What Makes Rainbow Shine? A Modular Study of DQN Enhancements in Practice

Max Fong[1], William Kiem Lafond[1], Denis Tsariov[1]
[1]Department of Computer Science,
McGill University
{max.fong, william.lafond, denis.tsariov}@mail.mcgill.com

Abstract

This project recreates DeepMind's Rainbow—an algorithm that integrates six key improvements to the classic DQN (Deep Q-Learning) framework into a single, unified agent. We initially tested our implementation using Gymnasium's `AnyTrading` environment, but due to inconsistent results in financial domains, we pivoted to three Atari games—*Seaquest*, *Asterix*, and *Road Runner*—to conduct controlled ablation studies. Our implementation was built from scratch and structured to support modular experimentation. Denis developed the foundational agents and training loop. William implemented several of the Rainbow enhancements, including prioritized replay, dueling networks, and noisy layers. Max contributed the multi-step returns and distributional RL modules, and also helped refactor the codebase into a clean, toggleable architecture. Together, we conducted systematic ablation studies to evaluate the individual and combined impact of these components on agent performance.

1 Introduction

Rainbow (4) achieved state-of-the-art results on the Atari 2600 benchmark by combining six key improvements to the original DQN framework[1]—Double Q-learning, dueling networks, prioritized replay, multi-step returns, distributional RL, and noisy layers—into a single, unified agent (we provide a detailed overview of these components in Section 2). These components were shown to be largely complementary, offering both greater data efficiency and final performance.

Motivated by its general-purpose design, we set out to test whether Rainbow could extend to real-world, high-volatility environments—specifically, quantitative trading, where reinforcement learning faces additional challenges such as stochastic transitions, sparse rewards, and non-stationarity[2].

We reimplemented Rainbow DQN from scratch and evaluated it in both financial and game-based settings. While initial tests in Gymnasium's `AnyTrading` environment yielded inconsistent results—likely due to the domain's noisy dynamics—we shifted to three Atari games: *Seaquest*, *Asterix*, and *Road Runner*, where we conducted controlled ablation studies to validate our implementation.

Our key contribution lies in the design of `CombinedAgent`, a ready-to-use modular class that allows Rainbow components to be toggled independently. This efficient ablation tool supports experimentation with the full Rainbow architecture and simplifies isolating the contribution of each component. Our full implementation—including training logs, ablation toggles, and plots—is available at `https://github.com/Waterfountain10/comp579_project`.

[1] DQN was introduced by Mnih et al. (3) as a deep RL algorithm that learns directly from raw pixels.

[2] RL in finance is difficult due to noise, sparse signals, and non-stationarity (1; 2).

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

2 Background

2.1 Rainbow DQN and Its Components

Rainbow builds on Deep Q-Learning (DQN) (3), which approximates the optimal action-value function using neural networks and a replay buffer for stabilization. Rainbow integrates six major extensions that each address different limitations of the original DQN. (For full mathematical definitions and formula derivations, see Appendix A.

Double Q-learning. Mitigates Q-learning's tendency to overestimate action values by decoupling action selection from evaluation, improving stability (5). The updated target becomes:

$$r + \gamma Q_{\theta'} \left(s', \arg\max_{a'} Q_\theta(s', a') \right) \tag{1}$$

Prioritized Replay. Rather than sampling transitions uniformly, this strategy replays transitions with higher TD error more frequently to speed up learning (6). Transitions are sampled with probability:

$$p_t \propto \left| r + \gamma \max_{a'} Q_{\theta'}(s', a') - Q_\theta(s, a) \right|^\omega \tag{2}$$

where ω controls prioritization strength.

Dueling Networks. Separates the estimation of state value and action advantage, enabling better learning in states where the choice of action matters less (7):

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \tag{3}$$

Noisy Networks. Replaces ϵ-greedy exploration with trainable noise in the weights (8):

$$W' = W_\mu + W_\sigma \odot \varepsilon_w, \quad b' = b_\mu + b_\sigma \odot \varepsilon_b \tag{4}$$

allowing the agent to learn when and how to explore.

Multi-step Returns. Uses cumulative discounted rewards over multiple steps to propagate information more effectively (12):

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} \tag{5}$$

Distributional RL. Instead of predicting expected returns, Distributional RL models a full distribution (9):

$$Q(s, a) = \sum_{i=1}^{N_z} z_i \cdot p_i(s, a) \tag{6}$$

where z_i are fixed support atoms and p_i are learned probabilities.

2.2 Quantitative Trading

Quantitative trading involves the systematic execution of trading strategies based on mathematical models and statistical patterns in financial data. Rather than relying on human intuition, quantitative systems automate decision-making using data-driven signals, often across large volumes of assets (10).

Framing as an MDP. A task can be formalized as a Markov Decision Process (MDP), where:

- **States** represent market observations, such as recent price trends.
- **Actions** correspond to discrete trading decisions—e.g., `buy`, `sell`, or `hold`.
- **Rewards** reflect gains or losses, typically computed as changes in portfolio value.

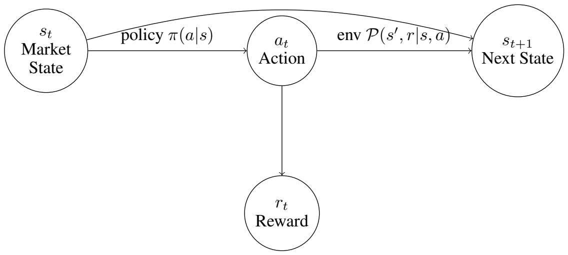

Figure 1: MDP structure for reinforcement learning in quantitative trading.

Episodes unfold over fixed trading windows, and the agent learns a policy that maximizes cumulative returns. Due to the non-stationary nature of financial markets, traditional RL assumptions such as stationary dynamics and dense rewards often break down (11).

3 Related Work

Recent work has explored deep RL in finance for tasks like portfolio optimization and trading (1; 2; 11). While Rainbow DQN performs well in game-based benchmarks (4), its application to real-world financial data is rare. We extend this line by evaluating Rainbow in volatile Forex markets and performing ablations to assess each component's contribution under non-stationary dynamics.

4 Methodology

4.1 Setting up the Environments

Forex: We used the `gymnasium-anytrading` environment (14) as our financial benchmark. Instead of the default stock datasets, we imported real-world foreign exchange (Forex) data for currency pairs such as USD/CHF and inverted them to CHF/USD to match our reward semantics. Preprocessing was done via a custom script that reshaped the columns, inverted prices, rounded to 6 decimal places, and filtered by time granularity (4-hour candles).

Atari: To validate our Rainbow implementation in dense-reward settings, we benchmarked three standard ALE Atari environments: *Seaquest*, *Asterix*, and *Road Runner*.

4.2 Implementing Each Rainbow Component

To support modular experimentation, we implemented each Rainbow enhancement as a standalone module that integrates into the base DQN structure.

- **DQN:** We built a 3-layer feedforward neural network. For Atari environments, the hidden size is set to 512; for Forex, we use 256 for faster convergence on lower-dimensional input.
- **Prioritized Experience Replay (PER):** We replaced the standard replay buffer with a `PrioritizedReplayBuffer` that uses segment trees for efficient TD-error-based sampling. During training, losses are scaled by importance-sampling weights, and priorities are updated post-optimization based on TD magnitudes. See Appendix A for our segment tree implementation.
- **Dueling Networks:** We added a second value stream to the Q-network. The final output computes $Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')$, improving learning in states where the action choice has limited impact.
- **Noisy Networks:** We replaced linear layers with `NoisyLayer` modules that inject factorized Gaussian noise into weights and biases. These noise parameters are trainable and reset after each update step to enable adaptive exploration.
- **n-Step Returns:** We accumulated rewards across $n = 3$ transitions using a fixed-length queue. When filled, we compute discounted returns for the oldest experience and store it in the main buffer, helping the agent learn from longer-term reward signals.
- **Distributional RL (C51):** The final layer of the network outputs a categorical distribution over 51 fixed return atoms between $v_{\min}$ and $v_{\max}$. We compute the expected Q-value via a softmax-weighted sum and apply cross-entropy loss to train the full return distribution.

4.3 Combining Everything with `CombinedAgent`

After implementing each Rainbow ingredient, we packaged them into one central hub: `CombinedAgent`. This class manages environment interaction, model training, and toggling between features. Two submodules encapsulate most architectural differences: `CombinedNeuralNet` handles Dueling, Noisy, and Distributional features, while `CombinedBuffer` manages Prioritized Replay and n-step returns. All components are fully modular and controlled via CLI flags.

We provide a `main.py` script that supports launching agents with any combination of enhancements. For instance:

```
python main.py -useNoisy -usePrioritized -lr 0.003
```

spawns an agent with noisy exploration and prioritized replay. If no flags are set, the script defaults to a baseline DQN agent.

To assess the contribution of each Rainbow component, we implemented an `ablation` flag that activates the full Rainbow agent and systematically disables individual components across controlled experiments. (For pseudo-code of components, see Appendix B.)

5 Experiments

5.1 Final Runs and Results

After sweeping hyperparameters with the Forex environments, we noticed that results were too unstable and decided to commit to Atari benchmarks. We conducted full-scale experiments on three Atari games using the tuned hyperparameters from our earlier sweep (see Appendix C). *Seaquest* and *Road Runner* were trained for 2 000 episodes each, while *Asterix* required 2 000 episodes as well, given its slower convergence. A testing phase of 250 episodes followed, with learning disabled to evaluate generalization. Each setup was repeated across 3 random seeds, with results averaged and smoothed using a 50-episode moving average. Details on the compute resources used are specified in Appendix E.

We performed two experiment studies: base agent comparisons and full Rainbow ablations with each component isolated. Full experiment plots appear in Appendix D.

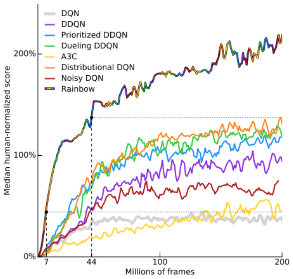

Figure 2: Original paper: Rainbow=rainbow color

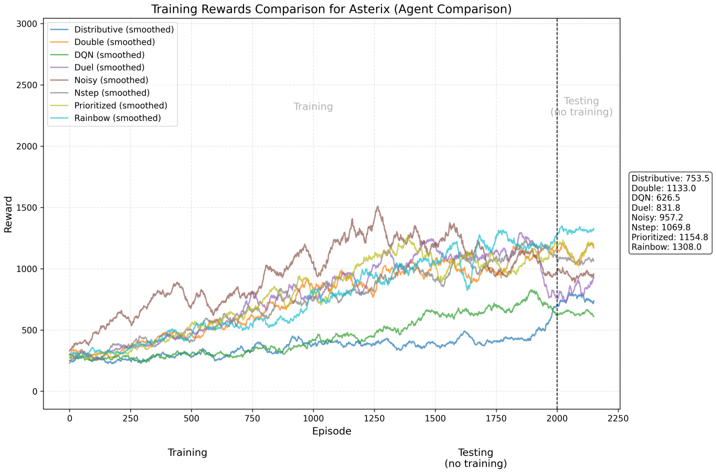

Figure 3: Our reproduction: Asterix - Agent Comparison (3 seeds, smoothed). Rainbow=light blue

We can see similar trends across agents. For instance, Rainbow beats all other agents, and DQN has the lowest performance. The original paper exhibits greater variance, but this is expected since they trained on millions more frames, whereas our experiments comprised only $2{,}250$ episodes $\times$ 700 steps $= 1{,}575{,}000$ frames. This suggests further tuning.

Looking at Table 1, the full Rainbow agent not only converges fastest but also achieves the highest test reward, underscoring how its components complement each other. Conversely, removing **Noisy Networks** yields the slowest convergence, likely because noise-driven exploration is critical for early performance. These trends mirror those reported in the original Rainbow paper (4).

Table 1: Seaquest Ablation Study (3 seeds, 900 training eps. → 200 test eps.)

Removed	% to Converg.	Avg. Test Reward
None (Rainbow)	60%	153.9
Noisy	**83%**	77.7
PER	71%	22.7
Dueling	67%	136.8
Distributive	70%	3.9
N-step	63%	165.4
DDQN	61%	138.3

6 Conclusions and Limitations

Our modular reimplementation of Rainbow confirms that its combined enhancements—especially **Noisy Networks** and **Prioritized Experience Replay**—consistently improve performance over vanilla DQN and standalone variants. In Atari benchmarks, the full Rainbow agent achieved faster convergence and higher rewards, with ablation studies highlighting the complementary nature of its components. Although our initial goal was to evaluate Rainbow in high-volatility financial settings, results in the Forex domain were unstable, prompting a shift to more controlled environments.

While our exploratory Forex runs showed some promise, they were constrained by non-stationary dynamics, a limited asset set (CHF/USD), and simplified assumptions like discrete actions and no trading costs. Our Distributional RL module also underperformed, suggesting the need for further hyperparameter tuning. Future work will explore continuous control methods (e.g., PPO, D4PG), multi-asset portfolios, and more realistic trading constraints to better assess Rainbow's practical viability.

References

[1] J. Moody and M. Saffell. Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12(4):875–889, 2001.

[2] Z. Jiang, D. Xu, and J. Li. A deep reinforcement learning framework for the financial portfolio management problem. *arXiv preprint arXiv:1706.10059*, 2017.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] M. Hessel, J. Modayil, H. Van Hasselt, et al. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[5] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[7] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2016.

[8] M. Fortunato, M. G. Azar, B. Piot, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[9] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017.

[10] Investopedia. What Is Quantitative Trading? Definition, Examples, and Profit. *Investopedia*. Retrieved from `https://www.investopedia.com/terms/q/quantitative-trading.asp`

[11] Souradeep Chakraborty. Capturing Financial Markets to Apply Deep Reinforcement Learning. *arXiv preprint arXiv:1907.04373*, 2019.

[12] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[13] Y. Zhang, D. Zhang, X. Zhou, Y. Li, and C. Zhou. R-DDQN: Optimizing algorithmic trading strategies using a reward network in a double DQN. *Mathematics*, 12(11):1621, 2024.

[14] Z. Liu, Z. Shi, H. Li, and W. Zhang. Gym-anytrading: Benchmarking reinforcement learning for financial trading environments. *arXiv preprint arXiv:2407.17032*, 2024.

[15] Anonymous. Reinforcement learning for financial portfolio management. *arXiv preprint arXiv:2411.07585*, 2024.

[16] Anonymous. Enhancing DQN-based trading agents with adaptive risk modeling. In *Proceedings of the ACM International Conference*, 2023.

Appendix

A Mathematical Formulas and Component Intuition

This appendix provides detailed explanations of the mathematical formulas associated with each component of the Rainbow DQN architecture. Each formula is annotated with variable definitions and additional context for interpretation.

Double Q-learning Target (DDQN)

$$r + \gamma Q_{\theta'} \left(s', \arg\max_{a'} Q_\theta(s', a') \right)$$

- r: immediate reward from the environment

- γ: discount factor (controls importance of future rewards)

- Q_θ: current network with parameters θ

- $Q_{\theta'}$: target network with frozen parameters θ'

- s': next state

- a': candidate action for next state

This formula decouples action selection and evaluation to reduce overestimation bias present in standard DQN.

Prioritized Experience Replay Probability (PER)

$$p_t \propto \left| r + \gamma \max_{a'} Q_{\theta'}(s', a') - Q_\theta(s, a) \right|^\omega$$

- p_t: probability of sampling transition t

- ω: prioritization exponent, usually in [0.4, 0.6]

The TD error magnitude prioritizes transitions more likely to improve learning.

Dueling Network Aggregation

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a')$$

- $V(s)$: estimated value of being in state s

- $A(s, a)$: advantage of action a in state s

- $\mathcal{A}$: action space

Subtracting the average advantage ensures identifiability and prevents the agent from learning redundant V and A values.

Noisy Network Weight Perturbation

$$W' = W_\mu + W_\sigma \odot \varepsilon_w, \quad b' = b_\mu + b_\sigma \odot \varepsilon_b$$

- W_μ, b_μ: learnable means

- W_σ, b_σ: learnable standard deviations

- $\varepsilon_w, \varepsilon_b$: random noise samples from factorized Gaussian

The element-wise product $\odot$ injects trainable noise into the weights, replacing static exploration like ϵ-greedy.

Multi-step Return (n-step learning)

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1}$$

- $R_t^{(n)}$: cumulative reward over n steps

- r_{t+k+1}: reward at time $t + k + 1$

By looking ahead n steps instead of one, the agent propagates rewards more efficiently, especially in sparse-reward settings.

Distributional Q-learning (C51)

$$Q(s, a) = \sum_{i=1}^{N_z} z_i \cdot p_i(s, a)$$

- N_z: number of atoms (e.g., 51)

- z_i: fixed support values (e.g., linearly spaced in $[v_{\min}, v_{\max}]$)

- $p_i(s, a)$: predicted probability of z_i under state-action pair (s, a)

This formulation models a full return distribution instead of a single expected value, capturing the uncertainty in outcomes.

B Pseudo-Code Implementation

B.1 Base DQN Neural Network

```
def BasicNet(x):
    h = ReLU(Linear1(x))          # hidden_dim
    h = ReLU(Linear2(h))          # hidden_dim
    q = Linear3(h)                # output_dim
    return q                      # Q(s, a)
```

Intuition. The *Base DQN* head is a plain multi-layer perceptron that maps the flattened state vector $x \in \mathbb{R}^{d_{\text{in}}}$ through two ReLU-activated hidden layers to a vector of action values. Each output entry corresponds to the predicted return of taking action a in state s.

$$\begin{aligned} h_1 &= \mathrm{ReLU}\big(W_1 x + b_1\big), \\ h_2 &= \mathrm{ReLU}\big(W_2 h_1 + b_2\big), \\ Q(s, a) &= \big(W_3 h_2 + b_3\big)_a. \end{aligned}$$

Here W_k and b_k are the learned weight matrices and bias vectors for layer k. This simple architecture serves as our control baseline for evaluating the added capacity and exploration benefits of the Noisy, Dueling, and Distributional variants.

B.2 Noisy Net Neural Network

```
def NoisyNet(x, s0=0.5):
    h = ReLU(NoisyLinear(in_dim, hid_dim, s0)(x))
    h = ReLU(NoisyLinear(hid_dim, hid_dim, s0)(h))
    q =       NoisyLinear(hid_dim, out_dim, s0)(h)
    return q

class NoisyLinear(in_f, out_f, s0=0.5):
    # trainable means (mu) and std devs (sigma)
    w_mu,  w_sigma  = Param(out_f, in_f), Param(out_f, in_f)
    b_mu,  b_sigma  = Param(out_f),       Param(out_f)
    def reset_parameters():
        k = 1 / sqrt(in_f)
        w_mu.uniform_(-k, k);  b_mu.uniform_(-k, k)
        w_sigma.fill_(s0 * k); b_sigma.fill_(s0 * k)

    def forward(x):
        eps_in   = _f(in_f);                 # noise  eps_in
        eps_out  = _f(out_f);                # noise  eps_out
        eps_w    = outer(eps_out, eps_in)   # weight noise
        eps_b    = eps_out                   # bias noise
        w = w_mu + w_sigma * eps_w
        b = b_mu + b_sigma * eps_b
        return linear(x, w, b)

    @staticmethod
    def _f(size):
        z = randn(size)                      # N(0,1)
        return sign(z) * sqrt(abs(z))        # transform
```

Intuition. Each forward pass draws a fresh, factorised Gaussian noise sample, perturbing the weights and biases. Because the noise scale is trainable, the network can reduce exploration in well-understood states while retaining stochasticity where it is helpful.

Code written from equations:

$$W' = W_\mu + W_\sigma \odot \varepsilon_w, \qquad b' = b_\mu + b_\sigma \odot \varepsilon_b.$$

B.3 Duel Neural Network

```
def DuelingNet(x):
    h = ReLU(Linear1(x))
    h = ReLU(Linear2(h))
    V = Linear_value(h)           # state value
    A = Linear_adv(h)             # advantage
    q = V + (A - A.mean(dim=-1, keepdim=True))
    return q
```

Intuition. We separate a state's overall value $V(s)$ from the action-specific advantage $A(s,a)$; this lets the network learn which states are good even when the best action is uncertain. The mean-subtraction term keeps the resulting Q-values properly centred and makes the decomposition unique.

Code written from equation:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a') \ .$$

B.4 Distributional Neural Network

```
def DistNet(x, support, atom_size=51):
    h = ReLU(Linear1(x))
    h = ReLU(Linear2(h))
    logits = Linear3(h)                      # (B, A*atoms)
    logits = logits.view(-1, A, atom_size)
    prob   = softmax(logits, dim=-1).clamp(min=1e-3)
    q = (prob * support).sum(dim=-1)         # expectation
    return q, prob
```

Intuition. Rather than predict a single expected return, the agent outputs a full probability mass over N_z fixed "atoms," capturing uncertainty about future rewards. The scalar Q-value is simply the expectation of that distribution.Code written from equation:

$$Q(s,a) = \sum_{i=1}^{N_z} z_i \, p_i(s,a), \qquad N_z = 51, \ z_i \in [v_{\min}, v_{\max}].$$

C Hyper-parameters

Table 2: Rainbow hyper-parameters used in our experiments.

Parameter	Value
Train Window size () Window size (price history)	200 steps
Steps per episode N_{steps}	700
Replay buffer size $\lvert\mathcal{D}\rvert$	80K transitions
Batch size	256
Target network period	32K steps
Adam learning rate α	5×10^{-4}
Adam ε	1.5×10^{-4}
Exploration $\varepsilon_{\min}$	0.10
ε decay schedule	343K steps (70% of training)
ε decay rate	Linear
Noisy Nets σ_0	0.5
Prioritization exponent ω	0.6
Importance-sampling β	$0.4 \to 1.0$
Multi-step returns n	3
TD-error clip ε_{TD}	10^{-6}
Distributional atoms (N_z)	51
Distributional range $[v_{\min}, v_{\max}]$	$[-100,\ 100]$
Discount factor γ	0.99

D Results learning curves (Agents comparison and Ablation)

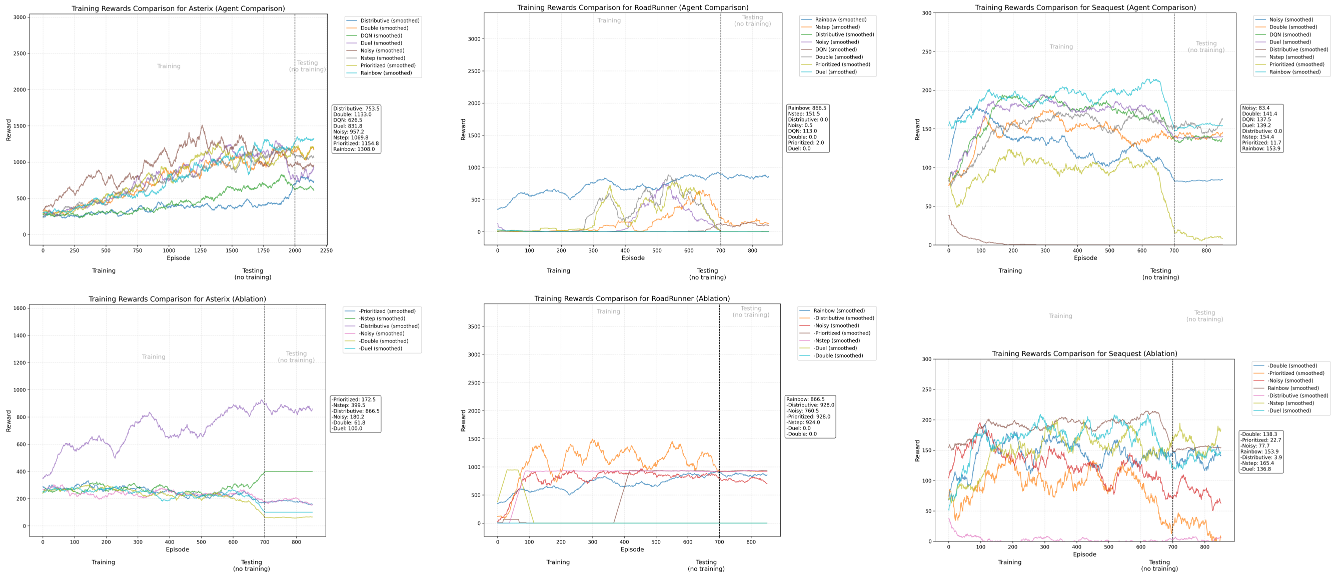

Figure 4: Agent comparison and ablation study results across three Atari environments averaged over 3 seeds and smoothed with a 50-episode moving average. Top Row: Agent Comparison, Bottom Row: Ablation Studies. Left to Right : Asterix, Roadrunner, Seaquest

E Compute Resources

Each training run (1.2M environment steps) took approximately 1.5 hours of wall-clock time. In total, we launched 57 experiments, resulting in roughly 85.5 GPU/CPU-hours.

Table 3: Compute resources used across local machines and McGill SLURM cluster.

Location	**Member**	**CPU / System**	**GPU**
Local	William	Apple M2 (10 cores)	–
Local	Max	Intel i9-12900K	RTX 3070 (8 GB)
Local	Denis	Apple M1 Pro (8 cores)	–
Home	Denis	Intel i5	RTX 3060 (12 GB)
SLURM Cluster	–	Various nodes on `mimi.cs.mcgill.ca`	A5000 / A2000 GPUs

We used McGill's SLURM infrastructure to run parallel jobs using `sbatch` scripts that passed different flag combinations to `main.py`, automating the full experimental sweep across all environments and configurations.