Deep Reinforcement Learning for Navigation in AAA Video Games

Eloi Alonso* , **Maxim Peter*** , **David Goumard** and **Joshua Romoff**
Ubisoft La Forge
eloi.alonso@unige.ch, {maxim.peter, david.goumard, joshua.romoff}@ubisoft.com

Abstract

In video games, *non-player characters* (NPCs) are used to enhance the players' experience in a variety of ways, e.g., as enemies, allies, or innocent bystanders. A crucial component of NPCs is navigation, which allows them to move from one point to another on the map. The most popular approach for NPC navigation in the video game industry is to use a *navigation mesh* (NavMesh), which is a graph representation of the map, with nodes and edges indicating traversable areas. Unfortunately, complex navigation abilities that extend the character's capacity for movement, e.g., grappling hooks, jetpacks, teleportation, or double-jumps, increase the complexity of the NavMesh, making it intractable in many practical scenarios. Game designers are thus constrained to only add abilities that can be handled by a NavMesh. As an alternative to the NavMesh, we propose to use Deep Reinforcement Learning (Deep RL) to learn how to navigate 3D maps in video games using any navigation ability. We test our approach on complex 3D environments that are notably an order of magnitude larger than maps typically used in the Deep RL literature. One of these environments is from a recently released AAA video game called Hyper Scape[1]. We find that our approach performs surprisingly well, achieving at least 90% success rate in a variety of scenarios using complex navigation abilities.

1 Introduction

Realistic navigation for *non-player characters* (NPCs) is an important component in most video games to enhance the players' experience. The traditional pipeline for NPC navigation is as follows:

1. A graph representation of the world is pre-generated from the game geometry.
2. At runtime, a pathfinding algorithm like A* [Hart *et al.*, 1968] is applied on this graph to find the shortest path between any pair of locations in the game.
3. A controller tailored to the character is used to follow this path.

The *navigation mesh* (NavMesh) [Snook, 2000] is the most used representation of the world geometry [McAnlis, 2008]. This graph, whose nodes represent the traversable surfaces of the 3D environment as convex polygons, is a compact representation of the world, independent of character abilities. Adding character constraints or abilities is traditionally done through other means, such as tweaking the pathfinding algorithm or extending the NavMesh with additional links (more details can be found in Section 2.1). However, these approaches impose limitations on the kind of abilities that NPCs can use to navigate, which detract from its realism.

In this paper, we set out to replace classical graph-based navigation with a system that can learn how to navigate between any two points on a map using all of the navigation options available to the character. Replacing the NavMesh with a learning system in modern AAA video games is challenging for several reasons. First, as the game worlds are increasingly more realistic, they have become both larger and more complex. Second, the maps can be dynamic, as objects and other characters can move in the world. Finally, to replace the existing NavMesh, solutions need to run on a tight budget at runtime and ideally be relatively cheap to train.

To tackle these issues, we opt for a *model-free* Reinforcement Learning (RL) approach [Sutton and Barto, 2018] to the navigation problem, where an agent learns a *policy* that maximizes a reward signal through interacting with an environment. Specifically, we train an agent with Deep RL to navigate to locations in the game world using Soft Actor-Critic (SAC) [Haarnoja *et al.*, 2018a] as our learning algorithm. We also build off of recent work that augments the agent's state with memory to effectively solve navigation tasks in complex 3D environments [Mirowski *et al.*, 2016; Kapturowski *et al.*, 2018; Wijmans *et al.*, 2019].

We begin by providing the relevant background and related work on navigation in Section 2, followed by a detailed description of our system in Section 3. Then, in Section 4, we demonstrate the performance of our Deep RL system and examine several ablations on two maps created using the Unity

*Equal contributions, alphabetical order.

[1]Hyper Scape was used for experimentation purposes only. Our agent was not shipped in the game. Video in Hyper Scape: youtu.be/DKdQFajLfzk

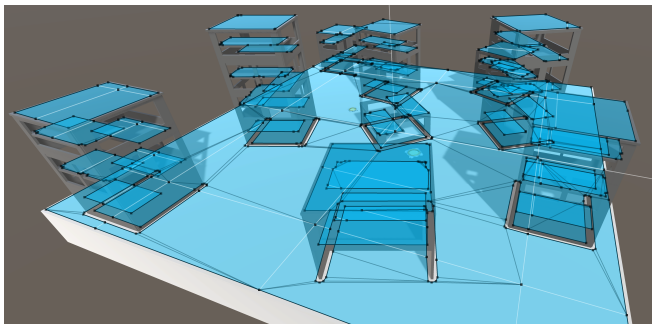

Figure 1: The NavMesh is automatically generated from the map geometry. Each blue convex polygon is a node of the NavMesh and represents a traversable region of the map. NavMesh edges associate adjacent polygons. Here, the NavMesh is not a connected graph since building floors and rooftops are not connected to anything.

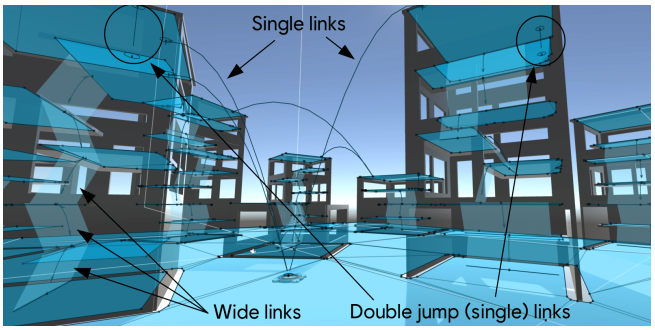

Figure 2: NavMesh links (either single or wide) are added to enable navigation between disconnected components of the graph. We added the minimum number of links such that the agent can reach any point on the map, e.g., a link that connects the rooftop to its top floor is added since the rooftops are accessible via double jumps.

game engine [Juliani *et al.*, 2018]. We also train our agent on several map sizes in the engine of a recently released AAA video game to validate its potential as an alternative to the NavMesh. Notably, in contrast to previous RL-based approaches to navigation, we successfully train using continuous actions on 3D maps an order of magnitude bigger than simulators used in the research community [Wydmuch *et al.*, 2018; Manolis Savva *et al.*, 2019].

2 Background and Related Work

2.1 Navigation in AAA Games

Waypoint Graph. Prior to the NavMesh, the most popular approach to game navigation was to use a *waypoint graph* [Lidén and others, 2002], which consists of a set of points of interest connected to each other. Unfortunately, this approach has several main drawbacks: its manual construction can be cumbersome and prone to human errors, it cannot handle dynamic objects, it is expensive to build since it needs to check all $n(n-1)$ combinations of paths, and the paths tend to not look realistic since all agents follow the same set of constrained paths [Tozour and Austin, 2002].

NavMesh. The NavMesh solves most of the aforementioned problems with waypoint graphs and is thus currently the main navigation tool used in video games [McAnlis, 2008]. Specifically, the NavMesh divides the game map into a set of convex regions, which can each be trivially navigated within. It can be generated from either the raw geometry of the world using voxel-based approaches, or using pre-processed inputs like planar layers [Van Toll *et al.*, 2016; Oliva and Pelechano, 2011]. It is thus a compact representation of the world's traversable terrains, independent of any character ability. Once the polygons have been placed, a graph is created by using the polygons as nodes and by connecting adjacent polygons with edges. With the built graph, search algorithms such as A* and Dijkstra's algorithm can be leveraged to find the shortest path between two nodes. Then, the path is smoothed to look more realistic [Brand, 2009].

Using Navigation Abilities With a NavMesh. As alluded to in the introduction, it is increasingly common in modern games to offer additional navigation options to the players that enable the full use of the 3D space. For example, by using grappling hooks, jump-pads, jetpacks, teleportation, or other *navigation abilities*, a player can very quickly navigate a map. These abilities add versatility to the players' gameplay but come at the cost of an increase in the number of feasible paths, making the NavMesh increasingly expensive to search through and labor intensive to create [Van Waveren, 2001; Van Toll *et al.*, 2011]. Specifically, in order to allow NPCs to use navigation abilities, the prevalent solution is to add an edge, called *link*, connecting the nodes at both ends of the use of the ability. An animation is then played on the character as it goes through the link to give the illusion of using the ability.

A visual illustration of the NavMesh-based approach can be found in Figures 1 and 2. We used Unity [Juliani *et al.*, 2018] to build a minimal map, called *Toy Map*. We then used Unity's built-in generation tool to create a NavMesh from the map geometry (Figure 1). We added a small set of complex navigation abilities (jumps, jump pads, and double jumps), and added links so that the NavMesh can take these actions into account (see Figure 2).

The search for all possible links that could be added to the graph can sometimes be done automatically. In the case of a jump, for example, possible jump trajectories are simulated and links are added between nodes that can be reached [Axelrod, 2008; Roumimper, 2017; Budde, 2013]. However, as this automatic search relies on connecting all the nodes that can be at the start and end of a movement trajectory, only the simplest trajectories can be used so that the search is tractable and the number of edges added is reasonable.

In practice, having navigation abilities makes the number of links and the graph connectivity increase dramatically and results in extremely long computation time [Axelrod, 2008] with many redundant links [Budde, 2013]. In such cases, adding a limited number of links manually is the favored option. However, not adding enough links can result in unrealistic behaviour, e.g., many NPCs aggregating in navigation bottlenecks around the map to use the specifically hand-designed way-points. As adding more links improves the realism of AI behaviors but increases the labor and runtime costs of the NavMesh, a compromise has to be made.

Another fundamental issue with links is that they do not

correspond to a topological reality. For example, if a link to allow NPCs to climb a ladder is added, it is hard to interrupt the character once it starts climbing. Once on the link, the character is no longer moving in a traversable area, as the link does not correspond to a topological reality. Interrupting the character, e.g., by making it fall off the ladder, would require additional work to make it remain on the NavMesh.

Thus, existing solutions to support navigation abilities are far from perfect. Supporting navigation abilities comes at expensive labor and runtime costs, and fundamental flaws limit the realism of the obtained behaviors. In the case of our toy map, we manually placed a small number of additional links[2] to bridge disconnected parts that should be accessible by using the jump, the double jump or the jump pads (Figure 2). While we are definitely not experts, adding links on such a small map still took us a few hours. Even if imperfect, this minimal example helps to illustrate the main limitations of the NavMesh and motivates our use of Deep RL.

2.2 Navigation in Robotics and RL

SLAM. The classical approach to navigation in robotics is called *simultaneous localization and mapping* (SLAM) [Leonard and Durrant-Whyte, 1991], which builds a high-level map (usually a top-down view) of the world from experience and locates the agent within this map. The high-level map is generated using data from sensors, such as LIDARs or RGB cameras. Once the map has been built, classical path-finding algorithms can be used to plan and extract the shortest paths between any two points. Recently, several works have extended the SLAM framework to learn the high-level mapping using differentiable neural networks [Zhang *et al.*, 2017; Beeching *et al.*, 2020] with some approaches even having success in video games [Bhatti *et al.*, 2016]. Similar to the NavMesh, SLAM-based approaches struggle to integrate navigation abilities in the mapping. Moreover, in the case of video games, we already have exact localization and mapping and thus do not need to use an estimate.

Model-Based RL. Alternatively, in *model-based* RL, a model of the *transition dynamics*, i.e., a mapping from the current state and action of the agent to its next state, can be used for planning [Sutton, 1991]. The model of the transition dynamics can either be estimated from interacting with the environment [Kaiser *et al.*, 2019], or in some cases be given to the agent beforehand [Silver *et al.*, 2016]. In the case of video games, a model of the world can be used if we can manipulate the underlying game engine, in the sense that actions can be undone and different actions can be tried.

Like SLAM and NavMesh-based approaches, model-based RL approaches could theoretically handle complex navigation actions and potentially be used to plan shortest paths. However, there are two notable drawbacks to using model-based approaches for planning. Firstly, like SLAM and NavMesh-based approaches, they are expensive to run at inference/mapping time as they need to compute many forward passes of the model to determine the best path using the complex navigation abilities. Secondly, when the model is estimated from data, they tend to suffer from compounding error due to model imperfections, which makes planning challenging [Feinberg *et al.*, 2018].

Model-Free RL. *Model-free* RL does not use a model to plan but learns which actions to take in the environment through pure trial and error [Sutton and Barto, 2018]. Previous works have used model-free RL for navigation [Mirowski *et al.*, 2016; Wijmans *et al.*, 2019] but have been mostly limited to relatively small 2D environments with simple action spaces. Recent works have circumvented the lack of planning in model-free RL by using a hierarchical architecture, where intermediate goals are given to a controller by a high level planner [Eysenbach *et al.*, 2019]. As navigation in a visually complex environment is usually modeled as a *partially observable markov decision process*, the importance of using memory has been previously acknowledged [Mirowski *et al.*, 2016]. While unstructured memory such as LSTMs [Hochreiter and Schmidhuber, 1997] can be used, architectures involving spatially structured memory have also been explored [Parisotto and Salakhutdinov, 2017; Beeching *et al.*, 2020]. The use of auxiliary tasks to accelerate the learning of challenging goal-based RL problems has also been studied [Mirowski *et al.*, 2016; Andrychowicz *et al.*, 2017].

3 Approach

The following section describes our approach to solve point-to-point navigation on a fixed 3D map using navigation abilities available to the agent.

States. The state is composed of local perception in the form of a 3D occupancy map and a 2D depth map, as well as scalar information about physical attributes of the agent and its goal (velocity, relative goal position, absolute goal position, and previous action). The 3D occupancy map, referred to as *BoxCasts*, can be generated and cached offline. This makes it efficient to compute at runtime which is critical when running inside a video game engine. 2D depth maps have been used in several recent works [Manolis Savva *et al.*, 2019; Wijmans *et al.*, 2019]. They can be extracted from a rendering camera, or in our case by casting *RayCasts*. The absolute positions of the agent and its goal pass through their own network to extract an absolute position embedding similar to [Wijmans *et al.*, 2019].

Actions. The actions are continuous values $\in [-1, 1]$, and correspond to *jump, forward, strafe, rotate*. The jump is treated as a continuous action on the algorithmic side and binarized in the environment, i.e., jump if > 0.

Rewards. To avoid complications associated with long term credit assignment when using a sparse reward, we densify the reward signal to be:

$$\max\left(\min_{i\in[|0,t-1|]}(D_i(agent, goal)) - D_t(agent, goal), 0\right) - \alpha + \mathbb{1}_{D_t(agent,goal)\leq\epsilon}, \quad (1)$$

where D_t is the Euclidean distance between the positions of its arguments at time t, α is a penalty given at each step and ϵ is the distance below which the agent is considered to have

[2]The exact number of additional links is 54.

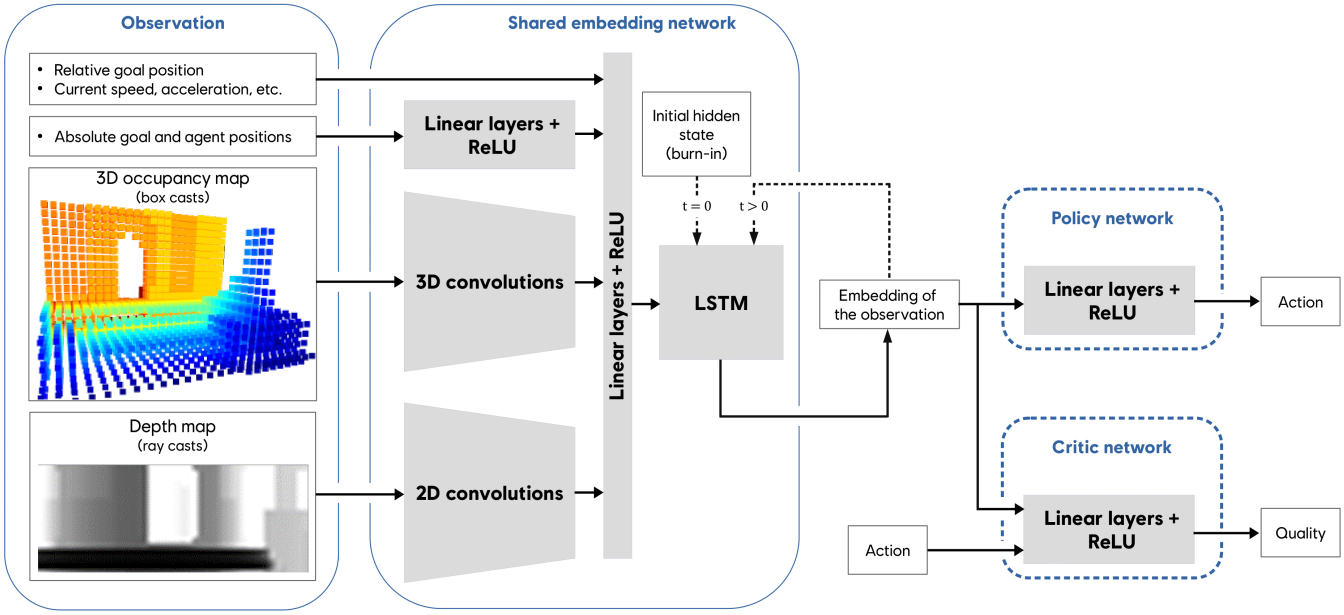

Figure 3: Architecture of our system. The 3D occupancy map, 2D depth map, and absolute goal and agent positions, pass through independent feature extraction layers (3D convolutions, 2D convolutions, and linear layers respectively). The output of each feature extractor is then combined with other state variables, such as relative goal position, speed, acceleration, and previous action. The combined output is fed through several linear layers, followed by an LSTM, and shared by both the policy and critic heads (trained using the critic loss).

reached its goal. We note that the first term rewards the agent for getting closer to its goal than it has ever been so far in this episode. While this reward is non-markovian, since it depends on all previous states visited in the episode, we use an LSTM [Hochreiter and Schmidhuber, 1997] to capture a compressed history of the trajectory.

Training Procedure. As running a game engine is costly, we use Soft Actor-Critic (SAC) [Haarnoja *et al.*, 2018a], a sample efficient off-policy RL algorithm. As described in [Haarnoja *et al.*, 2018b], we modify SAC so that the entropy coefficient is learned and there is no state value network. The critic and policy networks share layers that are tasked with extracting an embedding from local perception as well as previous steps by using Convolutional Neural Networks [LeCun *et al.*, 1989] and an LSTM [Hochreiter and Schmidhuber, 1997]. Furthermore, we use a burn-in to initialize the hidden states when sampling from the replay buffer [Kapturowski *et al.*, 2018; Paine *et al.*, 2019].

During training, we spawn the agent and its goal in a cylinder with a variable radius. An episode is considered over when the agent has reached its goal or when the number of steps is over a certain budget. To allow the agent to have informative trajectories at all stages during training, we use a training curriculum [Bengio *et al.*, 2009] and increase the radius of the spawning cylinder until the full map is covered. Specifically, when the agent achieves a success rate of $> 80\%$ over the last 200 episodes we increase the spawning radius of the goal. All the networks are trained using the Adam optimizer [Kingma and Ba, 2014]. More details on our hyperparameters can be found in the supplementary material, and Figure 3 describes our architecture.

4 Experiments

To highlight the capabilities of our Deep RL system, we train an agent as described in Section 3 on both a Toy Map (see supplemental) and a Big Map (see Figure 4), which we built using the Unity game engine [Juliani *et al.*, 2018]. Both of these environments have many jump pads scattered across the map and allow for the agent to perform double jumps. In order to determine the factors that lead to its success, we run state-based and algorithmic ablations. For the state-based ablation, we evaluate our agent without BoxCasts, without RayCasts, without BoxCasts and RayCasts, and without absolute positions. For the algorithmic ablation, we test removing the training curriculum, the LSTM, and using Hindsight Experience Replay (HER) [Andrychowicz *et al.*, 2017].

Ablation Results. The results for both ablation studies can be found in Figure 5 and in the supplemental. Concerning the state ablation, we find that, on both the Toy Map and the Big Map, the Base agent is significantly faster (in terms of samples) at reaching the target success rate on the final curriculum level than without BoxCasts, without RayCasts, and without BoxCasts and RayCasts (see supplemental for more details). Furthermore, we find that removing the absolute position from the agent's state does not negatively impact performance. In fact, the agent performs slightly better on average (over 5 seeds) which we find to be statistically significant ($p < 0.05$) on the Big Map but not on the Toy Map.

Moreover, we find that by removing both BoxCasts and RayCasts, the agent fails to reach the final curriculum level in the training period (see Figure 5) on both the Toy Map and the Big Map. As this agent is training without local perception, it is perhaps unsurprising that it is less sample efficient. How-

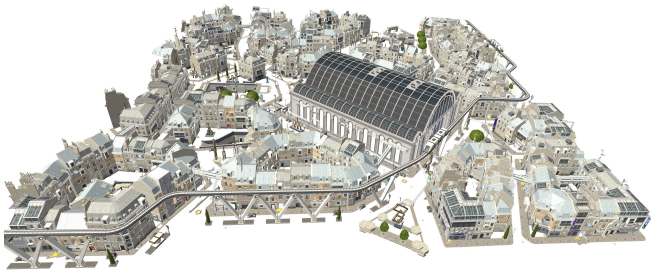

Figure 4: Overview of the Big Map in Unity. The map is $300m \times 300m \times 100m$, with many buildings and jump pads.

ever, we do note that since the performance of the agent is still increasing (albeit slowly, see supplemental), given more training samples it may attain the target success rate (100% on the Toy Map and 90% on the Big Map) on the final curriculum level. To verify this, we ran that configuration for $100M$ steps on the Big Map and confirmed that it could not reach the target success rate of 90% (see supplemental), but plateaus at $\sim 80\%$ success rate on the final curriculum level.

In terms of the algorithmic ablation, none of the algorithmic changes make a significant difference on the Toy Map (see supplemental). On the other hand, on the Big Map we find that removing the LSTM significantly hurts sample efficiency. We hypothesize that the discrepancy between the Toy Map and Big Map with regards to the LSTM can be explained by the simplicity of the Toy map, with the local perception being sufficient to accurately represent the space. We also find that removing the training curriculum improves performance significantly on the Big Map, whereas it does not have a significant impact on the Toy Map. Upon further inspection, we found that we increased the curriculum radius by 5 meters at every iteration on the Big Map and by 10 meters on the Toy Map. A hyperparameter search would need to be done in order to find the optimal setting. However, we note that the strong performance without a curriculum on the Big Map indicates that the dense reward signal that we use is sufficient for the agent to learn efficiently without the need for a curriculum. Finally, we find that performance does not increase significantly with the use of HER on the Toy Map but it does (insignificantly) on the Big Map. Since HER has been shown to be effective in sparse reward regimes, the minimal gains could be attributed to the dense reward signal.

Deep RL vs. NavMesh. To provide a comparison to the NavMesh, we made a video[3] that shows how the two systems behave on the Toy Map. We note that without additional links (which are manually intensive to create), the NavMesh based solution would not be able to navigate to the rooftops. Thus, whereas our Deep RL approach achieves a 100% success rate on this map, the NavMesh based approach without additional links would have a significantly lower success rate. As explained in Section 2.1, such a comparison is imperfect as the NavMesh-based navigation could be made better by spending more time adding links and animating characters on link traversals. Nevertheless, we believe that this visual comparison provides interesting insights to understand how our approach relates to classical solutions for navigation.

Results in Hyper Scape. To validate that our system can scale beyond toy maps to the needs of AAA video game productions, we integrated our solution in the engine of a recently released game called Hyper Scape. A video of our results in the game is available[4], in which we can see the agent navigate around the map by moving sticks on a virtual controller. Apparent in the video is that the agent jumps frequently as it navigates. Upon further inspection we discovered that jumping is slightly faster than walking. We found that this can be alleviated by adding a sprinting action (common in games). Nevertheless, the agent is able to navigate successfully on complex maps with a size of $400m \times 400m \times 90m$, reaching a 90% success rate. We also trained the agent on the full map ($1000m \times 1000m \times 90m$) on which it reached 74% success rate. Success rates and performance benchmarks are provided in the supplemental.

5 Discussion

The results of our ablation study indicate that local perception (through both RayCasts and BoxCasts) drastically improves the sample efficiency of the Deep RL system. We also found that the LSTM was crucial for optimal sample efficiency on large maps. To further improve sample efficiency, auxiliary tasks [Jaderberg *et al.*, 2016] could be used to train the representation layers more efficiently.

All of the experiments conducted in this paper were concerned with the sample efficiency of the Deep RL agent on a single map. We emphasize that sample efficiency is indeed important when faced with a *slow* simulator, which is the case with AAA games. This can be alleviated by implementing the game engine into a binary that can be run much faster than real-time, however, this may not be feasible due to engineering complications. Sample efficient solutions, however, often come at the cost of increased inference time. Our results in the AAA game suggest that Reinforcement Learning is a viable alternative to NavMesh-based navigation.

Finally, while not addressed in this paper, generalization is an interesting direction of future work. Typically the agent is tasked with navigation within a game with dynamic components, e.g., players or objects. As for the NavMesh, it is usually prebuilt to optimize for runtime efficiency and thus cannot easily handle dynamic components [McAnlis, 2008]. Instead, NavMeshes are either updated dynamically at runtime [Marden, 2008] or navigate around non-static objects by using heuristics [Brand, 2009]. Thus, a generalizable Deep RL agent may be key to adapt to dynamic components that may not have appeared during training. Generalization could also enable the reusability of the trained system on novel maps. This can have drastic cost saving impacts when the number of maps becomes large, e.g. when procedurally generated, where retraining the agent is not feasible.

In this paper, we showed that Deep RL can be an alternative to the NavMesh for navigation in complicated 3D maps, such as the ones found in AAA video games. In comparison with

[3]Video comparison with a NavMesh: youtu.be/WFIf9Wwlq8M

[4]Hyper Scape video: youtu.be/DKdQFajLfzk

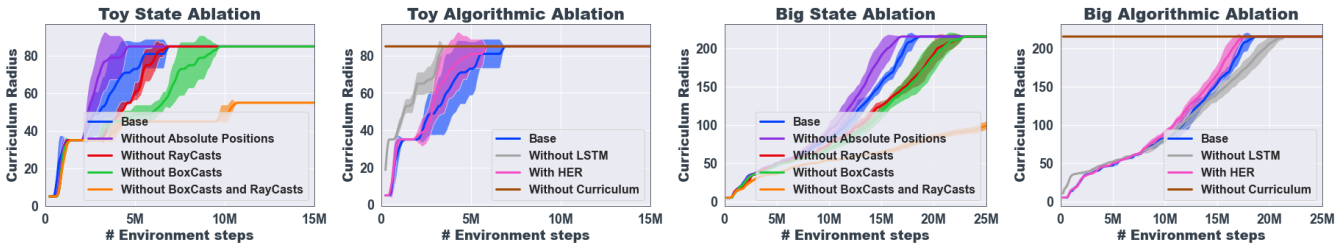

Figure 5: Results from the state ablation and algorithmic ablation on the Toy Map (left and center left) and the Big Map (center right and right). Each curve represents the mean over 5 seeds with 95% confidence intervals in the shaded regions.

previous works exploring Deep RL for navigation, our Big Map and AAA Game map are an order of magnitude bigger [Manolis Savva *et al.*, 2019; Wydmuch *et al.*, 2018]. Unlike the NavMesh, the Deep RL system is able to handle navigation actions without manually specifying individual links. We believe that our work can be used as a stepping stone for future Deep RL applications inside modern video games.

Acknowledgments

We would like to thank Julien L'Heureux, Claudine Combe, Nicolas Landron, Bérenger Bailly, Mike Yurick, Philippe Marcotte and Adrien Logut for many engineering contributions and thorough feedback during the development of this project. We also wish to thank Pierre Falticska, Olivier Pomarez, Paul Barde and Julien Roy for insightful discussions and their feedback on an earlier draft of this paper. Finally, we thank Olivier Delalleau, Batu Aytemiz and Sahand Rezaei-Shoshtari for contributions to an early iteration of this project.

References

[Andrychowicz *et al.*, 2017] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, 2017.

[Axelrod, 2008] Ramon Axelrod. *Navigating graph generation in highly dynamic worlds*, volume 4 of *AI Game Programming Wisdom*, chapter 2.6, pages 125–141. Charles River Media, 2008.

[Beeching *et al.*, 2020] Edward Beeching, Christian Wolf, Jilles Dibangoye, and Olivier Simonin. Egomap: Projective mapping and structured egocentric memory for deep RL. *arXiv preprint arXiv:2002.02286*, 2020.

[Bengio *et al.*, 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[Bhatti *et al.*, 2016] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, and Philip HS Torr. Playing Doom with slam-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*, 2016.

[Brand, 2009] S Brand. Efficient obstacle avoidance using autonomously generated navigation meshes. Master's thesis, Delft University of Technology, 2009.

[Budde, 2013] Sara Budde. Automatic generation of jump links in arbitrary 3d environments. Master's thesis, Humboldt University of Berlin, 2013.

[Eysenbach *et al.*, 2019] Ben Eysenbach, Russ R. Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

[Feinberg *et al.*, 2018] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018.

[Haarnoja *et al.*, 2018a] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[Haarnoja *et al.*, 2018b] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Jaderberg *et al.*, 2016] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[Juliani *et al.*, 2018] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[Kaiser *et al.*, 2019] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H.

Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for Atari. *arXiv preprint arXiv:1903.00374*, 2019.

[Kapturowski *et al.*, 2018] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[Kingma and Ba, 2014] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[LeCun *et al.*, 1989] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[Leonard and Durrant-Whyte, 1991] John J Leonard and Hugh F Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *IROS*, volume 3, pages 1442–1447, 1991.

[Lidén and others, 2002] Lars Lidén et al. Strategic and tactical reasoning with waypoints. In *AI Game Programming Wisdom, Charles River Media*. Citeseer, 2002.

[Manolis Savva *et al.*, 2019] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[Marden, 2008] Paul Marden. *Dynamically updating a Navigation Mesh via Efficient Polygon Subdivision*, volume 4 of *AI Game Programming Wisdom*, chapter 2.3, pages 83–94. Charles River Media, 2008.

[McAnlis, 2008] Colt McAnlis. *Intrinsic Detail in Navigation Mesh Generation*, volume 4 of *AI Game Programming Wisdom*, chapter 2.4, pages 95–112. Charles River Media, 2008.

[Mirowski *et al.*, 2016] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

[Oliva and Pelechano, 2011] Ramon Oliva and Nuria Pelechano. Automatic generation of suboptimal navmeshes. In *International Conference on Motion in Games*, pages 328–339. Springer, 2011.

[Paine *et al.*, 2019] Tom Le Paine, Caglar Gulcehre, Bobak Shahriari, Misha Denil, Matt Hoffman, Hubert Soyer, Richard Tanburn, Steven Kapturowski, Neil Rabinowitz, Duncan Williams, et al. Making efficient use of demonstrations to solve hard exploration problems. *arXiv preprint arXiv:1909.01387*, 2019.

[Parisotto and Salakhutdinov, 2017] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017.

[Roumimper, 2017] Nick Roumimper. Mesh navigation through jumping. Master's thesis, Utrecht University, 2017.

[Silver *et al.*, 2016] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[Snook, 2000] Greg Snook. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. *Introduction to reinforcement learning*. MIT press Cambridge, second edition, 2018.

[Sutton, 1991] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[Tozour and Austin, 2002] Paul Tozour and IS Austin. Building a near-optimal navigation mesh. *AI game programming wisdom*, 1:298–304, 2002.

[Van Toll *et al.*, 2011] Wouter Van Toll, Atlas F. Cook, and Roland Geraerts. Navigation meshes for realistic multi-layered environments. In *2011 International Conference on Intelligent Robots and Systems*, 2011.

[Van Toll *et al.*, 2016] Wouter Van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*, pages 91–100, 2016.

[Van Waveren, 2001] JMP Van Waveren. The Quake III arena bot. Master's thesis, University of Technology Delft, 2001.

[Wijmans *et al.*, 2019] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv preprint arXiv:1911.00357*, 2019.

[Wydmuch *et al.*, 2018] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. ViZDoom competitions: Playing Doom from pixels. *arXiv preprint arXiv:1809.03470*, 2018.

[Zhang *et al.*, 2017] Jingwei Zhang, Lei Tai, Joschka Boedecker, Wolfram Burgard, and Ming Liu. Neural Slam: Learning to explore with external memory. *arXiv preprint arXiv:1706.09520*, 2017.