Università degli Studi di Milano
Dipartimento di Informatica Giovanni Degli Antoni

Natural Interaction and Affective Computing: A NS Agent for Pacman Maze

Stefano Capelli, Matteo Onger

May 7, 2025

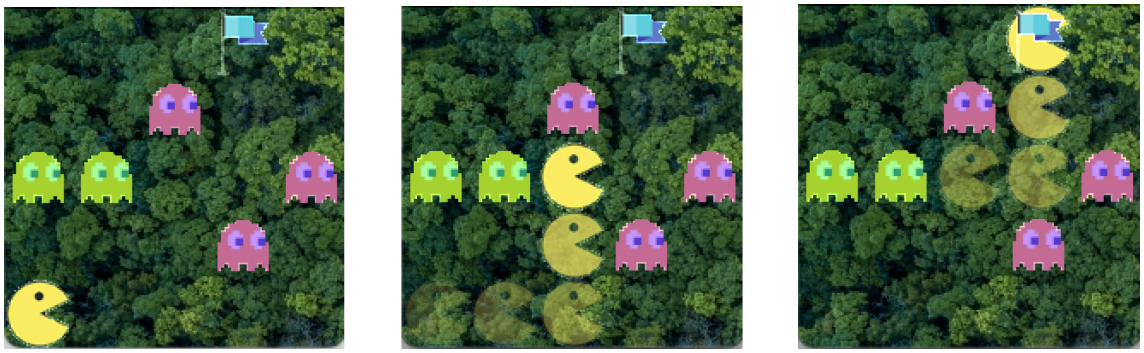
Abstract

This project builds upon an existing Pacman Maze simulation developed by the Scallop team as an official demonstration of their neuro-symbolic programming framework. The original demo combines a convolutional neural network (CNN) for game state perception with a symbolic reasoning component that determines the next agent's action. The aim of this work is to replace the CNN module with a simplified architecture based on Active Inference, while also investigating potential improvements to the symbolic reasoning component to enhance the agent's decision-making capabilities.

1 A Starting Point

1.1 Pacman Maze

The Pacman Maze game [5] is a simplified version of Pac-Man: an agent, positioned within a $M \times N$ grid, must reach a target, represented by a blue flag, in a maximum of T moves and without colliding with one of the K enemies (ghosts). The initial positions of all entities are randomly assigned at the beginning of the game. Thereafter, while the enemy and target positions remain fixed, the agent can perform one of four possible actions (*right*, *up*, *left* and *down*) to move by one cell vertically or horizontally. If the chosen action takes the agent out of the grid, the agent's position remains unchanged. In any case, after each move, the agent receives a reward whose value depends on the state of the cell reached: empty, occupied by an enemy, or containing the target. The game ends when the agent hits an enemy, reaches the goal, or exceeds the maximum number of moves.



(a) Step 0. (b) Step 4. (c) Step 7.

Figure 1: Three states of one game-play session.
Source: [5].

From an implementation perspective, the game was developed in accordance with the de facto standard established by Gymnasium [8]: the class `AvoidingArena` extends the `Env` class and overrides its main methods:

- `reset` sets the environment in an initial state;
- `step` executes a selected action;
- `render` returns a representation of the current state, either as an image or as a string, depending on the selected rendering mode.

1.2 The Agent

The agent is a Q-learning agent: at each time step, it receives as input an RGB image representing the current state of the grid and selects the next action to perform based on the learned policy. As previously mentioned, during the training phase, the agent learns the policy through the Q-learning algorithm, using an ϵ-greedy strategy to balance exploration and exploitation, i.e., at each step, with probability ϵ an action is selected randomly, instead of following the policy learned up to that point, to explore the environment. Typically, the value of ϵ decays with time.

Algorithm 1 Q-learning algorithm

```
Set values for learning rate α and discount rate γ
Initialize Q(s, a) = 0  ∀(s, a)
for all episode ∈ episodes do
    Select state s randomly
    for all step ∈ episode do
        x ← U[0, 1]                                   ▷ sample x uniformly at random
        if x ≤ ε then                                 ▷ use ε-greedy strategy
            Select a randomly
        else
            Select a using policy derived from Q
        Take action a to obtain the reward r and the next state s'
        Update Q(s, a) ← Q(s, a) + α[r + γ max_a' Q(s', a') − Q(s, a)]
        Update ε                                      ▷ discretionary
```

In this specific case, the policy is implemented using a neuro-symbolic architecture [4], and thus consists of two components: a neural component, discussed in more detail in Section 1.2.1, and a logical component, discussed in Section 1.2.2. Further details on the training procedure, loss function, and backpropagation are provided in Section 1.2.3.

1.2.1 Neural Component

The neural component receives the previously mentioned RGB image as input and classifies the cells of the game grid, assigning to each cell one of four possible labels based on its content, so the possible labels are: *agent*, *target*, *enemy* and *empty*.

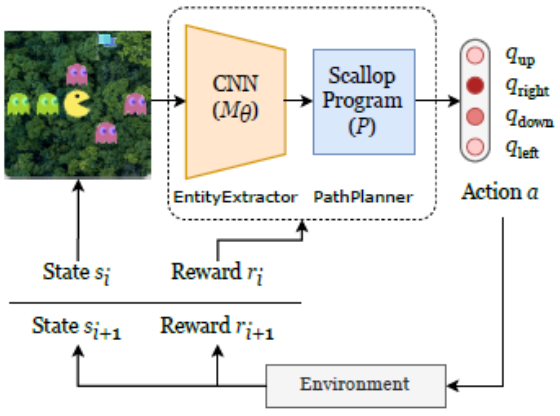


Figure 2: Architecture of the agent.
Source: [5].

This is achieved using a convolutional neural network (CNN) with a softmax activation function in the final layer, producing a soft classification, that is, a probability distribution over the four possible labels for each grid cell.

1.2.2 Logical Component

The logic component, as shown in the figure 2, receives as input the soft classification produced by the neural component and outputs a probability distribution over the possible actions. In practice, the grid is modeled as a graph over which a probabilistic search is performed: each node represents a cell, with an associated probability equal to the complement of the probability that the cell contains an enemy; an edge between two nodes represents the possibility of moving from one cell to the other by executing a specific action.

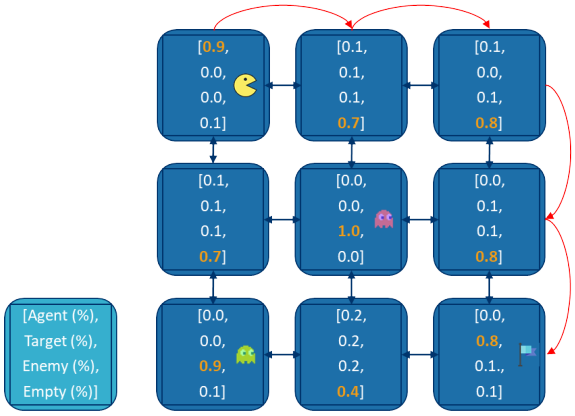


Figure 3: Probabilistic path search on a graph.

Once the graph is constructed, a path is computed for each pair of nodes, with its associated probability defined as the conjunction of the probabilities of the nodes along the path, thus favoring paths that are less likely to pass through cells containing an enemy. Finally, a probability distribution over the possible moves is computed by taking into account the probabilities of each path, as well as the probabilities associated with the initial and final positions.

```
rel path(x, y, x, y) = node(x, y)
rel path(x, y, xp, yp) = edge(x, y, xp, yp, _)
rel path(x, y, xpp, ypp) = path(x, y, xp, yp) and \
    edge(xp, yp, xpp, ypp, _)
```

Listing 1: Connectivity conditioned on no enemy on the path.

```
rel next_position(xp, yp, a) = agent(x, y) and \
    edge(x, y, xp, yp, a)
rel next_action(a) = next_position(x, y, a) and \
    path(x, y, gx, gy) and target(gx, gy)
```

Listing 2: Distribution over actions.

1.2.3 Training, Loss and Backpropagation

From the above, it follows that the method employed is more accurately described as Deep Q-Learning, given the specific implementation of the agent's policy. Consequently, updating the Q-function corresponds to updating the weights of a neural network. So, typically, the loss is computed by measuring the difference between the expected and predicted $Q(s, a)$; in this case, the Huber loss was selected for this purpose. Equation 1 defines the Huber loss for a single prediction instance.

$$Huber(x_n, y_n) = \begin{cases} 0.5(x_n - y_n)^2 & if|x_n - y_n| < \delta \\ \delta(|x_n - y_n| - 0.5\delta) & otherwise \end{cases} \tag{1}$$

To speed up the training, a second loss term, directly estimating the quality of the output produced by the neural component, is added to the primary loss. This loss term, given the soft classification produced by the CNN, computes the probability p that more than one cell is identified as the target or that more than K cells are identified as enemies, deviating from the ideal output distribution. The Smooth-L1 loss was selected for this purpose and Equation 2 defines it for a single prediction instance.

$$SmoothL1(x_n, y_n) = \begin{cases} \frac{0.5}{\beta}(x_n - y_n)^2 & if|x_n - y_n| < \beta \\ |x_n - y_n| - 0.5\beta & otherwise \end{cases} \tag{2}$$

So, the final loss becomes:

$$loss = Huber(\hat{Q}(s,a), Q(s,a)) + SmoothL1(p, 0). \tag{3}$$

Obviously, the neural component is, by its very nature, differentiable, but, in order to perform training with a single backpropagation step, the logical component must also be differentiable, despite the absence of trainable parameters in it. This is achieved by using a differentiable *provenance* for logical equations, i.e. logical operators (AND, OR, ...) are mapped into differentiable operators ($\times$, $+$, ...), resulting in differentiable equations.

2 Improving the Agent

2.1 Generalizing the Logical Component

A possible critique of the previous section is that, in Q-learning, the policy network's output represents an estimate of the state-action value function. In contrast, here, the policy network's output corresponds to the logical component's output, which is a probability distribution over the possible actions, as specified in Section 1.2.2. The reason why the algorithm works equally lies in the choice of rewards: at each step, the agent receives a reward of zero, except when it reaches the target cell, in which case the reward is one. This deliberate design choice ensures that the policy network's output, representing a distribution over the possible actions, aligns with the Q-function.

However, this approach limits the generality of the agent and the range of applicable domains. To address this limitations, a possible solution is to extend the logical component to handle different rewards. This can be achieved in two ways: by customizing the provenance (1) or by incorporating the rewards directly into the tuples as additional elements (2).

```
# path struct: from node A --> node B
#   => (A_x, A_y, B_x, B_y, distance, reward)

path(x, y, x, y, 1, r) = node(x, y, r)

path(x, y, xp, yp, 1, r) =
    edge(x, y, xp, yp, _) and \
    node(xp, yp, r)

path(x, y, xpp, ypp, L, R) =
    node(x, y, _) and \
    node(xpp, ypp, _) and \                    # L and R are const.
    $abs(x - xpp) + $abs(y - ypp) > 1          # used as lower bound

path(x, y, xpp, ypp, l + 1, rp + rn) =
    path(x, y, xp, yp, l, rp) and \
    edge(xp, yp, xpp, ypp, _) and \
    node(xpp, ypp, rn) and \
    path(x, y, xpp, ypp, _, r) and \
    r < rp + rn and \
    l + 1 <= L
```

Listing 3: Extended connectivity to integrate rewards and distances.

While the second solution is presented in Listing 3, the first is omitted for brevity. In essence, it involves implementing a class that can serve as a provenance and uses rewards instead of probabilities as tuple tags. To be used as provenance, this class must implement the methods defined in the abstract class `ScallopProvenance`.[1]

Both solutions are theoretically valid alternatives, but in practice, once implemented, they yielded poor results mainly in terms of efficiency. This is due to certain limitations currently present in the Scallop library, which are discussed in more detail in Section 2.1.1.

2.1.1 Limitations of Scallop

The current version of Scallop (0.2.4) has some limitations, bugs and/or missing features, that affect its overall usability.

[1] The code for both solutions is available in the Extras section of the project repository notebook.

Lack of Documentation The first, relatively significant, shortcoming concerns the documentation: while there is a basic documentation covering the language and the general capabilities of Scallop, more specific and detailed documentation for its more complex operations is somewhat lacking. This is especially true for *Scallopy*, the Python integration of Scallop.
For example, in order to implement a custom provenance correctly, a high-level description of the abstract methods may not be enough, users would benefit from a cleaner understanding of how these methods are used internally within the library's codebase.

Lack of Support for Filtering Another missing feature that is likely to impact usability, in particular the scalability of algorithms based on Scallop, is the inability to filter tuples on the fly, i.e., as they are being generated. Instead, the process requires two steps: first, all tuples must be generated, and only then they can be filtered. This risks producing unnecessary memory overhead. Consider, for example, the previously proposed Solution 2 for extending the rewards: a tuple is considered distinct from others if it differs at least in one element. So, in this case, a tuple must be stored for every path between two nodes that differs in length or reward, and filtering is applied only afterward to retain the best paths. However, many of these tuples could be discarded immediately, as they are associated with paths that are already known to be suboptimal. This leads to a large memory footprint even for small game grids, which significantly impacted the performance of this solution.

Limited Provenance Customizability A third possible issue, although more open to debate, concerns the limited flexibility in customizing provenance. This limitation comes from the structure of the interface, particularly the arguments required by its abstract methods. This point is debatable, because the current design is perfectly valid if the goal is simply to enable a different form of differentiability for logical equations. However, it becomes insufficient if the aim is to extend the tags associated with tuples to something more complex than a probability.
The Solution (1) proposed earlier is directly affected by this limitation, as it involves replacing the probability with the reward associated with each path. This approach requires more flexibility in how provenance information is represented and handled, which the current interface does not fully support.

GPU Usage The final issue is a bug related to GPU usage: although GPUs are available and Scallop is built on top of PyTorch tensors, it does not appear to leverage them to accelerate computation. While this does not perhaps directly prevent the use of Scallop, it significantly limits certain scenarios: for instance, training neural networks upstream a logical component in a single backpropagation step becomes impractical.[2]

2.2 Improving the Neural Component

2.2.1 Active Inference: Frameworks and Strategies

Active Inference is a theoretical framework grounded in Bayesian principles and the minimization of variational free energy. It is employed to model perception, learning, and decision-making processes [7]. The framework is based on the following core concepts:

1. Define a generative model by specifying states and possible observations;
2. Perceive the environment by gathering observations;
3. Minimize variational free energy to update beliefs about hidden states;
4. Define a policy, compute expected free energy (EFE) and minimize it;
5. Act as defined by the policy;
6. Receive new observations and return to Step 2.

In the context of this project, Active Inference appears to be a suitable approach, as it enables the identification of the Pacman Maze through a generative model and the use of a logical policy to act and minimize the energy after each new observation. Preliminary investigations have focused on some existing researches approaching this topic with Monte Carlo Tree Search and deep learning [2].

[2] A GitHub Issue is currently open regarding this problem.

Deep Active Inference with Variational Policy Gradient (DAI_{VPG}) This architecture consists of three neural networks:

- A *transition network*, which predicts future observations based on the current observation and the executed action;
- A *policy network*, which models the variational distribution over actions;
- A *critic network*, which estimates the EFE of each action given the current observation.

Notably, DAI_{VPG} does not include an encoder or decoder, and therefore does not maintain a representation of hidden states. This omission is a significant deviation from full Active Inference, where latent state inference is central, and it introduces uncertainty regarding the computation of intrinsic value. Furthermore, the Kullback–Leibler (KL) divergence term is approximated by the mean squared error (MSE), which constitutes a strong simplification. The resulting agent operates in a setting that lies between a Partially Observable Markov Decision Process (POMDP) and a Markov Decision Process (MDP), representing an adaptation rather than a strict implementation of the POMDP formalism.

Deep Active Inference based on Free Action (DAI_{FA}) This approach incorporates four deep neural networks, including an encoder and a decoder. However, several limitations must be considered. First, it does not explicitly optimize expected free energy; instead, the agent is trained to minimize the cumulative variational free energy over time. Second, the implementation constructs an extensive computational graph at each action-perception cycle, including the encoder, decoder, transition, and policy networks. This design leads to high computational costs and can become infeasible for long time horizons due to memory and scalability constraints. Finally, the DAI_{FA} framework requires manual specification of prior preferences within the distributions predicted by the encoder and transition networks. This manual encoding limits the method's applicability in real-world scenarios, where defining such priors can be highly non-trivial or domain-specific.

Deep Active Inference with Monte Carlo Tree Search (DAI_{MC}) DAI_{MC} is currently one of the most comprehensive and promising implementations of Active Inference. It integrates all the key components of a complete Active Inference model and employs Monte Carlo Tree Search (MCTS) to evaluate the EFE of different policies and to guide action selection. Action decisions are made by simulating future trajectories within the agent's internal generative model, computing the EFE for each trajectory, and selecting the action that minimizes it. While this approach enables principled planning under uncertainty, it requires multiple forward passes through deep neural networks, resulting in significant computational overhead. Training DAI_{MC} is particularly challenging due to numerical instabilities and the high variance introduced when estimating EFE through sampling. These difficulties are exacerbated when the encoder is insufficiently trained, potentially leading to inaccurate posterior estimates and degenerate or unstable behavior.

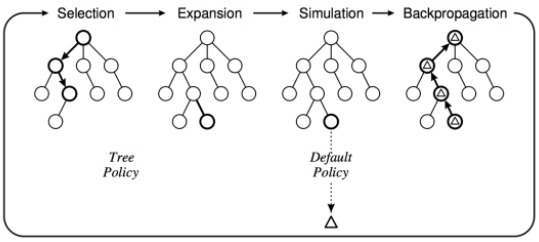


Figure 4: Diagram of the phases of an MCTS iteration.
Source: [1]

2.2.2 Monte Carlo Dropout

A practical approach to addressing the challenges of training and evaluation is the use of Monte Carlo Dropout (MCD). While MCD does not constitute a full implementation of Active Inference, it represents a meaningful step in that direction. As shown in [3], MCD offers a computationally efficient and scalable approximation of Bayesian inference in deep neural networks. In addition, the predictive uncertainty estimated through dropout can be used to dynamically adjust the ϵ

hyper-parameter of the policy network during the training. This allows for a more informed balance between exploration and exploitation: higher uncertainty promotes exploration, while lower uncertainty favors exploitation. So this mechanism enables faster convergence and helps prevent overfitting.

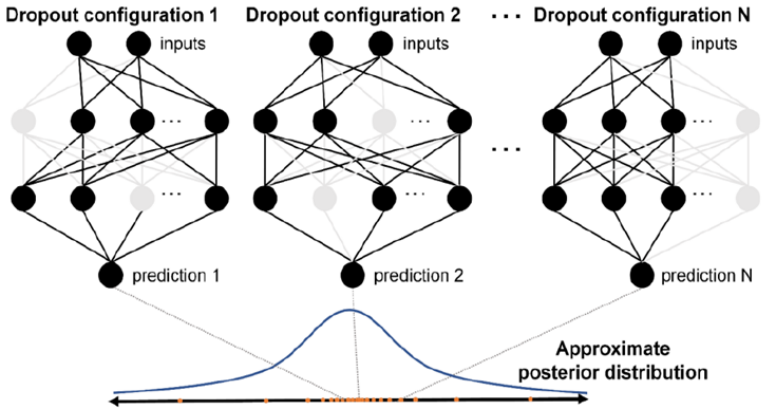


Figure 5: MC Dropout illustrative example.

Theory During the training, dropout randomly sets a fraction of the neurons to zero at each forward pass in order to prevent the network from relying too heavily on any neuron, forcing it to learn more robust features across the network. When dropout is also applied during inference, it transforms the network into a stochastic model, where each forward pass yields slightly different outputs. By performing multiple forward passes through these randomized sub-networks, we can sample from the posterior distribution of the network's weights, effectively turning a deterministic model into a stochastic one without incurring significant computational overhead.

Implementation To implement MC Dropout, dropout layers are kept active even during the testing phase, whereas they are typically disabled during inference. For each input sample, multiple forward passes are performed, denoted as N, which is a crucial hyperparameter that must be carefully chosen to balance computational complexity and prediction variance. The mean and standard deviation of the N predictions are then computed, providing both the expected output and a measure of uncertainty associated with the prediction. As previously mentioned, this uncertainty measure can be utilized to dynamically adjust the ϵ hyper-parameter. Importantly, these N forward passes can be executed in parallel, leveraging modern hardware capabilities such as GPUs. This parallelization significantly reduces the overall inference time, making the MC Dropout approach more practical for real-world applications.

Algorithm 2 Forward pass with MC Dropout.

Require: A tensor x of shape $(batch, 3, H, W)$
Set the number N of MC samples and $training$ to $True$
$x \leftarrow repeat(x, N)$ ▷ *shape* $(batch \cdot N, 3, H, W)$
$y \leftarrow forward(x)$ ▷ *forward pass, shape* $(batch \cdot N, 4)$
$y \leftarrow reshape(y, shape = (-1, N, 4))$ ▷ *shape* $(batch, N, 4)$
$avg \leftarrow mean(y, axis = 1)$ ▷ *mean of the N predictions for each datapoint in x*
$sd \leftarrow std(y, axis = 1)$ ▷ *standard deviation of the N predictions*
return avg, sd

3 Experimental results

First Experiment The first test aimed to evaluate the scalability of the proposed model with respect to the number of samples used with the MC Dropout. The agent was trained and tested seven times on the same dataset, each time using a different number of samples. Table 1 reports the average training and testing time per epoch as the number of samples varies. In this experiment, the number of episodes per epoch was significantly reduced to speed up data collection. As a result, the reported computation times are not representative of realistic performance, and the agent's accuracy under these conditions would be extremely poor.

Execution Time (sec)		
$\#Samples$	T_{train}	T_{test}
1	2.43	0.54
10	2.49	0.55
20	2.69	0.55
30	2.88	0.56
50	3.21	0.60
100	3.53	0.67
250	4.74	0.81

Table 1: Average training and testing time per epoch as the number of MC Dropout's samples increases. The mean was estimated using 50 epochs.

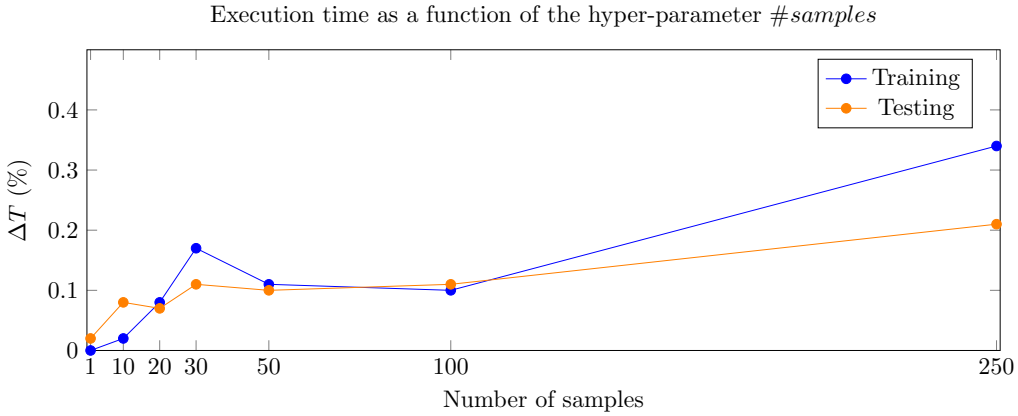


Figure 6: Percentage change in computation times shown in Table 1.

As shown in Table 1 and also in Figure 6, execution time grows slowly with the number of samples and this growth is mainly due to the preparatory steps needed to enable N parallel forward passes per input. This not only demonstrates the scalability of the algorithm with respect to the number of samples, but also shows that, when comparing both training times, which are consistently higher, and test times with those of an equivalent agent without MC Dropout (first row of Table 1), the difference remains quite acceptable, particularly given that in most cases fewer than 50 samples are sufficient to obtain a reliable uncertainty estimate [6].

Second Experiment The second experiment aimed to assess the success rate achieved by the agent during both training and testing phases. The performance of the agent augmented with MC Dropout was subsequently compared to that of the original agent developed by the Scallop team. Success rate is quantified as the percentage of episodes successfully completed, providing a direct metric for evaluating the impact of MC Dropout on agent performance.

As shown in Figure 7, accuracy reaches around 80% after 32 epochs. Fluctuations in performance are expected in reinforcement learning, where it's also common for the training error to be higher than the test error due to the ϵ-greedy policy, which drives the agent to act randomly during training to explore the environment.
Due to the issues discussed in Section 2.1.1, training and testing the agent, even with only 1024 episodes, take about an hour, even on a T4 GPU.

Compared to the baseline agent provided as an example in Scallop, the version using MC Dropout achieves higher accuracy across both training and testing epochs and episodes. A substantially higher success rate is reported in [5]; however, that result was obtained using 10,000 episodes, while only 1,000 were used in our experiments. Notably, even training on 1,000 episodes requires nearly two hours. It is important to note, however, that MC Dropout is not the sole difference between the two implementations. The architectures of the neural components differ, as does the ϵ update strategy, as previously discussed, as well the target network update policy. In the Scallop agent, the target network is updated periodically with fixed intervals, and this becomes an additional hyper-parameters, while our implementation adopts a soft update mechanism.

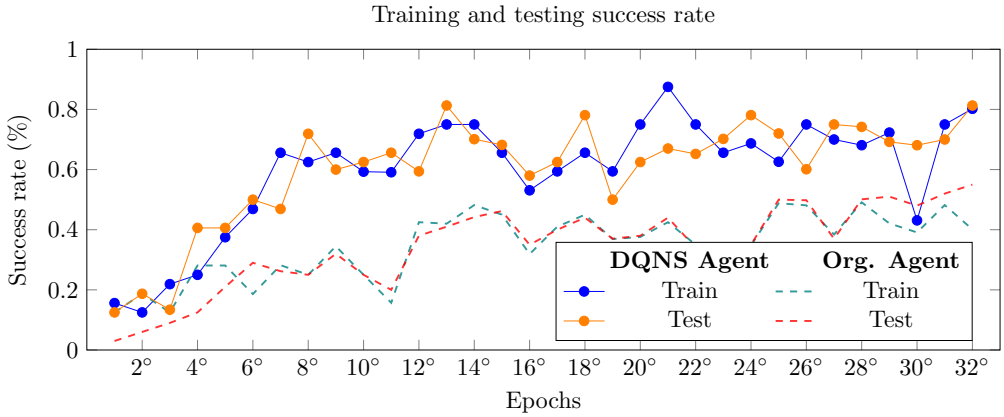


Figure 7: Success rate during training and testing over the first 32 epochs, each consisting of 32 episodes. The environment is a 5×5 grid with 4 enemies and a maximum of 30 moves per episode. Solid lines correspond to the performance of the implemented DQNS agent, while the dashed lines indicate the performance of the original Scallop's agent.

4 Concluding remarks

In conclusion, we tested the applicability of Scallop as a framework for neuro-symbolic programming, highlighting both its strengths and potential limitations. While full integration with active inference models remains challenging, due to their inherent nature, computational complexity, and comprehensive structure, hybrid approaches like the one proposed are still feasible. As our results suggest, such solutions can be competitive both in accuracy and in terms of resource usage, although more extensive testing and significantly longer training sessions would be needed to draw more reliable conclusions. An effort currently limited by our available computational resources.

References

[1] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[2] Théophile Champion et al. "Deconstructing deep active inference". In: *arXiv preprint arXiv:2303.01618* (2023).

[3] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 1050–1059. URL: https://proceedings.mlr.press/v48/gal16.html.

[4] Pascal Hitzler and Md Kamruzzaman Sarker. *Neuro-symbolic artificial intelligence: The state of the art*. IOS press, 2022.

[5] Ziyang Li, Jiani Huang, and Mayur Naik. "Scallop: A language for neurosymbolic programming". In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 1463–1487.

[6] Daily Milanés-Hermosilla et al. "Monte carlo dropout for uncertainty estimation and motor imagery classification". In: *Sensors* 21.21 (2021), p. 7241.

[7] Christopher J. Whyte Ryan Smith Karl J. Friston. "A step-by-step tutorial on active inference and its application to empirical data". In: *Journal of Mathematical Psychology* (2022). URL: https://doi.org/10.1016/j.jmp.2021.102632.

[8] Mark Towers et al. "Gymnasium: A standard interface for reinforcement learning environments". In: *arXiv preprint arXiv:2407.17032* (2024).