Extended Abstract

Motivation Large language models (LLMs) have shown strong reasoning abilities in static tasks but have yet to demonstrate similar capabilities in real-time, feedback-driven environments. Reinforcement learning (RL), in contrast, enables agents to learn from interaction but lacks the flexibility and generalization of LLMs. This project explores whether open-source multimodal LLMs, specifically QWEN2.5, can be trained as active agents that learn from reward signals over time. We aim to understand if LLMs can move beyond static prompting and act effectively in dense control tasks that require spatial reasoning, rapid decision-making, and long-horizon planning.

Method We evaluate two types of agents in the Super Mario Bros environment. The first is a vision-based DDQN agent trained on stacks of grayscale frames using a convolutional neural network. This model uses the Double Q-learning update to reduce overestimation bias and improve stability. The second is a fine-tuned QWEN2.5-VL model trained with Proximal Policy Optimization (PPO). It receives a structured prompt combining an image of the current frame and a serialized JSON-based state description. The LLM produces hidden states that are passed to lightweight MLP heads for both policy and value prediction. We compute advantages using Generalized Advantage Estimation (GAE) and optimize the PPO loss with entropy regularization and a value error term. The goal is to assess whether LLMs, when trained in this setup, can learn competitive policies relative to traditional deep RL agents.

Implementation Both agents are implemented in PyTorch. The DDQN model uses standard neural network modules to define its convolutional and fully connected layers, along with a replay buffer and target network for stable training. For the LLM-based PPO agent, we fine-tune QWEN2.5-VL using the `transformers` and `accelerate` libraries, paired with `peft` for efficient LoRA-based updates. Mixed-precision (`bf16`) training and gradient checkpointing are used to reduce memory usage. The LLM's policy and value heads are implemented as lightweight MLPs attached to the final hidden states. Training and evaluation metrics are logged via Weights & Biases.

Results The DDQN agent outperformed random and greedy baselines on level 1, with performance improving as training progressed. After 10,000 episodes, it reached an average reward of 892.3 but still fell short of the scripted Lookahead agent. On level 2, DDQN generalized better than Greedy but underperformed Random, suggesting overfitting or limited adaptability.

The LLM agent, without PPO fine-tuning, behaved similarly to Greedy, often failing early in the episode. With PPO training, it improved to an average reward of 720.7—substantially better than the unfine-tuned version (446.1), though still below DDQN. Training plots showed noisy but improving reward trajectories, indicating partial policy learning.

Discussion Q-learning proved effective but sample-inefficient—training DDQN for 10,000 episodes required several days and still displayed inconsistent behavior. The agent's weaker performance on new levels points to limited generalization and possible overfitting.

For the LLM agent, PPO fine-tuning allowed it to learn beyond greedy heuristics, though convergence was slow. Despite not outperforming DDQN, the LLM's post-finetuning improvements suggest it can act as a viable policy learner, especially when enhanced with prompt engineering, LoRA adaptation, and multimodal inputs. However, training remains resource-intensive.

Conclusion Our findings show that both DDQN and PPO-finetuned LLMs can learn useful policies in dynamic environments. DDQN currently offers better reward performance and efficiency, but LLMs demonstrate the potential for flexible, high-level reasoning with continued training.

Future directions include improving input representations, extending training duration, and exploring hybrid architectures that fuse LLMs with efficient RL backbones or model-based planning. These enhancements could yield more generalizable and sample-efficient agents.

TOKEN UP A NOTCH: TEACHING LLMS TO PLAY MARIO

Ishan Khare
Department of Computer Science
Stanford University
iskhare@stanford.edu

Gabe Seir
Department of Computer Science
Stanford University
gseir@stanford.edu

Anthony Zhan
Department of Computer Science
Stanford University
azhan9@stanford.edu

Abstract

Large language models (LLMs) have achieved remarkable success on static reasoning tasks, but their capabilities as interactive agents in dynamic, real-time environments remain underexplored. In this work, we investigate whether open-source multimodal LLMs can learn effective policies via reinforcement learning. Specifically, we fine-tune a QWEN2.5-VL model using Proximal Policy Optimization (PPO) to play Super Mario Bros, combining image and structured state inputs into a chat-style prompt. As a baseline, we train a vision-based DDQN agent on stacked grayscale frames. The DDQN agent outperformed random and greedy policies, achieving an average reward of 892.3 after 10,000 training episodes. The LLM agent, while initially weak, improved to an average reward of 720.7 after PPO fine-tuning. Qualitatively, we observed that PPO training helped the LLM generalize past simple heuristics, learning to dodge enemies and navigate hazards. However, both agents struggled to generalize to novel levels, with signs of overfitting and limited exploration capacity. Our findings show that LLMs can learn grounded behavior through policy gradient training, but currently fall short of traditional deep RL agents in sample efficiency and robustness. This suggests a promising but still-maturing frontier for LLM-based reinforcement learning. We share our codebase at `https://github.com/iskhare/MarioLLM`.

1 Introduction

Can large language models learn to act through *real-time* interaction?

Large language models (LLMs) have demonstrated impressive reasoning and generalization capabilities across a range of natural language tasks—from instruction following (Zhou et al., 2023a) to code generation and multi-step planning (Grattafiori et al., 2024; OpenAI et al., 2024; Qwen et al., 2025). Yet their deployment as agents in *real-time, feedback-driven environments* remains underexplored. Most LLM applications rely on static prompts, assume single-shot inference (Zhong et al., 2021), and lack any form of adaptation. In contrast, reinforcement learning (RL) offers a complementary paradigm where agents learn to optimize behavior through trial-and-error, grounded in reward signals (Sutton et al., 1998).

We investigate whether **open-source LLMs** can be trained as *interactive agents* in dense, high-speed control settings. Specifically, we embed a fine-tuned QWEN2.5 model into the `SuperMarioBros` environment (Kauten, 2018), where it receives serialized observations of game state and learns to output discrete actions through proximal policy optimization (PPO) (Schulman et al., 2017). Unlike prompt-based or zero-shot approaches (Kojima et al., 2022), our method fine-tunes the LLM in a *closed feedback loop*, treating it as a policy network augmented with a lightweight policy-value head. This transforms the model from a passive predictor into an *active decision-maker* capable of learning from its own gameplay.

`SuperMarioBros` presents a non-trivial benchmark: agents must reason over spatial dynamics, plan over long horizons, and react to unpredictable environment events—all within a fast, partially observable domain. By evaluating LLMs in this setting, we surface practical questions around sample efficiency and generalization.

Key Contributions of this work.

- **Baselines.** We implement standard benchmark agents—including random, greedy, and depth-2 lookahead policies—to establish a lower bound for gameplay performance and isolate the contribution of learning-based methods.
- **Double Deep Q-Network (DDQN).** We train a vision-based DDQN agent (Van Hasselt et al., 2016) using a convolutional architecture on stacked grayscale frames, providing a strong deep RL baseline for comparison and helping characterize the sample efficiency demands of the environment.
- **LLM-based RL Agent.** We integrate a fine-tuned QWEN2.5 model into the control loop via PPO, combining structured game-state inputs with a trainable policy-value head to study whether LLMs can learn grounded, reward-optimized behavior in a real-time setting.

Together, our results provide early evidence on the viability of open-weight LLMs as *trainable, interactive agents*—contributing to a growing body of work at the intersection of language modeling, control, and reinforcement learning.

2 Related Work

LLMs as symbolic agents. Much of the early work on LLM-based agents focused on symbolic reasoning tasks such as tool use, code execution, and web interaction. Agents like ReAct (Yao et al., 2023), Toolformer (Schick et al., 2023), WebArena (Zhou et al., 2023b), and ReSearch (Chen et al., 2025) use chain-of-thought prompting and in-context demonstrations to execute complex action sequences. While effective at language-level planning, these models do not learn from feedback or improve with experience. They operate in stateless or few-shot modes, rather than optimizing policies over time.

LLMs in long-horizon environments. Recent efforts have applied LLMs to open-ended environments requiring sequential decision-making. For instance, Voyager (Wang et al., 2023) uses GPT-4 to explore and complete tasks in Minecraft, combining scripted APIs with auto-curriculum learning. Claude has been shown to play Pokémon via structured inputs and outputs (Lee, 2025). However, these agents typically rely on large proprietary models, tool calling, and fixed world abstractions—sidestepping the learning challenges of exploration, sparse rewards, and online adaptation.

Reinforcement learning for control. Classic deep RL methods such as DDQN (Van Hasselt et al., 2016), PPO (Schulman et al., 2017), and model-based agents like Dreamer (Hafner et al., 2019a) and PlaNet (Hafner et al., 2019b) have proven effective in structured, high-frequency control tasks. These agents learn policies end-to-end from reward and visual input, and set the standard for sample efficiency and stability in dynamic environments. However, they often lack the abstraction and generalization capabilities that LLMs bring.

RL with LLMs. Bridging LLMs and reinforcement learning remains a nascent field. RLHF approaches (Ouyang et al., 2022) use preference models to align LLM outputs with human intent, but typically rely on offline data. Recent work has begun exploring online LLM adaptation through tool interaction (Wang et al., 2025), reward shaping (Shinn et al., 2023), or imitation. Yet few studies evaluate whether LLMs can function as full-fledged RL agents capable of optimizing long-term return in continuous feedback settings.

Positioning of this work. Our study combines these threads by evaluating whether open-source LLMs—specifically a fine-tuned QWEN2.5 model—can act as real-time, reward-optimizing agents in the `SuperMarioBros` environment (Kauten, 2018). We implement a PPO-based control loop with structured inputs, benchmark against DDQN and scripted baselines, and investigate how LLMs perform under dense, temporally extended reward signals. In doing so, we move beyond prompt-only agents and contribute empirical insights into the feasibility of LLMs for embodied control.

3 Method

3.1 Overview

Our system investigates reinforcement learning (RL) for large language models (LLMs) in the context of a discrete-action platformer (Super Mario Bros). We implement two separate agents:

1. A vision-based **DDQN agent** trained with Q-learning on stacked grayscale frames as displayed in Figure 1.
2. A **PPO-trained LLM agent** that conditions on serialized game states and screenshots.

We describe the architecture, learning objectives, and reward design for each agent, along with the action space and prompting format used for policy conditioning.

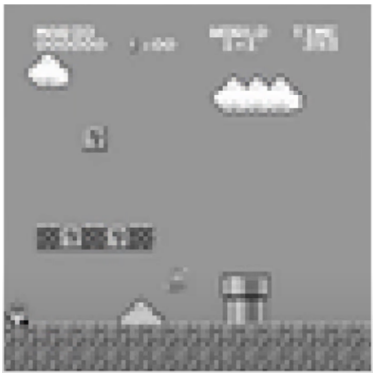

Figure 1: Example grayscale input frame down-scaled to 84×84 pixels.

3.2 Double Deep Q-Network (DDQN) Agent

We use a standard Double Deep Q-Network (DDQN) architecture (Van Hasselt et al., 2016), with the following components:

- **Input:** A stack of 4 preprocessed grayscale frames (84×84). An example of a single frame is shown in Figure 1.

- **CNN Backbone:** Three convolutional layers: 32 filters (kernel=8, stride=4), 64 filters (kernel=4, stride=2), and 64 filters (kernel=3, stride=1), all with ReLU activations.
- **Multilayer Perceptrons (MLPs):** Two MLPs with 512 hidden units each.
- **Action selection:** Epsilon-greedy policy.

An visual depiction of the DDQN architecture is given in Figure 2.

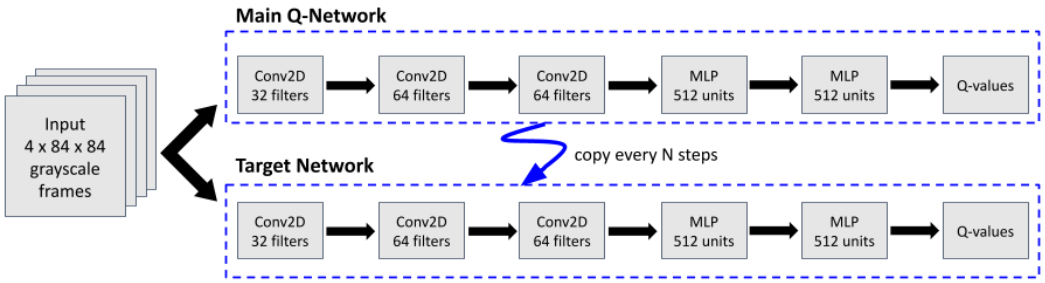

Figure 2: Diagram of the DDQN Architecture.

3.2.1 Why DDQN over DQN.

In traditional Deep Q-Networks (DQN) (Mnih et al., 2015), the target for the Q-learning update is computed using the same network both to select and evaluate the next action:

$$\mathcal{L}_{\text{DQN}} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right], \tag{1}$$

where θ are the online network parameters and θ^- are the target network parameters. This can lead to an **overestimation bias**, since $\max_{a'} Q(s', a')$ may overestimate the value of the next state due to being both the selector and evaluator.

Double DQN (DDQN) addresses this by *decoupling the action selection from action evaluation*. The action is selected using the online network, but its value is estimated using the target network. This reduces over-optimistic value estimates and improves stability.

3.2.2 Temporal Difference Loss in DDQN.

In DDQN, the target is computed by first selecting the action with the highest value according to the online network, then evaluating that action using the target network:

$$\text{Target} = r + \gamma Q(s', a^*; \theta^-), \text{ where } a^* = \arg\max_{a'} Q(s', a'; \theta). \tag{2}$$

Now, incorporating what we have from Equations 1 and 2 results in the following temporal difference loss which is used to learn the DDQN:

$$\mathcal{L}_{\text{DDQN}} = \mathbb{E}\left[\left(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s, a; \theta)\right)^2\right]. \tag{3}$$

This formulation ensures that the Q-value estimate used for bootstrapping is less biased, leading to more stable learning and better policy performance.

3.2.3 DDQN Agent Action Space and Reward

The DDQN agent operates within a constrained action space designed to simplify learning and encourage horizontal progression through the level. Specifically, the agent is limited to the `RIGHT_ONLY` action set provided by `gym-super-mario-bros`, which includes:

{ `NOOP`, `RIGHT`, `RIGHT+JUMP`, `RIGHT+RUN`, `RIGHT+RUN+JUMP` }

This restriction reduces the complexity of the exploration space and aligns with the design goal of emphasizing forward movement in the environment, making it more tractable for early-stage deep reinforcement learning.

The reward function is designed to encourage forward progress, penalize death, and promote efficient completion. Let x_t denote Mario's horizontal position at timestep t, and $\delta_t = x_t - x_{t-1}$ be the change in position. The reward r_t at time t is defined as:

$$r_t = \begin{cases} 0 & \text{if Mario dies at } t \\ \delta_t - \lambda & \text{otherwise} \end{cases} \tag{4}$$

where $\lambda > 0$ is a small time penalty (e.g., $\lambda = 0.01$) applied at each timestep to discourage idling and encourage faster level traversal.

3.3 Proximal Policy Optimization (PPO) LLM Agent

We use an open-source multimodal LLM (QWEN2.5-VL) fine-tuned with Proximal Policy Optimization (PPO). The agent conditions on both:

- **Screenshot:** Rendered game frame, passed as an image input.
- **Serialized State:** Textual JSON-based game state including Mario's position, score, world, and enemies.

These are combined into a prompt, formatted using a chat template as given in Section 3.3.1.

The model generates hidden states which are fed into a policy and value head:

$$\pi_\theta(a|s) = \text{Categorical}\left(\text{MLP}(h_T)\right) \text{ and } V_\theta(s) = \text{MLP}(h_T) \tag{5}$$

where h_T is the final hidden state.

3.3.1 LLM Agent Input Prompt

```
SYSTEM_PROMPT = """
You are an AI agent playing Super Mario Bros.
Analyze the game state and choose the best action.

Available actions:
0: NOOP            - Do nothing
1: RIGHT           - Move right
2: RIGHT+JUMP      - Move right and jump
3: RIGHT+RUN       - Move right and run
4: RIGHT+RUN+JUMP  - Move right, run and jump (best for long jumps)
5: JUMP            - Jump in place
6: LEFT            - Move left (avoid unless necessary)

Strategy:
- Always progress right to complete the level
- Jump over enemies and gaps
- Use running for speed and longer jumps
- Collect coins when safe
- Avoid getting stuck
- Complete the level as fast as possible
"""
```

3.3.2 Reward and Action Space

The LLM agent operates over a discrete action space consisting of 7 high-level commands: {`NOOP`, `RIGHT`, `RIGHT+JUMP`, `RIGHT+RUN`, `RIGHT+RUN+JUMP`, `JUMP`, `LEFT`}. These were selected to provide a compact yet expressive control interface suitable for language model reasoning.

The reward function used is identical to the DDQN reward from Equation 4. This reward structure simplifies training dynamics for PPO while still providing a dense signal for policy improvement.

3.3.3 PPO Loss Formulation

We optimize the clipped surrogate PPO objective:

$$\mathcal{L}_{\text{PPO}} = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right], \text{ where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{6}$$

with advantage estimates computed via Generalized Advantage Estimation (GAE) (Schulman et al., 2015):

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots$$
$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

The total loss includes three terms:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{PPO}} + c_1 \cdot \mathcal{L}_{\text{value}} - c_2 \cdot \mathcal{H}[\pi_\theta], \tag{7}$$

where $\mathcal{L}_{\text{value}}$ is the MSE loss between $V(s)$ and empirical return, and $\mathcal{H}$ is the entropy bonus.

3.3.4 Parameter-Efficient Fine-Tuning with LoRA

To make fine-tuning tractable for large multimodal LLMs, we adopt Low-Rank Adaptation (LoRA) (Hu et al., 2022) as a parameter-efficient tuning strategy. Instead of updating all model weights, LoRA introduces small trainable rank-decomposition matrices into attention and feedforward layers, significantly reducing the number of updated parameters.

We apply LoRA to the QWEN2.5-VL model using rank $r = 16$, dropout rate of 0.05, and $\alpha = 32$, targeting key transformer modules such as query/key/value projections and feedforward blocks. The rest of the model weights remain frozen.

This setup enables gradient updates on a lightweight set of injected parameters, improving training speed and reducing GPU memory usage. Combined with mixed-precision (`bf16`) training, quantization, and gradient checkpointing, this allows stable PPO fine-tuning within a single-GPU setup.

The LoRA-adapted LLM outputs final hidden states h_T, which are then passed into separate MLP heads for policy and value prediction (see Equation 5).

4 Experimental Setup

4.1 Task and Environment

We evaluate agents in the `SuperMarioBros-1-1-v0` environment from the `gym-super-mario-bros` suite (Kauten, 2018). The agent's objective is to complete the level as efficiently as possible while avoiding obstacles and enemies. The environment emits pixel observations and game metadata at 60 FPS, which we downsample and preprocess (see Section 3).

4.2 Baselines and Metrics

To compare agents, we track the total reward which is the sum of timestep-level rewards per episode. All experiments are averaged over multiple episodes (typically 10–20) for evaluation.

We implement three non-learning baselines for comparison:

- **Random:** Uniformly samples an action at each timestep.
- **Greedy:** Repeatedly selects the action that maximizes the reward at the next time step.
- **Depth-2 Lookahead:** Enumerates all two-step action sequences and chooses the one that maximizes immediate reward using a lightweight heuristic model.

These provide simple behavioral references and help isolate the gains from learned policies.

4.3 DDQN Agent

The DDQN agent observes stacks of 4 grayscale 84×84 frames and selects from a constrained `RIGHT_ONLY` action set: {`NOOP`, `RIGHT`, `RIGHT+JUMP`, `RIGHT+RUN`, `RIGHT+RUN+JUMP`}. The network is trained with the Double Q-learning objective using a target network and replay buffer. The DDQN agent is trained using the settings described in Table 1.

Hyperparameter	**Value**
Learning rate	2.5×10^{-4}
Discount factor (γ)	0.9
Replay buffer size	100,000 transitions
Batch size	256
Target network sync interval	2,500 steps
Training episodes (Attempted)	25,000 steps
Training episodes (Actual)	10,000 steps
Exploration schedule (ϵ-greedy)	Decay from 1.0 to 0.1

Table 1: Key hyperparameters used for training the DDQN agent.

4.4 LLM PPO Agent

The LLM agent is trained using Proximal Policy Optimization (PPO) on serialized game-state JSON and base64-encoded screenshots, both of which are passed to a fine-tuned QWEN2.5-VL model via a structured system prompt (Section 3.3.1). The model outputs logits for 7 discrete actions: {`NOOP`, `RIGHT`, `RIGHT+JUMP`, `RIGHT+RUN`, `RIGHT+RUN+JUMP`, `JUMP`, `LEFT`}.

Training uses simple linear layers for both the policy and value head, trained on top of the final hidden state of the underlying LLM. We optimize for the standard PPO loss (see Section 3). The key hyperparameters for this agent are given in Table 2. Fine-tuning is performed with LoRA adapters using `bf16` precision along with 4-bit quantization using `bitsandbytes` for more efficient GPU utilization, and logged via Weights & Biases.

Hyperparameter	**Value**
Learning rate	1×10^{-5}
Discount factor (γ)	0.99
Clip range	0.2
GAE parameter (λ)	0.95
Mini-batch size	16
PPO epochs	4
Value loss coefficient	0.1
Entropy coefficient	0.01

Table 2: Key hyperparameters used for training the PPO-based LLM agent.

5 Results

5.1 Quantitative Evaluation

We began by evaluating the training of the DDQN agent across 10,000 episodes. Figure 3 shows how the agent received mostly random rewards, especially at the beginning of training, displaying how the agent was in its exploration phase. As ϵ decreased and the agent began to use exploitation more often, the average reward began to increase slightly as in Figure 4. However, the curve shows the reward was still mostly random showing that more training would be beneficial.

To evaluate the performance of our agents, we measured the average reward obtained by the agents in level 1 across 1,000 episodes. We also tested our DDQN agent after 2,500 training episodes and after 10,000 training episodes.

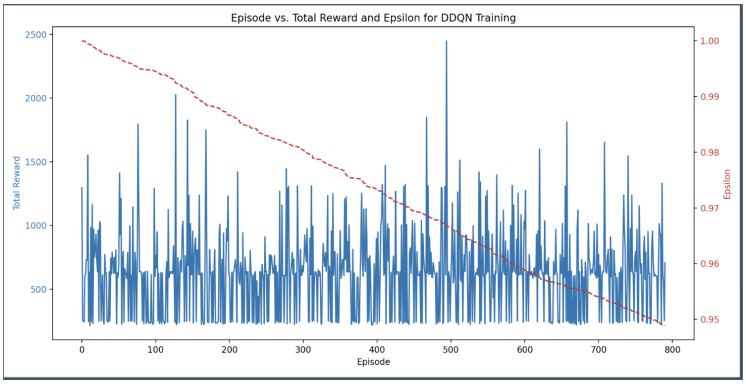

Figure 3: Reward over first 1,000 episodes of training.

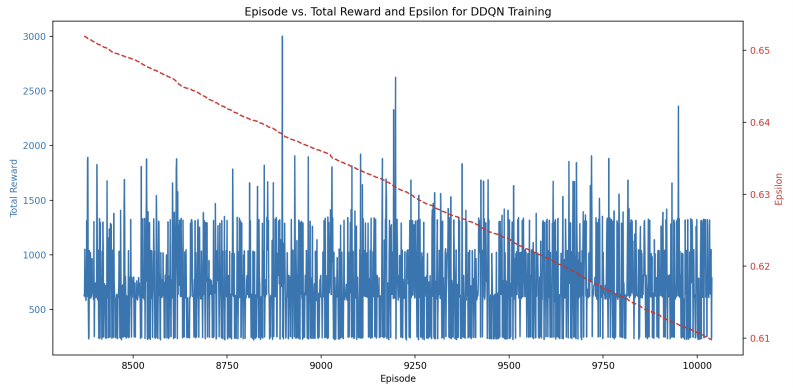

Figure 4: Reward over final 1,500 episodes of training.

From these results, we can see that that the DDQN agent was able to learn specific behavior that enabled it to perform better than random. Additionally, further training improved upon the average return for the DDQN agent indicating that further strengthening the indication that additional training would be beneficial.

Model	**Average Reward**
Random	616.8
Greedy	236.0
Lookahead (depth = 2)	1046.0
LLM (no RL fine-tuning)	446.1
DDQN (2,500)	827.9
DDQN (10,000)	892.3
LLM w/ PPO	720.7

Table 3: Average reward across 1,000 evaluation episodes for each model on level 1.

The LLM agent with no RL fine-tuning performed only slightly better than a Greedy agent, indicating that this approach only provided actions that maximized the additional next step reward even if this was not beneficial long-term. With PPO training, the model substantially improved but didn't reach the same level as the pure RL agents, presumably because of undertraining (due to time and resource constraints) or other causes discussed in the next section. As seen in the training plots (Figures 5 and 6), the reward curve is mostly noise, although the loss does seem to decrease (in an albeit unstable manner). The frequency of high reward (>1000) episodes also increased throughout training, suggesting that the model was indeed learning useful strategies, but was unable to fully break free of its failure modes.

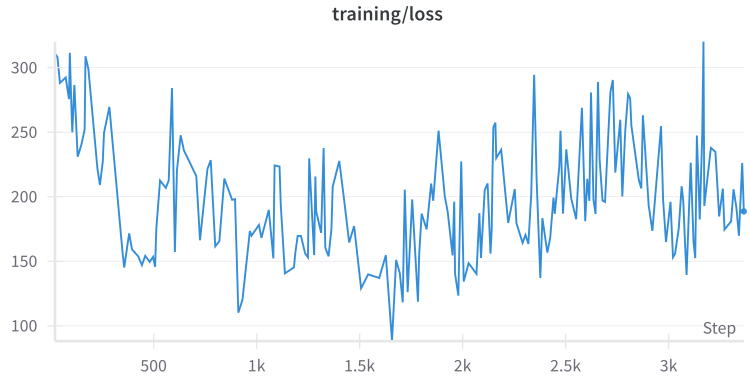

Figure 5: Loss curve for LLM PPO training.

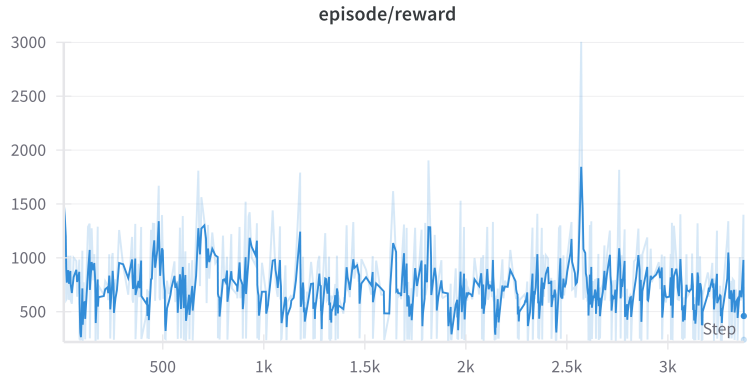

Figure 6: Reward curve for LLM PPO training.

Overall, the Lookahead agent with depth two was the highest performing agent demonstrating that taking into account future rewards was the most beneficial approach, further showing that as our DDQN agent improved it would likely lead to increased performance.

We additionally tested the baseline and DDQN models on level 2 to analyze the generalizability of our model from learning on level 1 to executing on level 2.

Model	**Average Reward**
Random	877.1
Greedy	105.0
Lookahead (depth = 2)	874.0
DDQN (2,500)	439.1
DDQN (10,000)	375.1

Table 4: Average reward across 1,000 evaluation episodes for each model on level 2.

Level 2 presents Mario with increased difficulty in the form of more enemies and harder to traverse obstacles. These results indicate that the DDQN agent was able to generalize some of its training from level 1 due to its superior performance to the Greedy baseline. However, the performance was still worse than the Lookahead or Random agents indicating that the training done on level 1 was not sufficient to create a performant agent on other levels. Surprisingly, we found that the DDQN agent that was trained for more episodes actually performed worse than the one trained on fewer episodes. This did not make sense to us as the increased training should allow the agent to more

expertly navigate the enemies and obstacles, even in a new environment. We hypothesized that the 10,000 episode agent may have overfit to the level 1 environment, but when observing the agent play through the level actually found a much more intuitive qualitative explanation to this behavior described below.

5.2 Qualitative Analysis

As our qualitative analysis, we observed our agents playing various levels in the Super Mario Bros environment. Specifically, we visually compared agents in the first two levels of the game consisting of obstacles (pipes that Mario needs to jump over, enemies (Goombas), and holes where Mario can fall and die.

Beginning with the first level, for the Greedy and LLM with no RL agents, we observed behavior consistent with the agent repeatedly choosing the "RIGHT" action. This means that Mario would move without jumping until it hit the first Goomba and then die (Figure 7). This behavior was expected from the Greedy agent as always moving right would maximize short-term reward (moving to the right quickly) while being unable to understand the benefit of not immediately dying. However, we were surprised to observe that the LLM with no RL agent wasn't able to understand the game-state context enough to jump over the first Goomba. This validated our hypothesis that we needed to include an RL in the loop approach to reinforce behavior that maximized improved long-term outcomes.

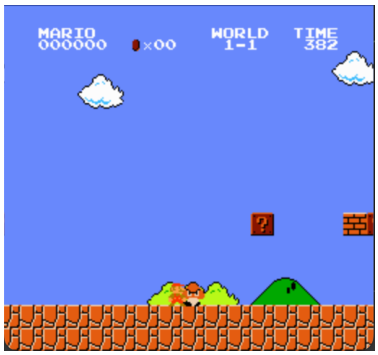

Figure 7: Greedy Mario Behavior.

The lookahead agent with depth 2 was the agent that most consistently was able to dodge the first round of enemies and advance through the level. However, it hit a block in progression when it was forced to time a jump to avoid enemies. Specifically, the agent consistently died when it came across a situation with moving enemies where it had to jump over a pipe and land between enemies (Figure 8). This shows how the lookahead agent lacked the ability to understand that its next move may put it in a poor position to succeed if that move lasted longer than one additional turn (as is the case with a long jump).

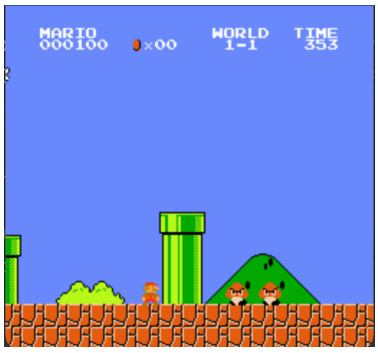

Figure 8: Mario jumping over pipe into two Goombas.

The random agent performed qualitatively as expected with Mario sometimes dying immediately, and other times progressing deep into the level and avoiding many enemies. This behavior was replicated early in the DDQN agent's training with the DDQN agent exhibiting random behavior during its early exploration phase. As training went on, the DDQN agent began to more consistently avoid the first Goomba in the level and progressed deeper into the level. Specifically, after 2,500 training episodes Mario often runs into a single enemy when jumping over a pipe (Figure 9). After 10,000 training episodes, Mario is able to consistently pass this enemy before dying at the next junction. This shows how many training episodes lead to incremental improvement and understanding of the environment, and underscore the need for further training.

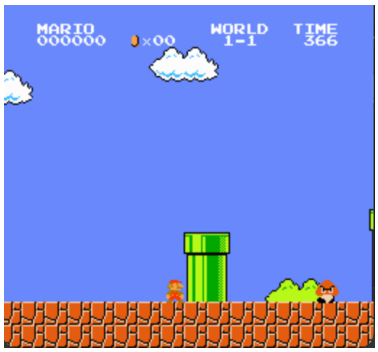

Figure 9: Mario jumping over pipe into Goomba.

With the LLM agent, we initially expected it to easily outperform pure RL baselines due to sheer size, scaling laws, zero-shot generalization, etc. However, without fine-tuning, the LLM was susceptible to dying (e.g. by greedily holding right) or falling into traps (e.g. running into a pipe over and over) early in the episode. With PPO training, performance improved slightly, and the model started to venture farther into the level more consistently (as seen in Figure 6). Despite these glimmers of hope, the LLM agent was still too prone to early traps to dethrone DDQN and Lookahead based on average reward. We hypothesize that either the fully general nature of pretrained + instruct-tuned LLMs isn't wholly compatible with this specific, out-of-distribution task of playing Mario, or the model couldn't learn to process the input in a way that fully realized the model's potential.

Progressing to the second level, we wanted to analyze if training solely on the first-level was generalizable. We were only able to analyze the DDQN agent in this scenario due to compute limitations with the LLM agent.

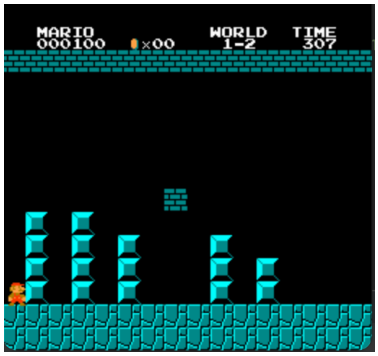

Figure 10: Mario stuck behind tower of blocks.

The DDQN agent is able to generalize many of the things it learned during training in the first level to the second level. It successfully dodges some of the enemies and is able to avoid initial obstacles. The main difference in the behavior observed in level two is that there are two places where the DDQN agent gets stuck. The first is a spot directly in front of a tall tower of blocks (Figure 10) where only a perfectly timed use of the RIGHT + JUMP action would be able to save the agent, a behavior that it would not have encountered in the first level at all. The second place where the agent gets stuck is a

box where moving right no longer works (Figure 11). This demonstrates (1) the limitation of our limited action space where Mario cannot move left and undo a failed move right, but also (2) a lack of context awareness that getting stuck is possible as this scenario was not encountered in the first level.

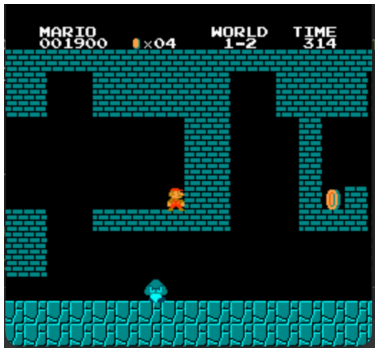

Figure 11: Mario stuck where it cannot proceed with RIGHT action.

Interestingly, we found that quantitatively the DDQN agent trained with 10,000 episodes performed worse than the one trained with only 2,500 episodes. However, when observing the gameplay of these agents we found that the agent trained on more episodes actually progressed deeper into the level more often, but got stuck in the scenario presented in Figure 11 often. This led to many negative rewards for the agent (as the agent would run out of time on that episode leading to negative reward accumulation) and explains the worse performance observed quantitatively.

The baseline algorithms perform similarly to the first level as expected. The Random agent often becomes stuck in the same places that the DDQN agent becomes stuck in leading to lower rewards. Additionally, however, as there are more enemies in the second level the Random agent often dies early in the run as well. The Greedy agent once again only moves to the right, and dies early in the level. The Lookahead with depth two agent is still able to perform quite well, managing to escape easily when it gets stuck in the scenario in Figure 10. It is able to progress past many of the enemies and even avoids getting stuck in the scenario in Figure 11. This was somewhat surprising as moving upwards should've enabled a similar outcome as moving right (within a depth of two moves). However, the agent still died shortly after this when encountering scenarios where there are many enemies and timing was required.

6 Discussion

Our results show how Q-learning is not only possible, but effective in this context. However, they have several limitations. We were not able to train the DDQN agent for sufficient episodes to enable true learned behavior. After 10,000 episodes of training our agent still displayed mostly random behavior, which is consistent with the fact that our ϵ parameter had not decayed sufficiently to significantly encourage exploitative behavior. We wished to train this agent for at least 50,000 episodes but were heavily compute limited as even training for 10,000 episodes required 5 days of computing. This indicates that there is a significant limitation with sampling efficiency that needs to be refined in order for long-term training.

Multimodal LLMs are a promising direction for game-playing agents, able to leverage vast amounts of general reasoning and language- and image-based data. Ultimately, however, these models aren't trained to maximize rewards in very specific game environments, which puts them at a disadvantage compared to pure RL methods. We explored PPO fine-tuning as a way to bridge these gaps, but were only able to partially do so, perhaps due to resource constraints or more fundamental limits with our setup. In this PPO setting, the 3B parameter size is both a blessing and a curse: it allows for very general understanding of the task and goal, with mostly coherent generations, but is extremely costly to run in terms of both memory and wall time (relative to other baselines). Nevertheless, the model's ability to string together high-reward actions over long time scales in certain episodes suggests that this approach still has promise.

7 Conclusion

Overall, our work shows that RL can be implemented in different ways — traditional methods such as DQNs, or newer methods such as with LLMs — to successfully train agents to play Mario, with varying degrees of success depending on the approach.

Quantitatively, the DDQN agent consistently outperformed random and greedy baselines on level 1, achieving an average reward of 892.3 after 10,000 training episodes. While it did not surpass the scripted lookahead agent, it demonstrated the ability to learn effective behaviors over time. The LLM agent, when trained with PPO, improved significantly over its non-finetuned counterpart (720.7 vs. 446.1 average reward), showing that policy gradient fine-tuning can extract useful decision-making behavior. However, it still underperformed compared to DDQN, likely due to limited training and the LLM's general-purpose nature. Notably, generalization to level 2 remained a challenge for all agents, with DDQN exhibiting unexpected degradation in performance — likely due to overfitting or limitations in action diversity.

This means that large language models, while not yet outperforming deep RL agents in sample efficiency or reliability, can learn grounded policies and execute multi-step reasoning through structured training. Their ability to improve with PPO signals that RL with LLMs is a viable and evolving paradigm.

In the future, we wish to explore deeper PPO fine-tuning for LLMs, alternative input encodings (such as temporal attention across frames), and hybrid systems that combine LLM priors with efficient RL modules. We are also interested in incorporating model-based planning or curriculum learning to enhance generalization and sample efficiency across diverse game environments.

8 Team Contributions

- **Ishan Khare:** Implemented the initial DDQN agent and conducted early deep RL experiments. Contributed to training diagnostics, refinement of reward structure, and overall coordination of the research direction. Assisted in writing and revising all sections of the report.
- **Gabe Seir:** Set up the core codebase and Super Mario Bros environment. Ran baseline experiments (random, greedy, lookahead) and extended the DDQN training. Developed evaluation scripts and reproducibility infrastructure. Led model evaluation and results analysis and assisted in writing and revising all sections of the report.
- **Anthony Zhan:** Led the implementation of the LLM-based agent, passing in pre-processed game-state info and using PPO fine-tuning. Developed the prompting logic and integrated LoRA-based training with quantization into the QWEN2.5-VL model. Assisted in writing and revising results, discussion, and conclusion.

Changes from Proposal Compared to the initial project proposal, our implementation shifted away from model-based planning (MCP loop) and expert trajectory generation toward direct reinforcement learning with PPO. We did not implement behavioral cloning or REINFORCE as originally planned. Instead, we focused on training the LLM agent via PPO with online experience collection and feedback. Additionally, our DDQN implementation took on a larger scope than initially expected, serving as a key benchmark. All team members contributed to analysis, discussion, and collaborative writing throughout the final report.

References

Mingyang Chen, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. 2025. ReSearch: Learning to Reason with Search for LLMs via Reinforcement Learning. arXiv:2503.19470 [cs.AI] https://arxiv.org/abs/2503.19470

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. 2019a. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603* (2019).

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2019b. Learning Latent Dynamics for Planning from Pixels. arXiv:1811.04551 [cs.LG] https://arxiv.org/abs/1811.04551

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

Christian Kauten. 2018. Super Mario Bros for OpenAI Gym. GitHub. https://github.com/Kautenja/gym-super-mario-bros

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

Benj Edwards Lee. 2025. *Why Anthropic's Claude still hasn't beaten Pokémon*. https://arstechnica.com/ai/2025/03/why-anthropics-claude-still-hasnt-beaten-pokemon/ Accessed: 2025-04-23.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, et al. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI] `https://arxiv.org/abs/2303.11366`

Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (2023).

Hanlin Wang, Chak Tou Leong, Jiashuo Wang, Jian Wang, and Wenjie Li. 2025. SPA-RL: Reinforcing LLM Agents via Stepwise Progress Attribution. arXiv:2505.20732 [cs.CL] `https://arxiv.org/abs/2505.20732`

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Ruiqi Zhong, Kristy Lee, Zheng Zhang, and Dan Klein. 2021. Adapting language models for zero-shot learning by meta-tuning on dataset and prompt collections. *arXiv preprint arXiv:2104.04670* (2021).

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023a. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911* (2023).

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023b. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).