Reinforcement Learning Assignement : Rainbow

Alinejad Kevin, Leo Bernouin

February 27, 2025

Contents

1 Introduction

Recent breakthroughs in deep Reinforcement Learning (RL) have demonstrated the ability of agents to learn complex behaviors directly from high-dimensional inputs such as raw pixels. A key milestone was the Deep Q-Network (DQN) algorithm [1], which combined convolutional neural networks with Q-learning and experience replay to achieve human-level performance on several Atari 2600 games.

Subsequent refinements have addressed various aspects of DQN. Double DQN [2] reduces overestimation bias by disentangling action selection from action evaluation. Prioritized experience replay [3] improves data efficiency by replaying transitions more often when they are more informative. The dueling architecture [4] separates the representation of state-values from action-advantages, thereby focusing the network on the most relevant states. Multi-step methods [5] speed up credit assignment by leveraging returns from multiple future steps. Distributional approaches [6] model the full return distribution rather than its mean, and noisy networks [7] add adaptive parameter noise for more efficient exploration.

Because many of these improvements tackle distinct limitations of DQN, they can often be combined. In this spirit, the Rainbow agent [8] integrates all of the above techniques, achieving state-of-the-art results on the Atari 2600 benchmark. Before examining how Rainbow merges these ideas, we first present general RL background needed to understand its foundations.

2 General Background and DQN Explanations

2.1 General Background

Reinforcement Learning studies how an agent can learn to act within an environment in order to maximize a numerical reward signal. Formally, we model the environment as a Markov Decision Process (MDP), defined by the tuple

$$\mathcal{M} = (S, A, T, r, \gamma),$$

where

- S is the set of possible states,
- A is the set of possible actions,
- $T(s, a, s') = P(S_{t+1} = s' \mid S_t = s,\, A_t = a)$ is the transition function,
- $r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s,\, A_t = a]$ is the reward function,
- $\gamma \in [0, 1]$ is the discount factor.

At each time step t, the agent observes a state $S_t \in S$, selects an action $A_t \in A$, and then receives a reward R_{t+1} and the next state S_{t+1}. The agent's goal is to choose actions so as to maximize the expected discounted return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1}.$$

A *policy* π is a mapping from states to a probability distribution over actions, $\pi(a \mid s)$. Value-based methods seek to learn the *action-value function* $Q^\pi(s, a)$, which is defined as the expected discounted return when taking action a in state s and thereafter following policy π:

$$Q^\pi(s, a) \;=\; \mathbb{E}_\pi\Big[G_t \;\Big|\; S_t = s,\, A_t = a\Big].$$

If the agent follows a *greedy* policy with respect to Q^π, then action selection at state s is:

$$A_t \;=\; \arg\max_a \, Q^\pi(s, a),$$

although in practice an ε-greedy strategy is often used for exploration, where with probability ε the agent chooses a random action.

2.2 Deep Q-Networks (DQN)

Deep Q-Learning extends classical Q-Learning to large state and action spaces by approximating the action-value function with a deep neural network. As shown in Figure 1, a convolutional neural network can be used to map raw pixel inputs (or other high-dimensional data) directly to estimates of $Q_\theta(s, a)$, where θ are the network parameters. The key insight is that each state S_t is fed into a neural network that outputs an approximate value for every possible action a, thus replacing the traditional Q-table which becomes infeasible for very large or continuous state spaces.

The optimization of this network is performed by minimizing a loss function derived from the Q-Learning target. Let (s_i, a_i, r_i, s'_i) be a transition sampled from a replay buffer of past experiences. Define the one-step target

$$y_i \;=\; r_i \;+\; \gamma \max_{a'} Q_{\theta^-}(s'_i, a'),$$

where θ^- are the parameters of a separate *target network*, periodically copied from the main (online) network θ. This target network remains fixed for a certain number of iterations to stabilize training, so that the target values do not move too quickly. maximisation step :

$$L(\theta) \;=\; \Big(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\theta'}(S_{t+1}, a') - q_\theta(S_t, A_t)\Big)^2 \tag{1}$$

where t is a time step randomly picked from the replay memory. The loss function for a mini-batch of N such transitions is:

$$L(\theta) \;=\; \frac{1}{N} \sum_{i=1}^{N} \big(y_i - Q_\theta(s_i, a_i)\big)^2.$$

Gradients of this loss with respect to θ are computed via backpropagation. By training on batches of transitions sampled uniformly at random from the replay buffer, the network avoids correlated updates that can otherwise destabilize learning. Experience replay thus improves data efficiency and helps smooth out training.

Algorithm 1 describes the basic DQN procedure [1]. States, actions, and rewards are gathered by interacting with the environment using an ε-greedy policy derived from $Q_\theta(s, a)$. These transitions are stored in the replay buffer, and random mini-batches from this buffer are used to update θ by stochastic gradient descent. After a fixed number of steps, the target network parameters θ^- are set to θ. This combination of a replay buffer and slowly updated target network has shown great stability and success across various Atari 2600 games.

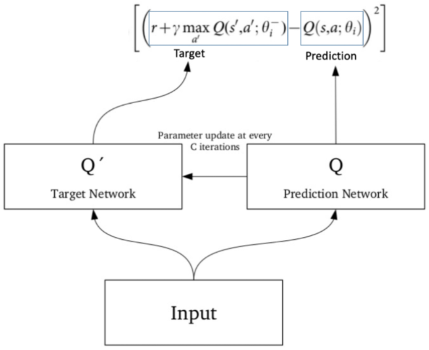
$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q´
Target Network
Q
Prediction Network
Input

Figure 1: Illustration of Deep Q-Learning using a replay buffer and a target network.

Algorithm 1 Deep Q-Network (DQN) [1]

```
Initialize the online network $Q_\theta(s,a)$ with random weights $\theta$.
Initialize the target network $Q_{\theta^-}(s,a)$ with $\theta^- \leftarrow \theta$.
Initialize the replay buffer $\mathcal{D}$.
for each episode do
  Observe and set initial state $s$.
  while not terminal do
    Select an action $a$ using the $\varepsilon$-greedy policy derived from $Q_\theta$.
    Execute $a$ in the environment, observe $r$ and next state $s'$.
    Store $(s,a,r,s')$ in $\mathcal{D}$.
    Sample a random mini-batch of transitions $\{(s_i,a_i,r_i,s'_i)\}$ from $\mathcal{D}$.
    Compute $y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s'_i,a')$ for each transition in the mini-batch.
    Update $\theta$ by minimizing $L(\theta) = \frac{1}{N}\sum_i \big(y_i - Q_\theta(s_i,a_i)\big)^2$.
    Periodically update $\theta^- \leftarrow \theta$.
    Set $s \leftarrow s'$.
  end while
end for
```

3 Extensions to DQN

In this section, we review several refinements that build upon the DQN framework to address its known limitations. Each method focuses on a distinct aspect of learning, and most can be combined to yield further improvements. We begin by listing six common extensions, but postpone detailed explanations of the first three. We then discuss the multi-step learning variant in more detail, leaving the remaining two extensions to follow.

3.1 Double Q-learning

Standard Q-learning suffers from *overestimation bias*, which arises due to the maximization step in equation(1). Specifically, since Q-values are estimated from sampled data, taking the maximum over estimated values introduces a positive bias when estimates are noisy or overoptimistic. This issue can degrade performance, particularly in stochastic environments.

Double Q-learning addresses this problem by *decoupling* the action selection and action evaluation steps. Instead of using the same Q-function for both action selection and target evaluation, Double Q-learning splits the estimation into two functions, thereby reducing the overestimation effect. This idea can be seamlessly incorporated into Deep Q-Networks (DQN) [2], leading to the *Double DQN* (DDQN) variant.

Double Q-learning Update Rule. Instead of the standard Q-learning target:

$$y_t = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'; \theta^-),$$

Double Q-learning introduces a *two-network* mechanism:

- One network (θ) is used for action *selection*, choosing $a^* = \arg\max_{a'} Q(S_{t+1}, a'; \theta)$.
- The other network (θ^-) is used for action *evaluation*, computing the value of the chosen action $Q(S_{t+1}, a^*; \theta^-)$.

This results in the *Double Q-learning* target:

$$y_t^{\text{DDQN}} = R_{t+1} + \gamma Q(S_{t+1}, a^*; \theta^-),$$

which mitigates the overestimation bias by ensuring that the same values are not used for both action selection and evaluation. The loss function in DDQN is thus:

$$L(\theta) = \mathbb{E}_{(S_t, A_t, R_{t+1}, S_{t+1}) \sim \mathcal{D}} \left[\left(y_t^{\text{DDQN}} - Q(S_t, A_t; \theta) \right)^2 \right].$$

Empirical Benefits. By reducing the overestimation bias, Double Q-learning improves the stability and accuracy of value estimates in reinforcement learning. This modification was found to significantly enhance performance in deep reinforcement learning, particularly in environments where overestimations lead to suboptimal policies. As demonstrated in [2], Double DQN reduces harmful Q-value overestimations in Deep Q-learning, leading to better convergence and more reliable decision-making.

Algorithm 2 Double DQN [2]

```
Initialize the online network $Q_\theta(s, a)$ with random weights $\theta$.
Initialize the target network $Q_{\theta^-}(s, a)$ with $\theta^- \leftarrow \theta$.
Initialize the replay buffer $\mathcal{D}$.
for each episode do
    Observe and set initial state $s$.
    while not terminal do
        Select an action $a$ using the $\varepsilon$-greedy policy derived from $Q_\theta$.
        Execute $a$ in the environment, observe reward $r$ and next state $s'$.
        Store $(s, a, r, s')$ in $\mathcal{D}$.
        If training is ready (e.g., $|\mathcal{D}| \geq$ batch_size):
        Sample a random mini-batch $\{(s_i, a_i, r_i, s'_i)\}$ from $\mathcal{D}$.
        For each transition, compute
        $a_i^* = \arg\max_{a'} Q_\theta(s'_i, a')$, $y_i = r_i + \gamma Q_{\theta^-}\big(s'_i, a_i^*\big)$.
        Update $\theta$ by minimizing $L(\theta) = \frac{1}{N} \sum_i \big(y_i - Q_\theta(s_i, a_i)\big)^2$.
        Periodically update $\theta^- \leftarrow \theta$.
        Set $s \leftarrow s'$.
    end while
end for
```

3.2 Prioritized Experience Replay

Using a replay memory leads to design choices at two levels: which experiences to store, and which experiences to replay (and how to do so). This paper addresses only the latter: making the most effective use of the replay memory for learning, assuming that its contents are outside of our control.

The central component of prioritized replay is the criterion by which the importance of each transition is measured. A reasonable approach is to use the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is. This algorithm stores the last encountered TD error along with each transition in the replay memory. The transition with the largest absolute TD error is replayed from the memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. One thing to note that new transitions arrive without a known TD-error, so it puts them at maximal priority in order to guarantee that all experience is seen at least once. (see store method)

We might use 2 ideas to deal with TD-error: 1. greedy TD-error prioritization, 2. stochastic prioritization. However, greedy TD-error prioritization has a severe drawback. Greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation, meaning that the initially high error transitions get replayed frequently. This lack of diversity that makes the system prone to over-fitting. To overcome this issue, we will use a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. In practice, we use additional term ϵ in order to guarantee all transactions can be possibly sampled: $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant.

One more. Let's recall one of the main ideas of DQN. To remove correlation of observations, it uses uniformly random sampling from the replay buffer. Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion corresponding to TD-error. We can correct this bias by using importance-sampling (IS) weights

$$w_i = \left(\frac{1}{N} \frac{1}{P(i)} \right)^{\beta}$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i \delta_i$ instead of δ_i. In typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training. We therefore exploit the flexibility of annealing the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning.

Algorithm 3 Deep Q-Network (DQN) with Prioritized Experience Replay [3]

```
Initialize online network $Q_\theta(s,a)$ with random weights $\theta$
Initialize target network $Q_{\theta^-}(s,a)$ with $\theta^- \leftarrow \theta$
Initialize prioritized replay buffer $\mathcal{D}$ (each new transition has priority $p_i = p_{\text{init}} > 0$)
for each episode do
  Observe initial state $s$
  while not terminal do
    Select action $a$ using the $\varepsilon$-greedy policy w.r.t. $Q_\theta(s,\cdot)$
    Execute $a$ in environment, observe reward $r$ and next state $s'$
    Compute temporary TD error (to set priority): $\delta = \left| r + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a) \right|$
    Compute priority: $p = (\delta + \epsilon)^\alpha$
    Store transition $(s,a,r,s')$ in $\mathcal{D}$ with priority $p$
    Set $s \leftarrow s'$
    if $|\mathcal{D}| \geq$ batch_size then
      Sample mini-batch of $N$ transitions $\{(s_i,a_i,r_i,s'_i)\}$ from $\mathcal{D}$ using probabilities $P_i = \frac{p_i}{\sum_k p_k}$
      Compute importance weights for each sample: $w_i = \left(\frac{1}{N}\frac{1}{P_i}\right)^\beta$
      Compute updated targets: $y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s'_i, a')$
      Compute new TD error $\delta_i$ for each sample: $\delta_i = y_i - Q_\theta(s_i,a_i)$
      Update each sample's priority: $p_i = (|\delta_i| + \epsilon)^\alpha$
      Perform weighted gradient descent step on $L(\theta) = \frac{1}{N}\sum_{i=1}^{N} w_i \left(\delta_i\right)^2$
      Periodically update $\theta^- \leftarrow \theta$
    end if
  end while
end for
```

3.3 Dueling Networks

The *Dueling Network Architecture* [4] is a neural network design tailored for value-based reinforcement learning. Unlike standard Deep Q-Networks (DQN), which estimate a single Q-value function for each action, the dueling network architecture separates the estimation into two distinct streams:

- A **value stream** $v_\eta(f_\xi(s))$ that estimates the value of being in a given state s, independent of the action taken.
- An **advantage stream** $a_\psi(f_\xi(s), a)$ that estimates the relative advantage of each action a compared to others in the same state.

Both streams share a common feature extraction module $f_\xi(s)$, typically a convolutional encoder when used with visual inputs.

Factorization of Action Values. The core idea behind the dueling network is to separate state-value estimation from action advantage estimation and merge them using a specialized aggregation function. The Q-value function is decomposed as follows:

$$q_\theta(s, a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}}.$$

Here:

- ξ represents the parameters of the shared feature encoder $f_\xi(s)$.
- η corresponds to the parameters of the value stream v_η.
- ψ refers to the parameters of the advantage stream a_ψ.

The term $\frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}}$ ensures that the Q-values remain identifiable by centering the advantage function.

Advantages of the Dueling Architecture. By separating value and advantage functions, the dueling network structure allows for more efficient learning in environments where the action choice does not always impact the overall value significantly. This is particularly beneficial in:

- Large state spaces where many actions have similar values.
- Scenarios where distinguishing the best action is difficult due to high variance in action rewards.

Empirical results [4] demonstrate that dueling architectures improve sample efficiency and robustness compared to traditional DQN.

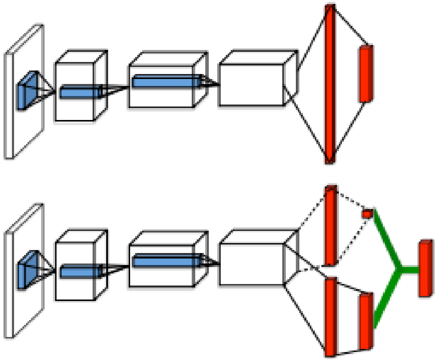

Figure 2: *Figure 1.* Taken from [4], A popular single stream Q-network **(top)** and the dueling Q-network **(bottom)**. The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module combine them. Both networks output Q-values for each action.

Algorithm 4 Dueling Deep Q-Network (Dueling DQN)

```
**Initialize:** Online network $Q_\theta(s,a)$ and target network $Q_{\theta^-}(s,a)$ with $\theta^- \leftarrow \theta$.
**Initialize:** Replay buffer $\mathcal{D}$.
**for** each episode **do**
  Observe initial state $s_0$.
  **while** not terminal **do**
    Select action $a_t$ using $\varepsilon$-greedy w.r.t. $Q_\theta(s_t,a)$.
    Execute $a_t$, observe reward $r_{t+1}$ and next state $s_{t+1}$.
    Store transition $(s_t,a_t,r_{t+1},s_{t+1})$ in $\mathcal{D}$.
    Sample a mini-batch of transitions $(s_i,a_i,r_i,s'_i)$ from $\mathcal{D}$.
    Compute target:
        $$y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s'_i,a').$$
    Compute dueling Q-values using:
        $$q_\theta(s,a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s),a) - \frac{\sum_{a'} a_\psi(f_\xi(s),a')}{N_{\text{actions}}}.$$
    Update network parameters using gradient descent on:
        $$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q_\theta(s_i,a_i))^2.$$
    Periodically update target network: $\theta^- \leftarrow \theta$.
  **end while**
**end for**
```

3.4 Multi-step Learning

The original one-step Q-learning target relies on the immediate reward plus a bootstrap from the subsequent state. In many tasks, longer-term credit assignment can improve efficiency. Instead of performing an update based on a single reward and then bootstrapping from $t + 1$, one can accumulate rewards over multiple steps before bootstrapping. This approach, often referred to as an n-step return, reduces bias in settings where returns are sparse or where more immediate feedback over a few steps leads to faster credit assignment.

An n-step return from a given state S_t is defined by

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

where R_{t+k+1} is the reward received $k + 1$ steps after time t. After these n steps, the target then bootstraps from $Q_{\theta^-}(S_{t+n}, a')$. The resulting update for a multi-step variant of DQN replaces the one-step target with

$$R_t^{(n)} + \gamma^n \max_{a'} Q_{\theta^-}(S_{t+n}, a').$$

Training proceeds in a manner similar to the one-step DQN algorithm, but each sample from the replay buffer must include enough consecutive transitions to construct n-step returns. Although incorporating multiple transitions in a single target can potentially introduce off-policy bias when the data were collected under a behavior policy different from the current one, in practice it has been observed that ignoring off-policy correction does not necessarily hurt performance. According to recent work, "it is possible to ignore off-policy correction without seeing an adverse effect in the overall performance of Sarsa and Q(σ). This finding is problem specific, but it suggests that off-policy correction is not always necessary for learning from samples from the experience replay buffer" [9].

Empirically, choosing an appropriate n can substantially reduce training time in problems where immediate rewards are not sufficient to guide learning. The multi-step mechanism partly compensates for the delayed nature of rewards by propagating newly obtained information more quickly, despite the potential for bias introduced by off-policy data.

Algorithm 5 DQN with n-step returns

```
Initialize online network $Q_\theta(s, a)$ with random weights $\theta$.
Initialize target network $Q_{\theta^-}(s, a)$ with $\theta^- \leftarrow \theta$.
Initialize replay buffer $\mathcal{D}$.
for each episode do
  Observe and set initial state $s_0$.
  while not terminal do
    Select an action $a_t$ using $\varepsilon$-greedy w.r.t. $Q_\theta(s_t, a)$.
    Execute $a_t$ in the environment, observe reward $r_{t+1}$ and next state $s_{t+1}$.
    Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$.
    Sample a random mini-batch of $N$ transitions $\{(s_i, a_i, r_i^{(1..n)}, s_{i+n})\}$ from $\mathcal{D}$, each containing
      up to $n$ steps of experience.
    For each sampled transition, compute the $n$-step target: $y_i = \left(\sum_{k=0}^{n-1} \gamma^k r_{i+k+1}\right) +$
      $\gamma^n \max_{a'} Q_{\theta^-}(s_{i+n}, a')$.
    Perform a gradient descent step on $L(\theta) = \frac{1}{N} \sum_i \big(y_i - Q_\theta(s_i, a_i)\big)^2$.
    Every fixed number of steps, update $\theta^- \leftarrow \theta$.
    Set $s_t \leftarrow s_{t+1}$.
  end while
end for
```

3.5 Distributional Reinforcement Learning

Standard Q-learning models the expected return $Q^\pi(s, a) = \mathbb{E}[G_t \mid S_t = s, A_t = a, \pi]$, thus discarding all higher-order information about the variability of returns. By contrast, *distributional RL* learns the entire *return distribution*, often denoted $Z^\pi(s, a)$. This random variable captures the range of possible outcomes (returns) one might observe from (s, a), rather than just their mean. Such a perspective can be useful to model risk or uncertainty, and it can also act as a richer training signal.

In the policy evaluation setting [10], one typically seeks the value function $V^\pi(s)$. The *distributional* analogue is the *value distribution* Z^π, which describes the entire distribution over returns rather than just the mean. We view the immediate reward $R(x, a)$ as a random variable, and define a *transition operator* P^π acting on distributions by

$$P^\pi Z(x, a) \overset{D}{=} Z(X', A'), \quad X' \sim P(\cdot \mid x, a), \quad A' \sim \pi(\cdot \mid X'),$$

where capital letters emphasize the randomness of the next state-action pair (X', A'). The *distributional Bellman operator* $\mathcal{T}^\pi$ then maps a distribution Z to

$$\mathcal{T}^\pi Z(x, a) \overset{D}{=} R(x, a) + \gamma P^\pi Z(x, a).$$

Hence, $\mathcal{T}^\pi Z(x, a)$ is the random return obtained by adding the (random) reward $R(x, a)$ and the discounted distribution of returns under π. In distributional RL, we aim to find Z^π by iterating $Z \leftarrow \mathcal{T}^\pi Z$ until convergence, capturing not just the expected return but its entire probability distribution.

Discrete Support (C51). One practical approach (called C51 [6]) discretizes the range of possible returns into N_{atoms} points, or *atoms*, usually spaced linearly between $v_{\min}$ and $v_{\max}$:

$$z_i \;=\; v_{\min} \;+\; (i-1)\,\frac{v_{\max} \;-\; v_{\min}}{N_{\text{atoms}} \;-\; 1}, \quad i = 1, \ldots, N_{\text{atoms}}.$$

Instead of outputting just one $Q_\theta(s,a)$ per action, the network now outputs a vector of probability masses

$$\big[p_\theta^1(s,a),\; p_\theta^2(s,a),\; \ldots,\; p_\theta^{N_{\text{atoms}}}(s,a)\big]$$

over these atoms, forming a parametric model of the return distribution. Specifically,

$$Z_\theta(s,a) \;=\; z_i \quad \text{with probability} \quad p_\theta^i(s,a) \;=\; \frac{\exp\!\big(\theta_i(s,a)\big)}{\sum_{j=1}^{N_{\text{atoms}}} \exp\!\big(\theta_j(s,a)\big)},$$

where $\theta_i(s,a)$ are the logits produced by the neural network (one for each atom). This yields a categorical distribution whose expectation approximates the classical Q-value, but which also preserves higher-order moments of the return.During training, for a sampled transition (s,a,r,s'), we:

1. Pick the next-action $a^* = \arg\max_{a'} \sum_i z_i\, p_\theta^i(s,a')$ (e.g. by maximizing the *mean* of the distribution for each action in state s').
2. Shift and discount the distribution from (s', a^*) by r and γ.
3. *Project* that shifted distribution back onto the fixed support $\{z_i\}$ via a piecewise linear operator (see Eq. (projection)).
4. Compute a cross-entropy (or KL divergence) loss between the current prediction $p_\theta(s,a)$ and this projected target distribution.

Projected Bellman Update. As shown in Figure 3 (adapted from [6]), the Bellman update for a single transition produces a *shifted* distribution $\hat{T} Z_\theta(s,a)$ centered at $r + \gamma\, z_j$. This shifted distribution may lie outside the original range $[v_{\min}, v_{\max}]$. Hence, C51 *clips* and projects it onto the $\{z_i\}$ support:

$$\big(\Phi_z\, \hat{T} Z_\theta\big)(s,a) \;=\; \sum_{j=0}^{N_{\text{atoms}}-1} \Big(1 \;-\; \big|\tfrac{\hat{T} z_j - z_i}{\Delta z}\big|\Big)_0^1\, p_\theta^j(s', a^*), \qquad \text{(projection)}$$

where $\Delta z = \frac{v_{\max} - v_{\min}}{N_{\text{atoms}} - 1}$ and $(\cdot)_0^1$ indicates clamping to the range $[0,1]$. Minimizing the Kullback-Leibler divergence between the resulting distribution and the predicted one $\big\{p_\theta^i(s,a)\big\}$ is essentially a multi-class classification problem.

Algorithm. Algorithm 6 illustrates a variant of DQN using the categorical distribution approach. The network architecture is similar to standard DQN, but the final layer outputs $N_{\text{atoms}} \times |\mathcal{A}|$ logits (one distribution per action). A softmax is applied to each action's logits to ensure valid probability distributions.

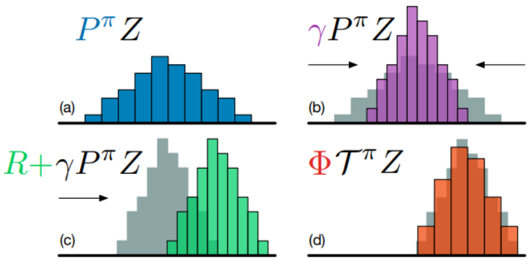

Figure 3: Distributional Bellman update idea: the return distribution for (s', a^*) is shifted by the immediate reward r, scaled by γ, and then projected onto the discrete support $\{z_i\}$.

Algorithm 6 Categorical DQN (C51) with Target Network

```
Input: $v_{\min}, v_{\max}$ (value bounds), $N_{\text{atoms}}$, discount $\gamma$, replay buffer $\mathcal{D}$, target update frequency, etc.
Initialize online parameters $\theta$ and target parameters $\theta^-$.
Initialize fixed atoms $\{z_i\}_{i=1}^{N_{\text{atoms}}}$ between $v_{\min}, v_{\max}$.
for each episode do
  Observe and set initial state $s$.
  while not terminal do
    Compute distribution $\{p_\theta^i(s,a)\}_{i=1}^{N_{\text{atoms}}}$ for each action $a$.
    Select $a = \arg\max_{a'} \sum_{i=1}^{N_{\text{atoms}}} z_i \, p_\theta^i(s,a')$.
    Execute $a$ in the environment, observe reward $r$ and next state $s'$.
    Store $(s,a,r,s')$ in $\mathcal{D}$.
    Set $s \leftarrow s'$.
    if $|\mathcal{D}| \geq \texttt{batch\_size}$ then
      Sample a mini-batch of $N$ transitions $\{(s_j, a_j, r_j, s'_j)\}_{j=1}^{N}$ from $\mathcal{D}$.
      for $j = 1$ to $N$ do
        Compute $a_j^* = \arg\max_b \sum_{i=1}^{N_{\text{atoms}}} z_i \, p_\theta^i(s'_j, b)$ using the online network.
        Retrieve $p_{\theta^-}^i(s'_j, a_j^*)$ from the target network for each atom $z_i$.
        Shift each atom by $r_j$ and scale by $\gamma$: $\hat{z}_i = \mathrm{clip}(r_j + \gamma\, z_i,\ v_{\min}, v_{\max})$.
        Distribute $p_{\theta^-}^i(s'_j, a_j^*)$ across $\{\hat{z}_i\}$ onto the fixed support $\{z_1, \ldots, z_{N_{\text{atoms}}}\}$ via the projection $\Phi_z$:
            $$\big(\Phi_z \hat{T} Z_{\theta^-}(s'_j, a_j^*)\big)_k.$$
      end for
      Define the cross-entropy between the projected target distribution and the online prediction:
            $$L(\theta) = \sum_{j=1}^{N} \Big[ - \sum_{k=1}^{N_{\text{atoms}}} \big(\Phi_z \hat{T} Z_{\theta^-}(s'_j, a_j^*)\big)_k \ln p_\theta^k(s_j, a_j) \Big].$$
      Perform a gradient descent step on $L(\theta)$.
      Periodically update $\theta^- \leftarrow \theta$.
    end if
  end while
end for
```

3.6 Noisy Nets

Exploration in reinforcement learning is often handled via simple schemes like ε-greedy, but these do not adapt to state or agent uncertainty in a fine-grained way. *Noisy Networks* [7] address this by injecting learnable, parameter-dependent noise into the agent's function approximator itself, enabling a form of state-conditional exploration. Instead of adding noise purely to actions or to the reward function, the network parameters are randomly perturbed in a way that is differentiable and thus can be learned through gradient descent.

Consider a standard linear layer:

$$y = Wx + b,$$

where $x \in \mathbb{R}^p$ is the input, $W \in \mathbb{R}^{q\times p}$ is the weight matrix, and $b \in \mathbb{R}^q$ is the bias vector. A *noisy linear layer* replaces these deterministic parameters with noisy ones:

$$y \;=\; \underbrace{\left(\mu^W + \sigma^W \odot \epsilon^W\right)}_{\text{noisy weights}} \; x \;+\; \underbrace{\left(\mu^b + \sigma^b \odot \epsilon^b\right)}_{\text{noisy bias}},$$

where $\mu^W, \sigma^W, \mu^b, \sigma^b$ are all learnable parameters, and ϵ^W, ϵ^b are random variables sampled from a fixed distribution (e.g. independent or factorized Gaussian). The symbol $\odot$ denotes element-wise multiplication. At each forward pass, new noise samples (ϵ^W, ϵ^b) are drawn (or periodically updated), causing the network's outputs to vary stochastically with state x. This randomness is not merely a global exploration parameter, but is tuned per-layer and per-weight, offering a more flexible, state-dependent exploration mechanism.

The loss function for training is taken in expectation over the noise distribution, $\mathcal{L}(\zeta) = \mathbb{E}_\epsilon[L(\theta(\epsilon))]$, where ζ denotes all learnable parameters $(\mu^W, \sigma^W, \mu^b, \sigma^b)$ and $\theta(\epsilon)$ denotes the effective parameters after adding noise. In practice, a single noise sample per training update often suffices (Monte Carlo approximation of the expectation). Because the noise parameters and their variances are adjusted by gradient descent, the agent can learn to increase or decrease noise in different parts of the state space over time, thus providing a form of self-annealing exploration.

Figure 4 taken from [7] shows a high-level schematic of a noisy linear layer. By substituting standard linear layers with noisy ones in DQN, the policy can shift from highly stochastic actions at the outset to more deterministic choices as soon as it has higher confidence in its value estimates.

Consequently, the Q-network itself becomes

$$Q_\zeta(x, a, \epsilon) \;=\; Q\big(x, a;\, \mu^W + \sigma^W \odot \epsilon^W,\; \mu^b + \sigma^b \odot \epsilon^b\big)$$

To train a NoisyNet-DQN, one typically replaces the final (or certain hidden) fully connected layers with noisy layers. Instead of using ε-greedy for exploration, the agent samples noise ϵ in the linear layers; the same noise sample is used for all transitions in a mini-batch update, and a separate noise sample ϵ' is used in the target network to reduce noise correlation. The standard DQN loss is then taken in expectation over the noise distribution:

$$\mathcal{L}(\zeta) \;=\; \mathbb{E}_{(x,a,r,y)\sim\mathcal{D}}\, \mathbb{E}_\epsilon\Big[\big(r \;+\; \gamma \max_{b\in A} Q_\zeta(y, b, \epsilon';\, \zeta^-) \;-\; Q_\zeta(x, a, \epsilon;\, \zeta)\big)^2\Big],$$

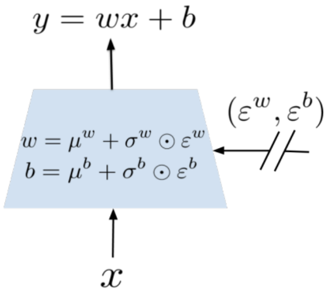
$y = wx + b$
$(\varepsilon^w, \varepsilon^b)$
$w = \mu^w + \sigma^w \odot \varepsilon^w$
$b = \mu^b + \sigma^b \odot \varepsilon^b$
x

Figure 4: Graphical representation of a noisy linear layer. The parameters $\mu^W, \mu^b, \sigma^W, \sigma^b$ are learnable, while ϵ^W, ϵ^b are noise variables (drawn from a chosen distribution). The output is $y = Wx + b$ where $W = \mu^W + \sigma^W \odot \epsilon^W$ and $b = \mu^b + \sigma^b \odot \epsilon^b$.

where (x, a, r, y) is a transition in the replay buffer, y is the next state, and ζ^- are target network parameters (periodically updated from ζ). In practice, we typically sample one realization of ϵ and ϵ' per update (a Monte Carlo approximation to the noise expectation). The key advantage is that the magnitude of noise in the Q-network becomes *learnable*; the network can tune σ^W, σ^b to increase or decrease stochasticity in different parts of the state space.

Algorithm 7 NoisyNet-DQN

```
Initialize online network $Q_\zeta(s, a, \epsilon)$ with random parameters $\zeta$ and random noise parameters
Initialize target network $Q_{\zeta^-}(s, a, \epsilon')$ with $\zeta^- \leftarrow \zeta$
Initialize replay buffer $\mathcal{D}$
for each episode do
  Observe and set initial state $s$
  while not terminal do
    Sample noise $\epsilon$ for the online network
    Select action $a = \arg\max_{a'} Q_\zeta\big(s, a', \epsilon\big)$
    Execute $a$ in the environment, observe reward $r$ and next state $s'$
    Store $(s, a, r, s')$ in $\mathcal{D}$
    Set $s \leftarrow s'$
    if training is ready (e.g. $|\mathcal{D}| \geq$ batch_size) then
      Sample a mini-batch of $N$ transitions $\{(s_i, a_i, r_i, s'_i)\}_{i=1}^N$ from $\mathcal{D}$
      Sample new noise $\epsilon$ for the online network and $\epsilon'$ for the target network
      for $i = 1$ to $N$ do
        Compute $y_i = r_i + \gamma \max_{b \in A} Q_{\zeta^-}\big(s'_i, b, \epsilon'\big)$
      end for
      Perform a gradient update on $L(\zeta) = \frac{1}{N} \sum_{i=1}^N \Big(y_i - Q_\zeta\big(s_i, a_i, \epsilon\big)\Big)^2$
      Periodically update $\zeta^- \leftarrow \zeta$
    end if
  end while
end for
```

4 Putting It All Together: The Rainbow Agent

Rainbow [8] combines all the DQN enhancements we have discussed (multi-step learning, distributional RL, double Q-learning, prioritized replay, dueling networks, and noisy nets) into a single integrated algorithm. Each piece addresses a distinct limitation of vanilla DQN, and in practice they are largely complementary. Below we outline how Rainbow merges these components.

(1) Multi-Step Distributional Loss. Rather than using a 1-step return for the distributional update, Rainbow constructs an n-step target. Concretely, for a given transition starting in state S_t, Rainbow accumulates

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

then bootstraps from the state S_{t+n}. The *distributional Bellman target* becomes

$$d_t^{(n)} \;=\; \bigl(R_t^{(n)} \;+\; \gamma^n \, z,\; p_\theta(S_{t+n}, a_{t+n}^*)\bigr),$$

where z is the vector of fixed *atoms* in C51 (the discrete support), and a_{t+n}^* is the action chosen under Double Q-learning (described next). Rainbow then applies a projection Φ_z to map this shifted distribution back onto the fixed atoms. The final loss is the Kullback–Leibler divergence between the online network's predicted distribution d_t and the n-step target distribution $d_t^{(n)}$:

$$L(\theta) \;=\; D_{\mathrm{KL}}\Bigl(\Phi_z \, d_t^{(n)} \;\|\; d_t\Bigr).$$

Because n-step returns propagate reward signals more quickly than 1-step returns, they can improve learning speed and stability in many environments.

(2) Double Q-Learning. Rainbow uses Double Q-learning to reduce overestimation bias. Specifically, the *online* network (with parameters θ) selects the greedy action a_{t+n}^* in the bootstrap state S_{t+n}:

$$a_{t+n}^* \;=\; \arg\max_{a'} \sum_{i=1}^{N_{\text{atoms}}} z_i \, p_\theta^i(S_{t+n}, a'),$$

while a *target* network (with parameters θ^-) evaluates that action:

$$Q_{\theta^-}\bigl(S_{t+n}, a_{t+n}^*\bigr).$$

For the distributional setting, it means the target network's probability distribution at (S_{t+n}, a_{t+n}^*) is used when constructing the target distribution $d_t^{(n)}$. Periodically, $\theta^- \leftarrow \theta$ to keep the target network slowly updated and stable.

(3) Prioritized Replay with KL-based Priorities. In standard proportional prioritized replay [3], the absolute TD error $\bigl|\delta_i\bigr|$ is used to define transition priorities p_i. Rainbow extends this idea to the distributional context. Instead of the 1-step absolute TD error, the agent often uses the *KL loss* between the predicted distribution d_t and the n-step target $\Phi_z \, d_t^{(n)}$:

$$p_t \;\propto\; \Bigl(D_{\mathrm{KL}}\bigl(\Phi_z \, d_t^{(n)} \,\|\, d_t\bigr)\Bigr)^{\omega},$$

where ω is an exponent controlling how sharply transitions are prioritized. This choice is more directly aligned with what the distributional agent actually minimizes. The replay buffer then samples transitions with probability proportional to p_t, and the agent can use importance sampling weights in the update to correct for this sampling bias.

(4) Dueling Network Architecture with Return Distributions. Rainbow adopts a *dueling* architecture to learn separate representations of state-value and action-advantages. Specifically, the shared feature extractor $f_\theta(s)$ feeds into:

- A *value stream* $v_\theta(f_\theta(s)) \in \mathbb{R}^{N_{\text{atoms}}}$, producing one logit per atom for the "state-value."
- An *advantage stream* $a_\theta(f_\theta(s), a) \in \mathbb{R}^{N_{\text{atoms}} \times |\mathcal{A}|}$, producing one logit per atom, per action.

For each atom z_i, the final logit for (s, a) is formed by combining $v_\theta^i(f_\theta(s))$ and $a_\theta^i(f_\theta(s), a)$, subtracting the mean advantage across all actions to ensure identifiability:

$$\text{logit}_\theta^i(s,a) \;=\; v_\theta^i\big(f_\theta(s)\big) \;+\; a_\theta^i\big(f_\theta(s),a\big) \;-\; \bar{a}_\theta^i(s),$$

$$p_\theta^i(s,a) \;=\; \frac{\exp\!\big(v_\theta^i(f_\theta(s)) + a_\theta^i(f_\theta(s),a) - \bar{a}_\theta^i(s)\big)}{\sum_{j=1}^{N_{\text{atoms}}} \exp\!\big(v_\theta^j(f_\theta(s)) + a_\theta^j(f_\theta(s),a) - \bar{a}_\theta^j(s)\big)},$$

where $\bar{a}_\theta^i(s) = \frac{1}{|\mathcal{A}|} \sum_{a'} a_\theta^i\big(f_\theta(s), a'\big)$ is the mean advantage across all actions at atom i. A softmax is then applied *per action* to turn these logits into a probability distribution $\{p_\theta^i(s,a)\}$ over the atoms z_i.

(5) Noisy Linear Layers. Finally, Rainbow replaces the standard linear layers in the value and advantage streams with *noisy* layers (Noisy Nets) [7], where each weight and bias has a deterministic component plus a learnable noise term. This provides state-dependent, parameter-based exploration that can be more efficient than a simple ε-greedy schedule. In Rainbow, a factorized Gaussian noise scheme is typically used to reduce computational overhead.

Overall Rainbow Update Procedure. Putting everything together, Rainbow proceeds much like DQN, but with several modifications:

1. **Interaction and Storage:** Interact with the environment under the noisy policy, store n-step transitions in a *prioritized* replay buffer (where priorities are updated using the distributional KL loss).
2. **Sample and Target Construction:** Sample a mini-batch of transitions from the buffer. For each transition, compute the *n-step* discounted reward $R_t^{(n)}$, find the *Double-Q* action with the online network, and retrieve the distribution from the target network. Shift and discount that distribution by $R_t^{(n)}$, then *project* it onto the discrete atom support.
3. **Loss Computation:** Compute the KL divergence (or cross-entropy) between the online network's predicted distribution and the projected target distribution. This *KL* also serves as a basis for updating the transition priorities.

4. **Network Update:** Update θ via stochastic gradient descent. Periodically synchronize $\theta^{-} \leftarrow \theta$.

With these enhancements, Rainbow was shown to achieve state-of-the-art performance on Atari 2600 games, illustrating how multi-step returns, distributional modeling, double Q-learning, dueling networks, noisy exploration, and prioritized replay can be synergistically combined into one cohesive agent.

Algorithm 8 Rainbow: Integrating All Extensions into DQN

```
Initialize online network $Q_\theta(s,a)$ (noisy dueling, C51) with random parameters $\theta$
Initialize target network $Q_{\theta^-}(s,a)$ with $\theta^- \leftarrow \theta$
Initialize prioritized replay buffer $\mathcal{D}$ (capable of $n$-step storage)
Initialize the set of atoms $\{z_k\}_{k=1}^{N_{\text{atoms}}}$ spanning $[v_{\min}, v_{\max}]$
for each episode do
  Observe initial state $s_0$
  while not terminal do
    # Action Selection
    Sample noise in the noisy layers of the online network
    $a_t \leftarrow \arg\max_a Q_\theta(s_t, a)$ # (Dueling net internally computes $\text{logit}_\theta$ and does softmax along atoms)
    Execute $a_t$ in the environment; observe reward $r_{t+1}$, next state $s_{t+1}$
    # Multi-step storage
    Accumulate up to $n$ steps of transitions to compute $n$-step return $R_t^{(n)}$
    Store $(s_t,\ a_t,\ R_t^{(n)},\ s_{t+n},\ \text{done})$ in $\mathcal{D}$
    $s_t \leftarrow s_{t+1}$
    if $|\mathcal{D}| \geq$ batch_size then
      Sample mini-batch of $N$ transitions $\big\{(s_i, a_i, R_i^{(n)}, s_i')\big\}_{i=1}^N$ from $\mathcal{D}$ with priorities $p_i$
      Compute importance weights $w_i$ for each sampled transition
      for $i = 1$ to $N$ do
        # the distribution $p_\theta(s_i, a_i)$ are already computed via Dueling-Net + C51:
        # Double Q-Learning step
        $a_i^* \leftarrow \arg\max_a \sum_{k=1}^{N_{\text{atoms}}} z_k\, p_\theta^k(s_i', a)$ # online net for action selection
        Retrieve target distribution $p_{\theta^-}(s_i', a_i^*)$ # target net for action evaluation
        # Distributional shift
        Shift and clamp each atom: $\widehat{z}_k \leftarrow R_i^{(n)} + \gamma^n\, z_k \in [v_{\min}, v_{\max}]$
        Project $\big\{\widehat{z}_k,\ p_{\theta^-}^k(s_i', a_i^*)\big\}$ onto the fixed support $\{z_k\}$ to form the target distribution $\Phi_z \hat{T} Z_{\theta^-}(s_i', a_i^*)$
      end for
      # Loss and update
      Let $p_\theta(s_i, a_i)$ be the predicted distribution by online net
      Compute cross-entropy or KL loss per sample, then weight by $w_i$
      $L(\theta) \leftarrow \frac{1}{N}\sum_{i=1}^N w_i\, D_{\text{KL}}\Big(\Phi_z \hat{T} Z_{\theta^-}(s_i', a_i^*) \,\|\, p_\theta(s_i, a_i)\Big)$
      Perform a gradient update on $\theta$ to minimize $L(\theta)$
      Compute updated priorities for each sampled transition and modify $\mathcal{D}$ accordingly.
      Reset noisy parameters in the online and target networks
      Periodically copy $\theta^- \leftarrow \theta$
    end if
  end while
end for
```

5 Empirical Experiment: Comparing DQN and Rainbow on CartPole-v1

In this section, we present an empirical study aimed at comparing the performance of DQN and Rainbow agents on the CartPole-v1 environment from the Gymnasium suite. The goal of this experiment is to assess the impact of the various enhancements introduced by Rainbow compared to a standard DQN agent.

5.1 Context and Motivation

The CartPole-v1 environment is a classic reinforcement learning task in which an agent must learn to balance a pole vertically by applying discrete forces to the left or right. The system consists of a cart that can move along a track and a pole attached to it via a hinge. The agent's objective is to prevent the pole from falling over by adjusting the cart's movement accordingly. If the pole exceeds a certain angle threshold or the cart moves beyond the track boundaries, the episode terminates. The problem can be modeled as a Markov Decision Process (MDP), where the agent receives a reward of +1 at each step as long as the pole remains upright and the episode does not terminate.

The maximum achievable score in CartPole-v1 is 200, as each episode is capped at 200 time steps. A well-trained agent is expected to reach this upper limit consistently, whereas suboptimal policies tend to score significantly lower. The average score over multiple episodes serves as a key metric for evaluating agent performance, reflecting both learning efficiency and stability.

The choice of CartPole-v1 for this study is motivated by several factors:

- It is a well-known and widely used environment for testing reinforcement learning algorithms.
- It allows for quick results due to its low complexity, making comparative analysis between agents easier.
- It highlights differences in performance in terms of stability, convergence speed, and robustness to hyperparameters.

5.2 Methodology

To compare DQN and Rainbow, both agents are trained on CartPole-v1 under the same experimental conditions:

- A total of 10,000 environment interactions.
- A replay memory size of 10,000 transitions.
- A mini-batch size of 128 for training.
- Target network update every 100 time steps.
- A fixed seed to ensure reproducibility of results.

While Rainbow incorporates several key improvements over DQN, they have been detailed earlier. This experiment focuses on evaluating how these enhancements affect learning efficiency and performance in the CartPole-v1 task.

The performance of the agents will be measured in terms of the average score per episode. Additionally, we analyze the rate of convergence to high scores and the consistency of performance across multiple episodes to assess the robustness of each approach.

5.3 Results and Analysis

The training results of DQN and Rainbow are illustrated in Figures 5 and 6, showing the episode scores and loss evolution over 10,000 frames.

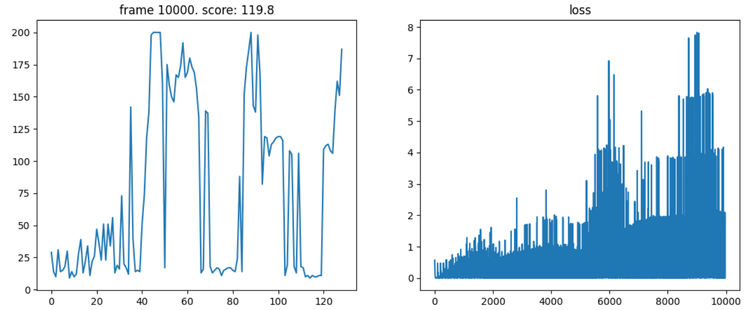

Figure 5: Training performance of DQN on CartPole-v1.

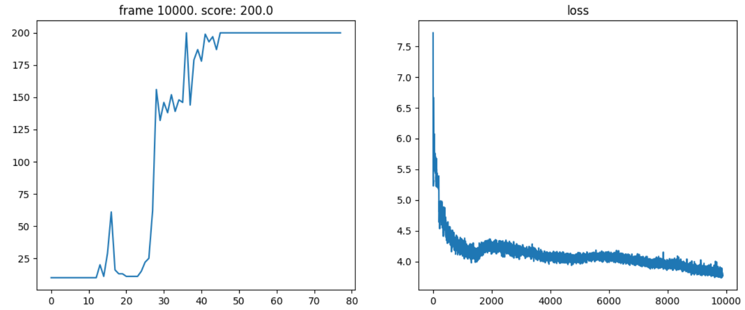

Figure 6: Training performance of Rainbow on CartPole-v1.

The analysis of the results reveals clear distinctions between the two approaches:

- The DQN agent achieves an average score of 119.8 after 10,000 frames, exhibiting significant fluctuations in performance and instability in learning, as seen in the varying episode scores.
- The Rainbow agent reaches the maximum score of 200 much faster and remains consistently at that level, indicating superior stability and efficiency in learning.

- The loss curves show that DQN experiences higher variance and instability in gradient updates, while Rainbow's loss stabilizes more quickly, demonstrating better convergence properties.

These results validate the benefits of Rainbow's enhancements, particularly in terms of learning stability, faster convergence, and robustness against instability. The prioritization of experiences, improved exploration via Noisy Networks, and multi-step learning contribute to this superior performance.

5.4 Conclusion

This study compared the performance of Deep Q-Network (DQN) and Rainbow on the CartPole-v1 environment. The results demonstrated that Rainbow, with its integration of multiple improvements, significantly outperformed the standard DQN, achieving the maximum possible score of 200 more consistently and with improved learning efficiency. The analysis of training curves highlighted the stability and effectiveness of Rainbow's components, such as multi-step learning, prioritized experience replay, and distributional RL.

While our experiment focused on a simpler environment with limited state and action spaces, the original Rainbow paper conducted evaluations on more complex Atari 2600 games, showcasing the scalability and efficiency of Rainbow in high-dimensional scenarios. The paper also studied the individual contributions of each component in Rainbow, isolating their effects through ablation studies.

References

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

[2] 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.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations*, 2016.

[4] Z. Wang, T. Schaul, M. Hessel, *et al.* Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[5] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1), 1988.

[6] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, 2017.

[7] M. Fortunato, M. G. Azhar, I. Piot, *et al.* Noisy networks for exploration. In *International Conference on Learning Representations*, 2018.

[8] 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.

[9] J. F. Hernandez-Garcia and R. S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the DQN target. *arXiv preprint arXiv:1901.07510*, 2019.

[10] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.