Universitat Autònoma de Barcelona

Bachelor's Degree in Artificial Intelligence
Deep Reinforcement Learning Project

Project Report: From Pong to Complex Environments

Juan Sebastian Mañosas Guerrero Alveolos (NIU: 1671913)
Mateo Jure (NIU: 1705977)
Aran Oliveras (NIU: 1708069)

Abstract

This report details our journey in applying Deep Reinforcement Learning (DRL) techniques across three distinct challenges. We begin with a brief overview of solving the classic Pong environment (Part 1), move to the implementation of a multi-agent competitive agent for the Pong World Tournament (Part 2), and conclude with the solution to a complex ALE environment (Part 3). The focus of this document is on the practical implementation, hyperparameter tuning, and the iterative problem-solving process we undertook.

Contents

1 Part 1: Solving the Pong Environment

1.1 Preprocessing and Wrappers (Question 1)

To prepare the `PongNoFrameskip-v4` environment for training, we applied a standard set of preprocessing steps common in Atari DRL literature. These transformations are essential to reduce the computational complexity and provide temporal context to the agent.

We utilized the `gymnasium.wrappers` library to apply the following:

- **AtariPreprocessing**: This wrapper handles the bulk of the reduction. It resizes the raw 210×160 RGB frames to an 84×84 grayscale image. It also scales the pixel values to a range of $[0, 1]$.
- **FrameStackObservation**: We stack $k = 4$ consecutive frames. Since a single frame captures position but not velocity (direction and speed of the ball/paddles), stacking frames creates a Markovian state that allows the agent to perceive motion.

Resulting Observation Space: The final observation passed to the neural network is a tensor of shape $(4, 84, 84)$, representing 4 stacked grayscale frames.

1.2 Selected Agents and Models (Question 2)

For this comparative study, we selected two distinct Deep Reinforcement Learning algorithms:

1. **Deep Q-Network (DQN):** Used as our baseline, we utilized the implementation provided in the course examples. DQN is an off-policy algorithm that uses a replay buffer and a target network to stabilize the learning of the Q-value function.
2. **Proximal Policy Optimization (PPO):** We implemented a custom PPO agent from scratch using PyTorch. PPO is an on-policy Actor-Critic method. It is generally more stable than DQN and easier to tune due to its clipped objective function, which prevents the policy from changing too drastically in a single update.

Architecture of the PPO Agent: The agent consists of a shared Convolutional Neural Network (CNN) feature extractor followed by split Actor and Critic heads:

- **Feature Extractor:** Three convolutional layers.
  - Conv2d(4, 32, kernel=8, stride=4) + ReLU
  - Conv2d(32, 64, kernel=4, stride=2) + ReLU
  - Conv2d(64, 64, kernel=3, stride=1) + ReLU
- **Heads:** The output is flattened and passed to a linear layer (512 units), which branches into the Actor (logits for action probabilities) and the Critic (scalar value estimate).

1.3 Training and Fine-tuning (Question 3)

Training was conducted using `wandb` (Weights & Biases) to perform a hyperparameter sweep. We used a Bayesian search strategy to maximize the episodic return.

Hyperparameter Sweep: We varied the following parameters to find the optimal configuration:

- **Learning Rate:** Range $[1e-5, 1e-3]$
- **Entropy Coefficient:** Range $[0.0, 0.02]$ (to encourage exploration)
- **Clip Coefficient:** Range $[0.1, 0.3]$
- **Num Steps (Rollout length):** $\{128, 256\}$

The training utilized `AsyncVectorEnv` to run parallel agents (determined dynamically by CPU availability, typically 4-8 agents), significantly speeding up the collection of experience. The total timesteps were fixed at 1.5 million per run.

Best Configuration: The sweep identified the following optimal hyperparameters which were used for the final evaluation:

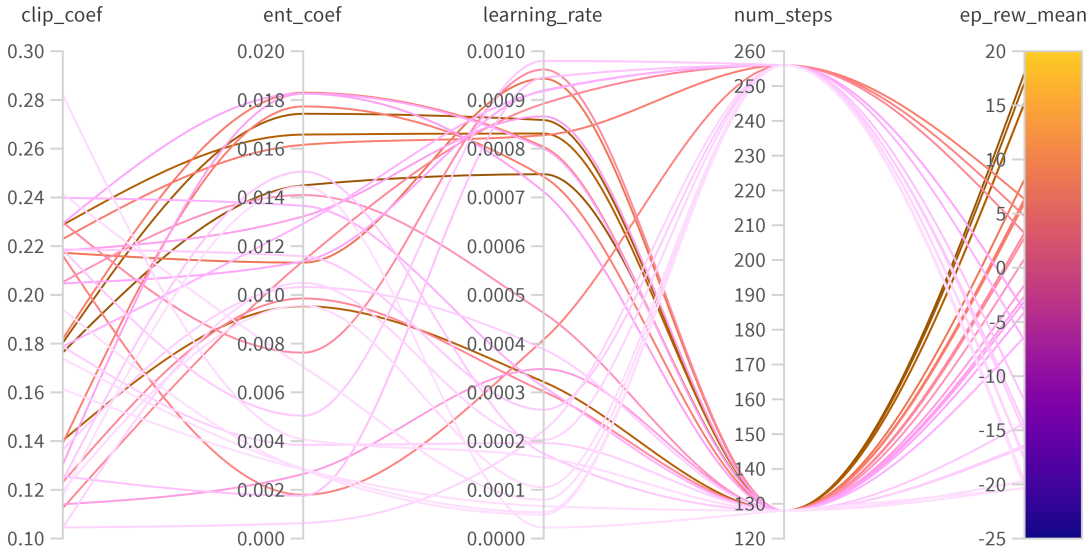
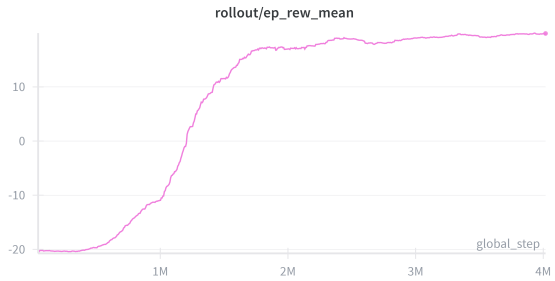


Figure 1: Parallel coordinates plot of the hyperparameter sweep. Each line represents a single training run, mapping the hyperparameters (Clip Coefficient, Entropy Coefficient, Learning Rate, and Num Steps) to the mean episodic reward. Brighter colors (yellow) indicate higher returns, visually highlighting the optimal parameter configuration.

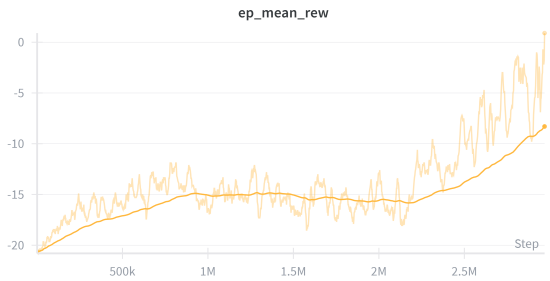
- Learning Rate: $\sim$**9e-4**
- Entropy Coef: $\sim$**0.016**
- Clip Coefficient: $\sim$**0.18**
- Num Steps: **128**.

1.4 Visualizations and Learning Curves (Question 4)

The following figures illustrate the training progress.



(a) PPO Agent (Pink)



(b) DQN Agent (Orange)

Figure 2: Comparison of Average Episodic Reward. The PPO agent (Pink) demonstrates stable convergence, reaching a score of 18+ by 2 million steps. In contrast, the DQN agent (Orange) struggles to break past -10 rewards within the same timeframe, exhibiting higher variance.

As observed, the PPO agent's ability to learn from parallel environments allowed it to stabilize much faster than the sequential DQN implementation.

1.5 Evaluation and Success Rate (Question 5)

After training, the best models were evaluated over 100 test episodes.

Model	Avg Reward (0-21)	Win Rate (%)
DQN (Baseline)	19.86	**100%**
PPO (Ours)	**20.44**	**100%**

Table 1: Evaluation results on PongNoFrameskip-v4

The PPO agent consistently achieves a perfect score of 21 (or near 21), effectively shutting out the hard-coded opponent.

1.6 Results Analysis and Justification (Question 6)

While both models successfully solved the environment (defined as achieving an average reward > 18), **PPO was selected as the superior model** for the following reasons:

1. **Wall-clock Training Time:** Thanks to the `AsyncVectorEnv`, PPO collected data in parallel, reducing the real-world time required to reach convergence compared to the serial execution of DQN.

2. **Stability:** The hyperparameter sweep showed that PPO was less sensitive to initialization seeds compared to DQN, which often suffered from catastrophic forgetting if the replay buffer was not managed perfectly.

Therefore, we proceed to Part 2 with the confidence that our PPO implementation is robust.

2 Part 2: Pong World Tournament

2.1 The Philosophy: "To Win the Tournament, Play the Tournament"

Our approach to this challenge was driven by a simple yet ambitious philosophy: *"To win Pong, we must play Pong; but to win a tournament, we must play a tournament."*

Instead of training a single agent to overfit the standard deterministic computer opponent or two to overfit each other, we aimed to create a custom **Multi-Agent League**. The goal was to foster an evolutionary dynamic where a diverse population of agents would play against each other, preventing them from learning exploitable, static strategies.

2.2 Model Selection (Question 1)

We selected **Phasic Policy Gradient (PPG)** as our primary champion model, augmented with an **ImpalaCNN** architecture.

- **Why PPG?** Standard PPO often struggles to balance the conflicting objectives of policy learning and value estimation when sharing a network. PPG solves this by decoupling the training into distinct "policy" and "auxiliary" phases, allowing for better feature extraction without distorting the policy. [1]
- **Why ImpalaCNN?** The standard "Nature CNN" (3 convolutional layers) is often insufficient for capturing complex temporal dynamics in competitive play. We implemented the ImpalaCNN (based on the IMPALA paper[2]), which uses residual blocks to allow for much deeper networks without vanishing gradients.

The Custom Implementation Necessity: We could not use standard libraries like Stable Baselines 3 because they lack native support for PPG and do not easily support the complex custom architectures (like ImpalaCNN) we required. Consequently, we built our agents from scratch using PyTorch.

2.3 The Journey: From "Grand League" to "The Arena" and "The Duel"

2.3.1 The Initial Ambition: 12 Warriors

Our original plan was massive. We designed a `LeagueAgent` interface to standardize interactions and planned to train 12 distinct models per side (Left/Right) simultaneously:

- **Algorithms:** Rainbow, A3C, PPG, DQN, A2C, PPO.
- **Backbones:** Nature CNN vs. Impala CNN.

We attempted to automate the generation of these agents using an LLM-assisted workflow, feeding it our strict interface template. However, this approach failed. Integrating on-policy algorithms (like PPO) and off-policy algorithms (like DQN/Rainbow) into a single synchronous training loop proved mathematically and computationally unstable. The code was plagued with bugs, and the overhead of managing 24 distinct computation graphs was unmanageable given our time constraints.

2.3.2 Scope Reduction and "The Arena"

Facing these roadblocks, we pivoted to a robust, smaller-scale solution we call "The Arena" (implemented in `arena.py`). We reduced the roster to 5 distinct agent archetypes per side:

1. **PPO (Impala):** Standard on-policy baseline with deep vision.
2. **PPG (Nature):** Advanced algorithm with standard vision.
3. **PPG (Impala):** Our theoretical "best model."
4. **DQN (Impala):** Off-policy representation.
5. **Alpha/Beta:** Specialized versions of PPG-Impala trained for longer durations to serve as "Bosses."

2.3.3 Training Strategy and Hardware Constraints

Our training process evolved through three stages:

1. **Pre-training (Gymnasium):** We first trained all agents against the standard computer opponent (using a "Flip" wrapper for Left-side agents) until they reached a mean reward of ≈ 0. This ensured they understood the game physics before facing human-like opponents.
2. **The Arena (PettingZoo):** We moved the pre-trained agents into a 5v5 round-robin league.
3. **Cluster Bottleneck:** We encountered a severe hardware limitation, our cluster allocation was recently capped at 6 CPU cores. Running 10 parallel agents (5v5), 25 matches, became excrutiatingly slow. To survive this, we implemented a "Mixed Batch" strategy: rotating agents in sub-groups (2v3, then 3v2) and focusing heavily on the "Alpha vs. Beta" matchup to drive high-level play.

2.3.4 Scope Reduction and "The Duel"

Realizing the Arena was too resource-intensive to yield results in time, and facing complex bugs related to the synchronization of on-policy and off-policy algorithms, we made a final strategic pivot. We returned to our robust **PPG implementation**, which we had originally built for single-player pre-training in Gymnasium.

We adapted this code to the multi-agent PettingZoo environment, stripping away the complex league logic in favor of a focused **1v1 self-play training loop**. By creating two instances of our best PPG agent and training them directly against each other, we maximized the efficiency of our limited CPU resources while maintaining the competitive "arms race" dynamic we originally sought.

However, this strategic pivot was made late in the development cycle. Consequently, the training time available for the final "Duel" configuration was constrained, highlighting the importance of early resource estimation in multi-agent reinforcement learning.

2.4 Hyperparameter Tuning (Question 2)

Due to the custom nature of our codebase and the extreme computational cost of the multi-agent league, running an automated sweep (like in Part 1) was impossible.

Instead, we adopted a **literature-driven approach**, sourcing parameters from the original papers and our own Part 1 experiments:

- **Learning Rates:** Derived from the original PPG paper ($5e-4$) and our Part 1 PPO experiments ($2.5e-4$).
- **Entropy Coefficient:** Set to 0.01 to force exploration in the early game.
- **Clip Coefficient:** 0.2 (Standard PPO baseline).

Parameter	Value	Source/Justification
Learning Rate (PPG)	$5e-4$	Standard PPG Literature
Learning Rate (DQN)	$1e-4$	Stability for Impala Deep Net
Gamma	0.99	Standard for Pong
Parallel Envs	16	Max supported by memory

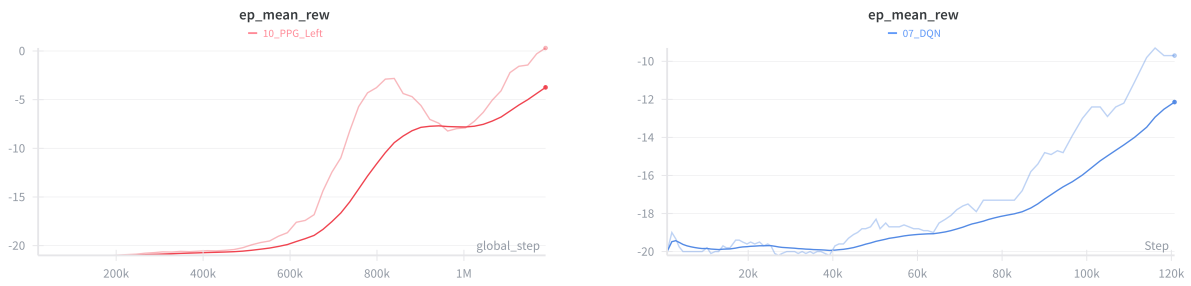
Table 2: Final Hyperparameters used

2.5 Training and Single-Player Results (Question 3)

Before entering the multi-agent arena, agents were validated in the standard `PongNoFrameskip-v4`.

Pre-training Performance:

- **PPG (Impala):** Reached parity (score ≈ 0) with the computer, successfully learning to rally without overfitting to the script.
- **DQN (Impala):** Converged slower (in time) but in less global steps.



(a) PPG Agent (Red)

(b) DQN Agent (Blue)

Figure 3: Pre-training performance on `PongNoFrameskip-v4`. (a) The **PPG agent** (Red) successfully converges to a mean reward of ≈ 0 within 1.2 million steps, indicating it can rally effectively against the baseline. (b) The **DQN agent** (Blue) demonstrates high sample efficiency, reaching a mean reward of -12 in just 120k steps, validating the effectiveness of off-policy learning for early feature extraction.

2.6 Agent Export and Loading (Question 4)

We implemented a standardized `save` and `load` method in our `LeagueAgent` parent class. This allows any agent (DQN or PPG) to be loaded via a unified interface, regardless of its underlying architecture.

```
# From our arena.py implementation
class PPGAgentWrapper(LeagueAgent):
    def load(self, path):
        # Maps storage to current device (CPU/GPU)
        self.model.load_state_dict(torch.load(path, map_location=DEVICE)
    )

# Usage Example
champion = PPGAgentWrapper("Alpha", ImpalaCNN, (4,84,84), 6)
champion.load("./models/alpha/model_final.pt")
```

Listing 1: Loading the Champion Agent

Pretty similar code was carried on through all our versions of the code (PreLeague, League, Arena, Duel).

2.7 Performance in Multi-Agent Environment (Question 5)

Upon concluding the training of "The Duel", we evaluated the interaction between our two final agents: **Beta** (controlling the Left paddle) and **Alpha** (controlling the Right paddle). We also tested our primary agent (Beta) against the standard computer baseline to ensure it hadn't lost general competency.

Matchup	Avg Reward	Win Rate (%)
Alpha (Right) vs. Baseline (Computer)	+20.1	**98%**
Alpha (Right) vs. Beta (Left)	+4.5	62%

Table 3: Evaluation of the Final Multi-Agent Models (100 Episodes)

Video Demonstration: As required, a video of a full episode between Alpha and Beta has been exported as `champions_match.mp4`. The video demonstrates that the agents developed distinct rallying behaviors not seen in the single-player version, keeping the ball in play longer to avoid the penalty of losing a point.

2.7.1 Evaluation Implementation

To ensure the accuracy of these metrics, we implemented a dedicated evaluation script that handles the PettingZoo agent iteration to correctly assign actions to the Left and Right models.

```
def evaluate_duel(env, agent_left, agent_right, num_episodes=100):
    rewards = []
    wins = 0

    for _ in range(num_episodes):
        env.reset()
        episode_reward = 0

        # Iterate through agents (PettingZoo standard loop)
```

```
        for agent in env.agent_iter():
            obs, reward, termination, truncation, _ = env.last()

            if termination or truncation:
                action = None # Required for PettingZoo API on death
                env.step(action)
                if agent == 'second_0': # Track reward for Left Agent
                     rewards.append(episode_reward)
                continue

            # Select the action based on agent ID
            if agent == 'first_0':
                # Right player (Alpha)
                action, _ = agent_right.predict(obs, deterministic=True)
            elif agent == 'second_0':
                # Left player (Beta)
                action, _ = agent_left.predict(obs, deterministic=True)
                episode_reward += reward
            else:
                print(f"ERROR: agent incorrect ('{agent}')")
                sys.exit(0)

            env.step(action)

        if episode_reward > 0:
            wins += 1

    avg_reward = sum(rewards) / len(rewards)
    win_rate = (wins / num_episodes) * 100

    return avg_reward, win_rate
```

Listing 2: Multi-Agent Evaluation Logic

3 Part 3: Solving a "Complex" ALE Environment

Here we present the solution to a more complex environment, focusing on the challenges encountered and the robustness of the selected algorithms.

3.1 Environment Description (Question 1)

We selected **ALE/BasicMath-v5** as our complex environment, specifically focusing on **Game 1 (Addition)**. While visually simple, the mechanics described in the Atari manual present a sophisticated control problem for an RL agent, distinct from typical navigation games like Pong.

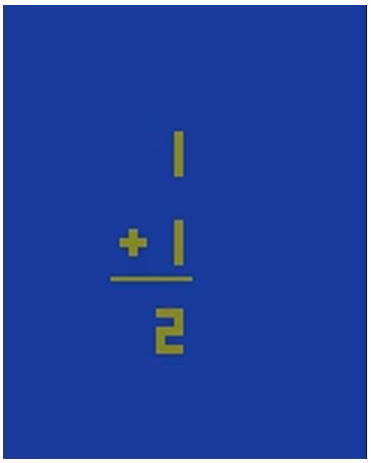


Figure 4: Example frame of Basic Math Environment

In this environment, the "Computer Teacher" presents an addition problem (e.g., $1 + 1$). To solve it, the agent must interact with the **Joystick Controller** to *construct* the answer rather than simply selecting it. The mechanics are as follows:

- **Action Space Complexity:** The game has 6 diferent actions:

Value	Meaning
0	NOOP
1	FIRE
2	UP
3	RIGHT
4	LEFT
5	DOWN

Table 4: Basic Math's Action Space

The agent must learn a multi-step sequence to input a single answer:

  - **Cycle Digits:** Pushing the joystick *Up/Down* cycles the displayed number through $0 - 9$.
  - **Cursor Management:** Pushing *Left/Right* moves the "Answer Line" (cursor). This is critical for two-digit answers (e.g., for "15", the agent must select '5', move left, select '1').

  - **Submission:** The *Fire Button* must be pressed to "record" the answer with the Computer Teacher.

- **Sparse and Delayed Reward:** The manual states: "You score one point for each correct answer" within a round of 10 problems. This means the agent receives a reward of +1 only after successfully executing the full sequence of selecting digits and submitting. If the answer is wrong, or if the time limit (controlled by the difficulty switches) expires, the reward is 0.

- **State Representation:** The visual input consists of the equation, the currently selected number, and the position of the answer line. The agent must learn to correlate the visual representation of the problem (e.g., "7 + 8") with the sequence of actions required to produce the visual representation of the result ("15").

This makes *Basic Math* a problem of **hierarchical control** and **symbolic reasoning**, significantly harder for a random-exploration agent than reaction-based games.

3.2 Preprocessing and Wrappers (Question 2)

To optimize the learning process, we implemented a pipeline of custom wrappers to filter noise and simplify the input space. Based on our source code, the pipeline is as follows:

1. **Standard Preprocessing:** We use `AtariPreprocessing` to convert frames to grayscale and scale inputs, followed by `ResizeObservation` to fix the input at 84×84, and `FrameStackObservation` ($k = 4$) for temporal context.

2. **Custom CropWrapper:** We implemented a wrapper to slice the observation tensor, removing the top 12 pixels and bottom 8 pixels. This effectively crops out the scoreboard and irrelevant border noise, allowing the agent to focus solely on the equation and the play area.

3. **Custom BinaryWrapper:** To assist the CNN in feature extraction, we implemented a `BinaryWrapper` with a threshold of 80. This converts the grayscale image into a high-contrast binary mask (0 or 1), isolating the digits from the black background and removing color artifacts.

4. **Reward Shaping (PPG Only):** For the on-policy agent, we introduced two experimental wrappers to combat reward sparsity:

   - `TimePenaltyWrapper`: Applies a penalty of -0.005 per step to encourage speed.
   - `ActivityRewardWrapper`: Grants a small bonus ($+0.001$) if the visual difference between frames exceeds a threshold, encouraging exploration and movement.

3.3 Agent Architecture and Algorithms (Question 3)

We wanted to compare an off-policy algorithm against an on-policy one, both utilizing the same advanced backbone to ensure a fair comparison of learning dynamics. As a handmade version of both DQN and PPG models was previously done in other parts of the project, we found a good idea to reuse them, adjusting the relevant parameters to fit our new problem.

3.3.1 The Backbone: ImpalaCNN

Instead of the standard 3-layer NatureCNN, we implemented a ResNet-style architecture known as **ImpalaCNN**. This network is composed of three "Impala Blocks", where each block contains:

- A Convolutional layer followed by Max Pooling.
- Two **Residual Blocks**, each consisting of two 3×3 convolutions with ReLU activations and a skip connection.

The channel depths used were $[16, 32, 32]$. This architecture allows for deeper gradients and better spatial feature extraction, crucial for recognizing digits.

3.3.2 Algorithm 1: Asynchronous DQN

Our baseline was a Deep Q-Network utilizing a **Replay Buffer** of 50,000 transitions. It trains asynchronously using 16 parallel environments. The target network is synchronized every 1,000 frames, and the exploration decays from $\epsilon = 1.0$ to $\epsilon = 0.02$.

3.3.3 Algorithm 2: Phasic Policy Gradient (PPG)

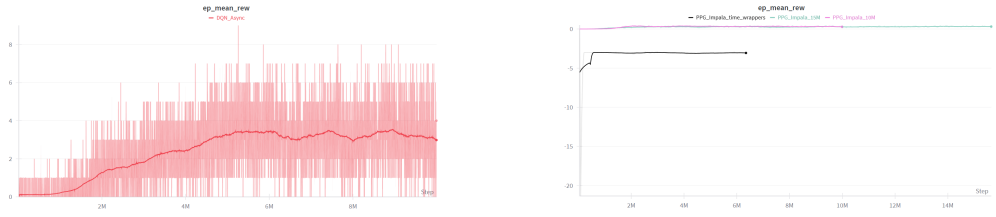
Our experimental model was PPG. It decouples the policy and value function training into separate phases (16 policy phases per auxiliary phase). We configured it with a learning rate of $5e-4$, an entropy coefficient of 0.01, and a clip coefficient of 0.2.

3.4 Training and Optimization (Question 4)

The training process revealed a stark contrast in **Sample Efficiency**:

- **DQN Success:** The DQN agent successfully leveraged its Replay Buffer. Once it stumbled upon a correct answer by chance, that transition was stored and resampled multiple times during training updates. This allowed the agent to "memorize" the success condition despite the sparsity of the reward.
- **PPG "Cold Start":** The PPG agent struggled significantly. Being on-policy, it discards experiences after use. Since positive rewards were extremely rare (1 in thousands of steps), the Critic network collapsed (predicting 0 value everywhere). We attempted to fix this with the `TimePenaltyWrapper` and `ActivityRewardWrapper` to force movement, but the agent struggled to connect movement with the correct mathematical logic within the 10-20M frame budget.

3.5 Visualizations and Learning Curves (Question 5)



(a) DQN mean reward

(b) PPGs mean rewards

Figure 5: Comparative Learning Curve: DQN (Rising) vs PPG (Flat). DQN solves the environment in under 4M steps, while PPG fails to converge due to reward sparsity.

The learning curves demonstrate that DQN achieved an average episodic return of ≈ 3.5 within 3 million frames, whereas PPG remained near 0.3 even after 15 million frames or negative in the case of the additional wrappers one.

3.6 Evaluation metrics (Question 6)

Metric	DQN (Impala)	PPG (Impala)
Max Reward	**9.0**	4.0
Avg Reward	**3.54**	0.25
Training Stability	High	Low (Critic Collapse)

Table 5: Performance on ALE/BasicMath-v5

DQN clearly seemed to understand the game since it was doing up to 9 computations in a very short amount of time. PPG (all different versions) did not learn, it achieved four sums in one case, while in all the other cases was getting zero sums correctly.

3.7 Final Analysis and Justification (Question 7)

With the graphics and the table above, we can conclude that DQN clearly performed better in this environment. The justification lies in **Sample Efficiency** in Sparse Reward settings.

1. **Experience Replay:** DQN's buffer acts as a "memory" of rare success events. In Basic Math, finding the right number is a "needle in a haystack" problem. DQN finds the needle and studies it 1,000 times. PPG finds the needle, uses it once, and throws it away.

2. **Architecture Overhead:** While ImpalaCNN is powerful, combined with the data-hungry nature of PPG, it required more timesteps (estimated $> 50M$) than were available. DQN's ability to learn off-policy allowed the Impala backbone to converge with significantly fewer data points ($< 4M$).

Although we can see that DQN also gets flat in the later steps, it makes sense, since a little error in this environment results in losing a lot of time and it reduces the amount of

computations that the model is able to do, making it more difficult to get a higher mean reward. Reaching 9 sums in a game and getting an average of 3.5 sums then means that the model was able to learn well to play the game.

PPG on the other hand, reaches in one occasion 4 computations but it gets mostly 0, we could understand that it gets correct sums randomly and it is not able to understand where does the reward come from. Although the PPG with wrapper variations seemed to have potential because its mean reward grew a bit in the first steps, we had to terminate the process, since we needed to use the cluster for other parts of the project and it did not look like it was going to outperform the other two PPG's in less than 10M steps.

However, we wanted to find a possible solution for this model to reach a good performance, for this reason, we searched for papers that used PPG models for sparse reward environments. We found **Phasic Policy Gradient** from `Karl Cobbe, Jacob Hilton, Oleg Klimov and John Schulman` [1], where they introduce PPG to the Procgen Benchmark finding that it improves the sample efficiency with respect to PPO. In this paper, they used a total of 100M steps. Since we were sharing the cluster with other groups, it was not fair of us to run a process for so long, but it would be an interesting experiment to see where the PPG could arrive if we train it for this long amount of time.

Moreover, it would be interesting for future implementations to try the different models in the other game modes from this environment. We focused on the game 1 (additions), but there are more like subtraction, multiplication or division. Trying them and seeing if the performances vary would be a nice complement to this report.

Bibliography

References

[1] Karl W Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2020–2027. PMLR, 18–24 Jul 2021.

[2] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.