STAG: Smart Tools and Applications in Graphics (2024)
A. Caputo, V. Garro, U. Castellani, and A. Giachetti (Editors)

Peek-a-bot: learning through vision in Unreal Engine

Daniele Della Pietra[1], Nicola Garau[1,2], Nicola Conci[1,2], Fabrizio Granelli[1,2]

[1]University of Trento, Via Sommarive 14, Trento, 38123 (Italy)
[2]CNIT - Consorzio Nazionale Interuniversitario per le Telecomunicazioni

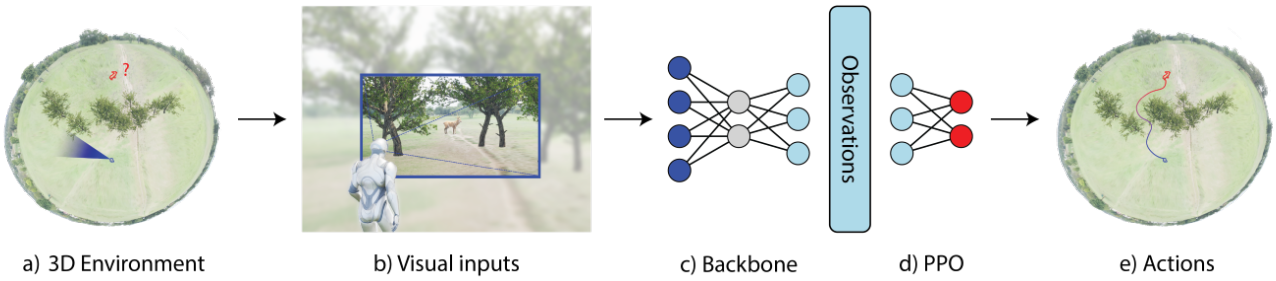

Figure 1: We present *Peek-a-bot*, a hybrid DL-RL framework that allows the training of reinforcement learning agents only through vision observations and fully in-engine. We (**a**) build arbitrarily complex and photorealistic environments for agents to navigate. Each agent (**b**) perceives the world only through its vision. *No other information is given at any time to any of the agents, including goal positions or distance information*. We (**c**) extract visual features in-engine from a pre-trained ONNX backbone and (**d**) use them as input to the PPO algorithm to optimize a set of given rewards (**e**). Every step is designed to run in real-time inside Unreal Engine 5, making it possible to train agents specifically for video games and complex simulations.

Abstract

Humans learn to navigate and interact with their surroundings through their senses, particularly vision. Ego-vision has lately become a significant focus in computer vision, enabling neural networks to learn from first-person data effectively, as we humans do. Supervised or self-supervised learning of depth, object location and segmentation maps through deep networks has shown considerable success in recent years. On the other hand, reinforcement learning (RL) has been focusing on learning from different kinds of sensing data, such as rays, collisions, distances, and other types of observations. In this paper, we merge the two approaches, providing a complete pipeline to train reinforcement learning agents inside virtual environments, only relying on vision, eliminating the need for traditional RL observations. We demonstrate that visual stimuli, if encoded by a carefully designed vision encoder, can provide informative observations, thus replacing ray-based approaches and drastically simplifying the reward shaping typical of classical RL. Our method is fully implemented inside Unreal Engine 5, from the real-time inference of visual features to the online training of the agents' behaviour using the Proximal Policy Optimization (PPO) algorithm. To the best of our knowledge, this is the first in-engine solution targeting video games and simulation, enabling game developers to easily train vision-based RL agents without writing a single line of code. All the code, complete experiments and analysis will be available at `https://mmlab-cv.github.io/Peek-a-bot/`.

1. Introduction and motivation

Reinforcement learning is a branch of machine learning used to train intelligent agents that interact with the environment through *actions*, with the objective of maximizing *cumulative rewards*. Unlike supervised, self-supervised and unsupervised learning, reinforcement learning does not rely on a fixed dataset for training, but it exploits the concept of *observations*, which are continuously gathered from the environment. For these reasons, RL is particularly well suited for highly dynamic and complex scenarios, such as video games, simulations, autonomous driving and robotics.

Before RL, in the field of video games and simulation, non-player characters (NPCs) and intelligent agents have been created using behaviour trees [Bro86], static mathematical models of plan execution that must be designed a priori, taking into account all the possible interactions between the agent and the environment. This is a significant limitation for two main reasons: (i) designing

behaviour trees is time-consuming and requires specific technical skills, and (ii) because they are static, behaviour trees cannot adapt to dynamically changing scenarios.

On the other hand, NPCs and intelligent agents require real-time decision-making, which is possible through decision trees but hard if modelled using deep learning (DL) alone, since that would require creating enormous datasets with a multitude of different observations, as well as fine-tuning them when new scenarios take place.

Current research in RL has focused on learning from non-visual observations [STZ*19], such as ray collisions, distances, lidar and sonar, achieving interesting results and good degrees of generalization to new environments. However, these kinds of observations are inherently sparse and not very informative. Some attempts have been made to train RL agents directly from pixels, with impressive results, demonstrating that visual cues can be used as a substitute for classic observations in RL, despite the increased complexity due to the high input resolution (W $\times$ H $\times$ C pixels, W=width, H=height, C=channels). Deep reinforcement learning can help the convergence to better local minima, but the complexity remains high.

One important example of combining RL with other learning paradigms is the mixing of supervised and reinforcement learning from human feedback in ChatGPT [AAA*23] and other generative pre-trained transformers. This DL-RL hybrid approach combines the strengths of deep learning in static tasks like text and image generation and understanding, with the creation of reward models to aid generalization to previously unseen scenarios.

Inspired by this double-sided approach, we present *Peek-a-bot*, a complete system for training intelligent DL-RL agents inside Unreal Engine 5, targeting video games and complex simulations. With Peek-a-bot, game developers can easily create large-scale custom scenarios, capture visual observations and train intelligent agents in real-time, all inside Unreal Engine 5, without writing a single line of code. Differently from existing works in literature, we don't rely on any game state information, nor do we directly train on raw pixels. Instead, we use features extracted from a pre-trained encoder as observations, greatly simplifying the shaping of inputs and rewards, while allowing us to run all the steps in-engine.

Here we summarize all the main contributions of our work:

1. **Visual observations are all you need**: we discard any non-visual observation, including the most common ones, such as the goal position or the environment map. We show how substituting classic ray-based observations with vision makes it much easier to shape the reward functions, as well as aiding better decision-making for the agents.
2. **Deep networks are excellent representation machines**: we argue that shaping reward functions from pixels alone is hard and not necessary. We therefore feed the visual representations processed by deep networks to our RL module, making it easier to generate informed actions. We provide extensive experiments with different kinds of networks and representations, as well as ablation studies.
3. **Real-time, dynamic decision making**: DL can provide highly informative features, making them ideal candidates for visual observations. However, these features alone are not sufficient to directly infer useful actions. By feeding them to the Proximal Policy Optimization algorithm as soon as they are produced, we can train agents in a continuous way in real time, thus bypassing the need to store any collected data or fine-tune deep networks.
4. **KISS ("*Keep it simple, stupid!*") design principle**: developing video games and complex simulations is a hard and technical task in itself. We exploit the KISS design principle to provide a way for game developers to train intelligent agents from within Unreal Engine without writing a single line of code. All the parameters, inputs, outputs, and observations can be changed and configured directly from inside Unreal Engine's interface, unlocking new possibilities for game development for everybody.
5. **Novel hybrid in-engine approach**: To the best of our knowledge, we are the first to explore the impact of visual observations inside Unreal Engine's Learning Agents. All the previous existing works solely relied on game data observations, making the deployment in real-world scenarios harder. Moreover, we are the first to combine NNE's in-engine inference with the Learning Agents' in-engine training, unlocking new possibilities for online vision-based agents training.

2. Related Work

2.1. Deep Networks as Representation Machines

Deep neural networks have proven to be powerful tools for extracting and representing visual features from raw images. In the context of object detection, models like Faster R-CNN [RHGS15] and YOLO [JCQ23] have set benchmarks for real-time and accurate object detection. These models are crucial for tasks where understanding and localizing objects within a frame is necessary.

For visual classification and representation learning, networks like ResNet [HZRS16], ConvNext [LMW*22], Vision Transformers (ViT) [DBK*20], Swin Transformer [LLC*21, LHL*22], DINO [MM20, ODM*23] and recently Mamba [GD23, ZLZ*24] have achieved state-of-the-art performance. Meanwhile, contrastive learning-based approaches such as SimClr [CKNH20], SimSiam [CH21] and SWAV[CMM*20] have demonstrated promising capabilities at modelling meaningful features to be used for multiple tasks. ViT and Swin Transformer have leveraged the transformer architecture for image recognition, showing that ViT-based models can outperform convolutional neural networks on various benchmarks. SimClr, SWAV, DINO and DINOv2 have further pushed the boundaries in contrastive and unsupervised learning, enabling the extraction of invariant features from images without the need for labelled data.

These advancements in visual feature extraction are critical for reinforcement learning, as they provide rich and informative representations that can be used as inputs for RL algorithms.

2.2. Vision-Aided Reinforcement Learning

Vision-aided reinforcement learning is an emerging area that leverages visual inputs to train RL agents. Traditional RL research has often focused on non-visual observations, such as rays, collisions, and distances, as seen in foundational works like [MKS*13], where

Area	Application Field	Method	Additional Observations	Multi-task agents	Runs in-engine	Resolution	Unbounded
Robotics	Robotic arm	[NPD*18]	✗	✗	✓(MuJoCO, PyGame)	Unknown	✗
	Robotic arm	[SK21]	✗	✗	✓(OpenAI Gym)	224x224	✗
	Robot soccer	[TWM*24]	✗	✗	✓(MuJoCo)	40x30	✗
Navigation	Indoor navigation	[MC*19]	✗	✗	✓(OpenAI Gym)	64x48	✗
	Indoor navigation	[CO21]	✓(keypoints)	✗	✗(Gibson dataset)	Unknown	✗
	Indoor navigation	[MSL*22]	✓(memory module)	✗	✗(Gibson dataset)	128x128	✗
	Indoor navigation	[HJK*23]	✓(language input)	✗	✗(Gibson dataset)	640x480	✗
	Indoor navigation	[ZHZ*24]	✓(multi-sensory)	✗	✗(Matterport3D)	Unknown	✗
	Indoor navigation	[DPGCG24]	✓(game state only)	✗	✓(Unreal Engine 5)	-	✗
Gaming	Atari games	[MKS*13]	✗	✓	✓(OpenAI Gym)	84x84	✗
	Doom	[LC17]	✓(game state)	✗	✓(ViZDoom)	108x60	✗
	2D+3D games	[HS18]	✗	✓	✓(OpenAI Gym, ViZDoom)	64x64	✓
	Quake III	[JCD*19]	✓(game state, previous frames)	✗	✓(DeepMind Lab)	Unknown	✓
	Fortnite	[FMH*24]	✓(game state only)	✗	✓(Unreal Engine 5)	-	✓
Multiple	3D games Simulation Indoor navigation Robotics	Ours	✗	✓	✓(Unreal Engine 5)	1024x128	✓

Table 1: Comparison of features between our *Peek-a-bot* framework and similar works. Peek-a-bot is not limited to a single task or game, but can generalize to multiple kinds of applications (games, simulations, navigation and robotics), while not requiring any other information from the engine, just visual data. Very few other works in literature ([DPGCG24, MKS*13, HS18] allow for creating multi-task agents that can play different games or in general conduct multiple tasks. Only two other recent works ([DPGCG24, FMH*24]) focus on implementing intelligent agents inside Unreal Engine 5, which allows to simulate much more photorealistic and complex environments, compared to the widely adopted OpenAI Gym. More importantly, most of the works that rely on visual observations only work in non-open-world environments and require a very small input resolution to train the agents, because they do so using raw pixels and not by first encoding visual observations as features.

the authors demonstrated the potential of deep Q-networks (DQN) to learn policies directly from pixel inputs, achieving human-level performance on several Atari games.

Proximal Policy Optimization (PPO) [SWD*17] provides a robust and scalable algorithm for training RL agents as an alternative to DQN. PPO's ability to handle high-dimensional inputs makes it well-suited for vision-based RL tasks. However, learning directly from raw pixels can be computationally expensive and challenging due to the high-dimensional nature of visual data.

Recent works in literature [MSL*22, CO21, MC*19] have shown that using features extracted from pre-trained deep networks is feasible and can sometimes improve the efficiency and performance of RL agents. [PAED17] introduced a method for intrinsic motivation, using visual features to drive exploration. Similarly, [NPD*18] demonstrated how visual inputs could be used to set and achieve goals in an RL framework. Other works [HJK*23, ZHZ*24] focus on visual navigation from hybrid vision-and-dialogue inputs, but they may not be ideal for gaming or simulation scenarios.

The authors in [SK21] propose a straightforward yet effective approach that uses features extracted from pre-trained ResNet models in conjunction with RL algorithms. This method has been shown to deliver results comparable to learning directly from rays, effectively using visual inputs to train RL agents.

2.3. Learning Inside Game Engines

Game engines provide rich and complex environments for training RL agents. Unity's ML-Agents [JBT*20] was one of the first solutions offering a platform for integrating RL with Unity environments, allowing for the training of intelligent agents in various simulated scenarios, providing a bridge between academic research and practical applications in gaming and simulation.

Unreal Engine, with its high-fidelity graphics and dynamic environment capabilities, has also been recently used for RL applications. Unreal Engine's Learning Agents[Epi24a] and Neural Network Engine (NNE)[Epi24b] modules provide tools for integrating deep learning and RL directly within the engine. The flexibility and realism offered by Unreal Engine make it an ideal platform for developing and testing vision-based RL systems. [DPGCG24] proposes an alternative to the Social Force Model leveraging Unreal's Learning Agents. Similarly, [FMH*24] shows how to train human-like players inside Fortnite using imitation learning.

The DeepMind Control Suite [TDM*18] and the OpenAI Gym [Bro16] provide a collection of continuous control tasks in a physics-based simulator and are widely used for benchmarking RL algorithms, particularly those involving visual inputs.

Our method, *Peek-a-bot*, builds upon these advancements by fully integrating our DL-RL hybrid training pipeline within Unreal Engine 5. We leverage the engine's capabilities to create photorealistic environments and real-time inference of visual features, which are then used to train RL agents using the PPO algorithm. This approach does not require external data processing and storage, streamlining the training process and making it accessible to game developers without requiring extensive machine learning expertise. An overview of the features of *Peek-a-bot* compared with related works in different application areas can be found in Table 1.

3. Method

Peek-a-bot is designed to be a complete system for game developers or people working with game engines to create large-scale

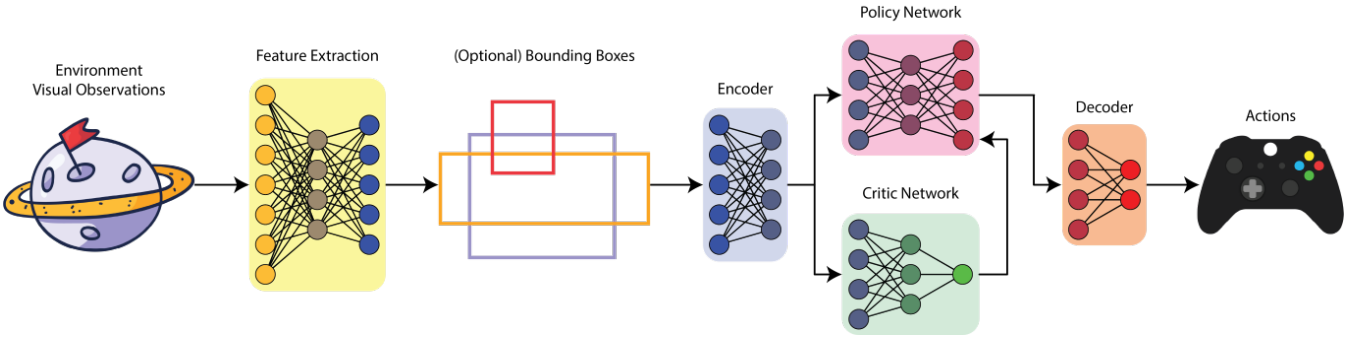

Figure 2: *Peek-a-bot* architecture. Left to right: the environment is sensed through visual observations (RGB images), which are fed to a backbone neural network chosen by the user. If the selected backbone is pre-trained for an object detection task (optional), the game developer can decide to use the bounding boxes as observations instead of the visual features. A small encoder processes the observations to create inputs to the Policy and Critic networks, which are trained using the PPO algorithm. A small decoder transforms the Policy network's output into actions, which can be seen as the movement inputs typical of a video game controller.

dynamic simulations. With our solution, the user can create custom scenarios, capture visual observations and train intelligent agents of diverse nature, all inside Unreal Engine 5 and in real-time, without touching a single line of code.

In this section, we show how the configuration of the training scenario works, as well as the architectural details and the rewards shaping.

3.1. Training scenarios

Historically, developing a video game required substantial programming skills, as the process involved writing complex code in languages such as C++ or C#. Despite it being still necessary to model specific behaviours, the game development industry is promoting the usage of alternatives, such as the Blueprint system in Unreal Engine or other node editors for building shaders or logic inside Blender [Ble24], Godot [God24] and Unity [JBT*20]. The possibility of developing video games without writing many lines of code allowed independent developers or non-tech people to create entire video games or complex simulations from scratch, only focusing on the game design part.

Inspired by this approach, we designed *Peek-a-bot* to be a complete solution for training visual agents without the need to write any code. As seen in Fig 3, we allow the developer to configure the entire training scenario in-engine, from the building of the environment to the game logic using Blueprints, to the finer training details and parameters using a GUI interface. From a technical point of view, we rely on Unreal Engine's Neural Network Engine (NNE) as a neural network inference engine. This allows us to use the popular ONNX format to load any type of backbone for extracting visual observations and run inference on CPU or GPU, depending on the scenario.

As an example, provided an arbitrarily complex open-world scene inside Unreal Engine, the game developer can use *Peek-a-bot* to drag-and-drop pre-trained or untrained agents inside the environment, configure their "brain" using only a GUI interface or loading a custom ONNX[The24] backbone and then run training or inference by pressing a single button. Trained weights can be stored inside a single file, that can be later used for inference, or fine-tuned.

For the Reinforcement Learning part, we rely on Unreal Engine's experimental Learning Agents plugin, which allows us to simplify the process of communication with an external Python module for training. After defining agents with custom view cones (as seen in Fig. 4), we extract the latent space vectors from the chosen backbone (see section 3.2 for more details) and encode them to serve as observations for the RL network. By using the PPO algorithm, each agent will learn to navigate the scene, identify goals and obstacles, and maximize their rewards, only using these visual observations.

A key advantage of using a unified visual representation as the sole observation is that it allows for the simultaneous training of multiple agents with varying rewards and action types within the same scenario. This approach is challenging to implement when agents have different observations instead. For example, inside an urban scenario, pedestrian, vehicle and drone agents sharing the same type of visual observation can be trained simultaneously, despite their different rewards and actions. This also promotes generalization to various situations, especially when training on multiple scenarios.

In Fig. 4 we show some examples of dynamic scenarios that can be modelled using *Peek-a-bot*. Many others can be designed, such as first responder, CCTV camera coverage or whatever other scenario can exploit visual observations or a combination of vision and other sensors.

3.2. Neural networks architectures and Actions

An overview of the whole *Peek-a-bot* architecture is shown in Fig. 2. As stated in Sec. 2, we want to distinguish from the usual approaches in the literature that aim to transform raw pixels into actions with a single, end-to-end network. By using a pre-trained feature extractor instead, it is possible to (i) use high-resolution input images, (ii) drastically reduce the training time, while (iii) mitigating the risk of overfitting to a specific scenario due to the end-to-end

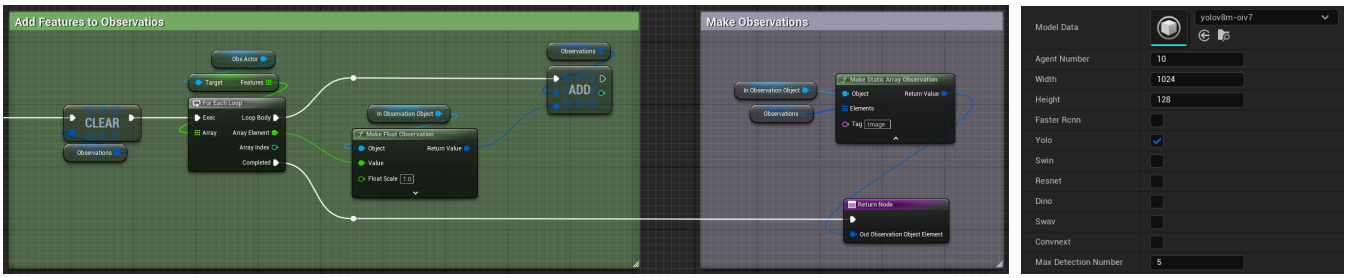

Figure 3: Our method is designed with the game developer in mind. All the visual inputs, backbone and RL network parameters, as well as the outputs and all the other parameters, are easily accessible through a GUI or Blueprint nodes, without the need for writing a single line of code. **Left**: a set of Blueprint nodes managing the visual observations. **Right**: a snippet of our GUI allowing the selection of the number of agents to be spawned, the desired backbone network architecture as well as the visual input configuration (resolution, FOV, etc.).

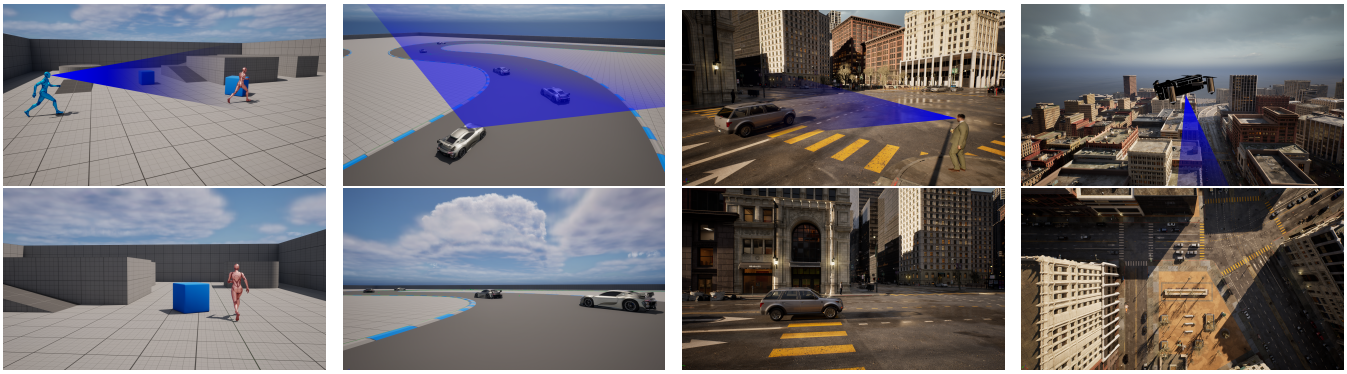

Figure 4: *Peek-a-bot* visual observations system applied to multiple dynamic and unbounded scenes. **Top row**: different types of training scenarios (hide and seek, car race, urban navigation, drone coverage) and visual observation cones (blue). **Bottom row**: corresponding visual observations to be encoded by the neural network into observation vectors. Our method allows us to manage different kinds of rich, dynamic scenes while dropping the need for different kinds of sensors for each scenario.

training. Please refer to Table 1 for a comparison to existing works in literature. From a high-level viewpoint, we want to build a system that takes a single image as input and outputs a combination of actions that can be seen as a button pressed on a standard game controller. The user can customize the number and kinds of actions that the trained agents should be allowed to perform.

To do so, we design a three-fold process, that we summarize below:

1. **Defining the visual stimuli**. Each agent can be equipped with one or more cameras. Each camera is fully customizable in terms of field of view and intrinsic/extrinsic camera parameters. The field of view (FOV) is particularly important for this application because it properly defines the visual field [Tra27] of the agent. Smaller FOV values will correspond to human macular to near-peripheral (up to 30°) area, while larger FOV values (60° to 100°) can be associated with the human's mid and far peripheral vision. We tested our method with different values and noticed that agents equipped with image classification-pretrained backbones tend to prefer smaller FOV values, while the opposite applies to agents with object detection-backed backbones. See Sec. 4.1 for further details.
   At this stage, the input to the backbone neural network can be defined as I, with size (B,W,H,C), where B is the batch size, and W, H and C are the width, height and channels of each image in the batch.
2. **Extracting visual features**. We use the NNE library to write a C++ parser that can load an ONNX neural network and its weights and run inference on the GPU. The input to the network is I and as for the output we have a tensor of features F.
   Depending on the backbone type, F can assume different shapes such as (B,D) for a latent space of a feature extractor with latent dimension D or (B,S,N) for object detection networks that output bounding boxes. S contains the spatial positions and the confidence values of each bounding box for each class and N contains all the detections. We apply batched non-maximum suppression (NMS) to this large matrix to filter and flatten the most accurate detections only, resulting in the same $F=(B,D)$ tensor for convenience.
   This stage can be summarized with Algorithm 1. The same concept applies to other techniques, such as semantic segmentation or depth estimation. The only requirement is for F to take the shape of (B,D), where D is parametrized via our GUI.
3. **Learning a set of policies**. Once the visual features F are extracted, they are used as inputs to train the policy and critic networks within a custom Proximal Policy Optimization (PPO) re-

Algorithm 1: Extracting Visual Features

Input: I: Input image tensor
Output: F: Output feature tensor
Load ONNX neural network and weights using NNE;
Run inference on the GPU with input I;
Obtain output tensor F;
if *backbone type is latent space feature extractor* **then**
    $F \leftarrow (B, D)$ `// D is the latent dimension`
else
    $F \leftarrow (B, S, N)$ `// S contains spatial positions and confidence values, N contains detections`
    $F \leftarrow NMS(F)$ `// Apply batched non-maximum suppression (NMS)`
    $F \leftarrow (B, D)$ `// Reshape for convenience`

inforcement learning framework that supports recurrent policies [WFPS10] and implements advantage normalization and gradient norm clipping [PMB13] to improve training stability. Here, we detail the steps involved in transforming F into meaningful actions.

Policy Network and Critic Network
The policy network $\pi_\theta(a|s)$ takes as input F and an optional memory buffer M and outputs the probability distribution over actions given the state, parameterized by θ. The critic network $V_\phi(s)$ estimates the value function, parameterized by ϕ. We design both networks to have a single hidden layer and a memory buffer, both of size 128.

Generalized Advantage Estimation (GAE)
The advantage function A_t at time step t evaluates the relative value of an action compared to the baseline (value function $V_\phi(s)$). It is defined as:

$$A_t = Q(s_t, a_t) - V_\phi(s_t)$$

where $Q(s_t, a_t)$ is the action-value function, approximated by the reward-to-go:

$$Q(s_t, a_t) = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$$

To compute the advantages efficiently, the Generalized Advantage Estimation (GAE) [SML*15] is used:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

$$A_t = \sum_{l=0}^{T-t} (\gamma\lambda)^l \delta_{t+l}$$

where γ is the discount factor and λ is the GAE parameter.

Clipped Surrogate Objective
The PPO algorithm optimizes the policy by maximizing a clipped surrogate objective function. The objective function $L_t(\theta)$ is defined as:

$$L_t(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

is the probability ratio between the new and old policies, $\hat{A}_t$ is the estimated advantage, and ϵ is a hyperparameter that controls the clipping range.

Value Function Loss
The value function is updated by minimizing the L_1 loss (rather than L_2 loss) between the predicted value and the empirical return to enhance stability:

$$L_v(\phi) = \mathbb{E}_t \left[\left| V_\phi(s_t) - R_t \right| \right]$$

where R_t is the empirical return computed as:

$$R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$$

Loss and Optimization
The final loss for the PPO algorithm combines the policy loss and the value function loss, with additional regularization and entropy terms to encourage exploration:

$$L(\theta, \phi) = \mathbb{E}_t \left[L_t(\theta) - c_1 L_v(\phi) + c_2 H(\pi_\theta(\cdot|s_t)) + c_3 R(\pi_\theta(\cdot|s_t)) \right]$$

where c_1, c_2, and c_3 are coefficients for the value loss, entropy bonus, and regularization terms, respectively, and $H(\pi_\theta(\cdot|s_t))$ is the entropy of the policy, and $R(\pi_\theta(\cdot|s_t))$ is the regularization term.

Actions Once the policy network $\pi_\theta(a|s)$ outputs the action-encoded vector, the actions are computed using a decoder network. The decoder processes the action-encoded vector to produce the final action distribution, as shown in Fig. 5.

3.3. Rewards

As for the observations and the actions, *Peek-a-bot* allows the user to customize the rewards for each agent or groups of agents. For example, in a hide-and-seek scenario, some agents will be positively rewarded whenever they see or reach other hiding agents, while the hiding agents will be negatively rewarded. However, differently from how we shape the observations, which are always and only visual observations, the rewards shaping is strictly linked to each scenario, thus they need to be defined based on the agents' goals.

In this section, we provide a brief description of the rewards given to agents in a "capture the flag" or "find an object in a scene" scenario. The same rewards can be applied in arbitrarily hard scenarios (e.g., find a key and then find a door to open it, or, follow an agent until you reach it and then go chase another one).

It is important to notice that agents will get rewards only at training time, at inference time agents will use visual observations and trained weights to generate the actions.

In this scenario type, the overall reward for each agent at time t is given by:

$$r^t = \lambda_g r_g^t + \lambda_c r_c^t + \lambda_h r_h^t + \lambda_p r_p^t \quad (1)$$

with $\lambda_g = 1$, $\lambda_c = 0.1$, $\lambda_h = \lambda_p = 0.001$, where

Pre-training	Observation type	Architecture	No Obstacles		Obstacles		Average	Ranking
			No BG	BG	No BG	BG		
Self-supervised	**Features**	DINO (ViT-B/16)	51,89	66,37	81,96	121,20	80,35	8
Contrastive	**Features**	SWAV (ResNet50)	30,45	41,60	69,69	54,68	49,11	4
Supervised	**Features**	ResNet34	44,04	61,99	64,06	71,37	60,36	6
		ConvNext-T	46,34	60,45	60,81	41,81	52,35	5
		SWINv2-T/16	19,90	25,77	63,45	33,52	35,66	2
	BBoxes	YOLOv8n	28,10	78,70	38,28	136,5	70,40	7
		Faster R-CNN	47,30	38,05	49,72	32,11	41,80	3
		YOLOv8m	13,26	21,11	33,39	56,28	31,01	1

Table 2: Quantitative results of our Peek-a-bot framework when using different kinds of backbones for extracting observations. All the numerical values represent the Average Episode Length for all the RL agents in different scenario configurations.

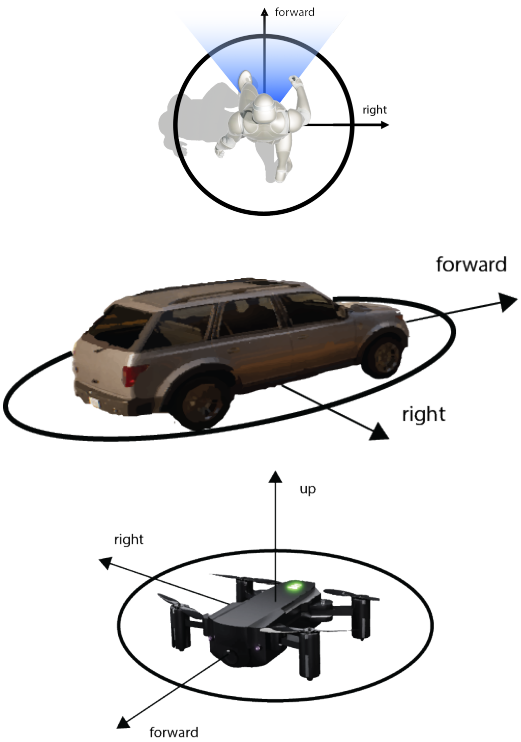

Figure 5: Example of movement actions for pedestrian, car and drone scenarios. The number and type of actions are not fixed and can be managed by the user, according to their needs.

- $r_g^t = 1$ (**goal**) is a fixed reward for reaching the goal, which can be both static or dynamic. Whenever an agent collides with a large enough defined bounding cube around the goal, the corresponding visual observation will be mapped to a positive reward. The main reason for the bounding cube usage is to prevent the creation of too zoomed-in visual observations, especially when using a small field of view.
- r_c^t (**collision**) is a fixed penalty given to the agent whenever it collides with an obstacle or wall in the scene. Please note that the agent has no explicit knowledge or observation of what an obstacle is, so they will learn it during training. Among our proposed configurations, the ones based on feature observations (and not bounding boxes) can better adapt to this reward, especially in case of object not belonging to the labels in the backbone's training set.
- r_h^t (**haste**) is a fixed penalty given at each frame in the simulation. While this may seem irrelevant to the training, it promotes the agent to reach its goal as fast as possible. The parameter λ_h can be used to decide the importance of this behaviour, and thus how "hasty" an agent should be.
- r_p^t (**proximity**) is a dynamically changing reward, that is positive whenever the agent is getting near its goal, and negative otherwise. Please note that the agent has no observation or prior knowledge of where the goal is, so it has to learn it from visual observations only. This reward can be expressed as:

$$r_p^t = -\frac{dist(p_a^{t-1}, p_g^{t-1}) - dist(p_a^t, p_g^t)}{100}, \quad dist\Big(A(x_1,y_1,z_1), B(x_2,y_2,z_2)\Big) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2} \tag{2}$$

where p_a is the agent position, p_g is the goal position, and t is the current time iteration.

4. Results and Ablation Studies

To evaluate the performance of our *Peek-a-bot* method, we design a set of experiments involving eight common pre-trained models. We group them based on their pretraining method (self-supervised, contrastive, and supervised) and on the type of output (features or bounding boxes). In these scenarios, agents are trained to accomplish a specific task, that is, reaching a goal. This is analogous to performing a task such as finding a key and opening a door, or following another agent. We train on each scenario for 1500 steps on a single NVIDIA RTX 4090.

Each training session was performed four times for every model, altering the environment settings. 50% of the sessions were conducted without obstacles and the other half with randomly spawned

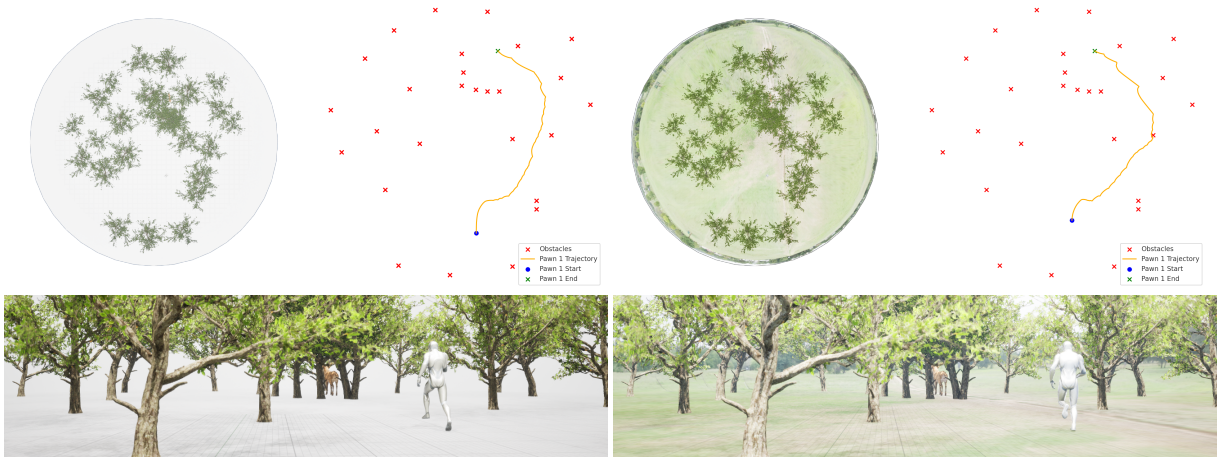

Figure 6: *Peek-a-bot* qualitative results on the same environment with and without an environment map, using the Swin transformer as the backbone. Both trajectories show that the agent is capable of finding the goal even in visually cluttered and challenging scenarios.

obstacles. For both cases, we trained 50% of the sub-sessions using a real-life environment map (as shown in Fig. 1) and the other half without any background, letting the agents focus on just the goals and obstacles. The images provided to the object detection models had a resolution of 1024x128 and a field of view (FOV) of 90 degrees. For the feature extraction models, we used images of 224x224 with an FOV of 30 degrees. Increasing the FOV of the latter to 60° resulted in a drastic decrease in performance, as shown in Table 3. We attribute this behaviour to the fact that models pre-trained for an image classification task require a tight crop of the subject at every time. The evaluation metric chosen for the quantitative results was the average episode length for reaching the goal.

In Table 2 we show an overview of the effect of each backbone in training RL agents within our *Peek-a-bot* framework. We observe that YOLOv8m, by performing object detection, generally achieves the best result. However, using SWINv2 as the backbone we achieve competitive results, despite using features instead of proper bounding box observations. Despite using smaller images and having a smaller model size, SWINv2 does not suffer from the same limitations that object detection models face during the detection phase. Specifically, if the detection fails because the object is not recognized or belongs to a class the model was not trained on, the observation will be null. This issue is evident in the significantly poorer performance of the smaller YOLO model. Feature extraction (FE) models, on the other hand, always extract some kind of features, resulting in richer and less sparse observations. This consistent capability of FE models contributes to their robust performance.

Another advantage of FE models is the ease with which the extracted features can be used without any need for post-processing. In contrast, models like YOLO require handling output data using methods such as non-maximum suppression. FE models, by providing readily usable features, simplify the integration into reinforcement learning pipelines and contribute to more efficient and streamlined training processes.

Among all the backbones shown in Table 2, we observe that supervised methods using feature observations (Fig. 7) are the most consistent at training time, with a smooth convergence, followed by the self-supervised and contrastive methods (Fig. 8).

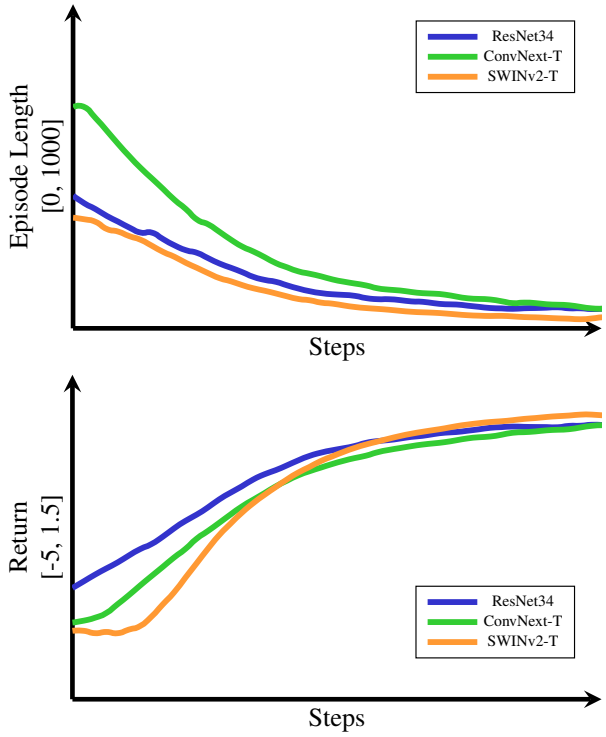

Figure 7: Convolutional vs Transformer visual features as observations in *Peek-a-bot*. The three backbones offer smoother results compared to methods in Figures 8 and 9. The features provided by the SWIN transformer are better suited for visual navigation in our tested scenarios.

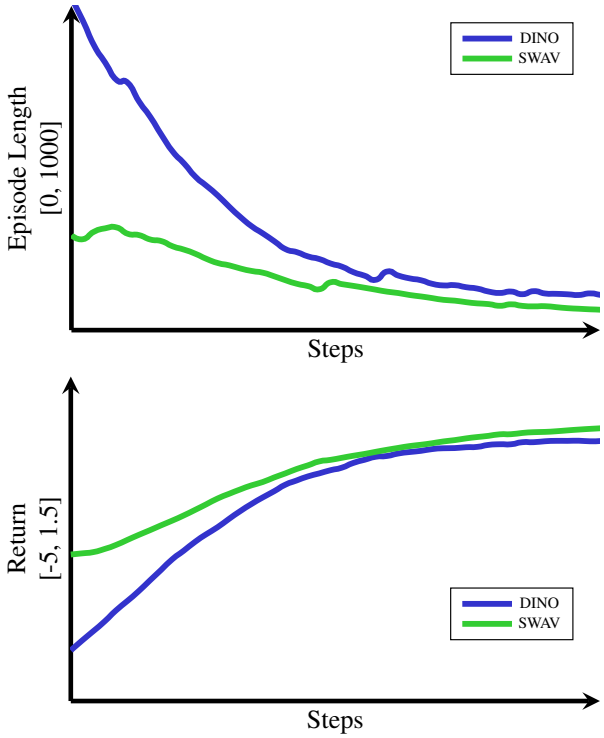

Figure 8: Self-supervised (DINO) vs contrastive (SWAV) feature extraction backbone. The kind of contrastive learning used in SWAV seems to produce better-suited features to be used as observations when compared to DINO.

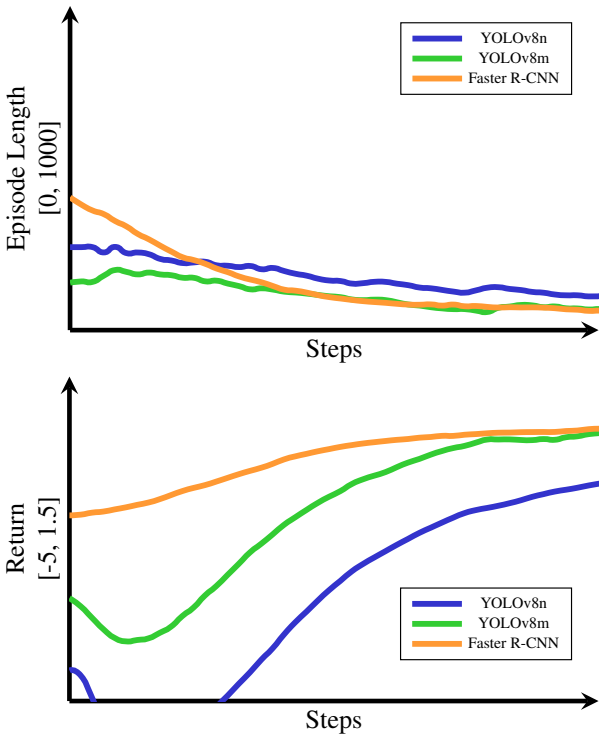

Figure 9: YOLO vs Faster R-CNN object detection backbone. The *nano* version of YOLO performs significantly worse than the *medium* one. Faster R-CNN provides slightly better results and a smoother training curve despite it having been trained on the 80 COCO classes instead of the 600 Open Images classes found in YOLOv8.

Detection-based methods, on the other hand, (Fig. 9) present less smooth training curves, despite winning in our benchmarks. We argue that this behaviour is due to the less sparse and uniform observations given by the combination of bounding boxes and labels. However, when the detection fails, the agent becomes virtually blind, resulting in the oscillating behaviours in Fig. 9.

4.1. Ablation studies

The main claim of this work is that by using only vision instead of relying on the game state to get observations, it is possible to simplify the pipeline of training RL systems for video games and simulations. We conducted multiple experiments comparing the game state observations with the visual ones, and unsurprisingly, the first usually resulted in faster convergence to the desired behaviour. However, when training agents using vision instead of the game state, we observed that the overall behaviour was more "human-like", promoting exploration and curiosity instead of directly completing the task as fast as possible. Despite this behaviour is difficult to quantify solely with an ablation study, it is important to notice that using visual observations would make it easier to move from a synthetic scenario to a real one, especially when dealing with embodied agents.

We provide extensive ablation studies on our PPO reinforcement learning pipeline, while keeping the backbone fixed (SWINv2 for simplicity), as shown in Table 3. The baseline configuration, shown in bold, is the one that achieves the best results. As a general observation, increasing or decreasing the policy and critic network dimensions or the number of hidden layers leads to worse performance, meaning that the network becomes over-parametrized or under-parametrized. The usage of memory as an additional input to the policy and critic networks is fundamental to making the agents remember recently visited locations. Different activation functions and epsilon clip values have a smaller impact over the average episode length, while a smaller FOV promotes FE backbones to extract features from a centrally cropped part of the environment in front of the agent, resulting in overall better results. See Fig. 6 for additional qualitative results.

5. Conclusions

In this work, we presented *Peek-a-bot*, an in-engine hybrid DL-RL solution to train intelligent agents using only visual observations inside Unreal Engine 5. Each agent has no other observation, such as rays, distances or metric clues, so it needs to navigate the environment using only visual features. Our method works in real-time and on a multitude of photorealistic environments, providing a high degree of generalization, making it ideal for video games and complex simulations. In the future, we plan to extend our method by

Memory (ON, OFF)	**23,01**	37,93	
Policy network (64, 128, 256)	39,38	**23,01**	26,26
Critic network (64, 128, 256)	53,89	**23,01**	36,57
Activation (ELU, ReLU, Tanh)	29,68	**23,01**	23,81
Epsilon Clip (0, 0.2, 0.5)	29,68	**23,01**	28,85
Hidden Layers (1, 2, 4)	**23,01**	56,89	157,50
FOV (30, 60, 90)	**23,01**	49,95	25,92

Table 3: Ablation studies for our RL architecture when using a SWINv2 backbone. All the numerical values represent the Average Episode length for every agent in the environment.

including an imitation learning module to mimic the behaviour of players in-game, as well as augmenting the agents' visual observations with other kinds of sensory stimuli.

6. Acknowledgements

Funded by the European Union - Next Generation EU, Mission 4 Component 2 - CUP E63C22000970007.

References

[AAA*23] ACHIAM J., ADLER S., AGARWAL S., AHMAD L., AKKAYA I., ALEMAN F. L., ALMEIDA D., ALTENSCHMIDT J., ALTMAN S., ANADKAT S., ET AL.: Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[Ble24] BLENDER ONLINE COMMUNITY: Blender - a 3D modelling and rendering package, 2024. URL: http://www.blender.org.

[Bro86] BROOKS R.: A robust layered control system for a mobile robot. *IEEE journal on robotics and automation 2*, 1 (1986), 14–23.

[Bro16] BROCKMAN G.: Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[CH21] CHEN X., HE K.: Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2021), pp. 15750–15758.

[CKNH20] CHEN T., KORNBLITH S., NOROUZI M., HINTON G.: A simple framework for contrastive learning of visual representations. In *International conference on machine learning* (2020), PMLR, pp. 1597–1607.

[CMM*20] CARON M., MISRA I., MAIRAL J., GOYAL P., BOJANOWSKI P., JOULIN A.: Unsupervised learning of visual features by contrasting cluster assignments. *Advances in neural information processing systems 33* (2020), 9912–9924.

[CO21] CHOI Y., OH S.: Image-goal navigation via keypoint-based reinforcement learning. In *2021 18th International Conference on Ubiquitous Robots (UR)* (2021), IEEE, pp. 18–21.

[DBK*20] DOSOVITSKIY A., BEYER L., KOLESNIKOV A., WEISSENBORN D., ZHAI X., UNTERTHINER T., DEHGHANI M., MINDERER M., HEIGOLD G., GELLY S., ET AL.: An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[DPGCG24] DELLA PIETRA D., GARAU N., CONCI N., GRANELLI F.: Dynamic crowd routing: Rl-driven crowd dynamics. In *2024 IEEE 25th International Workshop on Multimedia Signal Processing (MMSP)* (2024), IEEE, pp. 1–6.

[Epi24a] EPIC GAMES: Unreal Engine Learning Agents, 2024. URL: https://dev.epicgames.com/community/learning/courses/kRm/unreal-engine-learning-agents-5-4.

[Epi24b] EPIC GAMES: Unreal Engine Neural Network Engine (NNE), 2024. URL: https://dev.epicgames.com/community/learning/courses/e7w/unreal-engine-neural-network-engine-nne.

[FMH*24] FARHANG A. R., MULCAHY B., HOLDEN D., MATTHEWS I., YUE Y.: Humanlike behavior in a third-person shooter with imitation learning. In *2024 IEEE Conference on Games (CoG)* (2024), IEEE, pp. 1–4.

[GD23] GU A., DAO T.: Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[God24] GODOT FOUNDATION: Godot Engine, 2024. URL: https://godotengine.org.

[HJK*23] HWANG M., JEONG J., KIM M., OH Y., OH S.: Meta-explore: Exploratory hierarchical vision-and-language navigation using scene object spectrum grounding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2023), pp. 6683–6693.

[HS18] HA D., SCHMIDHUBER J.: Recurrent world models facilitate policy evolution. *Advances in neural information processing systems 31* (2018).

[HZRS16] HE K., ZHANG X., REN S., SUN J.: Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[JBT*20] JULIANI A., BERGES V.-P., TENG E., COHEN A., HARPER J., ELION C., GOY C., GAO Y., HENRY H., MATTAR M., LANGE D.: Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627* (2020). URL: https://arxiv.org/pdf/1809.02627.pdf.

[JCD*19] JADERBERG M., CZARNECKI W. M., DUNNING I., MARRIS L., LEVER G., CASTANEDA A. G., BEATTIE C., RABINOWITZ N. C., MORCOS A. S., RUDERMAN A., ET AL.: Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science 364*, 6443 (2019), 859–865.

[JCQ23] JOCHER G., CHAURASIA A., QIU J.: Ultralytics YOLO, Jan. 2023. URL: https://github.com/ultralytics/ultralytics.

[LC17] LAMPLE G., CHAPLOT D. S.: Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence* (2017), vol. 31.

[LHL*22] LIU Z., HU H., LIN Y., YAO Z., XIE Z., WEI Y., NING J., CAO Y., ZHANG Z., DONG L., ET AL.: Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2022), pp. 12009–12019.

[LLC*21] LIU Z., LIN Y., CAO Y., HU H., WEI Y., ZHANG Z., LIN S., GUO B.: Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision* (2021), pp. 10012–10022.

[LMW*22] LIU Z., MAO H., WU C.-Y., FEICHTENHOFER C., DARRELL T., XIE S.: A convnet for the 2020s, 2022.

[MC*19] MA L., CHEN J., ET AL.: Using rgb image as visual input for mapless robot navigation. *arXiv preprint arXiv:1903.09927* (2019).

[MKS*13] MNIH V., KAVUKCUOGLU K., SILVER D., GRAVES A., ANTONOGLOU I., WIERSTRA D., RIEDMILLER M.: Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[MM20] MISRA I., MAATEN L. V. D.: Self-supervised learning of pretext-invariant representations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2020), pp. 6707–6717.

[MSL*22] MEZGHAN L., SUKHBAATAR S., LAVRIL T., MAKSYMETS O., BATRA D., BOJANOWSKI P., ALAHARI K.: Memory-augmented reinforcement learning for image-goal navigation. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2022), IEEE, pp. 3316–3323.

[NPD*18] NAIR A. V., PONG V., DALAL M., BAHL S., LIN S., LEVINE S.: Visual reinforcement learning with imagined goals. *Advances in neural information processing systems 31* (2018).

[ODM*23] OQUAB M., DARCET T., MOUTAKANNI T., VO H., SZAFRANIEC M., KHALIDOV V., FERNANDEZ P., HAZIZA D., MASSA F., EL-NOUBY A., ET AL.: Dinov2: Learning robust visual features without supervision. *arXiv preprint arXiv:2304.07193* (2023).

[PAED17] PATHAK D., AGRAWAL P., EFROS A. A., DARRELL T.: Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning* (2017), PMLR, pp. 2778–2787.

[PMB13] PASCANU R., MIKOLOV T., BENGIO Y.: On the difficulty of training recurrent neural networks. In *International conference on machine learning* (2013), Pmlr, pp. 1310–1318.

[RHGS15] REN S., HE K., GIRSHICK R., SUN J.: Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems 28* (2015).

[SK21] SHAH R., KUMAR V.: Rrl: Resnet as representation for reinforcement learning. *arXiv preprint arXiv:2107.03380* (2021).

[SML*15] SCHULMAN J., MORITZ P., LEVINE S., JORDAN M., ABBEEL P.: High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[STZ*19] SHAO K., TANG Z., ZHU Y., LI N., ZHAO D.: A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944* (2019).

[SWD*17] SCHULMAN J., WOLSKI F., DHARIWAL P., RADFORD A., KLIMOV O.: Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[TDM*18] TASSA Y., DORON Y., MULDAL A., EREZ T., LI Y., CASAS D. D. L., BUDDEN D., ABDOLMALEKI A., MEREL J., LEFRANCQ A., ET AL.: Deepmind control suite. *arXiv preprint arXiv:1801.00690* (2018).

[The24] THE LINUX FOUNDATION: Open Neural Network Exchange (ONNX), 2024. URL: `https://onnx.ai`.

[Tra27] TRAQUAIR H. M.: *An introduction to clinical perimetry*. Kimpton, 1927.

[TWM*24] TIRUMALA D., WULFMEIER M., MORAN B., HUANG S., HUMPLIK J., LEVER G., HAARNOJA T., HASENCLEVER L., BYRAVAN A., BATCHELOR N., ET AL.: Learning robot soccer from egocentric vision with deep reinforcement learning. *arXiv preprint arXiv:2405.02425* (2024).

[WFPS10] WIERSTRA D., FÖRSTER A., PETERS J., SCHMIDHUBER J.: Recurrent policy gradients. *Logic Journal of IGPL 18*, 5 (2010), 620–634.

[ZHZ*24] ZHENG D., HUANG S., ZHAO L., ZHONG Y., WANG L.: Towards learning a generalist model for embodied navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2024), pp. 13624–13634.

[ZLZ*24] ZHU L., LIAO B., ZHANG Q., WANG X., LIU W., WANG X.: Vision mamba: Efficient visual representation learning with bidirectional state space model. *arXiv preprint arXiv:2401.09417* (2024).