

# NI & AC

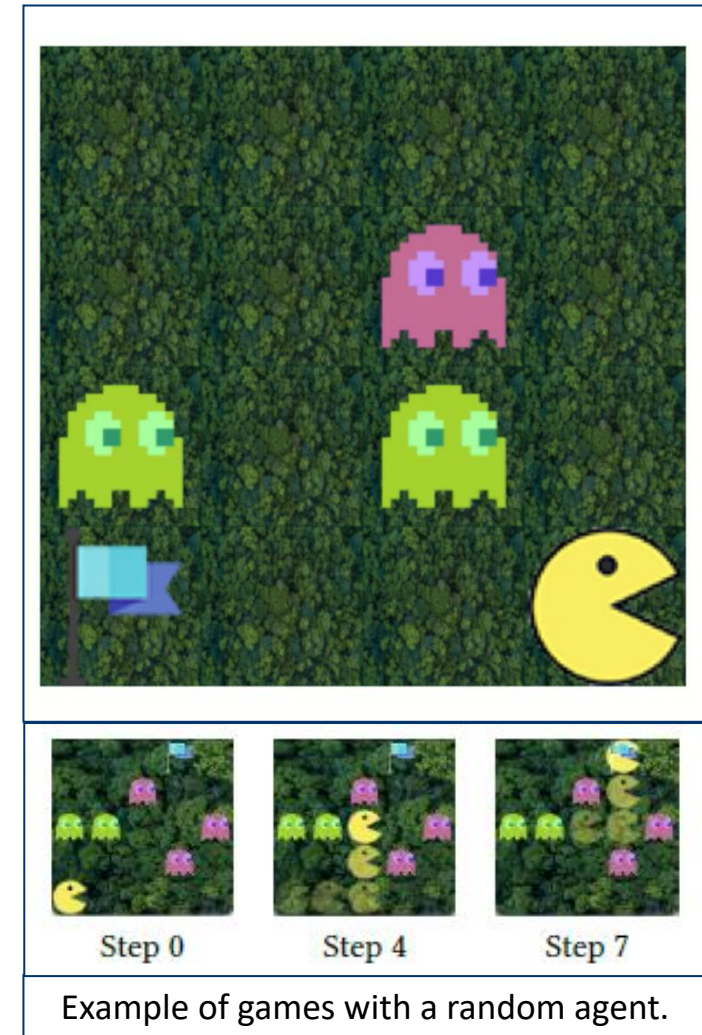
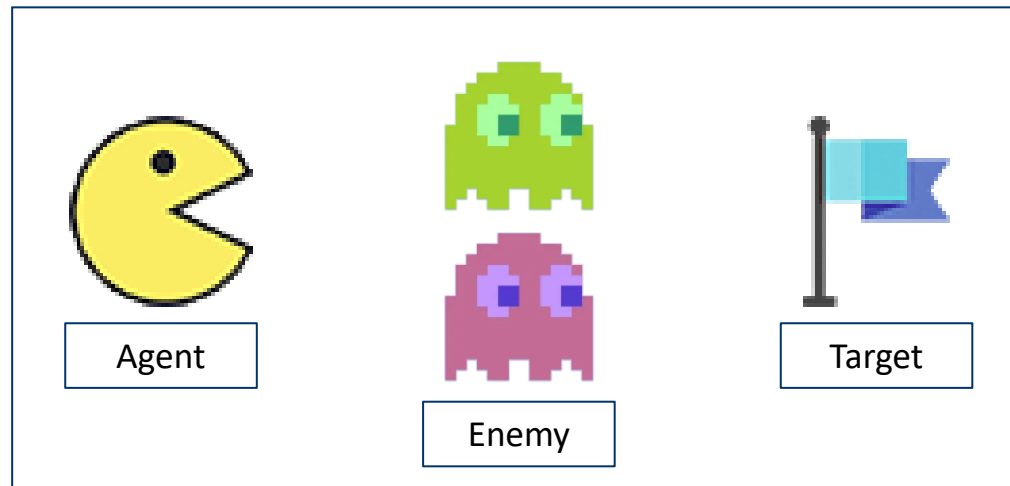
*A NS Agent for Pacman Maze*



# Pacman Maze

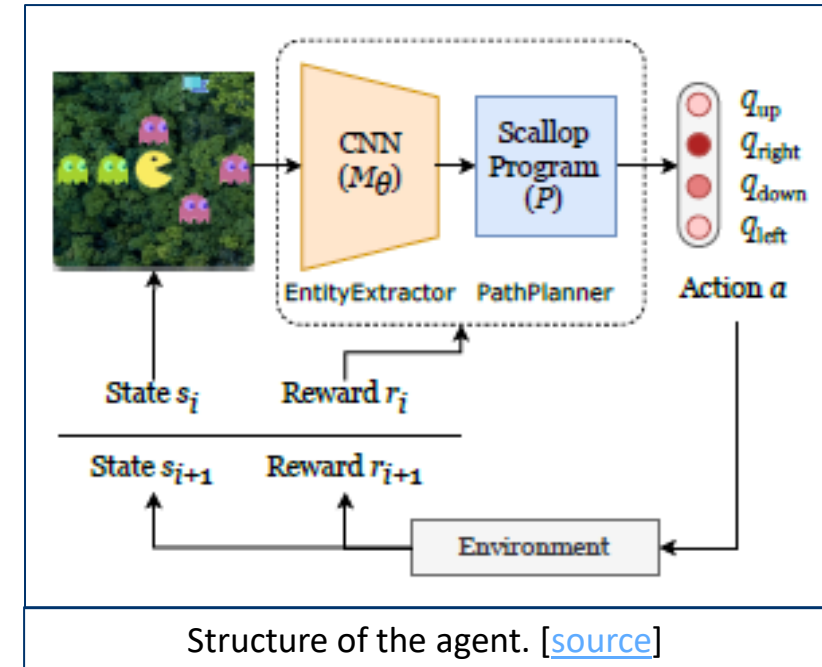
- Pacman Maze:

- Grid:  $M \times N$  cells.
- 4 possible moves:  $\uparrow \downarrow \leftarrow \rightarrow$ .
- $K$  enemies, all initial positions are randomly chosen.
- **Goal:** the agent must reach the flag within  $T$  moves without stepping on a cell occupied by an enemy.



# DQNS Agent

- Neural-symbolic agent:
  - **Neural component:**
    - CNN-like to extract entities (PyTorch).
    - From image to distribution over  $[agent, empty, enemy, target]$  for each cell.
  - **Logical component:**
    - Predict next move (Scallop).
    - Probabilistic path search on a graph.
- Agent trained using (Deep) **Q-learning**:
  - $\epsilon$ -greedy policy network = neural comp. + logical comp.
  - Soft update of the target network.



$$\theta' \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta$$

Soft update of the target network's parameters.

# Loss & Backpropagation

- Loss based on three factors:
  - **Expected vs predicted state-action value** → Humber loss.
  - **Constraints violation** → Smooth L1 loss:
    - Number (probability) of cells identified as targets (should be 1).
    - Number (probability) of cells identified as enemies (should be K).
- **Single backpropagation step:**
  - Neural component: inherently differentiable.
  - Logical component: logical equations are written using **differentiable provenance**, i.e. logical operators are mapped to differentiable ones.

AND, OR, ...  ×, +, ...

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

Huber loss. [\[source\]](#)

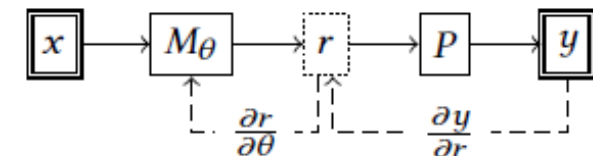


$$l_n = \begin{cases} 0.5(x_n - y_n)^2 / \beta, & \text{if } |x_n - y_n| < \beta \\ |x_n - y_n| - 0.5\beta, & \text{otherwise} \end{cases}$$

Smooth L1 loss. [\[source\]](#)



$$\text{loss}(X, Y) = \text{Humber}(X, Y) + \text{SmoothL1}(X, Y)$$



Backpropagation in a NS model. [\[source\]](#)

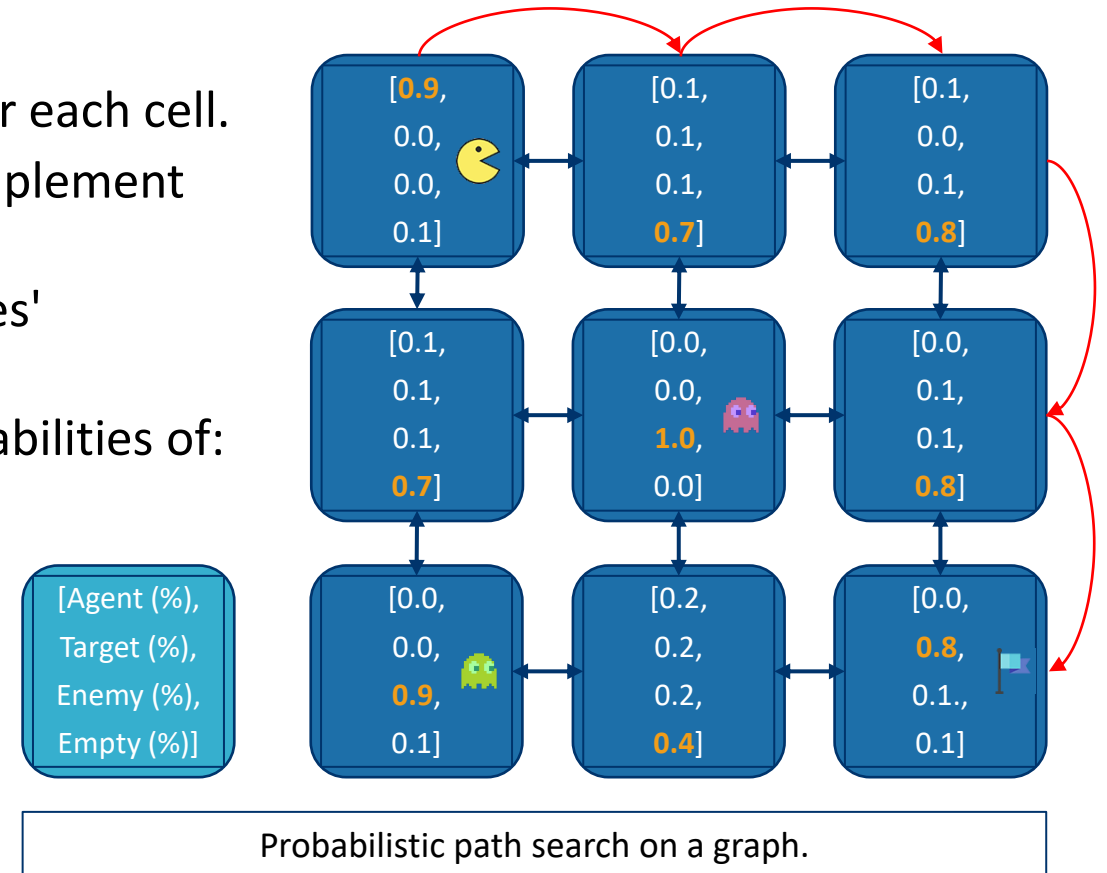
# Logical Component

- **Logical component:**

1. Read the prediction of the neural component for each cell.
2. Assign each node a probability equal to the complement of the enemy presence probability.
3. Path probability = conjunction of traversed nodes' probabilities.
4. Compute the next move by conjoining the probabilities of:
  1. The current position;
  2. The goal position;
  3. And the path to follow.

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

Formula used to compute the next action.



# Q-Learning

- In Q-Learning:
  - **Policy network's output** is an **approximation of the state-action value function  $Q$** .
  - At each step, this estimate is used, together with the reward obtained, to improve the quality of the next predictions.
- But here the **logical component returns a distribution over the action space**. It only works thanks to the chosen rewards.

```
class AvoidingArena(gym.Env):  
    def __init__(  
        self,  
        grid_dim: Tuple[int, int] = (5, 5),  
        cell_size: float = 0.5,  
        dpi: int = 80,  
        num_enemies: int = 5,  
        easy: bool = False,  
        default_reward: float = 0.00,  
        on_success_reward: float = 1.0,  
        on_failure_reward: float = 0.0,  
        remain_unchanged_reward: float = 0.0,  
    ):
```

$$Q^\pi(s, a) = r(s, a) + \gamma \sum_{s' \in S} V^\pi(s') \mathbb{P}(s' | s, a)$$

$Q$  function:  $\pi$  is the policy,  $s$  the state,  $a$  the action,  $r$  the reward and  $V$  is the state value function.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t (r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Q-Learning update rule.

# Extending the Rewards

## Custom Provenance

```
1 # ---- Custom provenance ----
2 class MyProvenance(scallopy.provenance.ScallopProvenance):
3     def __init__(self, min :float, max :float):
4         super(MyProvenance, self).__init__()
5         self.min = min
6         self.max = max
7         return
8
9     def name(self):
10        return "custom-provenance"
11
12    def zero(self):
13        return self.min
14
15    def one(self):
16        return self.max
17
18    def add(self, t1, t2):
19        raise Exception("Not implemented")
20
21    def mult(self, t1, t2):
22        return torch.clip(t1 + t2, min=self.min, max=self.max)
23
24    def negate(self, t):
25        return torch.clip(-t, min=self.min, max=self.max)
26
27    def saturated(self, t1, t2):
28        return bool(t1 > t2)
```

## Reward as Tuple Element

```
# connectivity: path(node_i, node_j, num_moves, reward)
# {L} is the maximum number of moves allowed
# {R} is a lower bound for the 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
    $abs(x - xpp) + $abs(y - ypp) > 1 # 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}
```



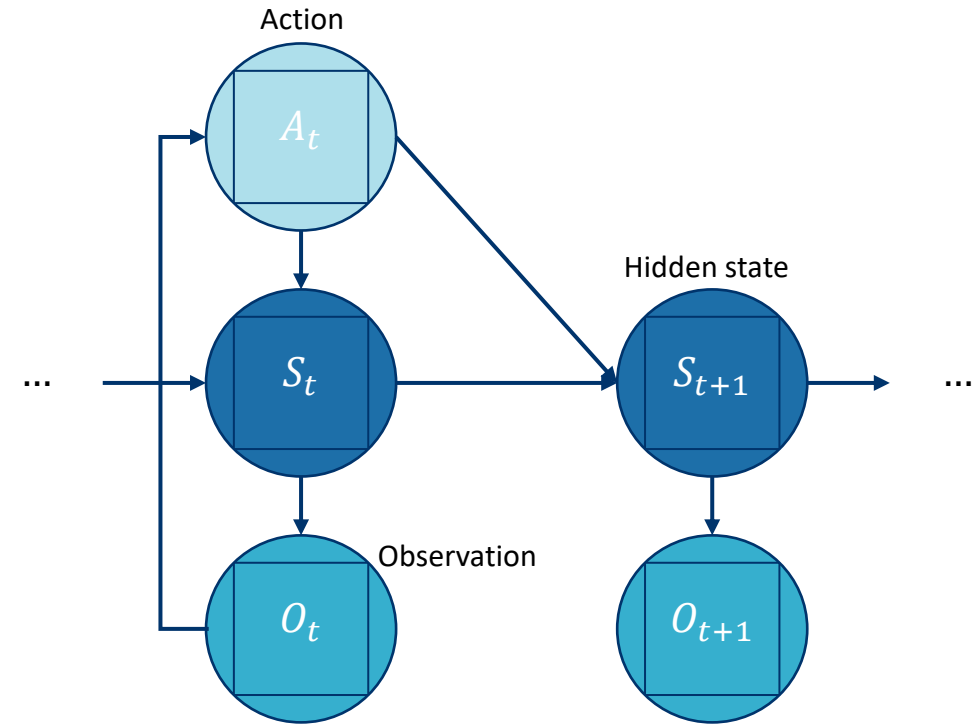
# Limitations of Scallop

- Main Scallop limitations:
  - A. **Lack of documentation** for advanced operations.
  - B. **Lack of support for filtering** tuples at generation time.
  - C. **Limited** provenance **customizability**.
  - D. **GPU unused** by logic component → High training time even for small neural component.  
[\[Git Issue\]](#)
- In practice, **proposed solutions** for handling different rewards are **ineffective**:
  - Reward as tuple element → Huge memory footprint even for small grids (due to point B).
  - Custom provenance → Incorrect rewards due to inability to implement the saturation criterion correctly (due to point C).



# Neural Component

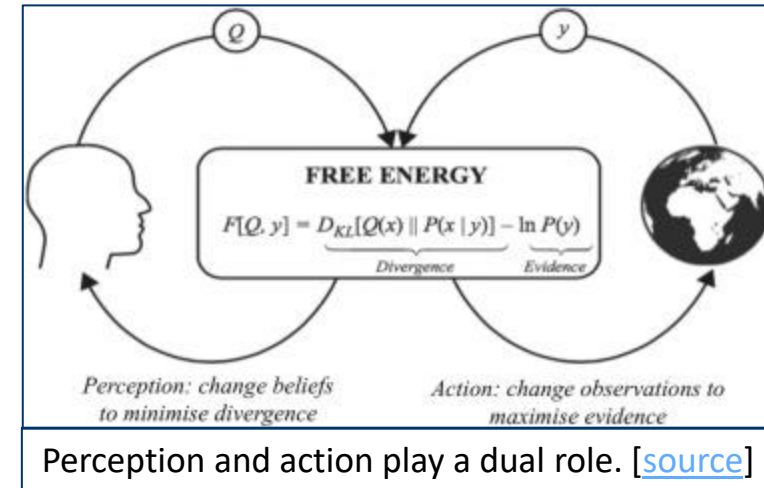
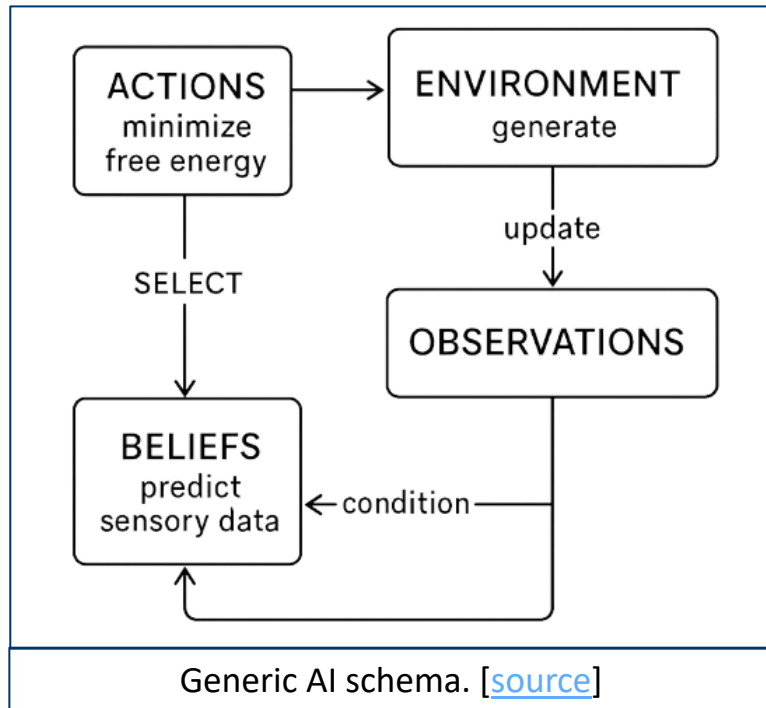
- How can the logical component be improved, considering the limitations of Scallop?
- Original neural component → CNN:
  - 2x 2D-Convolutional layers;
  - 2x Dense layers;
  - ReLu as activation function;
  - Softmax as last activation function.



PGM of the game: in dark blue the hidden states.

# Active Inference

- A different approach to the problem:
  - **Minimize uncertainty** instead of maximizing the reward.



- **Challenges:**
  - A detailed generative model is needed.
  - Simulating and managing uncertainty.
  - Scalability.

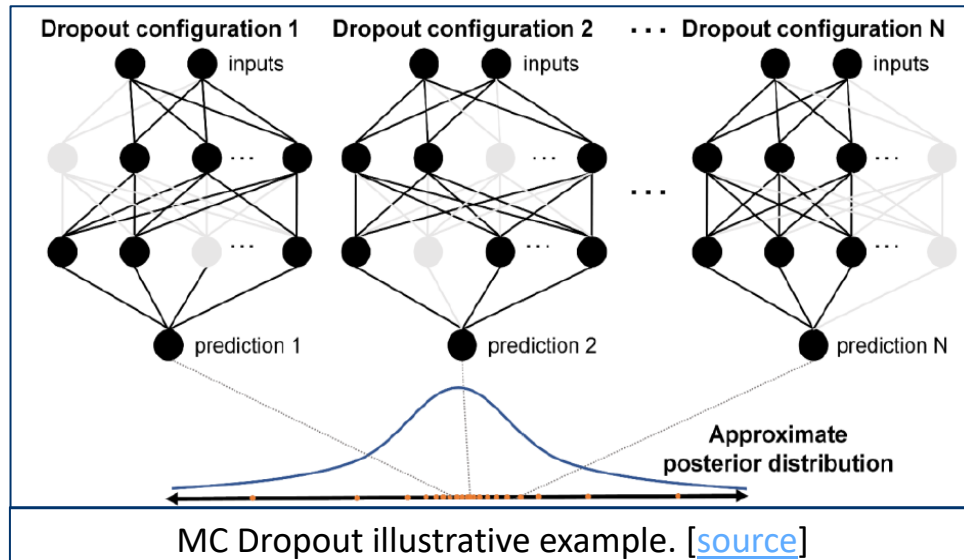
# Possible Implementations

- Deep Active Inference with Variation Policy Gradient (**DAI<sub>VP</sub>**):
  - Gradient descent to **minimize Expected Free Energy** (EFE).
    - Not full AI ✗
    - Inaccurate ✗
- Deep Active Inference Free Action Model (**DAI<sub>FA</sub>**):
  - **Minimizes the Variational Free Energy**.
    - No scalability ✗
    - Unclear preferences ✗
- Deep Active Inference with Monte Carlo Tree (**DAI<sub>MC</sub>**):
  - Monte Carlo Tree Search to explore action sequences.
    - Proper AI ↑
    - Heavy and complex to train ✗

# Our Approach: MC Dropout

- **Monte Carlo (MC) Dropout:**

- **Dropout** layers kept active also **during evaluation**.
- So multiple passes with different dropout masks for each input → multiple predictions.
- This allows **uncertainty estimation** in predictions, and it **approximates variational inference** ([Gal et al.](#)).



- Pros:

- Training time unaffected.
- Parallel evaluation → time unchanged.

- Cons:




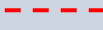
- Poorly approximates complex posterior distributions.
- Sensitive to hyperparameters (e.g., dropout rate).

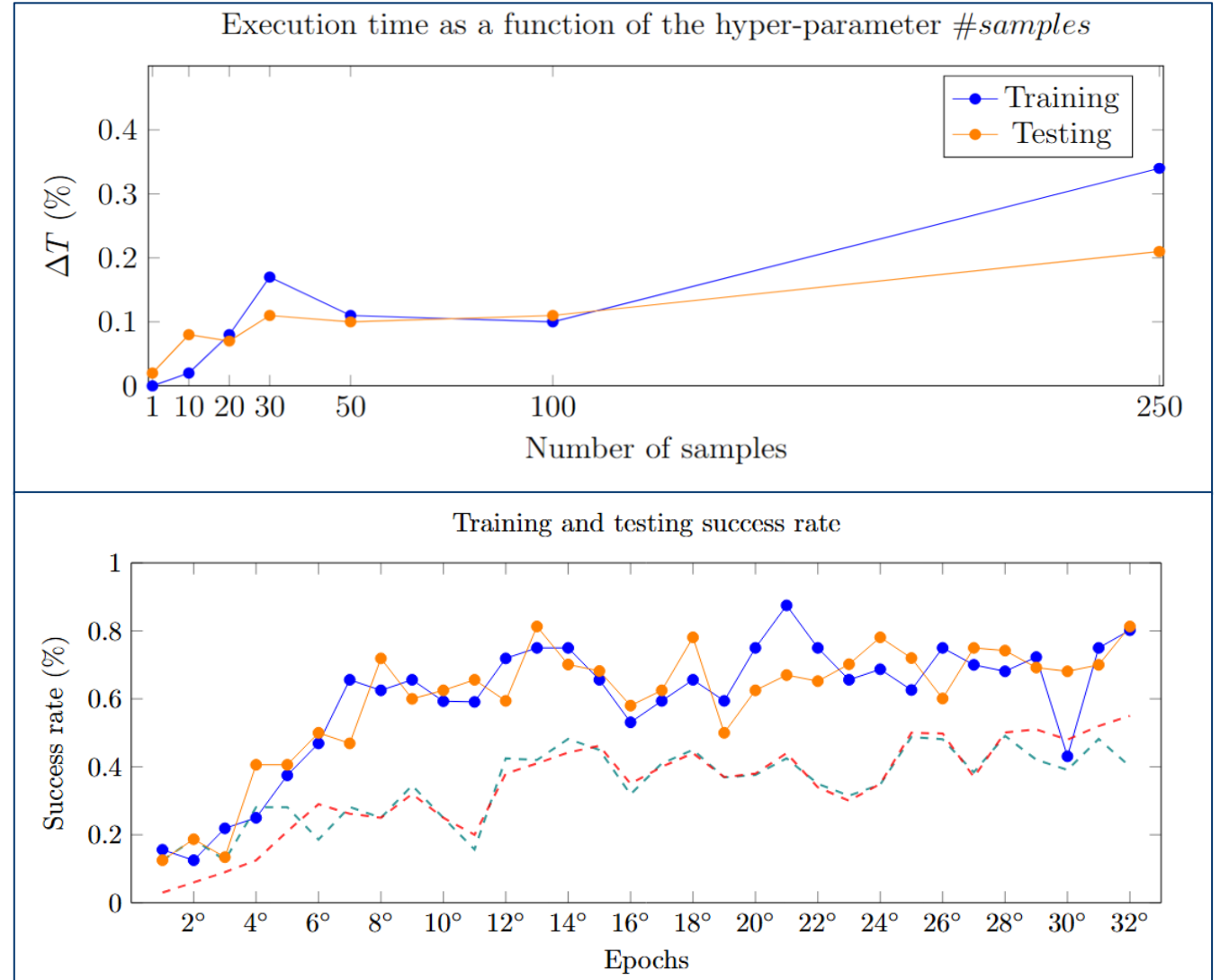
# Experimental Results

## Test 1

Execution Time (sec)		
#Samples	$T_{\text{train}}$	$T_{\text{test}}$
1	2,43	0,54
10	2,49	0,55
20	2,69	0,55
30	2,88	0,56
50	3,21	0,56
100	3,53	0,67
250	4,74	0,81

## Test 2

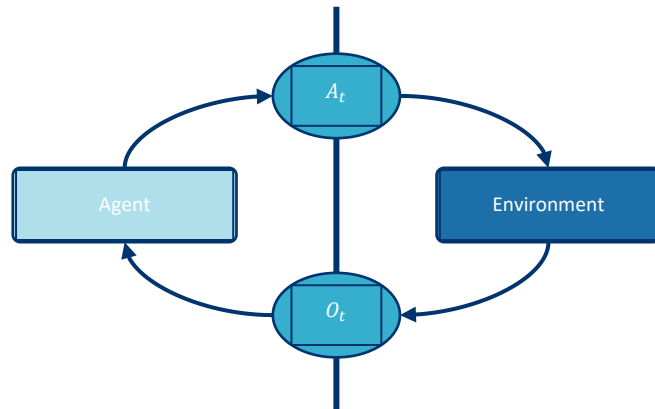
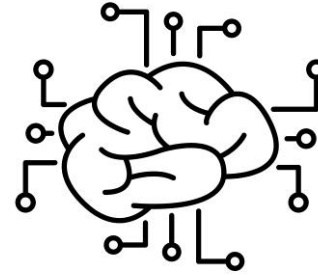
Legend		
	DQNS Agent	Original Agent
Train		
Test		



# Conclusions

- Scallop:

- A framework for neuro-symbolic programming .
- With its strengths and potential limitations.

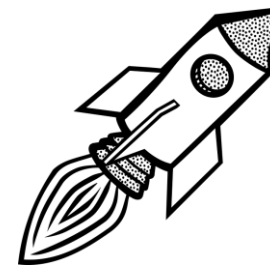


- MC Dropout:

- A trade-off between efficiency and Bayesian inference.
- Good accuracy and less prone to overfitting.

- Further developments:

- More training & testing → more reliable conclusions.
- More complex neural and/or logical components.



Grazie

