Solving Grid-World Problem with Reinforcement Learning

Jean Lucien Randrianantenaina
Applied Mathematics, Machine Learning and Artificial Intelligence
Stellenbosch University

***Abstract*—**

I. Introduction

When it comes automatic sequential decision making with a specific objective for an agent in given environment reinforcement learning is a best approach to use. In contrast with that, given grid-world, how we can train an agent to navigate form a point A to a point B, in an optimal way with the maximum score. To solve this problem, we explore four Reinforcement Learning algorithm.

For that, we start by defining the problem in Section II, and brief overview of reinforcement learning in the Section III. Then we present the tabular methods and deep learning techniques respectively in section IV and V.

Section VI delve on how we deal with our hyper-parameter, and model assessment. Finally in the Section VII, we presents our finding, compare and discuss about them.

II. Problem statement and modelling

We aim to build an agent (red triangle) that solves an empty 6×6 grid world, e.g., reaching the green cell as illustrated in Figure 1. The agent (red triangle) interacts with the environment through observations, actions and rewards. We use the `mini-grid` API from the Farama foundation to simulate the environment [3].

Depending on our configuration, the state can be a tensor or an image which is returned after each action. The agent has eight possible actions, but we only use three: turn left and right and move forward. In this setting, we deal with an episodic task, and for each episode, the agent can perform at most 256 steps.

If the agent reaches the target, it receives a reward r defined by:

$$r = 1 - 0.9 \times \frac{\text{Number of steps}}{\text{Maximum number of steps}} \tag{1}$$

otherwise, the reward is zero.

Therefore our task is to make the agent learn how to move in this grid in such a way that it maximizes its reward. For this particular environment, the agent can get at most a reward of 0.9613 with 11 steps. But for this work, we will consider 0.9578 as the optimal reward corresponding to 12 steps.

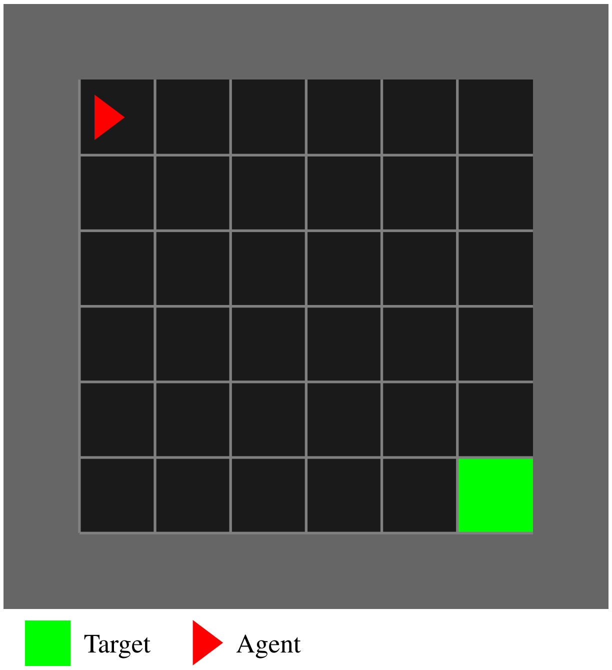

Fig. 1: An illustration of the `MiniGrid-Empty-8x8-v0` environment

III. Reinforcement Learning technique

In this era, reinforcement learning is widely used to solve the problem of programming intelligent agents that learn how to optimise a specific task such as playing a game, or autonomous vehicle. This is done by learning a policy for sequential decision problems that maximize the discounted cumulative future reward:

$$G_t \triangleq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2}$$

Where R is the reward, $\gamma \in [0, 1]$ is the discount factor that controls the importance fro immediate reward (near to 0) or future reward (near to 1). The agent learns the optimal policy without being explicitly told if its actions are good or bad using the reinforcement learning model depicted in Figure 2.

Several algorithms/techniques can be used to solve our problem with reinforcement learning, in the next section we will look:

- Q-Learning

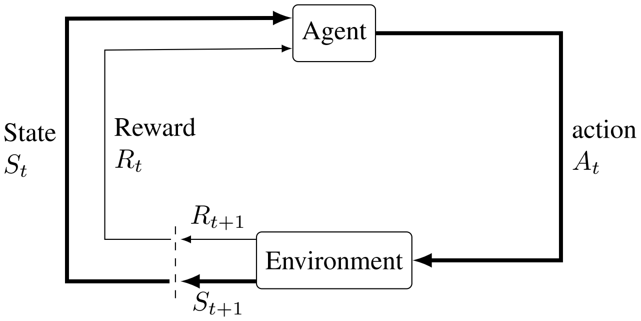

Fig. 2: Reinforcement learning Model, [1], [2]

- SARSA (State-Action-Reward-State-Action)
- Deep Q-Network
- Deep Q-Network with RGB Image techniques.

The algorithm 1 show the skeleton shared by these RL algorithms. In addition to that, we also use the ϵ-greedy strategy to select the action to be performed by the agent. The agent uniformly samples an action from the possible action for the given state with a probability ϵ and uses the optimal action for the given state with a probability $1 - \epsilon$. In the first case, we say that the agent is exploring, and in the second case we say that the agent is exploiting.

```
Parameters:...
foreach episode do
    (Re)Initialize the environment
    foreach step do
        Act and consider the observation and reward
        Steps specific to each methods
        if done or truncated then
            Some steps for metric and monitoring
        end
    end
end
```

Algorithm 1: RL Algorithm Skeleton

For this work we will start with a higher value of ϵ to allow the agent to explore the environment, then we decrease it slowly until we reach a small value, this will be controlled by :

$$\epsilon_t = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \exp\left\{-\frac{t}{\Delta}\right\} \tag{3}$$

Where Δ is the decay rate, a higher value corresponds to a slow decrease, and a small value corresponds to a fast decrease.

IV. Tabular methods: Q-Learning and SARSA

The tabular method refers to the creation of a table called Q-table, containing the value of $Q(S_t = s, A_t = a)$. This value indicates how good is acting a on the state s.

For the two algorithms that we present in this section, the observation (a $7 \times 3 \times 3$ array) will be converted into a string to create an MD5 hash to represent the given state.

A. SARSA

The SARSA algorithm is an on-policy method, which means it iteratively refines a single policy, that also generates control action within the environments. The update rule is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4}$$

From the equation (4), we obtain the Algorithm 2 for the SARSA learning.

```
Parameters: Step size $\alpha \in ]0,1]$, small $\epsilon > 0$
Initialize $Q(s,a)$ for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$
  arbitrarily, except that $Q(terminal - state, \cdot) = 0$
foreach episode do
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$
      ($\epsilon$-greedy)
    foreach step until $S$ is a terminal state do
        Take the action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from
          $Q$ ($\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A)$
                  $+\alpha[R_{t+1} + \gamma(S',A') - Q(S_t,A_t)]$
        $S \leftarrow S'$
        $A \leftarrow A'$
    end
end
```

Algorithm 2: SARSA: on-policy learners to estimate the optimal Q-table

B. Q-Learning

In contrast to SARSA, the Q-Learning algorithm is an off-policy algorithm, which means we optimise the target policy using different policies. The update rule is given by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(S_{t+1}, a) - Q(S_t, A_t)] \tag{5}$$

Since we have the Algorithm 3 for Q-learning methods.

```
Parameters: Step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialize $Q(s,a)$ for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$
  arbitrarily, except that $Q(terminal - state, \cdot) = 0$
foreach episode do
    Initialize $S$
    Choose $A$ from $S$ using $\epsilon$-greedy
    foreach step until $S$ is a terminal state do
        Take the action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using $\epsilon$-greedy
        $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} +$
          $\gamma \max_a(S_{t+1},a) - Q(S_t,A_t)]$
        $S \leftarrow S'$
    end
end
```

Algorithm 3: Q-learning algorithm to estimate the optimal Q-table

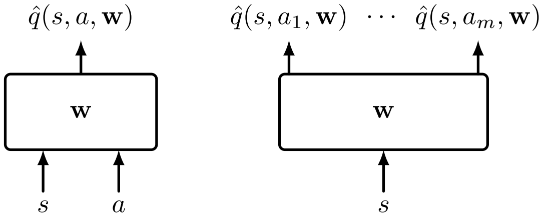
$\hat{q}(s,a,\mathbf{w})$
$\mathbf{w}$
s
a
$\hat{q}(s,a_1,\mathbf{w})$ $\cdots$ $\hat{q}(s,a_m,\mathbf{w})$
$\mathbf{w}$
s

Fig. 3: Illustration of function approximation

Layer	**Input**	**Output**	**# Param**
DQN	[1, 49]	[1, 3]	–
Sequential: 1-1	[1, 49]	[1, 3]	–
Linear: 2-1	[1, 49]	[1, 64]	3,200
ReLU: 2-2	[1, 64]	[1, 64]	–
Linear: 2-3	[1, 64]	[1, 32]	2,080
ReLU: 2-4	[1, 32]	[1, 32]	–
Linear: 2-5	[1, 32]	[1, 3]	99
Total params:			**5,379**
Trainable params:			**5,379**
Non-trainable params:			**0**
Total mult-adds (M):			**0.01**
Input size (MB):			**0.00**
Forward/backward pass size (MB):			**0.00**
Params size (MB):			**0.02**
Estimated Total Size (MB):			**0.02**

TABLE I: Model Summary for DQN

These two methods are powerful in dealing with a few states and actions, but not very efficient for numerous states and actions. We can replace the table with a function approximator as we will see in the next section.

V. Function approximation: Deep Q-network

The idea of function approximation is to replace the value function with its approximation instead of using a look-up table. The function can take

- the state and action as input, and $q(s,a)$ as output,
- or the state as input and $q(s,a)$ as output.

This is illustrated in the figure 3.

In our case, we will use a neural network as a function approximation. Thus we will consider two types.

A feed-forward neural network (MLP) which takes a vector $\mathbf{u} \in \mathbb{R}^{49\times 1}$ (state) as input and $q(s,a_1), q(s,a_2), q(s,a_3)$ as output. Similarly, Convolutional Neural networks (CNN), take a stack of four frames (images) at four successive time steps, e.g. a $4 \times 56 \times 56$ tensor as input and the same output as the MLP, The architecture of these neural networks are described in respectively in I and II.

As we do not have any labels to train the networks, we use two networks: the `policy_net` and `target_net`. The first one is optimised with the `Adam` optimizer, while the second one is fixed, and used to generate a sort of label for the `policy_net`. We also synchronize the weight of the two networks in a regular period of the training, to move toward the optimal values.

Layer (type:depth-idx)	Input	Kernel	Output	# Param
CNN_DQN	[1, 4, 56, 56]	–	[1, 3]	–
Sequential: 1-1	[1, 4, 56, 56]	–	[1, 512]	–
Conv2d: 2-1	[1, 4, 56, 56]	[3, 3]	[1, 16, 27, 27]	576
BatchNorm2d: 2-2	[1, 16, 27, 27]	–	[1, 16, 27, 27]	32
ReLU: 2-3	[1, 16, 27, 27]	–	[1, 16, 27, 27]	–
Conv2d: 2-4	[1, 16, 27, 27]	[3, 3]	[1, 32, 13, 13]	4,608
BatchNorm2d: 2-5	[1, 32, 13, 13]	–	[1, 32, 13, 13]	64
ReLU: 2-6	[1, 32, 13, 13]	–	[1, 32, 13, 13]	–
Conv2d: 2-7	[1, 32, 13, 13]	[3, 3]	[1, 64, 6, 6]	18,432
BatchNorm2d: 2-8	[1, 64, 6, 6]	–	[1, 64, 6, 6]	128
ReLU: 2-9	[1, 64, 6, 6]	–	[1, 64, 6, 6]	–
Conv2d: 2-10	[1, 64, 6, 6]	[3, 3]	[1, 128, 2, 2]	73,728
BatchNorm2d: 2-11	[1, 128, 2, 2]	–	[1, 128, 2, 2]	256
Flatten: 2-12	[1, 128, 2, 2]	–	[1, 512]	–
Sequential: 1-2	[1, 512]	–	[1, 3]	–
Linear: 2-13	[1, 512]	–	[1, 64]	32,832
ReLU: 2-14	[1, 64]	–	[1, 64]	–
Linear: 2-15	[1, 64]	–	[1, 3]	195
Total params				**130,851**
Trainable params:				**130,851**
Non-trainable params:				0
Total mult-adds (M):				**2.19**
Input size (MB):				**0.05**
Forward/backward pass size (MB):				**0.32**
Params size (MB):				**0.52**
Estimated Total Size (MB):				**0.89**

TABLE II: Model Architecture for DQN-Image

To train these networks, we need a dequeue (double end-queue) that stores a given number of the experience of the agent formed by the current state, action, next state, and reward, we call this a replay memory $\mathcal{D}$. As new experiences are added the oldest data is pushed out of the dequeue. Once we reach have batch size (enough number) of experience, we start to sample from $\mathcal{D}$ and optimize the policy_net parameter by minimizing the Mean Squared Error (MSE) given by:

$$\mathcal{L} = \frac{1}{N} \sum i = 1^N \left[R_i + \gamma \max_{a'} \hat{Q}(s'_i, a', \mathbf{w}^-) - \hat{Q}(s_i, a, \mathbf{w}) \right]^2 \tag{6}$$

where $R_i + \gamma \max_{a'} \hat{Q}(s'_i, a', \mathbf{w}^-)$ is computed using `target_net`, and $\hat{Q}(s'_i, a', \mathbf{w}^-)$ is 0 if s'_i is terminal state, while $\hat{Q}(s_i, a, \mathbf{w})$ is computed with `policy_net`.

Note that, the input of these networks (observation) is normalized by rescaling their value between $[-1, 1]$. In particular, we convert the RGB image into a grey-scale image for the CNN architecture. We can summarize the whole process in the algorithm 4.

VI. Hyper-parameters tuning and Evaluation

A. Hyper-parameters

like all machine learning problems we have several parameters to tune to get the optimal results. Grid search is one of the best methods, it iterates through all the possible combinations of predefined parameters set. The drawback of this method is the time complexity which can explode considerably when we hyper-parameter space is big and the algorithm is quite slow.

So, instead of using that approach, we will use trial and error to find good hyper-parameters such as γ, α, and the number of episodes. To do that we start we some values, and we adjust these parameters according to the obtained results.

```
Initialize policy_net and target_net
Initialize the environments
Set the decay rate for the epsilon decreasing
Set the updating period of target_net
Set the total step to 0
Create a replay memory $\mathcal{D}$
foreach episode do
    Set step to 0
    Make a new episode
    Observe the first state
    while not( done or truncated) do
        Choose $A$ from $S$ using policy derived from $Q$
         ($\epsilon$-greedy)
        Increment the total training step
        Execute $A$, observe $R$ and the new state $S'$
        Store Transition $< S, A, S', R >$ in $\mathcal{D}$
        Compute the $\mathcal{L}$ and do a gradient descent step
        if updating period then
            Copy the policy_net parameter to
             target_net
        end
    end
end
```

Algorithm 4: Training a DQN to estimate the optimal policy

B. Model evaluation

To evaluate the model, we run the agents in the environment for 1000 episodes, then compute the completion rate, the reward average, and the averaged steps for each method.

We also investigate these values for the training process to assess the speed and efficiency of each method, in addition to the plot of the loss and accumulated rewards.

VII. Results and Discussion

In this section we discus about our results an present them. We smoothed some of the curve using the exponential moving average wit a factor of 0.7, to obtain a better visualisation.

A. Q-Learning

After training the Q-Learning algorithm for 512 episodes, using a discount factor $\gamma = 0.9$, ϵ starting from 1 to 0.01 with decay rate of $\Delta = 3000$ and a learning rate $\alpha = 0.1$. We observe in Figure 4 the elution of the averaged cumulated squared TD-error and, reward, and steps.

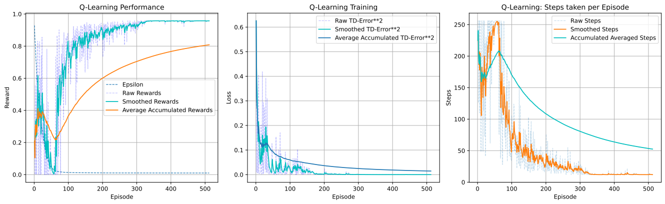

Fig. 4: Q-Learning per episode training metrics

	Train	Evaluation
Episodes	512	1000
Completion rate	92.97%	100%
Average rewards	0.808	0.958
Average steps	53	12

TABLE III: Training and evaluation summary for Q-Learning

	Train	Evaluation
Episodes	600	1000
Completion rate	95.67%	100%
Average rewards	0.844	0.958
Average steps	43	12

TABLE IV: Training and evaluation summary for SARSA-Learning

During the phase were we ϵ is big, the agent explore its environment. On it reach some point, the agent start to exploit the optimal policy found by algorithm. We resume the training and an evaluation phase in Table III, and the optimal policy is shown in figure 8.

B. SARSA-Learning

Similarly, we train the $SARSA$-Learning algorithm for 600 episodes, using a discount factor $\gamma = 0.9$, ϵ starting from 1 to 0.01 with decay rate of $\Delta = 3000$ and a learning rate $\alpha = 0.1$. We observe in Figure 5 the elution of the averaged cumulated squared TD-error and, reward, and steps.

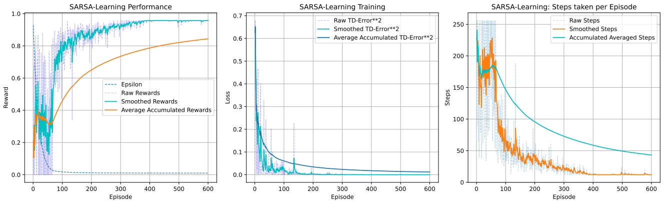

Fig. 5: SARSA-Learning per episode training metrics

We observe some similar pattern in the learning curve of the Q-Learning and SARSA. But SARSA required further episode before being stable. We resume the training and an evaluation phase in Table IV, and the optimal policy is shown in figure 8.

Overall the two Q-Learning and SARSA, performed well and give similar result, and yielded the same policy. However, as we can observe in Figure 8, several path with 12 steps is possible, and the choice of the agent seems to be the middle (nearest) one. It some how he summarize these path, but it may also because of the randomness introduced by the ϵ-greedy strategy.

Now the two next subsection presents our finding using deep learning technique.

C. Deep Q-Network Learning

We trained our Deep Q-Network for 2000 episodes, using a discount factor $\gamma = 0.9$, ϵ starting from 1 to 0.01 with decay

	Train	Evaluation
Episodes	2000	1000
Completion rate	99.2%	100%
Average rewards	0.912	0.958
Average steps	25	12

TABLE V: Training and evaluation summary for DQN

rate of $\Delta = 10^4$ and a learning rate $\alpha = 10^{-4}$. In addition, we use a replay memory of size 4096, and a batch 256. We synchronize the `policy_net` and `target_net` every 2048 steps of the agent. The evolution of the training process can be observed in Figure 6, it shows the averaged accumulated squared TD-error (MSE Loss) and, reward, and steps.

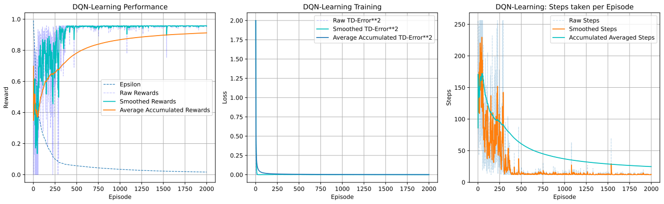

Fig. 6: DQN-Learning per episode training metrics

During the phase were we ϵ the network give some random value until it starts to find the right policy that he refine. Once ϵ is small enough, the agent start to exploit and refine the optimal policy found by algorithm. We resume the training and an evaluation phase in Table V, and the optimal policy is shown in figure 8.

Now, instead of feeding a vector, we use image as input and we report the results in the next section.

D. Deep Q-Network Learning with RGB Image

For this approach, we train the Deep Q-convolutional networks for 1500 episodes, using a discount factor $\gamma = 0.9$, ϵ starting from 1 to 0.01 with decay rate of $\Delta = 10^4$ and a learning rate $\alpha = 10^{-4}$. In addition, we use a replay memory of size 4096, and a batch 128. We synchronize the `policy_net` and `target_net` every 2048 steps of the agent. The evolution of the training process can be observed in Figure 7, it shows the averaged accumulated squared TD-error (MSE Loss) and, reward, and steps.

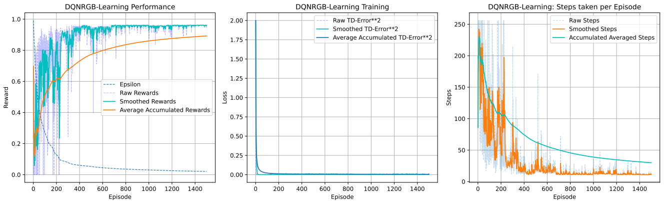

Fig. 7: DQN-Learning with RGB Image technique per episode training metrics

At the begging of the one episodes was slow. Then the duration of one epoch started slowly to decrease and become faster. This is due to the fact that the network/agent does not yet have any idea about the environment so it explore, (reson why we started with a higher value of ϵ). We resume the training and an evaluation phase in Table V, and the optimal policy is shown in figure 8.

	Train	Evaluation
Episodes	1500	1000
Completion rate	98%	100%
Average rewards	0.892	0.961
Average steps	30	11

TABLE VI: Training and evaluation summary for DQN-Image

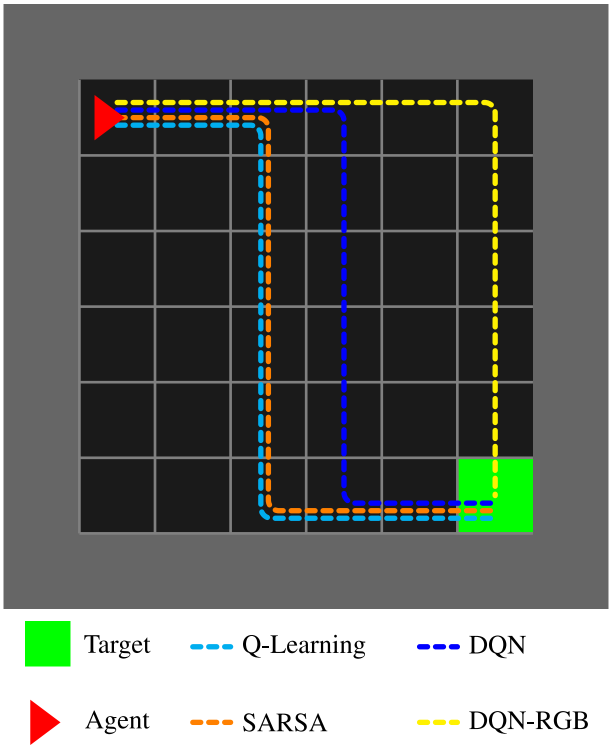

Fig. 8: Optimal Policy found by each algorithm

The two deep learning approach resulted in good performance, especially the DQ-CNN with 11 steps against 12 for the simple DQN. However, in terms of speed the simple DQN is faster, and may be with an efficient hyperameters tuning like grid search, we will possibly get same results for both methods.

VIII. Conclusion

In conclusion, we have explored four Reinforcement Learning algorithm. Q-Learning and SARSA for Tabular methods in one side and DQN and DQN-CNN for Deep leaning technique for reinforcement Learning in the other side. Each algorithm achieved 100% completion rate, with an average of 12 steps over 1000 episode for Q-Learning, SARSA, DQN, while the DQN-CNN did 11 steps, even it was a bit slow.

References

[1] Prof Herman Engelbrecht, Reinforcement Learning Lecture, Stellenbosch University, 2024.

[2] Richard S. Sutton, Andrew Barto, Reinforcement Learning: An Introduction.

[3] Farama Foundation, `minigrid` https://minigrid.farama.org/