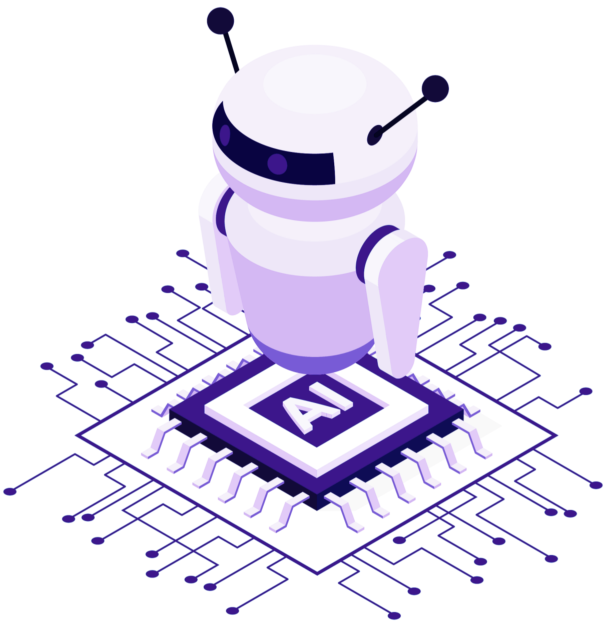

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 10:

Exploration Methods

Designed By:

Alireza Nobakht
a.nobakht13@gmail.com

Faezeh Sadeghi
fz.saadeghi@gmail.com

Benyamin Naderi
benjaminndr79@gmail.com

Spring 2025

Preface

One of the defining challenges in deep reinforcement learning (DRL) is effective exploration. Unlike other deep learning paradigms such as supervised or unsupervised learning, where models are trained on static datasets, DRL agents must collect their own data by interacting with an environment. This interaction-driven learning process makes exploration essential—without it, agents may fail to encounter the diverse and informative experiences needed for learning. In contrast, traditional deep learning models operate under the assumption that all necessary information is already available in the data. In DRL, the agent's ability to learn is directly influenced by how well it explores, making the process of data collection a core component of the learning itself.

What makes exploration particularly difficult in DRL is the nature of the environments in which agents operate. One major challenge is the sparsity of rewards—agents often receive little or no feedback for long sequences of actions, making it hard to determine which behaviors are beneficial. In such cases, the agent must persist in exploring without any immediate indication of progress. Furthermore, the state and action spaces can be vast or continuous, with only a small fraction leading to meaningful outcomes. Without exploration, the agent risks getting stuck in limited or suboptimal regions of the environment. This combination of delayed feedback, large search spaces, and uncertainty about what to explore makes exploration not only essential but also one of the most demanding aspects of training effective DRL agents.

In this homework, you will implement two algorithms designed to address the exploration problem in DRL: Bootstrap DQN variants and Random Network Distillation. These methods have been chosen because they represent distinct approaches to improving exploration, grounded in the challenges described above.

Grading

The grading will be based on the following criteria, with a total of 200 points:

Task	**Points**
Task 1	100
Task 2	100
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus	$20 - 80$

Submission

The deadline for this homework is 1404/03/16 (June 6th 2025) at 11:59 PM.

Please submit your work by following the instructions below:

- Place your solution alongside the Jupyter notebook(s).
  - Your written solution must be a single PDF file named `HW10_Solution.pdf`.
  - If there is more than one Jupyter notebook, put them in a folder named `Notebooks`.
- Zip all the files together with the following naming format:

  `DRL_HW10_[StudentNumber]_[FullName].zip`
  - Replace `[FullName]` and `[StudentNumber]` with your full name and student number, respectively. Your `[FullName]` must be in CamelCase with no spaces.
- Submit the zip file through Quera in the appropriate section.
- We provided this LaTeX template for writing your homework solution. There is a 5-point bonus for writing your solution in LaTeX using this template and including your LaTeX source code in your submission, named `HW10_Solution.zip`.
- If you have any questions about this homework, please ask them in the Homework section of our Telegram Group.
- If you are using any references to write your answers, consulting anyone, or using AI, please mention them in the appropriate section. In general, you must adhere to all the rules mentioned here and here by registering for this course.

Keep up the great work and best of luck with your submission!

Contents

1 Setup Instructions

1.1 Environment Setup

The detailed environment setup instructions for each task are provided within their respective sections of this document and the accompanying Jupyter notebooks. For Task 1 (Bootstrap DQN Variants), refer to the relevant section for guidance on installing dependencies and configuring the environment. Similarly, Task 2 (Random Network Distillation) includes its own setup steps and requirements. Please consult each task's section to ensure you follow the correct procedures for preparing your development environment.

1.2 Submission Requirements

Please note that the specific submission requirements for each task are detailed within their respective sections of this document. Be sure to review the instructions provided in each task description to ensure your submission meets all necessary criteria.

2 Problem Descriptions

2.1 Task 1: Bootstrap DQN Variants

2.1.1 Bootstrap DQN

Bootstrap Deep Q-Network (Bootstrap DQN) [2] is an extension of the standard DQN algorithm aimed at improving exploration through ensemble-based uncertainty estimation. Instead of training a single Q-network, Bootstrap DQN maintains an ensemble of Q-value "heads" sharing a common feature extractor but trained on different subsets of experience using bootstrapped masks. At the beginning of each episode, one head is randomly selected to guide the agent's actions using ε-greedy exploration. This structured randomness introduces a more consistent exploration strategy across time, known as *deep exploration*, which helps the agent discover rewarding strategies that might be missed by purely random exploration. By leveraging the diversity among heads, Bootstrap DQN improves both learning stability and performance in environments with sparse or deceptive rewards.

Algorithm 1 BootstrapDQN - Training

```
1: Input: minibatch size $K$, mask probability $p_m$, ensemble of $N$ $Q$-networks $\mathcal{Q}$, ensemble of $N$ target-networks $\mathcal{T}$
2: for each episode do
3:     Pick a $Q$-network from $\mathcal{Q}$ using $i \sim \text{Uniform}(1..N)$
4:     for each time step do
5:         $a \leftarrow \arg\max_\alpha \mathcal{Q}_i(s, \alpha)$
6:         Collect next-state $s'$ and reward $r$ from the environment by taking action $a$
7:         Sample bootstrap mask $M = m_l \sim \{\text{Bernoulli}(p_m) | l \in 1, ..., N\}$
8:         Add $(s, a, s', r, M)$ in buffer $\mathcal{B}$
9:         $s \leftarrow s'$
10:        Sample minibatch $D$ from replay buffer $\mathcal{B}$
11:        for $j$ in $[1 \dots N]$ do                                    ▷ Each member of the ensemble
12:            for each tuple $(s_k, a_k, s'_k, r_k, M_k)$ in minibatch $D$ do    ▷ Computing the targets
13:                $\hat{Q}_k \leftarrow \mathcal{Q}_j(a_k, s_k)$                  ▷ Expected $Q$-value
14:                $a'_k \leftarrow \arg\max_a \mathcal{T}_j(s'_k, a)$
15:                $T_k \leftarrow r_k + \gamma \mathcal{T}_j(s'_k, a'_k)$             ▷ Target
16:            end for
17:            Mask minibatch $D$ using $M_j$
18:            Update $\mathcal{Q}_j$'s parameters $\theta_j$ by minimizing $L2$ loss: $(\hat{Q} - T)^2$ over $D$
19:            Soft update of the $\mathcal{T}_j$'s parameters $\bar{\theta}_j \leftarrow (1 - \tau)\bar{\theta}_j + \tau\theta_j$
20:        end for
21:    end for
22: end for
```

2.1.2 Randomized Prior Functions (RPF)

To further enhance the exploratory behavior of Bootstrap DQN, *Randomized Prior Functions* (RPF) [3] can be added to each Q-network head. This technique introduces a fixed, randomly initialized and untrainable prior network whose output is added to the corresponding head's learned Q-values during both training and action selection. The purpose of the prior function is to inject structured, persistent randomness into the value estimates, encouraging each head to maintain distinct long-term behavior even early in training when data is scarce. By combining bootstrapped data exposure, ε-greedy exploration, and a unique prior function per head, this approach leads to richer and more temporally extended exploration—which is especially beneficial in environments with sparse rewards or delayed credit assignment.

Algorithm 2 BootstrapDQN with Randomized Prior Function. in blue, the differences with BootstrapDQN.

```
1:  Input: RPF scale $\delta_{\text{RPF}}$, minibatch size $K$, mask probability $p_m$, ensemble of $N$ $Q$-networks $\mathcal{Q}$, ensemble of $N$ target-networks $\mathcal{T}$, ensemble of $N$ prior-networks $P$
2:  for each episode do
3:      Pick a $Q$-network from $\mathcal{Q}$ using $i \sim \text{Uniform}\,(1..N)$
4:      for each time step do
5:          $a \leftarrow \arg\max_\alpha \left(\mathcal{Q}_i(s,\alpha) + \delta_{\text{RPF}} P_i(s,\alpha)\right)$    ▷ Random prior
6:          Collect next-state $s'$ and reward $r$ from the environment by taking action $a$
7:          Sample bootstrap mask $M = m_l \sim \{\text{Bernoulli}(p_m) | l \in 1, ..., N\}$
8:          Add $(s, a, s', r, M)$ in buffer $\mathcal{B}$
9:          $s \leftarrow s'$
10:         Sample minibatch $D$ from replay buffer $\mathcal{B}$
11:         for $j$ in $[1 \dots N]$ do    ▷ Each member of the ensemble
12:             for each tuple $(s_k, a_k, s'_k, r_k, M_k)$ in minibatch $D$ do    ▷ Computing the targets
13:                 $\hat{Q}_k \leftarrow \mathcal{Q}_j(a_k, s_k) + \delta_{\text{RPF}} P_j(s_k, a_k)$    ▷ Expected $Q$-value
14:                 $a'_k \leftarrow \arg\max_a \left(\mathcal{T}_j(s'_k, a) + \delta_{\text{RPF}} P_j(s'_k, a)\right)$
15:                 $T_k \leftarrow r_k + \gamma(\mathcal{T}_j(s'_k, a'_k) + \delta_{\text{RPF}} P_j(s'_k, a'_k))$    ▷ Target
16:             end for
17:             Mask minibatch $D$ using $M_j$
18:             Update $\mathcal{Q}_j$'s parameters $\theta_j$ by minimizing $L2$ loss: $(\hat{Q} - T)^2$ over $D$
19:             Soft update of the $\mathcal{T}_j$'s parameters $\bar{\theta}_j \leftarrow (1-\tau)\bar{\theta}_j + \tau\theta_j$
20:         end for
21:     end for
22: end for
```

2.1.3 Batch Inverse Variance (BIV)

Building on this, *Batch Inverse Variance* (BIV) [4] weighting can be incorporated into the RPF Bootstrap DQN framework to further improve sample efficiency. In this variant, the target Q-values are computed using a weighted average across the ensemble heads, where each head's contribution is inversely proportional to the variance of its predictions for a given state-action pair. This method places greater emphasis on more certain predictions while still leveraging the ensemble's diversity. By reducing the influence of noisy or uncertain estimates, BIV leads to more stable and data-efficient updates, enhancing learning performance particularly in environments with limited or costly interactions.

Algorithm 3 Batch Inverse Variance DQN. In red, the differences with BootstrapDQN with RPF.

```
1: Input: RPF scale $\delta_{\text{RPF}}$, minibatch size $K$, mask probability $p_m$, ensemble of $N$ $Q$-networks $\mathcal{Q}$, ensemble of $N$ target-networks $\mathcal{T}$, ensemble of $N$ prior-networks $P$, paramter $\xi$
2: for each episode do
3:     Pick a $Q$-network from $\mathcal{Q}$ using $i \sim \text{Uniform}(1..N)$
4:     for each time step do
5:         $a \leftarrow \arg\max_\alpha (\mathcal{Q}_i(s,\alpha) + \delta_{\text{RPF}} P_i(s,\alpha))$    ▷ Random prior
6:         Collect next-state $s'$ and reward $r$ from the environment by taking action $a$
7:         Sample bootstrap mask $M = m_l \sim \{\text{Bernoulli}(p_m) | l \in 1, ..., N\}$
8:         Add $(s, a, s', r, M)$ in buffer $\mathcal{B}$
9:         $s \leftarrow s'$
10:        Sample minibatch $D$ from replay buffer $\mathcal{B}$
11:        for $j$ in $[1 \dots N]$ do    ▷ Each member of the ensemble
12:            for each tuple $(s_k, a_k, s'_k, r_k, M_k)$ in minibatch $D$ do    ▷ Computing the targets
13:                $\hat{Q}_k \leftarrow \mathcal{Q}_j(a_k, s_k) + \delta_{\text{RPF}} P_j(s_k, a_k)$    ▷ Expected $Q$-value
14:                $a'_k \leftarrow \arg\max_a (\mathcal{T}_j(s'_k, a) + \delta_{\text{RPF}} P_j(s'_k, a))$
15:                $T_k \leftarrow r_k + \gamma(\mathcal{T}_j(s'_k, a'_k) + \delta_{\text{RPF}} P_j(s'_k, a'_k))$    ▷ Target
16:                for $l$ in $[1 \dots N]$ do    ▷ Computing action-state value of target of each network
17:                    $x_{j,k,l} \leftarrow \mathcal{T}_l(s'_k, a'_k) + \delta_{\text{RPF}} P_l(s'_k, a'_k)$
18:                end for
19:                $\sigma^2_{j,k} = \text{var}(x_{j,k,l})$    ▷ Uncertainty of the target for network $j$ on sample $k$
20:                $w_{j,k} = 1/(\gamma^2 \sigma^2_{j,k} + \xi)$
21:            end for
22:            Mask minibatch $D$ using $M_j$
23:            Update $\mathcal{Q}_j$'s parameters $\theta_j$ by minimizing $L2$ loss: $(\sum_k w_{j,k})^{-1} w_{j,k} (\hat{Q} - T)^2$ over $D$    ▷ Weighted loss
24:            Soft update of the $\mathcal{T}_j$'s parameters $\bar{\theta}_j \leftarrow (1-\tau)\bar{\theta}_j + \tau\theta_j$
25:        end for
26:    end for
27: end for
```

2.1.4 Implementation and Submission

The complete guidelines for implementing the Bootstrap DQN algorithm, including the RPF and BIV variants, are provided in the Jupyter notebook. You will find detailed instructions on how to set up the environment, implement the algorithms, and evaluate their performance. Make sure to read **Guidelines** section in the notebook carefully.

2.2 Task 2: Random Network Distillation (RND)

2.2.1 Introduction

A common way of doing exploration is to visit states with a large prediction error of some quantity, for instance, the TD error or even random functions. The RND algorithm [5] aims at encouraging exploration by asking the exploration policy to more frequently undertake transitions where the prediction error of a random neural network function is high.

Formally, let $f^*_\theta(s')$ be a randomly chosen vector-valued function represented by a neural network. RND trains another neural network, $\hat{f}_\phi(s')$, to match the predictions of $f^*_\theta(s')$ under the distribution of datapoints in the buffer, as shown below:

$$\phi^* = \arg\min_\phi \mathbb{E}_{s,a,s'\sim\mathcal{D}}\left[\left\|\hat{f}_\phi(s') - f^*_\theta(s')\right\|\right]$$

If a transition (s, a, s') is in the distribution of the data buffer, the prediction error $\mathcal{E}_\phi(s')$ is expected to be small. On the other hand, for all unseen state-action tuples, it is expected to be large.

In practice, RND uses two critics: - an exploitation critic $Q_R(s, a)$, which estimates returns based on the true rewards, - and an exploration critic $Q_E(s, a)$, which estimates returns based on the exploration bonuses.

To stabilize training, prediction errors are normalized before being used.

2.2.2 What You Will Implement

You will implement the missing core components of Random Network Distillation (RND) combined with a Proximal Policy Optimization (PPO) agent inside the MiniGrid environment.

Specifically, you will:

- Complete the architecture of TargetModel and PredictorModel.

- Complete the initialization of weights for these models.

- Implement the intrinsic reward calculation (prediction error).

- Implement the RND loss calculation.

You will complete TODO sections inside two main files:

```
Core/ppo_rnd_agent.py
Core/model.py
```

2.2.3 Environment Description: `MiniGrid-Empty-5x5-v0`

The environment used in this project is `MiniGrid-Empty-5x5-v0`, which is part of the `MiniGrid` benchmark suite. This environment consists of a small 5×5 empty grid world where a single agent navigates to reach a goal location. The observation provided to the agent is a partial view of the surrounding grid encoded as a $7 \times 7 \times 3$ tensor representing RGB channels.

This environment is particularly suitable for studying exploration strategies in sparse-reward settings because:

- Rewards are only given upon reaching the goal.
- The map is small but partially observable.
- It tests both spatial understanding and long-term exploration.

Environment Setup in Code

The environment is defined and constructed in the `Common/utils.py` module via the `make_env` function:

```
def make_env(env_id, max_episode_steps=None):
    def _init():
        env = gym.make(env_id)
        if max_episode_steps:
            env._max_episode_steps = max_episode_steps
        return env
    return _init
```

The specific environment ID `MiniGrid-Empty-5x5-v0` is passed through the configuration file `Common/config.py` as a default parameter:

```
default_params = {
    "env_name": "MiniGrid-Empty-5x5-v0",
    ...
}
```

This environment is then instantiated in both the `main.py` script for training and the `play.py` module for evaluation using this configuration.

2.2.4 Project Structure

```
RND_PPO_Project/
 main.py              # Main training loop and evaluation
 requirements.txt     # Python dependencies
 Core/
    ppo_rnd_agent.py        # Agent logic (policy + RND + training)
    model.py         # Model architectures (policy, predictor, target)
 Common/
    config.py        # Hyperparameters and argument parsing
    utils.py         # Utilities (normalization, helper functions)
    logger.py        # Tensorboard logger
    play.py          # Evaluation / Play script
```

2.2.5 Modules Explanation

Module	**Description**
`ppo_rnd_agent.py`	**Core agent logic.** Contains the PPO algorithm and handles RND intrinsic reward. It manages action selection, GAE, reward normalization, and model training. **You will implement intrinsic reward and RND loss functions here.**
`model.py`	**Neural network architectures.** Defines the policy, target, and predictor networks. **You will define `TargetModel` and `PredictorModel` and initialize them.**
`utils.py`	**Support utilities.** Includes random seeding, running mean/variance for normalization, and decorators.
`config.py`	**Experiment settings.** Contains hyperparameters and argument parsing.
`logger.py`	**Logging.** Logs rewards, losses, and other metrics to TensorBoard.
`play.py`	**Evaluation.** Runs a trained policy in the environment.
`runner.py`	**Parallel environment interaction.** Runs a Gym environment in a separate process using `torch.multiprocessing`. It communicates with the main process to exchange observations and actions, enabling parallel experience collection.
`main.py`	**Project entry point.** Sets up everything and launches training or evaluation.

Table 1: Overview of project modules and responsibilities.

2.2.6 TODO Parts (Your Tasks)

You must complete the following parts:

File	**TODO Description**
`Core/model.py`	Implement the architecture of `TargetModel` and `PredictorModel`.
`Core/model.py`	Implement `_init_weights()` method for proper initialization.
`Core/ppo_rnd_agent.py`	Implement `calculate_int_rewards()` to compute intrinsic rewards.
`Core/ppo_rnd_agent.py`	Implement `calculate_rnd_loss()` to compute predictor training loss.

Table 2: Summary of required TODO implementations

2.2.7 How to Complete the Homework

To complete the homework, follow the steps outlined in the provided notebook. These include:

1. Installing the required dependencies using `requirements.txt`.
2. Completing the specified TODOs in the source files.
3. Training the PPO agent using the RND exploration strategy.

4. Visualizing the training results using TensorBoard.
5. Once training is complete, try experimenting with different hyperparameters to observe their effects on performance.

These steps ensure you correctly implement and test all the required components of the project.

2.2.8 Questions to Answer

At the end of the homework, please reflect on and answer the following conceptual and experimental questions:

1. What is the intuition behind Random Network Distillation (RND)? Why does a prediction error signal encourage better exploration?
2. Why is it beneficial to use both intrinsic and extrinsic returns in the PPO loss function?
3. What happens when you increase the `predictor_proportion` (i.e., the proportion of masked features used in the RND loss)? Does it help or hurt learning?
4. Try training with `int_adv_coeff=0` (removing intrinsic motivation). How does the agent's behavior and reward change?
5. Inspect the TensorBoard logs. During successful runs, how do intrinsic rewards evolve over time? Are they higher in early training?

Answering these questions will help solidify your understanding of exploration methods and their practical impact.

References

[1] Cover image designed by freepik

[2] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016. arXiv:1602.04621

[3] Ian Osband, John Aslanides, and Albin Cassirer. Randomized Prior Functions for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. arXiv:1806.03335

[4] Vincent Mai, Kaustubh Mani, Liam Paull. Sample Efficient Deep Reinforcement Learning via Uncertainty Estimation In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2201.01666

[5] Yuri Burda, Harrison Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Exploration by Random Network Distillation. In *International Conference on Learning Representations (ICLR)*, 2019. arXiv:1810.12894