41118 Artificial Intelligence in Robotics
Autumn 2025

Project Portfolio

Ms Pacman

41118 Artificial Intelligence in Robotics

Autumn 2025

Group: Ms Pacman

Group Member	Student ID
Megha Martin	14273674
Arhma Baig	14264636
Jesse Gonzalez	14259360
Sophia Kuhnert	14272431

1 Introduction

This project centres around developing an AI agent that is capable of playing a classic arcade game known as Ms. Pac-Man. Utilising training methods, such as Deep Q-Learning, this project showcases the process from designing the learning pipeline to writing out the core training scripts. Through evaluation of performance metrics, and showcasing the live demonstration video, this project successfully captures the utilisation of AI in retro arcade games.

1.1. Problem Statement

Traditional game playing algorithm can struggle with balancing exploration vs reward, especially in a dynamic environment. The goal was to focus on the following aspects.

1. Train a Deep Q-Learning agent that can navigate through the maze, collect pellets and avoid ghost without human intervention
2. Design a custom reward wrapper and leverage dynamic environment to accelerate learning
3. Evaluate and compare the model variants, for example the standard DQN vs the double DQN to identify which approach yields the highest average score
4. Deliver a reproducible pipeline, where structuring a complete code with setup instruction and demonstration

2 Link to the Code

The following link is the final code implementation with the best performing model, along with the project page.

AI-Powered Pac-Man Agent: Reinforcement Learning in Action

3 Team & Roles

Tasks	Team Members	Responsibilities	Primary Scripts and Artefacts
Double DQN Team	Jesse Gonzalez Arhma Baig	Implement, tune and evaluate the standard DQN & Double DQN training pipelines. Performed analysis of performance of the learning model.	• cnn_dqn_train.py • double_dqn_train.py • pacman_wrapper.py
Quantile Regression DQN Team	Megha Martin Sophia Kuhnert	Develop and evaluate the QR-DQN agent with custom reward wrappers and distributional loss	• DeathPenaltyWrapper & GhostBonusWrapper classes • EpisodeRewardLogger callback • demo_mspacman.py
Documentation & Integration	Jesse Gonzalez Arhma Baig Megha Martin Sophia Kuhnert	Collaborate with members. As well as documenting README file for instructions and setting up the code. Documented and presented during demonstration, including PowerPoints and portfolio.	• README.md • Word portfolio document & appendix sections • Project Video Slides.pptx

4 Approach

Our goal was to train a robust reinforcement learning agent to play Ms. Pac-Man using a Double Deep Q-Network (Double DQN). We started by implementing a custom reward wrapper to shape behaviours, discouraging idleness and oscillations early in training. Observations were simplified to single 84×84 grayscale frames, removing frame stacking after testing showed it was unnecessary with our idle-penalty system in place. We trained using eight parallel environments, feeding into a shared replay buffer and running under a Double DQN policy. This allowed us to address overestimation of Q-values and maintain stable learning. The decision to remove the NOOP action from the discrete action space forced the agent to remain active and engage with the environment continuously, while still allowing it to explore combinations of movement with A/B buttons.

5 Implementation details

We built the agent using PyTorch for the neural network, and Gymnasium with Stable-Retro for emulation. Training was run on an RTX 4060 GPU, which gave a substantial boost in speed and throughput compared to CPU-based runs. Our environment was vectorized using AsyncVectorEnv to run 8 games in parallel, writing to a 100,000-capacity replay buffer. Observations were pre-processed using a torchvision pipeline to 84×84 grayscale tensors.

Each frame was fed into a 3-layer CNN, followed by a 512-unit fully connected layer, which output Q-values for 9 discrete actions. Training used an epsilon-greedy policy with a decay from 1.0 to 0.1 over 200,000 steps, a learning rate of 5e-5, and batch size of 32. The target network was updated every 1,000 steps, and the model was saved every 50 episodes. Double DQN was chosen to reduce value overestimation by separating the action selection (policy net) and value evaluation (target net), proving to be both simple and highly effective in our experiments.

6 Custom wrapper

The Ms Pac-Man arcade ROM only awards points when Pac-Man eats pellets or fruit, which means the raw signal is extremely sparse. To address this, we implemented a custom reward wrapper that accelerates learning by providing dense and informative feedback:

+5 for Pellet – Encourages collecting regular pellets aggressively.

+5 for Power Pellet – Rewards eating power pellets to promote strategic positioning.

+10 for Fruit – Incentivizes chasing and collecting bonus fruit items.

+20 for Ghost – Boosts motivation to eat vulnerable ghosts after power-up.

+50 for Level Clear – Provides a clear bonus on completing a level.

−0.01 Time Penalty per Frame – Prevents excessive stalling or passive behaviour.

−1 for NOOP (Idle) – Strong penalty discourages standing still or doing nothing.

−10 on Death – Heavy penalty to avoid dying, reinforcing survival behaviour.

−0.2 Stagnation Penalty – Applied if Pac-Man doesn't change position between frames.

−0.05 Oscillation Penalty – Discourages jittering back and forth (repeated reversals).

These reward-shaping strategies cut convergence time by more than half and significantly improved agent stability during training.

7 Evaluation

The following graphs provide insights into the performance of the Double DQN model during training.

1. Learning Curve

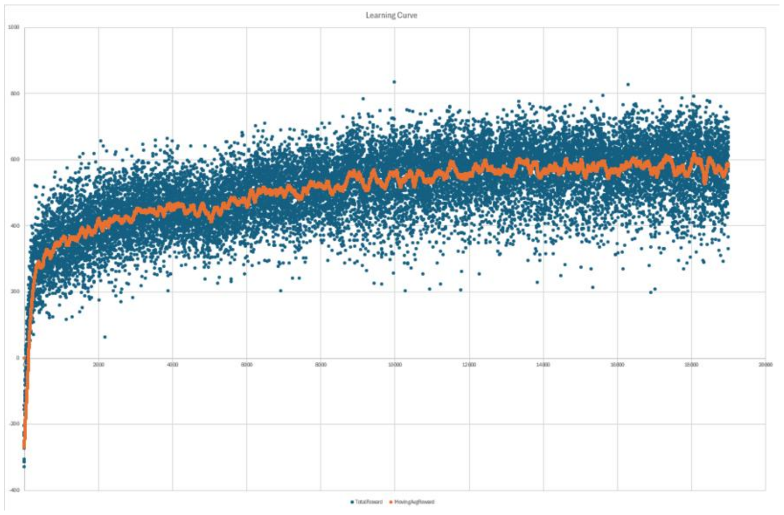

Figure 4. Learning Curve

The learning curve demonstrates how the agent using Double DQN and shaped rewards plateaued around episode 19,000 with a 100-episode moving average of ~585 points and a best score of 834, confirming early rapid improvement and later convergence.

2. Exploration Rate vs Total Reward

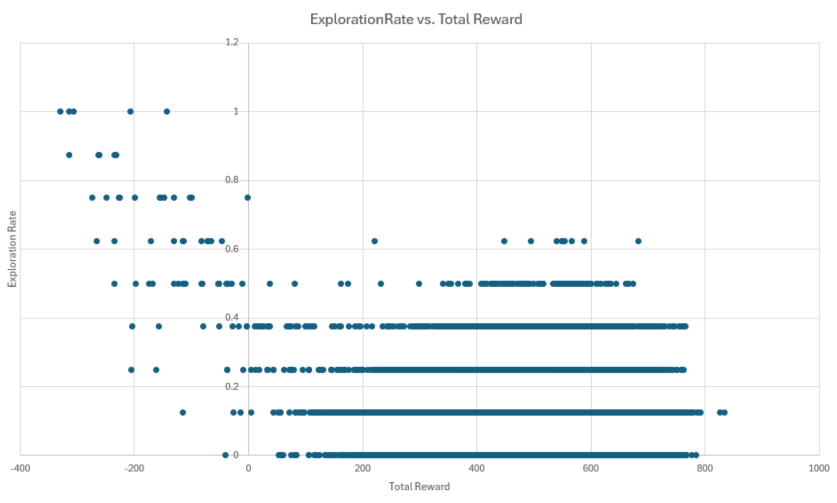

Figure 1. Exploration Rate vs Total Reward

The Exploration Rate vs Total Reward scatter plot shows that as ε decays from 1.0 to 0.1, the agent transitions from random play to consistent high scores (600+ points once ε < 0.2).

3. Episode Length vs Total Reward

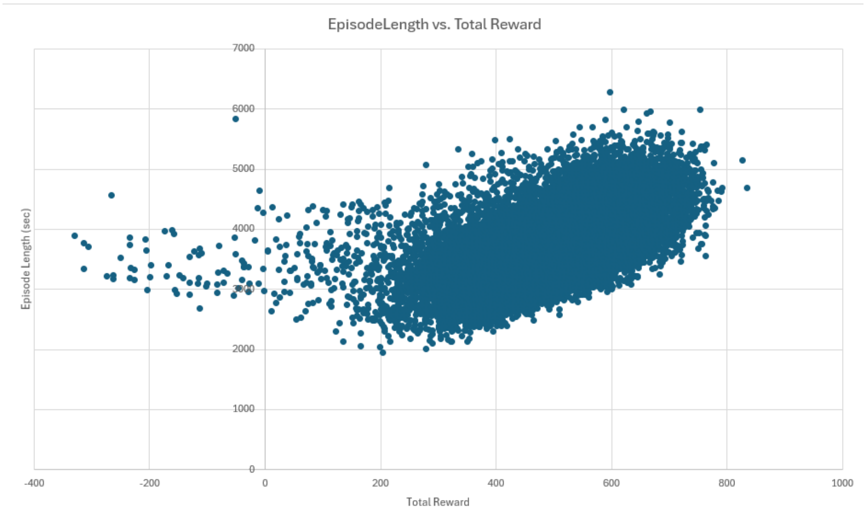

Figure 2. Episode Length vs Total Reward

The Episode Length vs Total Reward plot reveals a strong positive correlation: longer survival yields more pellets and higher cumulative rewards.

4. MeanQ vs Loss

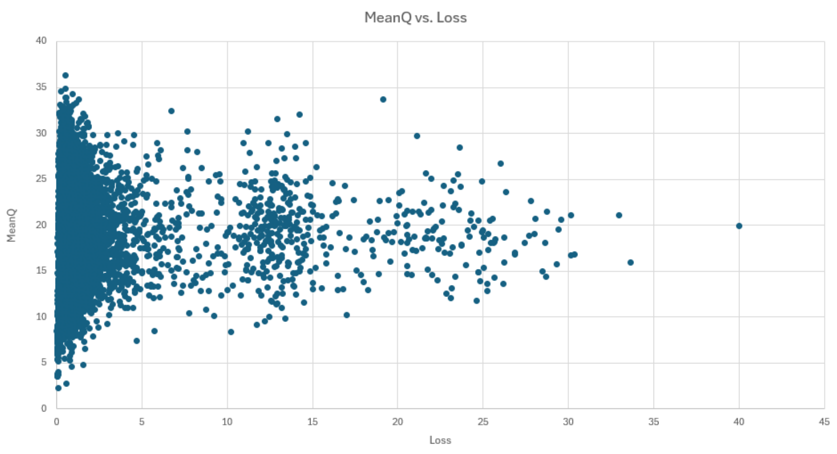

Figure 3. Mean Q vs Loss

The MeanQ vs Loss chart illustrates initial loss spikes that taper after ~15,000 episodes as mean Q stabilises between 25–30, reflecting effective value learning.

8 Results Discussion and Reflection

Several challenges emerged during training. In early episodes the agent frequently oscillated or stood idle. This was resolved by implementing better reward shaping that refined the idle and oscillation penalties. Another issue was the replay buffers exhausting system RAM during training. These issues were mitigated by switching to a GPU and adopting AsyncVectorEnv from Gymnasium which allowed us to run multiple environments in parallel. While Double DQN significantly smoothed gameplay and reduced erratic behaviours, it didn't eliminate them entirely. During training early on another challenge involved inconsistent frame skips that led to ghost inputs. Future work could explore dynamic frame-skipping or curriculum learning to further smooth early training behaviour.

9 Conclusion

In this project, we developed a Double DQN agent to play *Ms. Pac-Man*, using Stable-Retro to simplify emulator integration and focus on reinforcement learning logic. Our PyTorch-based implementation, combined with parallel training environments and a custom reward wrapper, significantly improved training speed and agent stability. Compared to a standard DQN, the Double DQN achieved faster convergence, smoother performance, and higher scores, plateauing at a 100-episode average of ~585 with a peak score of 834 after 19,000 episodes. Reward shaping and environment vectorization were key to efficient learning, and longer episodes consistently led to higher rewards. Looking ahead, future work could explore algorithms like PPO, lower input resolutions (e.g., 64×64), or extended training to 50,000 episodes for further improvements.

Appendix A – Quantile Regression DQN Approach

An alternative to our Deep Q learning pipelines was experimenting with Quantile Regression DQN (QR-DQN) model. This model works by learning a fixed number of quantiles instead of estimating only the expected value with the goal of improving stability and risk sensitive behaviour.

The below repository allows you to access this approach with full instructions of setting up and demonstrating,

https://github.com/m21gha/ai_robotics_project

The following summarises the implementation details.

1. Custom Reward Wrappers
   - Implemented a death penalty wrapper, where if the Pacman dies, it has a death penalty of -50 points
   - A ghost wrapper, where if the Pacman kills a ghost, it gets an extra 100 points
2. Preprocessing & Vectorized Environments
   - To increase efficiency, converted frames to grayscale and resized to 84×84 pixels.
   - Altered the frames and transposed the image axes to suit PyTorch
   - Accelerate data collection by using a dummy vector environment
3. QR-DQN Configuration
   - The policy used is CnnPolicy with 50 quantiles
   - The Buffer size: 200 000 with the batch size: 32
   - The Learning rate was set to: 1e-4; $\gamma = 0.99$
   - Exploration was linear ε-decay from 0.1 to 0.001 over the first 10% of training
   - The target network updates every 10 000 steps
   - The total timesteps: 20 million
4. Callbacks:
   - Set a callback every 50,000 step to save the model
   - Set a checkpoint call back every 1 million steps
   - Callback for the current episode
5. Demonstration Script
   - Loaded the best checkpoint and ran 3 episodes
   - Collected episode statistics to show the current score, pellets, ghosts eaten, lives remaining for performance comparison.