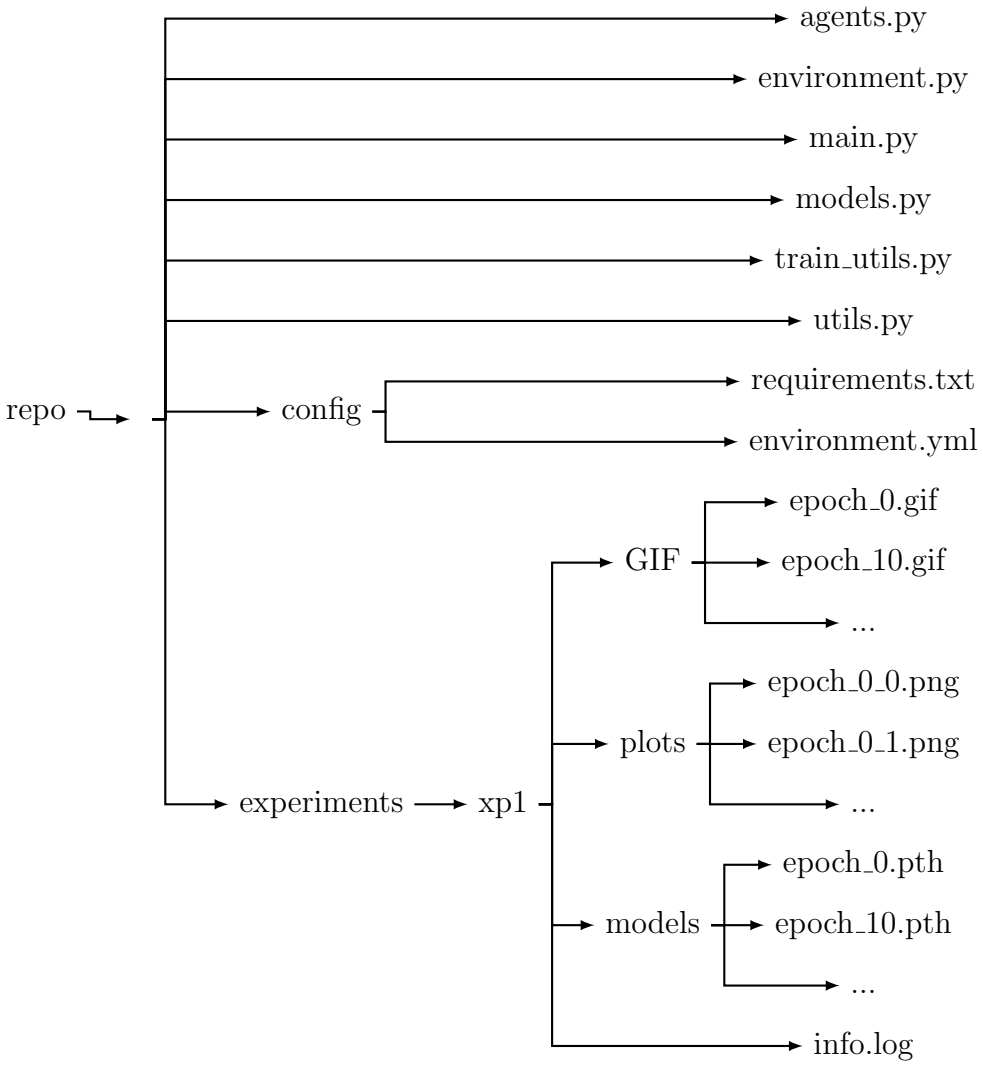
Deep Learning II **DL3AIISO**
PW4: Deep Reinforcement Learning: *From Notebook to Python Script*

Objective: The goal of the practical work is to learn how to manage a Deep Learning project using python scripts. During the session, we will implement a Deep Reinforcement Learning (DRL) and will go through different tools and techinques to improve the code quality and the project management.

Structure: The repository of the project is available at:
`https://www.lamsade.dauphine.fr/~averine/DL3AIISO/tp4.zip`. It contains the following files:



This code works for minimal example of a DRL project. Your task is to complete the implementation of the DQN algorithm.

---

1 Context

The goal of the project is to train a mouse to reach as much cheese in a grid filled with cheese and poison. The mouse can move in four directions: up, down, left, right. However, the mouse can only see two cells in each direction, thus a total of 5x5 cells. The mouse receives a positive reward when it reaches a cheese and a negative reward when it reaches a poison. The goal is to train the mouse to maximize the reward.

1.1 Mathematical formulation

The environment is a finite Markov Decision Process (MDP) of size T defined by the tuple (S, A, P, R, γ):

- S: set of states s. Naively, we can represent the state as a 5x5 grid of the board around the mouse. We will use a more detailed state representation in the code.
- A: set of actions a. The mouse can move in four directions: up, down, left, right. In the code, we will use the following encoding: 0 for up, 1 for down, 2 for left, 3 for right.
- $P(s'|s, a)$: transition probability. The environment can be deterministic if the mouse moves in the direction it wants to go and thus $P(s'|s, a) = 1$ if the mouse can move in the direction a and 0 otherwise. In the code, we will use a stochastic environment for training and a deterministic environment for evaluation.
- $r(s, a, s')$: reward function. The mouse receives a positive reward when it reaches a cheese and a negative reward when it reaches a poison. In this example the reward is a function of s' only and is deterministic.
- γ: discount factor. The discount factor is used to give more importance to the immediate reward than the future reward.
- T: horizon. The mouse can move for a fixed number of steps T.

In RL, the goal is to find a policy $\pi(a|s)$, the probability of taking action a in state s, that maximizes the expected reward:

$$R(\pi) = \mathbb{E}_{s_0,P,\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t, s_{t+1})\right]. \tag{1}$$

One way to find the optimal policy π^* is the Q-learning algorithm.

1.2 Q-learning

The Q-learning algorithm is a model-free reinforcement learning algorithm that learns the optimal action-value function $Q^{\pi^*}(s, a)$. The Q-value is the expected reward of taking action a in state s and following the policy π. The Q-value is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}_{P,\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s,\ a_0 = a\right]. \tag{2}$$

The Q-learning algorithm learns the optimal action-value function $Q^*(s, a)$, using the Bellman equation:

$$Q^{\pi^*}(s, a) = \mathbb{E}_{s' \sim \pi^*} \left[r(s, a, s') + \gamma \max_{a'} Q^{\pi^*}(s', a') \right]. \tag{3}$$

The Q-learning algorithm is an off-policy algorithm, meaning that it learns the optimal policy π^* while following a different policy π.

1.3 Deep Q-learning

The Q-learning algorithm can be extended to deep reinforcement learning by using a neural network Q_θ to approximate the Q-values. The target Q-value is defined as:

$$y = r(s, a, s') + \gamma \max_{a'} Q_{\theta^-}(s', a'), \tag{4}$$

where θ^- are the a fixed parameters of the target network. The loss function is defined as:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,s'} \left[(y - Q_\theta(s, a))^2 \right]. \tag{5}$$

It can be shown that the loss can be written as:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,s'} \left[\left(r(s, a, s') + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right)^2 \right]. \tag{6}$$

The Q-learning algorithm is an off-policy algorithm, meaning that it learns the optimal policy π^* while following a different policy π. We can sample some trajectories from the environment, store them in a replay buffer and train the agent using the loss function (6). We can use the following algorithm to train the agent:

1. Initialize the replay buffer D.
2. Initialize the Q-network Q_θ.
3. For each episode:
   (a) Initialize the state s.
   (b) For each step:
      i. The agent chooses an action a and get the reward r and the next state s'.
      ii. Store the transition (s, a, r, s') in the replay buffer D.
      iii. Sample a batch of transitions from the replay buffer D. (*Batched*)
      iv. Compute the target $y(a)$ using the target network Q_{θ^-}. (*Batched*)
      v. Add $\gamma + \max_{a'} y(a')$ to the target $y(a)$ if the episode is not done. (*Batched*)
      vi. Compute the loss using the loss function (6). (*Batched*)
      vii. Update the Q-network Q_θ using the loss function (6). (*Batched*)

With a Q-network learned, we can use the following policy to choose the action:

$$\pi(a|s) = \begin{cases} 1 & \text{if } a \in \arg\max_{a'} Q_\theta(s, a') \\ 0 & \text{otherwise} \end{cases}. \tag{7}$$

2 Project Management

2.1 Setting the python environment

For each deep learning project, you might need specific version of the libraries. To avoid conflicts between different projects, it is recommended to create a virtual environment for each project, i.e. a specific python environment with the required libraries. To do so you can use `virtualenv` or `conda`. You can find the files to configure the environment in the `config` folder of the repository.

To Do 1. *Create a virtual environment using either* ***`conda`*** *or* ***`virtualenv`****.*

To Do 2. *Activate the environment and install the required libraries using the file in the* ***`config`*** *folder.*

2.2 Running the code

The code is divided into different files:

- `main.py`: contains the main function to train/eval the agent.
- `agents.py`: contains the implementation of the DQN agent.
- `environment.py`: contains the implementation of the environment.
- `models.py`: contains the implementation of the neural network.
- `train_utils.py`: contains the implementation of the training loop.
- `utils.py`: contains utility functions.

To understand how the code works and have a detailed explanation of the different arguments, you can run the following command:

```
python main.py --help
```

Observe in the file `main.py` how the arguments are parsed using the `argparse` module. You can define the argument, its type, its default value, and its help message.

To Do 3. *Add an* ***`int`*** *argument to the* ***`main.py`*** *file to specify the maximum number T of steps per episode using the* ***`argparse`*** *module. Call the argument* ***`max_time`*** *with a default value of 100.*

To run the code, you can use the following command:

```
python main.py
```

It should run a naive agent (going up) for `n_epoch_eval`, compute the average reward. The code should save the last episode in the `GIF` folder and the plots in the `plots` folder.

To Do 4. *Run the code with the default arguments and observe the output in the* ***experiments*** *folder. Use the argument* ***--path naive*** *to specify the path of the experiment.*

By default, the code print the logs in the console. You can redirect the logs to a file using the module `logging`. The logs are saved in the `info.log` file in the experiment folder. Using the argument `--log_type`, you can specify what the function `log` should do.

To Do 5. *Implement the* ***logger*** *function in the* ***utils.py*** *file to save the logs in the* ***info.log*** *file. You just need to set the path of the log file. To do so, use* ***os.path.join*** *to concatenate the path of the experiment and the name of the log file.*

To Do 6. *Rerun the code with the naive agent and use the argument* ***--log_type file*** *to save the logs in the file and observe the output in the* ***experiments*** *folder.*

Logging the information in the console is useful to debug the code. However, it is not recommended to print the logs in the console when running the code on a server. Additionally, if you want to run the code in the backgroud On Windows, you can use the following command:

```
pythonw main.py --log_type file --path naive
```

On MacOS/linux, you can use the following command:

```
python main.py --log_type file --path naive &
```

To Do 7. *Run the command in the background and observe the output in the* ***experiments*** *folder.*

3 Implementation

3.1 The Random Agent

The file `agents.py` contains the implementation of the DQN agent. The framework to define as `class Agent`.

To Do 8. *Explain the role of* ***self.epsilon*** *in the* ***Agent*** *class and explain why it is essential.*

To Do 9. *By lookin at the method* `Environment.act` *of the class* `Environment` *in the file* `environment.py`*, explain the behavior of the naive agent when it hits the wall.*

The policy applied (and ideally) learned by the agent are implemented in the function `Agent.learned_act`. The action made by the agent is implemented in the function `Agent.act`.

To Do 10. *Implement the function* `Agent.learned_act` *in the file* `agents.py` *of the* `RandomAgent` *class. The function should return a random action.*

You can test your code by running the following command:

```
python main.py --agent random --path random
```

3.2 The Environment

The class `Environment` is used to simulate the environment. The environment is a grid of size `args.grid_size`×`args.grid_size` with a mouse and some cheese and poison.

To Do 11. *By looking at* `Environment.__init__` *and* `Environment.reset` *in the file* `environment.py`*, explain the role of the tensors* `self.board` *and* `self.position`*.*

To Do 12. *By looking at the method* `Environment.act` *of the class* `Environment` *in the file* `environment.py`*, explain the behavior of the naive agent when it hits the wall.*

For the moment, we will consider that the state observed by the mouse is both the board and the position of the mouse around the mouse. The state is a tensor of size $5 \times 5 \times 2$.

To Do 13. *In the methods* `Environment.reset` *and* `Environment.act`*, change the state to a tensor of size* $5 \times 5 \times 2$ *using the function* `Environment.get_state`*.*

3.3 The Neural Network

The property `self.model` of the class `Agent` is a neural network defined in the file `models.py`. The neural network is trained to approximate the Q-values. We will start with a simple neural network with three hidden layers of size 64 with ReLU activation functions between the layers.

To Do 14. *Implement the function* `DenseModel.__init__` *in the file* `models.py` *to define the neural network.*

To Do 15. *Using the equation* (7)*, implement the function* `DQN.learned_act` *in the file* `agents.py` *to compute the predicted action of the agent.*

Once the neural network is defined and the agent can choose an action, we can train the agent using the Q-learning algorithm. To do so, we use the function `train_model` in the file `train_utils.py`. We play `args.n_epoch` episodes and update the Q-network at each step using tuples of transitions stored in the replay buffer. Loading the replay from the buffer and computing the loss is done in the function `DQN.reinforce`.

To Do 16. *Using the equation* (6)*, complete the function* `DQN.reinforce` *in the file* `agents.py` *to compute the loss.*

To update the Q-network, we use an optimizer. The optimizer is loaded in the main function `main.py` using the function `get_optimizer` in the file `utils.py`.

To Do 17. *Complete the function* `get_optimizer` *in the file* `utils.py` *to load an SGD optimizer with a learning rate of* `args.lr`*, a momentum of 0.0 and a weight decay of 0.0001.*

4 Training Deep Q-learning Models

Now that the code is implemented, you can train the agent using the following command:

```
python main.py --agent fc --path fc --epoch 100 --epoch_eval 10 --lr 0.05
  --batch_size 32 --epsilon 0.2
```

4.1 Different models

To Do 18. *Train the agent using the command above and observe the output in the* `experiments` *folder.*

We can try a more complex model by using a convolutional neural network. The convolutional neural network is defined in the file `models.py`.

To Do 19. *Complete the function* `ConvModel.forward` *the file* `models.py` *to compute the Q-values using a convolutional neural network.*

You can use `--agent cnn` to use the convolutional neural network.

To Do 20. *Train the agent using a convolutional neural network and observe the output.*

4.2 ϵ-greedy policy

To train the model to explore, we need to randomize the action taken by the agent but we also want the model to exploit the knowledge learned. We can use an ϵ-greedy policy to balance the exploration and exploitation. The ϵ-greedy policy is defined in the function `Agent.act` in the file `agents.py`.

To push the agent to explore more, we can decrease the value of ϵ over time. In practice, the value of ϵ is decreased at every epoch, and during the epoch, the value of ϵ is decreased at every step. The value of ϵ is defined in the function `Agent.act` in the file `agents.py`.

We can try this method by using the flag `--explore`:

```
python main.py --agent fc --path fc_explore --explore
```

To Do 21. *Modify the function* ***`train_model`*** *in the file* ***`train_utils.py`*** *to include the decay of ϵ.*

Additionally, another way is to penalize the agent when it takes goes on the same cell. We can use a large state representation of size $5 \times 5 \times 3$ where the third channel is the number of time the mouse has been on the cell.

To Do 22. *Modify the function* ***`Environment.act`*** *in the file* ***`environment.py`*** *to penalize the agent when it goes on the same cell using the property* ***`self.malus_position`****.*

To Do 23. *Test your code using the flag* ***`--explore`****.*