Wrocław University of Science and Technology

Politechnika Wrocławska

Faculty of Electronics, Photonics and Microsystems

Applying Reinforcement Learning to the Classic Game Pong

Project Definition, Midterm and Final Report

Artificial Inteligence and Computer Vision

Autor: Mateusz Naryniecki
Indeks: 278154

Wrocław, January 25, 2026

Contents

1 Project name

RLPong [Name WIP] GitHub Repo

2 Scope and short description of project

I am going to apply reinforcement learning to the game Pong where the Player (in this case neural network AI) is put up against a typical algorithm that tracks the ball without anything else.

3 Case studies

Inspiration
https://www.youtube.com/watch?v=DmQ4Dqxs0HI

The approach I'm going to use
https://www.youtube.com/watch?v=vXtfdGphr3c&t=13s

Article on Github by Andrej Karpathy
https://karpathy.github.io/2016/05/31/rl

4 How you want to solve the problem?

The goal is to train an AI agent to play Pong using reinforcement learning. Instead of raw images, the agent will use 8 numerical values as input: the position and velocity of both paddles and the ball. This makes the problem simpler and faster to train.

4.1 Approach

First, I will build a simple version of Pong where these 8 values are available each frame. The AI will choose one of three actions: move up, move down, or stay still. The goal is to learn which actions help it win more often.

4.2 Neural network

A small neural network will be used to decide what to do. It will take the 8 input values, process them through a few layers of neurons, and output the probability of each action. The network will be trained to pick actions that lead to higher scores over time.

4.3 Pretraining and learning

At the beginning, I plan to train the network in a supervised way by copying a simple rule-based opponent that tracks the ball. This will give the AI some basic skills before reinforcement learning starts. Then I will use a standard reinforcement learning algorithm, such as a policy gradient method, to let the AI improve through trial and error.

4.4 Rewards

The agent will get a reward of +1 for scoring a point and -1 for losing a point. These rewards will guide the learning process so the AI learns to keep the ball in play and eventually beat the opponent.

4.5 Evaluation

Progress will be measured by how often the AI wins against the basic opponent and how its average reward increases over time. I will also record and plot training results to show improvements during learning.

5 Short plan with some big milestones together with dates

- **31.10 – 16.11.2025 (Design)** — Set up the Pong environment and define the 8 input values: the positions and velocities of both paddles and the ball. Make sure the basic game loop and the rule-based opponent work correctly.
- **17.11 – 30.11.2025 (Neural network setup)** — Build a small neural network (MLP) that takes the 8 inputs and outputs the three possible actions (up, down and optionally stay). Test the network to ensure it produces reasonable outputs.
- **01.12 – 14.12.2025 (Supervised pretraining)** — Collect gameplay data from the rule-based opponent and train the network to imitate it. Verify that the AI can follow the ball and react properly.
- **15.12.2025 – 04.01.2026 (Reinforcement learning)** — Train the AI using reinforcement learning to improve by playing against the opponent and learning from rewards.
- **05.01 – 11.01.2026 (Evaluation and fine-tuning)** — Measure the AI's performance, record win rates, adjust hyperparameters if needed, and save gameplay for analysis.
- **12.01 – 18.01.2026 (Final report)** — Write and finalise the documentation, including explanations of methods, results, and ideas for future improvements, ensuring it is ready for submission by the deadline.

6 Agreement

(proposal) I, Mateusz Naryniecki agree to deliver the project within defined timeline, within defined scope. Mateusz Cholewiński, PhD is confirming to grade it in an appropriate way, taking following document as a base. All changes, especially in timeline, scope, has to be agreed by both parties.

7 Midterm report: implementation status

This section describes what has been implemented so far, how it matches the original plan, and what remains to be done.

7.1 Environment and game dynamics

A custom Pong environment `PongEnv` has been implemented in Python. The game is played on a fixed-size window of `800x600` pixels with a square ball of size `20` and paddles of height `100` and width `20`. The ball speed is fixed at a constant magnitude, but its direction is initialised with a random angle so that it always moves either towards the left or towards the right side of the screen.

On reset, the environment:

- Resets both scores to zero.
- Places both paddles in the vertical centre.
- Resets the ball to the centre with a random initial direction biased horizontally.

Collisions with the top and bottom walls are handled by inverting the vertical component of the ball velocity. Paddle collisions are implemented such that the outgoing angle depends on where the ball hits the paddle (more extreme angles near the edges, more horizontal in the centre). This produces more interesting and less predictable trajectories.

A game terminates if either player reaches a fixed number of points (e.g. 5) or when a maximum number of frames is exceeded, in which case the episode ends with a "timeout" result. The environment returns both a scalar reward and an `info` dictionary that stores who won, the scores, and the total number of frames.

7.2 State representation and action space

As planned, the state is represented by 8 normalised numerical values:

- Ball horizontal position: x_{ball}/WIDTH.
- Ball vertical position: y_{ball}/HEIGHT.
- Ball horizontal velocity: v_x/speed.
- Ball vertical velocity: v_y/speed.
- Right paddle vertical position: y_{right}/HEIGHT.
- Right paddle movement direction (last action): $\{-1, 0, +1\}$.
- Left paddle vertical position: y_{left}/HEIGHT.
- Left paddle movement direction: $\{-1, 0, +1\}$.

This matches the initial design decision to avoid raw pixel inputs, simplifying the learning problem and speeding up training.

The action space for the learning agent (right paddle) consists of three discrete moves:

- **0**: move up,
- **1**: stay still,
- **2**: move down.

The left paddle is controlled by a simple rule-based policy that tries to track the ball vertically with a fixed speed. This opponent acts as a baseline and as a teacher for pretraining.

7.3 Reward structure

The reward function follows the original plan:

- The right (learning) agent receives $+1$ for scoring a point.
- It receives -1 for losing a point.
- Intermediate frames give 0 reward.

I might change the reward system to penalize and reward later actions as they are probably the most impactful on the victory/loss

7.4 Neural network policy

The policy network, `PongNet`, is a small multilayer perceptron (MLP) with the following structure:

- Input layer: dimension 8 (state vector).
- Two hidden layers with 64 units each and ReLU activations.
- Output layer: 3 units corresponding to the three possible actions.

The network outputs raw logits. During action selection, either a greedy policy (argmax over logits) or a stochastic policy (sampling from a softmax distribution with optional temperature) can be used. A wrapper class `PongAgent` handles device selection (CPU/GPU), model loading, and action selection in both single-state and batched mode.

8 Midterm report: supervised pretraining

8.1 Synthetic teacher data

Instead of recording data from live games, synthetic supervision data is generated directly from the state description. Each sample consists of:

- A random ball position (x, y), uniformly sampled in $[0, 1] \times [0, 1]$.
- A random ball velocity (v_x, v_y), where each component is sampled from $[-1, 1]$.
- Random paddle positions $y_{\text{left}}, y_{\text{right}}$ in $[0, 1]$.
- Paddle directions initially set to 0.

Given this state, a "teacher" rule determines the desired action for the right paddle mimicking the opponent in order to have enough "understanding" to continue learning :

- If the ball is clearly above the right paddle (difference $< -$margin), the correct action is `up`.
- If the ball is clearly below the right paddle (difference $>$ margin), the correct action is `down`.
- If the ball is already aligned (within the margin), the action is `stay`.

This generates a dataset of (state, action) pairs that approximate how a simple tracking algorithm would behave, but in a much more diversified set of situations than just replaying a few games.

8.2 Supervised training procedure

A separate script performs supervised training of `PongNet` on the synthetic dataset. The key components are:

- Loss function: cross-entropy between the predicted logits and the teacher action labels.
- Optimiser: Adam with a learning rate of 10^{-3}.
- Number of samples: e.g. 20 000 synthetic states.
- Training loop: multiple epochs over all data, printing the loss for monitoring.

After training converges, the model parameters are saved to a file (e.g. `pong_pretrained_teacher.pt`). The `PongAgent` automatically attempts to load these weights on initialisation and switches to evaluation mode for inference.

At this midterm stage, the supervised pretraining phase is implemented and ready to be combined with reinforcement learning.

9 Midterm report: evaluation and visualisation

9.1 Vectorised evaluation

To efficiently measure the performance of the current agent, a vectorised evaluation script is provided. It creates multiple independent `PongEnv` instances, runs them in parallel using batched action selection (`act_batch`), and counts:

- Number of games won by the right (learning) agent.
- Number of games won by the left (rule-based) opponent.
- Number of timeouts (no player reaches the winning score before the frame limit).

By increasing the number of environments and episodes, this script allows for statistically meaningful estimates of win rates and for timing measurements (e.g. average time per environment per game). These metrics will later be used to compare:

- The initial random policy.
- The supervised-pretrained policy.
- The final reinforcement learning policy.

9.2 Graphical interface

For qualitative inspection and debugging, a simple visualisation using `pygame` has been implemented. It:

- Opens a window of size `WIDTH` by `HEIGHT`.
- Draws the left and right paddles and the ball each frame.
- Renders the current score of both players.
- Lets the right paddle be controlled by the current `PongAgent` policy.

This visualisation is useful for checking whether the pre-trained policy behaves as expected: tracking the ball, reacting to bounces, and recovering after losing points.

10 Midterm report: comparison with original plan

Here is how the current status relates to the milestones defined in the project proposal:

- **31.10 – 16.11.2025 (Design):** The Pong environment, state representation, reward function, and rule-based opponent have been fully implemented and tested. The environment supports both single-episode runs and batched evaluation.
- **17.11 – 30.11.2025 (Neural network setup):** The MLP architecture has been implemented and integrated with an agent wrapper. Action selection supports both greedy and stochastic modes, and the network can be run on CPU or GPU.
- **01.12 – 14.12.2025 (Supervised pretraining):** The synthetic teacher data generation and supervised training script are implemented. The network can be pretrained to imitate a ball-tracking policy and saved to disk for later use in reinforcement learning.
- **15.12.2025 – 04.01.2026 (Reinforcement learning):** Reinforcement learning (e.g. policy gradients or another on-policy method) is the next step. The environment and pretrained policy are ready, but the RL training loop and logging have not yet been implemented at this midterm stage.
- **05.01 – 11.01.2026 (Evaluation and fine-tuning):** The evaluation infrastructure is already partially in place via the parallel evaluation script. It will be extended to log learning curves (win rate, average reward) during RL training.
- **12.01 – 18.01.2026 (Final report):** The current document serves as a combined project definition and midterm report. A final extended version will include detailed experimental results, plots, and a discussion of limitations and possible improvements.

11 Next steps

The main next steps towards the final project are:

- Implement a reinforcement learning algorithm (e.g. REINFORCE or an actor-critic variant) on top of the existing environment and pretrained policy.
- Add logging of episode returns, win rates, and possibly entropy or KL-divergence to monitor exploration.
- Run experiments comparing:
  - Training from scratch vs. starting from the supervised teacher.
  - Different reward shaping or curriculum variations (e.g. starting with slower ball speed).
- Produce plots and visualisations of learning progress.
- Integrate selected games into the report via screenshots or short descriptions of interesting behaviours (e.g. long rallies, exploiting paddle angles).

These steps will complete the transition from a working Pong environment and supervised policy to a fully trained reinforcement learning agent, and will provide the material for the final report.

12 Final Report: Reinforcement Learning Implementation

Following the midterm milestones, the focus shifted to implementing the Reinforcement Learning (RL) loop, automating the experimental workflow, and establishing a robust evaluation pipeline.

12.1 Algorithm and Training Loop

The REINFORCE algorithm (Monte Carlo Policy Gradient) was selected for training. The implementation in `train_rl.py` utilizes:

- **Parallel Environments:** To stabilize gradients and speed up data collection, 256 independent `PongEnv` instances run in parallel. This is crucial for collecting diverse trajectories efficiently.
- **Hyperparameters:**
  - Learning Rate: 1×10^{-4} (Adam optimizer).
  - Discount Factor (γ): 0.99.
  - Entropy Coefficient: 0.01 (to encourage exploration).
- **Batch Updates:** Gradients are computed and applied after collecting trajectories from all parallel environments.

- **Return Normalization:** Discounted returns are normalized across the batch to reduce variance and improve training stability.

To ensure the agent learns a robust policy, the training process includes a "waterfall" loading mechanism: it attempts to resume from a previous RL checkpoint, falls back to the supervised "teacher" model, and finally defaults to random initialization if no weights are found.

12.2 Experiment Orchestration

A dedicated script, `run_experiment.py`, was developed to automate the full experimental lifecycle. It performs the following steps sequentially:

1. **Cleanup:** Removes artifacts from previous runs (unless resuming) to ensure a clean state.
2. **Baseline Evaluation:** Evaluates a random agent ("Scratch") and the supervised agent ("Teacher") to establish performance benchmarks before RL begins.
3. **Incremental Training:** Runs the training loop in blocks of 16 updates (approx 4096 games per block), pausing to evaluate the current model against the rule-based opponent.
4. **Logging:** Results (Win Rate, Average Reward) are logged to CSV files (`results_scratch_stocha` and `results_teacher_stochastic.csv`) after every evaluation step.

12.3 Model Selection and Evaluation

Evaluation is performed using `eval.py`, which tests the agent in 128 parallel environments for a fixed number of episodes (16 per evaluation step).

Crucially, the training loop tracks the **Average Reward** to identify the best-performing model. The weights corresponding to the highest average reward are saved separately as `[prefix]_best.pth`. This ensures that the final deployed agent represents the peak of training performance rather than the latest (potentially unstable) iteration.

12.4 Results Visualization

A plotting utility, `plot_results.py`, was implemented to visualize the training progress. It reads the generated CSV logs and produces a dual-axis plot showing:

- **Win Rate (%):** The percentage of games won against the rule-based opponent.
- **Average Reward:** The mean reward per game, providing a more granular measure of performance than binary win/loss.

12.5 Codebase Refactoring

To support these advanced features, the codebase underwent significant refactoring:

- **Configuration:** All game constants (dimensions, speeds, rewards) were centralized in `config.py` to ensure consistency across simulation, training, and visualization.

- **Robustness:** The environment includes checks (e.g., `is_catchable`) to ensure fair initialization, preventing the agent from being penalized for physically impossible scenarios.
- **Visualization:** The `visual_game.py` script was updated to load the best model by default and supports command-line arguments for flexibility.

12.6 Updated Experimental Methodology

12.6.1 Stochastic Evaluation

To provide a more rigorous assessment of the agent's capabilities, the evaluation protocol in `eval.py` was updated to use a **stochastic policy** (`stochastic=True`). Unlike greedy evaluation, which deterministically selects the action with the highest probability, stochastic evaluation samples from the action distribution output by the policy network. This approach ensures that the reported Win Rate and Average Reward metrics reflect the agent's general robustness and ability to handle variability, rather than its ability to exploit deterministic trajectories against the fixed rule-based opponent.

12.6.2 Automated Comparison: Scratch vs. Teacher

The experiment orchestration script, `run_experiment.py`, was refactored to conduct a direct comparative analysis. It now automates the execution of two sequential training regimes:

1. **Scratch Stochastic:** Training an agent initialized with random weights to establish a baseline learning curve.
2. **Teacher Stochastic:** Fine-tuning an agent pre-trained on the rule-based opponent's behavior to evaluate the benefits of transfer learning.

Results from these experiments are logged to separate CSV files, enabling side-by-side visualization of convergence speed and final performance.

12.6.3 Robustness and Recovery

To support long-running experimental campaigns, a resume mechanism was implemented in the training loop. The system checks for existing result logs and model checkpoints before starting. If an experiment is interrupted (e.g., due to a system restart), it automatically resumes training from the last recorded update step, preserving previously collected data and computational resources.

12.7 Results and Discussion

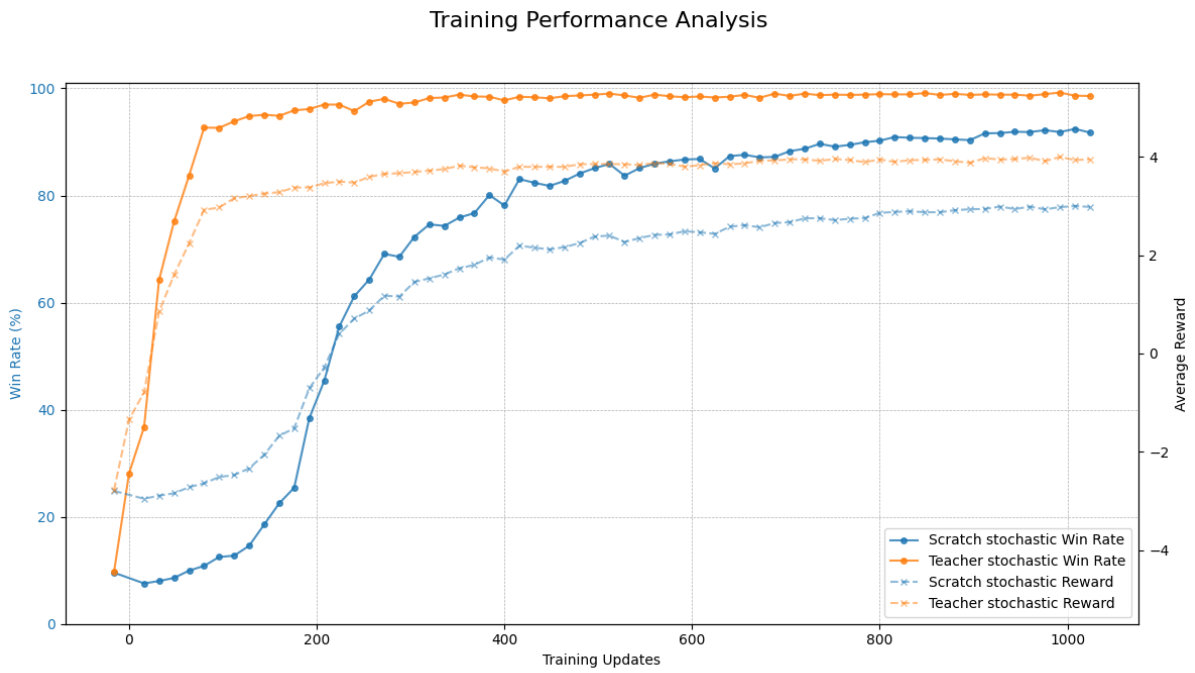

Figure 1: Comparison of learning curves between Scratch (Blue) and Pre-trained Teacher (Orange) using Stochastic Evaluation. The Teacher model starts with a higher win rate and converges significantly faster to a stable 99% win rate.

As shown in Figure 1, the agent initialized with "Teacher" weights demonstrated significantly faster convergence compared to the "Scratch" agent. The pre-trained model began with a non-zero win rate and quickly optimized its policy to exploit the opponent's weaknesses. In contrast, the scratch model required approximately 200 updates (roughly 50,000 games) to simply reach positive average rewards, indicating the difficulty of discovering the ball-tracking strategy from random actions.

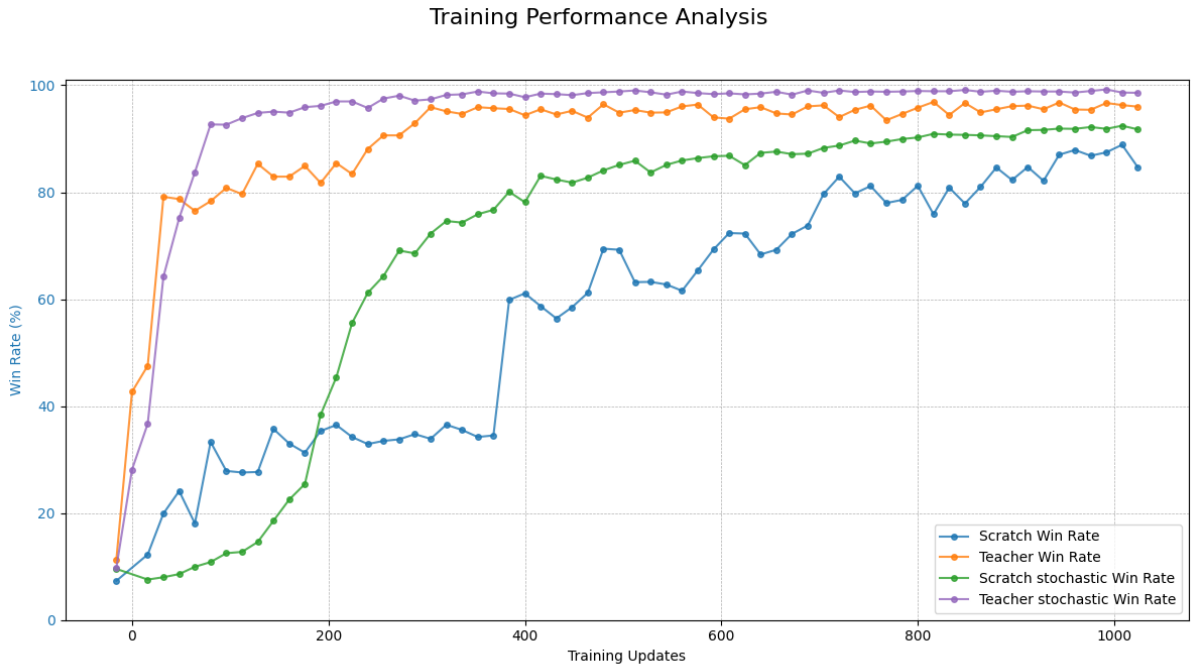

Figure 2: Performance gap between Greedy (Orange and Blue) and Stochastic (Purple and Green) evaluation policies. The stochastic metrics provide a more conservative and realistic estimate of the agent's true capability.

Figure 2 highlights a critical observation regarding the agent's stability. Surprisingly, the model performed better overall in **stochastic mode** compared to **greedy mode**. The greedy policy (taking the argmax of logits) proved to be more volatile: if the agent made a single suboptimal decision, the deterministic nature of the policy could cause it to get stuck in a failure loop or commit to a rigid trajectory that missed the ball.

In contrast, stochastic evaluation allowed the model to sample actions based on probability, effectively "smoothing out" jittery decision-making. This randomness served as a recovery mechanism, preventing the agent from becoming locked into a bad decision and allowing it to adjust its trajectory dynamically. Consequently, stochastic evaluation provided a more stable and higher-performing metric, confirming that a slight degree of entropy is beneficial for robustness in continuous-time tracking tasks.

13 Summary

This project successfully developed and evaluated a reinforcement learning agent for the classic game of Pong. The primary objective was to train a neural network to master the game using a state-based approach, bypassing the complexity of raw pixel inputs. The agent's "brain" is a multi-layer perceptron that processes an 8-dimensional state vector—containing the normalized positions and velocities of the ball and both paddles—to decide between moving up, down, or staying still.

The core of the training process utilized the **REINFORCE** (Monte Carlo Policy Gradient) algorithm. To ensure stable and efficient learning, the implementation was enhanced with several key techniques:

- **Parallel Environments:** Training data was collected from 256 simultaneous game instances, providing a diverse batch of experiences for each policy update.
- **Return Normalization:** The cumulative rewards were normalized within each batch, a crucial step that acts as a baseline to reduce gradient variance.
- **Entropy Regularization:** A small entropy bonus was added to the loss function to encourage exploration and prevent the agent from prematurely converging to a suboptimal deterministic policy.

A significant aspect of this project was the rigorous, automated experimental framework designed to compare two fundamental training strategies:

1. **Training from Scratch:** A baseline approach where the agent learns with no prior knowledge.
2. **Training from a Teacher:** A transfer learning approach where the agent is first pre-trained via supervised learning to imitate a simple, rule-based opponent. It is then fine-tuned using reinforcement learning.

The entire pipeline, from environment simulation and agent training to vectorized evaluation and results visualization, was automated. The system was designed not only to produce a capable Pong-playing agent but also to serve as a robust platform for conducting comparative machine learning experiments. The final evaluation, based on stochastic action selection, ensures that the reported metrics reflect the agent's general capability rather than its performance on deterministic trajectories.