, non-stationary autonomous entities (which all our tasks involve). We note that the last two rows of the figure visualizing model reconstructions in the original Dreamer v3 paper shows that its world model fails to reconstruct an observed pig (Hafner et al., 2023), supporting our hypothesis. This highlights the need for robust representations that are conducive to world model learning, with PR2L's capabilities to elicit task-relevant visual semantic features via prompting being one possibility for doing so.

F Ablations

We run four ablations on *combat spider*, *milk cow*, and *shear sheep* to isolate and understand the importance of various components of PR2L. First, we run PR2L with *no prompt* to see if prompting with task context actually tailors the VLM's generated representations favorably towards the target task, improving over an unprompted VLM. Note that this is not the same as just using the image encoder (comparison (a)), as this ablation still decodes through the VLM, just with an empty prompt. Second, we run PR2L with our chosen prompt, but *no generation* of text – i.e., the policy only receives the embeddings associated with the image and prompt (the left and middle red groupings of tokens in Figure 2, but not the right-most group). This tests the hypothesis that representations of generated text might make certain task-relevant features more salient: e.g., the embeddings for "Is there a cow in this image?", might not encode the presence of a cow as clearly as if the VLM generates "Yes" in response, impacting downstream performance. Third, to check if our prompt evaluation strategy provides a good proxy for downstream task performance while tuning prompts for P2RL, we run PR2L with alternative prompts that were not predicted to be the best, as per our criterion in Appendix A. We thus remove the auxiliary text from the prompt for *combat spider* and add it for *milk cow* and *shear sheep*. Lastly, to see if PR2L embeddings are just better due to them encoding entity detection, we train a VLM image encoder policy with an additional ground truth oracle target entity detector as a feature.

Results from these additional experiments are presented in Table 8. In general, all ablations perform worse than PR2L. For *milk cow*, we note the most performant ablation is no generation, perhaps because the generated text is often wrong; among the chosen prompts, it yields the lowest true positive and negative rates for classifying the presence of its corresponding target entity (see Table 5 in Appendix A), though adding auxiliary text makes it even worse, perhaps explaining why *milk cow* experienced the largest performance decrease from adding it back in. Based on these overall trends, we conclude that (i) the *promptable* and *generative* aspects of VLM representations are important for extracting good features for control tasks and (ii) our simple evaluation scheme is an effective proxy measure of how good a prompt is for PR2L.

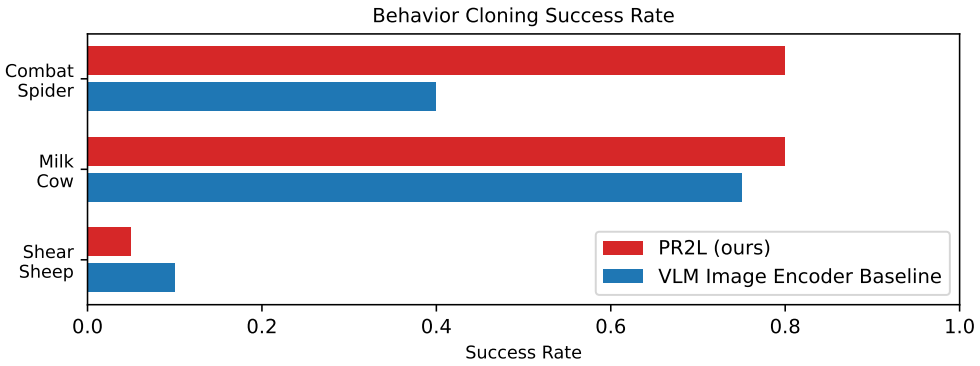

Figure 6: Success rates for BC on either PR2L or VLM image encoder baseline representations for all original tasks. PR2L excels at *combat spider*, even after the policy is trained for a single epoch.

G Minecraft Behavior Cloning Experiments

We collected expert policy data by training a policy on MineCLIP embeddings to completion on all of our original tasks and saving all transitions to create an offline dataset. We then embedded all transitions with either PR2L or the VLM image encoder. Finally, we train policies with behavior cloning (BC) on successful trajectories under a specified length (300 for *combat spider*, 250 for *milk cow*, and 500 for *shear sheep*) from either set of embeddings for all three tasks, then evaluate their task success rates.

Results are presented in Figure 6. We first note that, since the expert data was collected from a policy trained on MineCLIP embeddings, the *shear sheep* policy is not very effective (as we found in Table 2). Both resulting *shear sheep* BC policies are likewise not very performant. We find that *combat spider* in particular shows a very large gap in performance: the PR2L agent achieves approximately twice the success rate of the VLM image encoder agent *after training for just a single epoch*. The comparatively small amount of training and data necessary to achieve near-expert performance for this task supports our hypothesis that promptable representations from general-purpose VLMs do not help with exploration (they work better in offline cases, where exploration is not a problem), but instead are particularly conducive to being linked to appropriate actions even though the VLM is not producing actions itself. Further investigation of this hypothesis is presented in Appendix H.

H Representation Analysis

Why do our prompts yield higher performance than one asking for actions or instruction-following? Intuitively, despite appropriate responses to our task-relevant prompts not directly encoding actions, there should be a strong correlation: e.g., when fighting a spider, if the spider is in view and the VLM detects this, then a good policy should know to attack to get rewards. We therefore wish to investigate if our representations are conducive to easily deciding when certain rewarding actions would be appropriate for a given task – if it is, then such a policy may be more easily learned by RL, which would explain PR2L's improved performance over the baselines.

H.1 Minecraft Analysis

To investigate this, we use the embeddings of our offline data from the BC experiments (collected by training a MineCLIP encoder policy to high performance on all of our original three tasks, as discussed in Appendix G). We specifically look at the embeddings produced by a VLM when given our standard task-relevant prompts and when given the instructions used for our RT-2-style baseline. We then perform principal component analysis (PCA) on the tokenwise average of all embeddings for each observation, thereby projecting the embeddings to a 2D space with maximum variance.

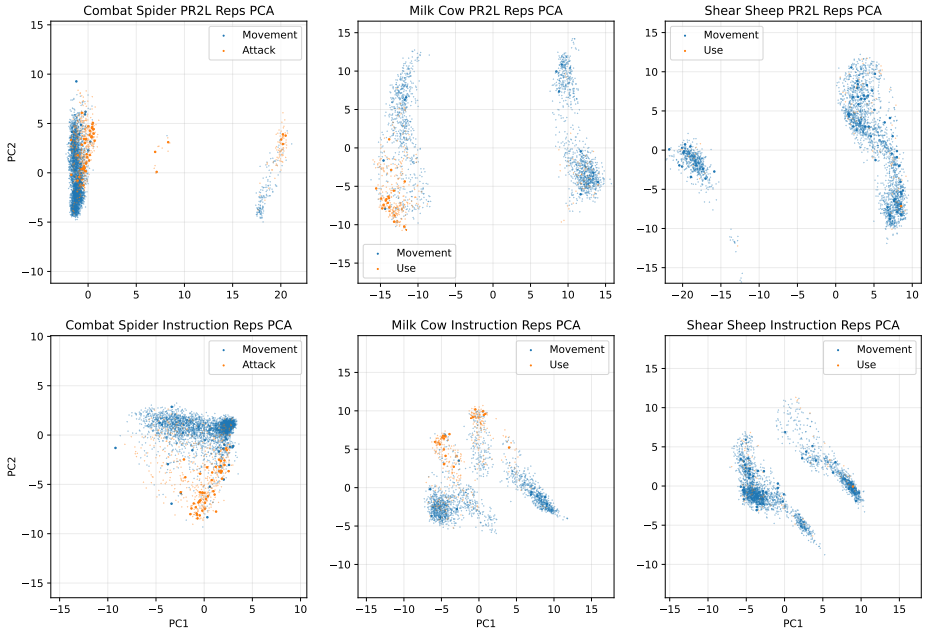

Figure 7: **PCA of PR2L representations of observations from twenty episode rollouts of expert policies in all three Minecraft tasks.** Larger points correspond to transitions where the expert received > 0.1 reward. We vary the prompt to be either our task-relevant prompt or the RT-2-style baseline instruction prompt. Our prompt's representations are bi-modal, with the clusters on the left corresponding to the VLM outputting "yes" (the entity is in view). We find that most functional actions (orange points) that yielded rewards are located in said clusters. Note that, since these expert policies are trained on top of MineCLIP embeddings, the *shear sheep* policy is not very performant, as seen in Table 2.

We visualize these low-dimensional space in Figure 7 for the final 20 successful observations from each task, with the point colors of orange and blue respectively indicating whether the observation results in a functional action (attack or use item) or movement (translation or rotation) by the expert policy. Additionally, we enlarge points corresponding to when the agent received rewards in order to recognize which actions aided in or achieved the task objective.

We find that our considered prompts resulted in a bimodal distribution over representations, wherein the left-side cluster corresponds to the VLM outputting "yes (the entity is in view)" and the right-side one corresponds to "no." Additionally, observations resulting in functional actions that received rewards (large orange points in Figure 7) tend to be on the left-side ("yes") cluster for representations elicited by our prompt, but are more widely distributed in the instruction prompt case, in agreement with intuition. This is especially clear in the *milk cow* plot, wherein nearly all rewarding functional actions (using the bucket on the cow to successfully collect milk) are in the lower left corner.

This analysis supports that the representations yielded by InstructBLIP in response to our chosen style of prompts are more structured than representations from instructions. Such structure is useful in identifying and learning rewarding actions, even when said actions were taken from an expert policy trained on unrelated embeddings. This suggests that such representations may similarly be more conducive to being mapped to good actions via RL, which we observe empirically (as our prompt's representations yield more performant policies than the instructions for the RT-2-style baseline).

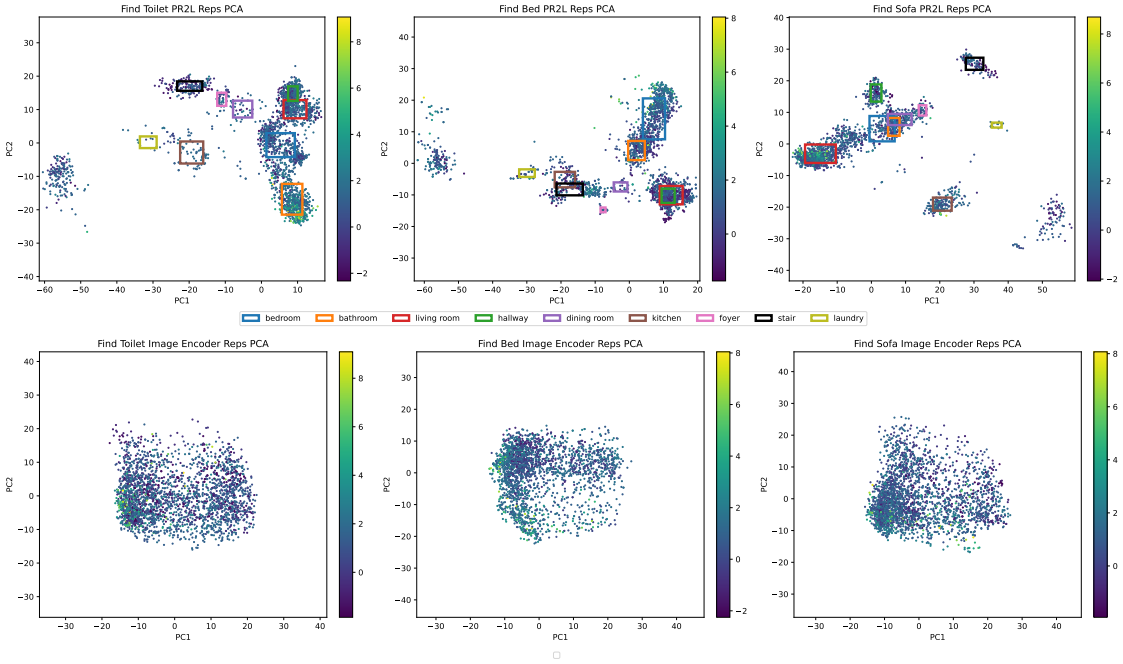

Figure 8: **PCA of PR2L (above) and image encoder (below) representations of observations from thirty episode rollouts of expert policies in all Habitat tasks.** The points' colors correspond to their value under Habitat's built-in oracle shortest path follower (a near-optimal policy). More yellow is better. Boxes correspond to points the VLM has labeled as a given household room, in response to the task prompt of "What room is this?" This analysis aligns with intuition: for *find toilet*, high value observations tend to be labeled as bathrooms (orange box), *find bed*'s tend to be labeled as bedrooms (blue), and *find sofa*'s are labeled as living rooms (red).

H.2 Habitat Analysis

Likewise, we conduct a similar analysis on the Habitat data from our simplified setting. Specifically, we wish to see if PR2L produces representations that are conducive to extracting the *value function* of a good policy. Since the chosen Habitat ObjectNav prompt is "What room is this?" we expect the state representations to be clustered based on room categories. Intuitively, states corresponding to the room one is likely to find the target object should have the highest values.

As shown in Figure 8, we thus used PCA to project expert trajectories' PR2L and general image encoder state representations (generated with Habitat's geodesic shortest path follower) to two dimensions, then colored each one based on their value under said near-optimal policy. We also plotted the mean and standard deviation of all points labeled as each room, visualizing them as axis-aligned bounding boxes. Note that each upper subplot in Figure 8 has a cluster of points far from all boxes. These correspond to the VLM generating nothing or garbage data with no room label.

This visualization qualitatively agrees with intuition. High value states tend to be grouped with the room the corresponding target object is often found in: *find toilet* corresponds to bathrooms, *find bed* to bedrooms, and *find sofa* to living rooms. Comparatively, the general image encoder features do not have such semantically meaningful groupings; all observations are clustered together and, within that single grouping, high-value observations are more spread out. This all supports the idea that prompting allows representations to take on structures that correlate well to value functions of good policies.

I Code Snippets

We provide some code snippets showcasing instantiations of PR2L.

```
class Policy(torch.nn.Module):
    def __init__(self, num_actions, tf_embed_dim=4096):
        """Policy that accepts promptable reps as input"""
        super().__init__()
        # Project down VLM embed dimensions
        self.embed_fc = torch.nn.Linear(tf_embed_dim, 1024)
        # Predict actions
        self.action_fc = torch.nn.Linear(1024, num_actions)
        # Transformer layer to condense promptable reps to 1 token
        self.transformer = torch.nn.Transformer(
            1024,
            1,
            num_encoder_layers=1,
            num_decoder_layers=1,
            dim_feedforward=1024,
            batch_first=True,
        )
        self.cls = torch.nn.Embedding(1, 1024)  # cls tokens

    def forward(self, x):
        seq, mask = x
        bs, traj_len, num_tokens, _ = seq.shape

        # [batch*traj_len, num tokens, token size]
        seq = seq.reshape(bs * traj_len, num_tokens, -1)
        # [batch*traj_len, num tokens]
        mask = mask.reshape(bs * traj_len, num_tokens)

        # Project down
        # [batch*traj_len, num tokens, tf dim]
        seq = self.embed_fc(seq)

        # Get CLS embedding
        cls = self.cls(torch.zeros([bs * traj_len, 1],
            device=seq.device, dtype=int))

        # Get summary embedding
        # [batch*traj_len, 1, tf dim]
        cls_embed = self.transformer(
            seq,  # Encoder input
            cls,  # Decoder input
            # Apply mask
            src_key_padding_mask=mask,
            memory_key_padding_mask=mask,
        )

        # [batch, traj_len, d_model]
        cls_embed = cls_embed.reshape(bs, traj_len, -1)

        # Predict actions
        # [batch, traj_len, actions]
        return self.action_fc(cls_embed)
```

Listing 1: Example policy for PR2L.

```
def process_obs(model, processor, image, prompt, device, last_n=2):
    inputs = processor(images=image, text=prompt, return_tensors="pt").to(device)

    # Generate text in response to prompt and extract embeddings
    outputs = model.generate(
        **inputs,
        output_hidden_states=True,
        return_dict_in_generate=True,
        # Any other generation parameters (min/max tokens, temp, etc)
```

```
    )
    hs = outputs["hidden_states"]

    # Get image and prompt token embeds
    # Any additional processing should happen here (eg pooling of visual tokens)
    # [last_n, num img + prompt tokens, tf_embed_dim]
    image_and_prompt_embs = torch.cat(hs[0], dim=0)[-last_n:]

    # Get decoded token embeds
    # [last_n, num decoded tokens, tf_embed_dim]
    dec_embs = []
    for dec_hs in hs[1:]:
        # [last_n, 1, tf_embed_dim]
        dec_hs = torch.cat(dec_hs, dim=0)[-last_n:]
        dec_embs.append(dec_hs)
    # [last_n, num decoded tokens, tf_embed_dim]
    dec_embs = torch.cat(dec_embs, dim=1)

    # [last_n, num total tokens]
    seq_embs = torch.cat([image_and_prompt_embs, dec_embs], dim=1)
    tf_embed_dim = seq_embs.shape[-1]

    # [bs=1, seq_len=1, last_n*num total tokens, tf_embed_dim]
    seq_embs = seq_embs.reshape(1, 1, -1, tf_embed_dim)

    mask = torch.zeros(seq_embs[:-1], type=int)

    return seq_embs, mask
```

Listing 2: Example code for extracting promptable representations from a VLM.

```
# Create VLM and processor (InstructBLIP, for example)
model = InstructBlipForConditionalGeneration.from_pretrained(
    "Salesforce/instructblip-vicuna-7b"
)
processor = InstructBlipProcessor.from_pretrained("Salesforce/instructblip-vicuna-7b")

# Set device, can also change dtype if desired
device = "cuda:0"
model = model.to(device)

# Create env
env = ...

# Create policy. This can be trained via RL or BC as needed.
policy = Policy(env.num_actions).to(device)

# Define task-relevant prompt
prompt = "Would a toilet be found here? Why or why not?"

# To predict an action, get an RGB obs from the env and process it with the VLM
obs = env.reset()
seq, mask = process_obs(model, processor, obs, prompt, device)

# Then, pass it through the policy to get action logits and step env
act_logits = policy.forward((seq, mask)).reshape(env.num_actions)
action = torch.argmax(act_logits)
obs, _, _, _ = env.step(action)
```

Listing 3: Example usage of the above function and policy.

J Extended Literature Review

Learning in Minecraft. We now consider some current approaches for creating autonomous learning systems for tasks in Minecraft. Such works highlight some of the difficulties prevalent in tasks in said environment. For instance, since Minecraft tasks take place in a dynamic open world, it can be difficult

for an agent to determine what goal it is attempting to reach and how close it is to reaching that goal. Cai et al. (2023) tackles these issues by introducing and integrating a training scheme for self-supervised goal-conditioned representations and a horizon predictor. Zhou et al. (2023) learns a model from visual observations to discriminate between expert state sequences and non-expert ones, which provides a source of intrinsic rewards for downstream RL tasks (as it pushes the policy to learn to match the expert state distribution, which tend to be "good" states for accomplishing tasks in Minecraft).

Foundation Models and Minecraft. Likewise, there has been much interest in applying foundation models – especially (V)LMs – to Minecraft tasks. Baker et al. (2022) pretrains on large scale videos, which enabled the first agent that could learn to acquire diamond tools (thereby completing a longstanding challenge in the MineRL competition Kanervisto et al. (2022)). LMs have subsequently also been used to produce graphs of proposed skills to learn or technology tree advancements to make in the form of structured language (Nottingham et al., 2023; Zhu et al., 2023; Yuan et al., 2023; Wang et al., 2023b). Other works propose to use the LLM to generate actions or code submodules given textual descriptions of observations or agent states (Wang et al., 2023a). Finally, VLMs have been used largely for language-conditioned reward shaping (Fan et al., 2022; Ding et al., 2023). In contrast, we use VLMs as a source of representations for learning of atomic tasks (as defined by Lin et al. (2023a)) that have pre-defined reward functions; the latter works can thus be used in conjunction with our proposed approach for tasks where these vision-language reward functions are appropriate.