Deep Reinforcement Learning for Non-Player Character Navigation in Open-World Games

Zerui Lyu
lv000013@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

The incorporation of Deep Reinforcement Learning (DRL) into Non-Player Character (NPC) navigation has changed how NPCs navigate in a complex game environment. DRL combines classic reinforcement learning with the use of neural networks.. Gomes, Vidal, et. al [4] explored the effectiveness of this approach using four different neural network (NN) architectures in a DRL system utilizing the Advantage Actor-Critic (A3C) algorithm . This paper explores the mechanisms of DRL, providing background for the neural network architectures used by Gomes, Videl et.al and describes the important details of the A3C algorithm which was used to train the NPCs in their study. Their result shows that the DRL system is efficient for the NPC navigation and that the simplest NN architecture was the most effective.

Keywords: Deep Reinforcement Learning (DRL), Feedforward, Long-Short Term Memory(LSTM), Actor-Critic (A3C) algorithm, reward functions

1 Introduction

Non-Player Characters are integral to open-world games, enriching the player's experience through interaction. Traditionally, NPC navigation relied on Waypoint Graphs (a method of connecting predefined points to form a navigable path)[14] and NavMeshes which is a collection of two-dimensional convex polygons (a polygon mesh) that define which areas of an environment are traversable by agents[2]. While these approaches were once effective for simpler game environments, the increasing complexity of modern game worlds, with their intricate terrains, dynamic obstacles, and complicated gameplay, has exposed the limitations of these traditional systems. In response to these limitations, Deep Reinforcement Learning (DRL) has emerged, allowing NPCs to learn and adapt in complex game environments. The authors, Gilzamir Gomes, Creto A. Vidal, Joaquim B. Cavalcante-Neto, and Yuri L. B. Nogueira[4], applied deep reinforcement learning techniques to train NPCs for navigation. This paper delves into the mechanisms of Deep Reinforcement Learning, with a focus on the use of two configurations of the deeper layers: feedforward and Long-Short Term Memory, reward function and A3C algorithm. This paper is organized as follows: in Section 2 , we will discuss a detailed background on traditional NPC navigation methods, neural networks, deep learning, reinforcement learning . In Section 3, the paper introduces the methods being used for training the NPC, such as configurations of the deeper layers, the Asynchronous Advantage Actor-Critic algorithm and the reward function. Section 4 explores case studies on different neural network configurations. Finally, in Section 5, we will compare both the training phase and the testing phase results among different neural network configurations and discuss the efficiency of the DRL system.

2 Background

Understanding the core technologies behind Deep Reinforcement Learning (DRL) is essential to appreciate its impact on NPC navigation in open-world games. This section provides an overview of neural networks, reinforcement learning principles, and how adaptive learning processes occur within dynamic virtual environments. Furthermore, it examines the evolution of traditional NPC navigation systems, such as Waypoint Graphs and NavMeshes, outlining their operational mechanisms and limitations compared to DRL-based approaches.

2.1 NPC Navigations Methods in the past

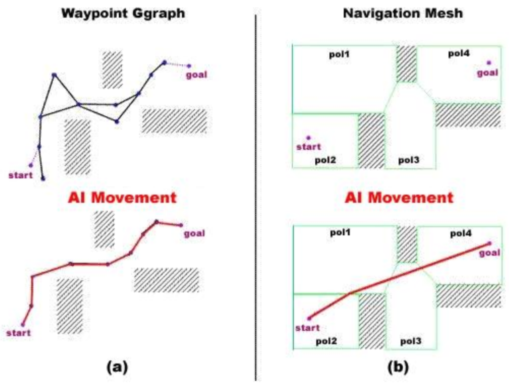

Figure 1. Waypoint Graph and NavMesh [11]

Two main methods have traditionally been used for NPC navigation [4]. The first is the Waypoint Graph, where developers manually place points on the map and connect them.

This method can cause navigation issues, especially if human errors occur during point placement[4]. The second, NavMesh, divides the map into convex regions, allowing easier navigation. However, NavMesh struggles with dynamic environments, as changes like destructible objects require updates to the mesh[4]. Due to these limitations, both methods have become less suitable for complex and dynamic game environments.

2.2 Neural Networks

A neural network is an approach within the field of artificial intelligence, used to teach computers to process data in a way that mimics the human brain [3]. A neural network consists of a set of interconnected layers. Layers consist of a sequence of numeric values, graphically represented by circle known as nodes or neurons.

1. **Input Layer**: This layer receives raw data and passes it to the next layers in the network. Each input neuron represents a feature or element of the data.
2. **Hidden Layers**: These intermediate layers apply complex transformations to the data using non-linear functions.
3. **Output Layer**: This final layer processes the transformed data from the hidden layers and produces the network's prediction or classification result.

A Feedforward Neural Network (FNN) is a type of artificial neural network where information moves only in one direction, from the input layer through any hidden layers and finally to the output layer [9]. Figure 2 illustrates a feedforward neural network specifically designed to map environmental data to potential NPC actions. The network consists of an input layer with four features, a hidden layer, and an output layer.

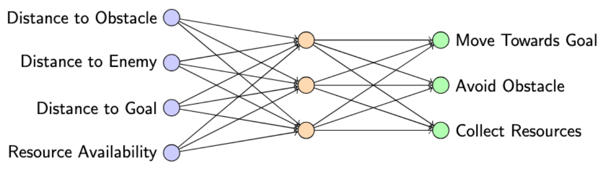

Figure 2. Illustration of a feedforward neural network with input, hidden, and output layers (The input layer on the left, the hidden layers in the middle, and the output layer on the right).

Figure 2 illustrates how data flows through the network:

- **Input Layer:** This layer takes in environmental data, including "Distance to Obstacle," "Distance to Enemy," "Distance to Goal," and "Resource Availability." Each input node corresponds to one of these features, forming the initial layer of the network.
- **Hidden Layers:** Hidden layers process inputs by applying weights, which represent the importance of connections between neurons and influence how information flows through the network.
- **Output Layer:** The output layer contains three nodes representing potential actions: "Move Towards Goal," "Avoid Obstacle," and "Collect Resources." The network outputs a probability distribution of actions known as a policy.

Activation Functions. Activation functions decide if a neuron should pass its signal forward, helping the model handle complex patterns.

- **ReLU (Rectified Linear Unit):** An activation function used in hidden layers that outputs the input if it's positive or zero if it's negative. In NPC navigation, ReLU helps the model quickly learn features like detecting obstacles or targets by focusing on important signals.
- **Softmax:** A function applied in the output layer to convert raw scores into probabilities. For example, in NPC navigation, Softmax can assign probabilities to actions like "move forward" (70%), "turn left" (20%), or "jump" (10%).

2.2.1 Neural Networks Training.

Training a neural network is the process of using training data to find the appropriate weights of the network for creating a good mapping of inputs and outputs[5]. The following explanation is based on the work presented in[5].

1. **Forward Propagation:** Input data passes through the network layer by layer. Each layer processes the data using weighted connections and activation functions.
2. **Loss Calculation:** The loss function measures the error between the predicted output and the target value.
   - **Predicted Output:** The output generated by the neural network based on its current weights and input data. For example, in NPC navigation, the predicted output would be the action probabilities (e.g., move forward or avoid obstacle).
   - **Target Value:** The desired value that the network aims to match. In DRL systems there are not specific target value for the NN training, instead the interaction of the NPC with its environment creates rewards and valuations that serve the same role. This is discussed in greater detail in Section 3.6.
3. **Backward Propagation:** The error calculated by the loss function is propagated backward through the network to compute gradients, which are measures of how much a weight or bias contributes to the error, with respect to each weight and bias.
4. **Weight Updates:** In training neural networks, the optimization process aims to find a set of weights W that minimize the loss function L, which measures

the error between the predicted and expected outputs. Gradient Descent (GD) is the most widely used optimization algorithm for this purpose. The process involves taking iterative steps in the direction opposite to the gradient of L with respect to W. In the policy network, the weights are updated to maximize an expected reward (Section 3.7.1, bullet point e Policy Update) instead of minimizing a loss function, but the mechanisms are almost identical to GD. In the value network the update rules use a traditional GD approach.

2.3 Recurrent Neural Network

Recurrent neural networks (RNNs) contains layers that store a history of their past values [13]. Their built-in memory enables RNNs to retain information from earlier time steps and apply it in later steps. Unlike feedforward neural networks, which process each input independently, RNNs include loops in their structure, allowing information to be passed forward and built on previous inputs.

Long Short-Term Memory (LSTM) networks, a type of RNN, use gates to selectively retain or discard information, allowing them to remember important patterns over extended sequences [7]. In NPC navigation, LSTMs are used for capturing temporal dependencies, such as tracking the sequence of past movements or predicting future positions based on the previous movements.

2.4 Reinforcement Learning

Reinforcement learning (RL) is an interdisciplinary area of machine learning and optimal control concerned with how an intelligent agent should take actions in a dynamic environment in order to maximize a reward signal[15]. Unlike supervised learning, where the agent learns from labeled data, RL emphasizes decision-making, where the agent must choose a series of actions that affect both its environment and future choices.

As illustrated in Figure 3, the RL model consists of key elements: an agent, a state, an action, an environment, and a policy.

The state (S_t) represents the current situation or condition of the environment as observed by the agent. It encapsulates all the relevant information needed to make a decision. The environment is everything external to the agent that the agent interacts with, and it evolves based on the agent's actions. An action (A_t) is a decision or move made by the agent to influence the environment. A policy is the agent's strategy for choosing actions based on the current state. It can be learned through experience as the agent attempts to maximize cumulative rewards. After each action, the agent receives feedback from the environment in the form of a reward (R_t), indicating how successful the action was in achieving the desired goal, such as reaching a target or avoiding obstacles in NPC navigation.

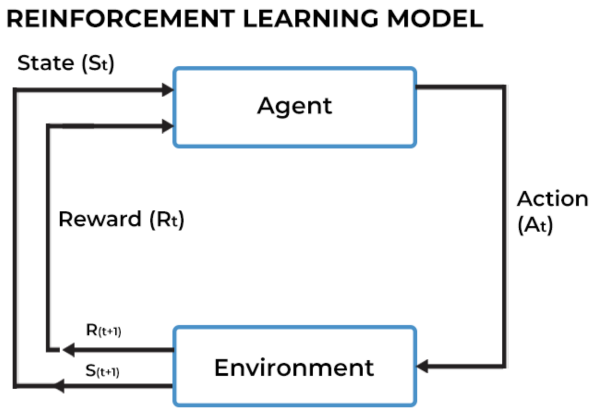

Figure 3. A Visual Representation of the Reinforcement Learning Model [12]

2.5 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines the strengths of neural networks with the decision-making process of reinforcement learning[10]. Unlike traditional supervised learning, DRL operates through trial and error, where an agent interacts with the environment to learn optimal actions.

Deep reinforcement learning (Section 3) involves the use of two neural networks. One is used for developing an effective policy (the policy network) for driving NPC actions. The other is used to assess the expected value accessible to the NPC (the value network) based upon the current state of the game system. The authors of the paper used a single neural network with two output layers (See Section 3.3) to perform both calculations, but it's easier to understand the process if the two NN are treated as being separate.

In NPC navigation, DRL enables agents to navigate complex environments by learning from rewards and penalties. For instance, an NPC learns to avoid obstacles, and reach a target location by maximizing cumulative rewards through repeated interactions with the game world. The Section 3.6.1 introduces how rewards influence the NPC's behaviors.

3 Method

3.1 Linear and Visual Data Collection

The NPC collects two types of inputs: **linear data** and **2D visual data**. A linear sensor gathers spatial and directional information, including:

- **Distance Vector** d_t: Unit vector pointing from the agent to the target.
- **Orientation** δ_t: Alignment between the agent's view and target direction.
- **Contact Status** S_t: Indicates if the agent is grounded or airborne.
- **Touch Signal** T_t: Signals contact with walls or the target.

The 2D visual sensor generates a 30×30 matrix at each timestep using ray-casting to capture the agent's immediate visual field.

3.2 Features Extraction

After collecting the linear data and the visual data, feature extractors need be used to process the raw data and simplify the complexity of the input data. There are two feature extractors in the neural network: a convolutional bidimensional feature extractor (a neural network layer that processes 2D data by applying filters to detect patterns) for processing the visual data, and a linear feature extractor for processing the input data. After the features extraction stage, The input features are concatenated to be processed by the deeper layers (See Figure 4)[4].

3.3 Two Configurations of the Deeper Layers

After feature extraction, the processed data feeds into deeper layers for advanced processing. These deeper layers use two distinct configurations designed to handle different levels of environmental complexity. Each configuration produces two outputs:

1. **Value (1 Linear Unit):** Represents the expected reward for the current state. This helps the agent evaluate the desirability of a state and make decisions that maximize cumulative rewards.
2. **Actions (6 Softmax Units):** Represents the probabilities of six possible actions to guide the agent's decisions. The six actions are: **Forward**, **Backward**, **Turn Left**, **Turn Right**, **Jump**, **Jump Forward**

3.4 Feedforward Configuration (Base Model)

The simpler configuration, shown in Fig. 4, employs traditional fully connected (dense) layers to process the extracted features and directly produce the two outputs (value and actions).

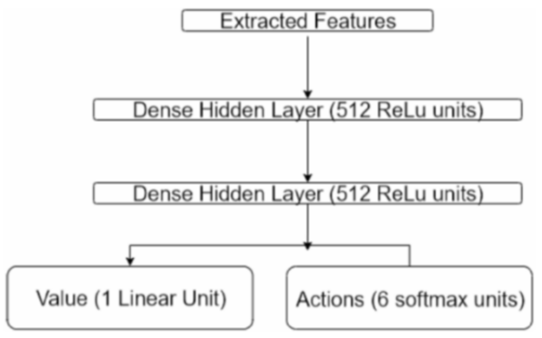
Extracted Features
Dense Hidden Layer (512 ReLu units)
Dense Hidden Layer (512 ReLu units)
Value (1 Linear Unit)
Actions (6 softmax units)

Figure 4. Feedforward Neural Network Configuration[4]

3.5 LSTM Configuration (Sequential Memory Model)

The more advanced configuration, shown in Fig. 5, incorporates two stacked Long Short-Term Memory (LSTM) layers.

- **Key Features:**
  - **Sequential Memory:** Retains relevant past information, helping the agent recognize patterns over time.
  - **Dynamic Adaptability:** Adapts to changes in the environment, such as obstacles or target shifts.
- **Applications:** The LSTM model has been shown to improve performance in complex, dynamic environments like NPC navigation tasks [7].

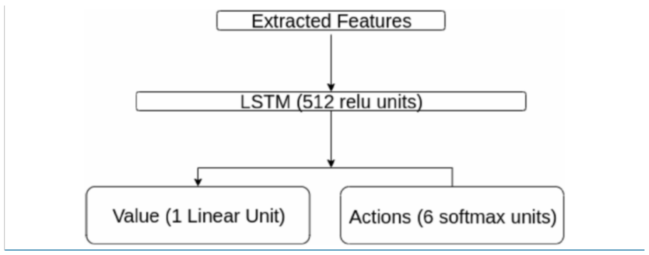

Figure 5. LSTM Neural Network Configuration [4]

3.6 Agent's Training Methodology and Reinforcement Process

The NPC in this study is trained using the **Asynchronous Advantage Actor-Critic (A3C)** algorithm [8]. The A3C algorithm's structure consists of two key components: the **Actor** and the **Critic** networks. Together, these networks allow the NPC to optimize navigation by iteratively adjusting weights based on rewards, reinforcing actions that lead to successful outcomes.

3.6.1 Reward Function for Optimizing Navigation. The reward function provides feedback to the NPC's learning process, encouraging efficient navigation behavior while penalizing inefficient movements. Inspired by prior work [6], the reward function at each time step, R_t, is designed to balance progress toward the goal, penalize inefficiency, and reward goal completion. The function is defined as:

$$R_t = \max\left(\min_{\forall i \in [0,t-1]} E(t,i), 0\right) - \alpha + 100 \cdot \text{touch}(agent, goal), \tag{1}$$

where:

- $E(t,i) = \text{dist}_i(agent, goal) - \text{dist}_t(agent, goal)$:
  - Measures the change in distance to the goal, from a previous time step i to the current time step t.
  - A positive $E(t,i)$ indicates the agent has moved closer to the goal.
  - A negative $E(t,i)$ indicates the agent has moved further away.
- α: Represents a penalty for each action taken, discouraging unnecessary or inefficient movements.

- $100 \cdot \text{touch}(agent, goal)$: Grants a reward of 100 if the agent successfully reaches the goal, reinforcing goal completion behavior.

3.7 Two-Stage Training Process

The training process of the NPC is divided into two stages of increasing complexity, each with a specific success rate requirement. Success rate is defined as the percentage of episodes in which the NPC successfully reaches its target within a set distance and action limit. The agent successfully learned to navigate a scene with ramps, walls, and stairs, avoiding obstacles and efficiently reaching its target despite potential wandering in open spaces [4].

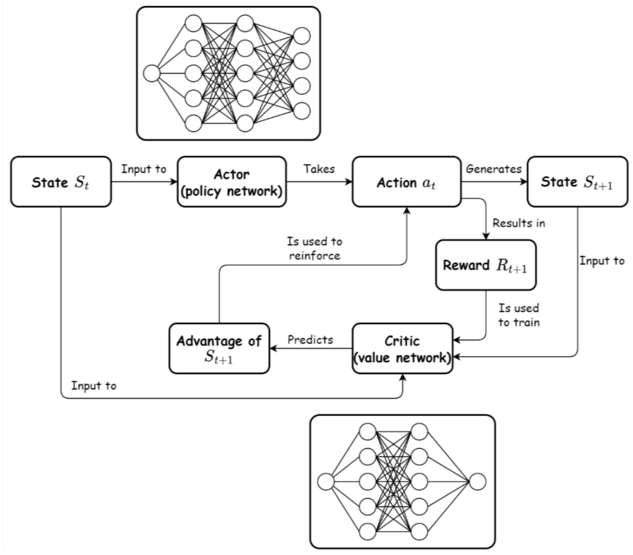

Figure 6. Actor Network and Critic Network [1]

3.7.1 Easy Navigation Training. Setup: In the initial stage, the target is positioned within 4.5 units of distance from the NPC in a simplified, obstacle-free environment. This scenario enables the NPC to quickly learn basic navigation skills. Training continues until the NPC achieves a success rate of 50%, meaning it successfully reaches the target in at least half of the episodes. An episode refers to a single trial or attempt where the NPC starts from a specific position and tries to reach the target within the defined environment.

Training Process:

1. **Initial State and Actor's Decision:** The NPC observes its surroundings and inputs its current state (S_t)—for example, its distance to the target. The Actor (policy network) processes this state and outputs a probability distribution over six possible actions (e.g., "move forward" or "turn left"). The action with the highest probability is selected and executed.
2. **Taking Action and Receiving Reward:** Based on the selected action (A_t), the NPC interacts with the environment. If the action improves its position (e.g., reduces distance to the goal), the environment provides a positive reward (R_t). For example, moving closer to the target might yield $R_t = 2.5$.
3. **Critic's Evaluation:** The Critic (value network) evaluates the new state (S_{t+1}) after the action is taken and predicts its value ($V(S_{t+1})$), which represents the expected future reward from this state. For instance, the Critic might assign $V(S_{t+1}) = 1.0$.
4. **Calculating Advantage:** The advantage ($A(S_t, A_t)$) quantifies how much better or worse the action performed compared to the Critic's prediction:
$$A(S_t, A_t) = R_t - V(S_{t+1})$$
For example, if $R_t = 2.5$ and $V(S_{t+1}) = 1.0$, then $A(S_t, A_t) = 1.5$, indicating that the action was beneficial.
   - A positive advantage ($A > 0$) reinforces the action, meaning it was more effective than expected.
   - A negative advantage ($A < 0$) discourages the action, indicating it was less effective than expected.
5. **Policy Update:** Using the advantage $A(S_t, A_t)$, the weights of Actor network is updated to produce a reliable policy:
$$\Delta\theta = \alpha \cdot A(S_t, A_t) \cdot \nabla_\theta \log \pi_\theta(A_t|S_t)$$
where:
   - $\Delta\theta$: The change in the Actor's weights.
   - α: The learning rate, controlling the update magnitude.
   - $\nabla_\theta \log \pi_\theta(A_t|S_t)$: The gradient of the log-probability of the chosen action.

   For example, if "move forward" yields a positive advantage, the probability of selecting this action in similar states is increased in subsequent iterations.
6. **Critic Update:** Simultaneously, the Critic is trained to minimize the difference between its predicted value ($V(S_{t+1})$) and the actual reward (R_t), using a loss function:
$$L(\theta_c) = \frac{1}{2}\left(R_t - V(S_{t+1})\right)^2$$
This updates the weights of Critic network and it increases Critic's accuracy in estimating state values over time.

Although the authors combined both actor and value network into a single NN, the two different output layers allow the weight updates to occur separately in a way that allows their NN to fulfill both roles. The authors are silent on the advantage gained by having the two networks sharing weights.

3.7.2 Progressive Training for Complexity. Once the NPC achieves a 50% success rate in the easy navigation scenario, the training progresses to more complex environments.

4 Comparison of Configurations

In evaluating the effectiveness of the Deep Reinforcement Learning (DRL) system for NPC navigation, four configurations were tested to assess the impact of different NN architectural choices and state information on the NPC's performance. The details of the four configurations of the policy network are summarized in Table 1 based upon the work presented in [4]. Table 1 summarizes these configurations:

Table 1. Reinforcement Learning System Configurations

Configuration	Architecture	State Information
Base	Feedforward	Includes animation status
VAR1	Feedforward	Excludes animation status
VAR2	LSTM	Includes animation status
VAR3	LSTMs	Includes animation status

Note: Animation status refers to jump status and NPC height.

Each configuration represents a distinct approach:

- **Base Configuration:** A feedforward (FF) neural network incorporating animation status (jump status and NPC height) as part of the state. Serves as a baseline in this experiment.
- **VAR1:** Similar to the base, but excludes animation status information to examine its impact on the NPC's adaptation.
- **VAR2:** Uses a single-layer Long Short-Term Memory (LSTM) network with animation status included, leveraging temporal information to assess the effect of memory on navigation.
- **VAR3:** A two-layer stacked LSTM network (128 neurons per layer) to capture complex temporal dependencies, hypothesizing improved navigation by retaining more historical data.

All configurations were trained for 6,000 episodes using identical initial seeds to ensure fair comparison. This comparison reveals the influence of different neural network architectures and state information on the NPC's ability to navigate effectively using the A3C algorithm.

5 Results and Conclusion

The experimental evaluation demonstrates the effectiveness of the proposed Deep Reinforcement Learning (DRL) system in training Non-Player Characters (NPCs) to navigate complex 3D environments with point-to-point navigation objectives. Figure 7 illustrates the evolution of the success rate, represented using a moving average, for each configuration over the course of 6,000 episodes during training. The base configuration achieved the highest success rate both in terms of stability and final performance. In contrast, VAR1 shows a slower learning rate.

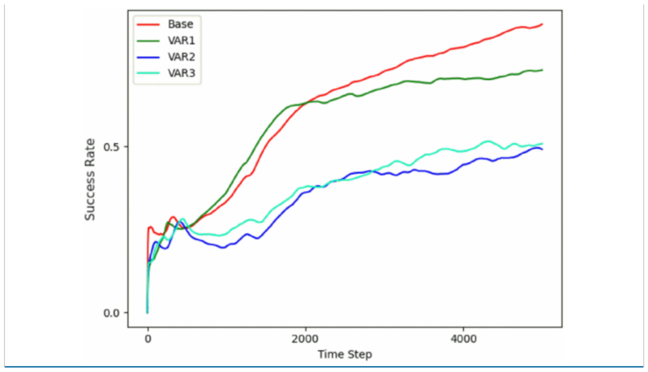

Figure 7. The success rate's moving average of the last hundred episodes during training of the agents [4].

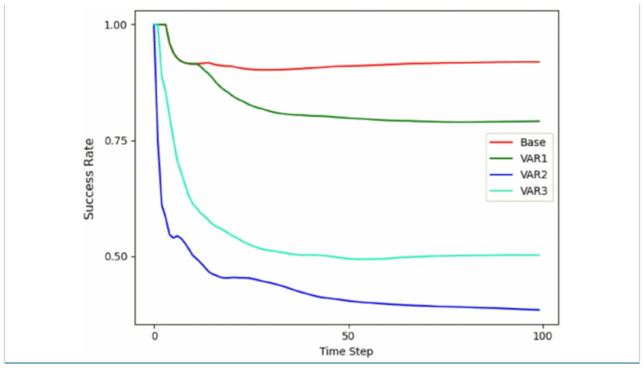

Figure 8. The success rate's moving average during testing of the agents [4].

The configurations utilizing LSTM layers (VAR2 and VAR3) exhibited lower success rates compared to the feedforward configurations. This result aligns with the hypothesis that, for this navigation task, the inclusion of LSTM (which models temporal dependencies) does not significantly enhance performance. The LSTM configurations may have added unnecessary complexity.

During the test phase, it is natural that the success rate at the beginning is high and decreases with time until it stabilizes (see Figure 8), given that the probability of the NPC making a mistake increases with time.

In conclusion, the results show that the feedforward neural network configuration with animation state information (Base) was the most efficient and effective model tested by the authors[4]. The problem's partially observable nature ensures the agent consistently has access to all relevant information, making LSTM memory redundant in this scenario. Moreover, LSTMs may require more training to reach their full potential. Future work could explore scenarios with greater sequential dependencies, where LSTM architectures may provide a stronger advantage.

Acknowledgments

I would like to thank Professor Peter Dolan for invaluable guidance throughout the writing process, and Professor Elena Machkasova for helpful feedback on my paper. Additionally, I'd like to thank Scott Steffes for taking the time to review my paper.

References

[1] 2024. *Actor-Critic Methods.* https://wikidocs.net/175903 Accessed: 2024-11-16.

[2] Wikipedia contributors. 2024. Navigation mesh. https://en.wikipedia.org/wiki/Navigation_mesh. Accessed: 2024-12-02.

[3] Eastgate Software. 2023. *Neural Networks Explained.* https://eastgate-software.com/neural-networks-explained/ Accessed: 2024-10-28.

[4] Gilzamir Gomes, Creto A. Vidal, Joaquim B. Cavalcante-Neto, and Yuri L. B. Nogueira. 2021. Two Level Control of Non-Player Characters for Navigation in 3D Games Scenes: A Deep Reinforcement Learning Approach. In *2021 20th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames).* 182–190. https://doi.org/10.1109/SBGames54170.2021.00030

[5] Shih-Chia Huang and Trung-Hieu Le. 2021. Chapter 2 - Neural networks. In *Principles and Labs for Deep Learning*, Shih-Chia Huang and Trung-Hieu Le (Eds.). Academic Press, 27–55. https://doi.org/10.1016/B978-0-323-90198-7.00006-9

[6] Miguel Liu, Abhinav Gupta, and Saurabh Gupta. 2020. Learning by Cheating: Using Imitation Learning to Train Agents for Tactical Navigation. *arXiv preprint arXiv:2011.04764* (2020).

[7] Wookhee Min, Bradford Mott, Jonathan Rowe, and James Lester. 2017. Deep LSTM-Based Goal Recognition Models for Open-World Digital Games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. WS-17-13. AAAI, 851–858.

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783* (2016).

[9] Sasirekha Rameshkumar. 2023. Deep Learning Basics - Part 10: Feed Forward Neural Networks (FFNN). https://medium.com/@sasirekharameshkumar/deep-learning-basics-part-10-feed-forward-neural-networks-ffnn-93a708f84a31 Accessed: 2024-10-28.

[10] P. V. Rao, V. B., M. Manjeet, A. Kumar, M. Mittal, A. Verma, and D. Dhabliya. 2024. Deep Reinforcement Learning: Bridging the Gap with Neural Networks. *International Journal of Intelligent Systems and Applications in Engineering* 12, 15s (2024), 576–. https://ijisae.org/index.php/IJISAE/article/view/4792

[11] ResearchGate. [n. d.]. Different representations of waypoint graph and NavMesh. https://www.researchgate.net/figure/Different-representations-of-waypoint-graph-and-NavMesh_fig2_303369993. Accessed: 2024-10-28.

[12] Spiceworks. [n. d.]. What Is Reinforcement Learning? https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/. Accessed: 2024-10-28.

[13] Vivek Veeriah, Matteo Hessel, Junhyuk Oh, Hado Van Hasselt, David Silver, Satinder Singh, and Rémi Munos. 2019. Discovery of useful questions as auxiliary tasks. *arXiv preprint arXiv:1912.05911* (2019).

[14] N.M. Wardhana, H. Johan, and H.S. Seah. 2013. Enhanced waypoint graph for surface and volumetric path planning in virtual worlds. *The Visual Computer* 29 (2013), 1051–1062. https://doi.org/10.1007/s00371-013-0837-x

[15] Wikipedia contributors. 2024. Reinforcement learning. https://en.wikipedia.org/wiki/Reinforcement_learning. https://en.wikipedia.org/wiki/Reinforcement_learning Accessed: 2024-10-28.