به نام خدا

تمرین ششم

(یادگیری تقویتی)

هوش مصنوعی و یادگیری ماشین

نام دانشجو:

امیرعلی محمودزاده طوسی

۸۱۰۶۰۳۱۴۲

استاد درس:

دکتر شریعت پناهی

مرداد ۱۴۰۴

https://github.com/amiralimt/AI_HW6

درک مسئله و آماده‌سازی محیط

هدف اصلی این تمرین، پیاده‌سازی، ارزیابی و مقایسه عملکرد الگوریتم‌های گوناگون یادگیری تقویتی در یک محیط شبیه‌سازی شده بود.

محیط مسئله: اسم محیط MountainCar-v0 از کتابخانه Gymnasium هست، که در این محیط:

یک ماشین در یک دره بین دو تپه قرار دارد، موتور ماشین ضعیف است و نمی‌تواند با یک حرکت مستقیم از تپه بالا برود.

ماشین باید با حرکت‌های رفت و برگشتی، انرژی (مومنتوم) لازم را برای رسیدن به قله تپه سمت راست (جایی که پرچم قرار دارد) کسب کند.

ما این مسئله را در دو حالت بررسی می‌کنیم: یکی با اقدامات گسسته (مثل حرکت به چپ، راست یا هیچ حرکتی) و دیگری با اقدامات پیوسته (اعمال نیروی دلخواه در یک بازه).

قبل از شروع تمرین، در ادامه دو بخش SEED و Classes را تعریف میکنیم:

SEED

با هر بار اجرای کد، نتایج کمی تغییر خواهند کرد و این کاملاً طبیعی است.

این تغییر به دلیل وجود تصادفی بودن (Stochasticity) در چندین بخش از الگوریتم‌های یادگیری تقویتی است:

۱. **مقداردهی اولیه وزن‌ها**: وزن‌های اولیه در شبکه‌های عصبی (DQN) به صورت تصادفی مقداردهی می‌شوند.

۲. **اکتشاف (Exploration)**: سیاست اپسیلون-حریصانه (Epsilon-Greedy) با یک احتمال تصادفی، یک اقدام اکتشافی را انتخاب می‌کند.

۳. **نمونه‌برداری از حافظه**: در DQN، دسته‌های آموزشی به صورت تصادفی از حافظه بازپخش انتخاب می‌شوند.

٤. **نقطه شروع محیط**: حتی نقطه شروع ماشین در محیط نیز می‌تواند کمی تصادفی باشد.

این عوامل باعث می‌شوند که مسیر یادگیری در هر بار اجرا کمی متفاوت باشد و در نتیجه، نمودارها و نتایج نهایی اندکی تغییر کنند.

راه‌حل: استفاده از Seed

برای اینکه نتایج در هر بار اجرا یکسان و قابل تکرار (Reproducible) باشند، باید از یک دانه تصادفی

(Random Seed) استفاده کنیم. وقتی یک عدد ثابت را به عنوان Seed به کتابخانه‌هایی که از اعداد تصادفی استفاده می‌کنند می‌دهیم، آن‌ها همیشه دنباله یکسانی از اعداد تصادفی را تولید خواهند کرد.

فعال کردن حالت قطعی (Deterministic Mode)

برای اینکه نتایج دقیقاً و مطلقاً یکسان باشند، باید به PyTorch بگوییم که از الگوریتم‌های قطعی استفاده کند.

Classes

یک سلول جدید بعد از سلول seed خود ایجاد می کنیم و تمام تعاریف (کلاس‌ها و توابع کمکی) را در آن سلول قرار می دهیم.

این سلول شامل هر سه تعریف مورد نیاز ماست:

DQN, ReplayBuffer, evaluate_agent_detailed

تعریف تابع ارزیابی (evaluate_agent_detailed):

این تابع کمکی بعد از آموزش هر مدل، فراخوانی می شود تا معیار های ارزیابی را محاسبه کند.

زمان آموزش مدل (Training Time)

برای هر الگوریتم، باید مدت زمان آموزش آن را اندازه‌گیری کنیم. این کار بسیار ساده است. کافی است قبل از شروع حلقه آموزش، زمان را ثبت کرده و در پایان، اختلاف آن را محاسبه کنیم.

این کد باید هر حلقه آموزش را در بر بگیرد. یعنی start_time درست قبل از شروع حلقه و end_time درست بعد از پایان آن قرار می‌گیرد.

میانگین گام‌ها تا رسیدن به هدف (Average Steps to Goal)

بعد از اینکه هر مدل آموزش دید، باید عملکرد آن را در چند اپیزود آزمایشی بسنجیم. طبق تمرین، مدل را ۱۰ بار اجرا کرده و میانگین تعداد قدم‌هایی که طول می‌کشد تا به هدف برسد را محاسبه می‌کنیم. در این مرحله آزمایشی، دیگر اکتشافی در کار نیست (epsilon = 0) .

نحوه استفاده از تابع: بعد از اینکه مثلاً آموزش Q-Learning تمام شد، این تابع را به این شکل فراخوانی می‌کنیم:

```
avg_steps = evaluate_agent(env, q_table, 'q_table', get_discrete_state)
```

و برای یک مدل DQN :

```
avg_steps_dqn = evaluate_agent(env, policy_net, 'dqn', device=device)
```

این تابع به گونه‌ای طراحی شده که هوشمند است و می‌تواند تمام مدل‌های ما را ارزیابی کند:

- مدل‌های Q-table
- مدل‌های DQN برای محیط گسسته
- مدل DQN برای محیط پیوسته (با کمک پارامتر discrete_actions)

نمودار تابع هزینه (Loss Curve)

برای الگوریتم‌هایی که از شبکه عصبی استفاده می‌کنند (DQN و DDPG)، رسم نمودار تابع هزینه (Loss) در طول زمان بسیار مفید است. این نمودار نشان می‌دهد که خطای پیش‌بینی شبکه در طول آموزش چگونه کاهش می‌یابد.

یک نمودار Loss که به آرامی و به طور پایدار کاهش می‌یابد، نشان‌دهنده یک فرآیند یادگیری سالم است. اگر نمودار نوسانات شدیدی داشته باشد یا افزایش یابد، نشان‌دهنده ناپایداری در آموزش است. این به شناخت کاستی‌های فرآیند یادگیری کمک می‌کند. کافی است در حلقه آموزش ، مقدار loss را در هر مرحله ذخیره کرده و در انتها میانگین متحرک آن را رسم کنیم.

لیست losses قبل از حلقه آموزش تعریف می‌شود، losses.append() داخل حلقه و بعد از محاسبه loss قرار می‌گیرد و کد plot در انتها و بعد از اتمام آموزش می‌آید.

نرخ موفقیت و پایداری

- **نرخ موفقیت (Success Rate):** در اپیزودهای آزمایشی، عامل چند درصد مواقع موفق می‌شود به هدف برسد؟ (یعنی قبل از تمام شدن حداکثر گام‌ها).
- **انحراف معیار گام‌ها (Standard Deviation of Steps)** : این معیار پایداری عملکرد عامل را نشان می‌دهد. یک انحراف معیار پایین یعنی عامل همیشه در تعداد گام‌های مشابهی مسئله را حل می‌کند، که مطلوب است.

تابع evaluate_agent این دو مقدار را نیز محاسبه می کند. بعد از پایان آموزش هر مدل، این تابع فراخوانی می‌شود.

پیاده‌سازی محیط

طبق دستورالعمل تمرین، اولین کار نصب کتابخانه‌های لازم و ساختن محیط شبیه‌سازی است.

ما برای این تمرین از گوگل کولب (Google Colab) استفاده میکنیم:

۱. **دسترسی به GPU رایگان:** بخش‌های بعدی تمرین، مخصوصاً الگوریتم‌های DQN و DDPG ، نیاز به آموزش شبکه‌های عصبی دارند. این فرآیند روی CPU کامپیوتر معمولی می‌تواند خیلی زمان‌بر باشد. در کولب می‌توان به راحتی از GPU رایگان استفاده کرد و سرعت آموزش رو ده‌ها برابر افزایش داد.

۲. **عدم نیاز به نصب:** لازم نیست نرم افزاری روی کامپیوتر نصب و کانفیگ کرد. فقط با مرورگر وارد کولب می‌شویم و کدنویسی رو شروع می‌کنیم.

یک نوت‌بوک جدید در گوگل کولب باز می کنیم و کد، دستور نصب کتابخانه رو با یک علامت ! در ابتدای آن مینویسیم:

```
!pip install gymnasium

!pip install "gymnasium[classic_control,box2d]"
```

دستور دوم برای اینکه که مطمئن بشویم تمام نیازمندی‌های محیط‌های کلاسیک مثل MountainCar نصب میشوند.

*نکته: خطایی که در این بخش مشاهده می شود یک خطای رایج موقع نصب Box2D در کولب هست، ما برای تمرین MountainCar-v0 به این کتابخانه (Box2D) نیازی نداریم. این خطا فقط برای بخشی اتفاق افتاد که مخصوص محیط‌های فیزیکی پیچیده‌تر است. کتابخانه اصلی gymnasium و بخش classic_control که ما لازم داریم، با موفقیت نصب شده است.

فرمول‌های فیزیک محیط

فرمول‌هایی برای به‌روزرسانی سرعت و موقعیت ماشین آمده است:

- حالت گسسته:

$$velocity_{t+1} = velocity_t + (action - 1) * force - \cos(3 * position_t) * gravity$$

- حالت پیوسته:

$$velocity_{t+1} = velocity_t + force * 0.0015 - 0.0025 * \cos(3 * position_t)$$

این فرمول‌ها قوانین فیزیکی دنیای شبیه‌سازی شده هستند. ما این‌ها را مستقیماً در کد خودمان نمی‌نویسیم. این قوانین در داخل خود محیطی که از کتابخانه gymnasium فراخوانی می‌کنیم (gym.make(...))پیاده‌سازی شده‌اند. وقتی ما دستور env.step(action) را اجرا می‌کنیم، این کتابخانه این محاسبات را برای ما انجام می‌دهد و حالت بعدی را برمی‌گرداند.

پیاده‌سازی الگوریتم‌ها (حالت گسسته)

حالا که محیط آماده است، به بخش اصلی تمرین یعنی پیاده‌سازی الگوریتم‌ها می‌رسیم.

درک فضای حالت و اقدام

فضای حالت (State Space) : این فضا پیوسته است و از دو متغیر تشکیل شده:

- موقعیت (Position) : عددی بین ۱.۲- تا ۰.۶
- سرعت (Velocity) : عددی بین ۰.۰۷- تا ۰.۰۷

فضای اقدام (Action Space) : این فضا گسسته است و ۳ مقدار ممکن دارد:

- ۰ : حرکت به چپ
- ۱ : حرکت نکردن
- ۲ : حرکت به راست

مشکلی که در اینجا هست اینکه الگوریتم Q-learning با فضای حالت گسسته کار می‌کند، اما فضای حالت محیط ما (موقعیت و سرعت) پیوسته است.

راه حل: باید فضای حالت پیوسته رو به یک فضای گسسته تبدیل کنیم. به این کار گسسته‌سازی (Discretization) می‌گویند. ساده‌ترین راه این است که بازه موقعیت و بازه سرعت رو به تعداد مشخصی سطل (bin) تقسیم کنیم.

حالا می‌توانیم در یک اسکریپت، محیط را وارد و بسازیم. این کار با دستور gym.make() انجام می‌شود.

```
import gymnasium as gym

env = gym.make("MountainCar-v0")

initial_state, info = env.reset())
```

```
حالت اولیه محیط: [-0.41235882  0.0       ]
فضای اقدامات (Actions): Discrete(3)
فضای حالات (States): Box([-1.2  -0.07], [0.6  0.07], (2,), float32)
```

حالت اولیه محیط: این یعنی ماشین در شروع، در موقعیت ۰.۴۱۲۳۵۸۸۲- (یک جای تصادفی بین ۰.۴- و ۰.۶- طبق توضیحات تمرین) و با سرعت اولیه صفر قرار گرفته است.

فضای اقدامات (Actions): Discrete(3) : این به ما می‌گوید که عامل ما می‌تواند ۳ اقدام گسسته انجام دهد: ۰ (چپ)، ۱ (هیچ)، و ۲ (راست).

فضای حالات (States): Box([-۱.۲ -۰.۰۷], [۰.۶ ۰.۰۷] ...): نشان‌دهنده فضای حالت پیوسته است. حالت از دو عدد تشکیل شده: اولی موقعیت (بین ۱.۲- و ۰.۶) و دومی سرعت (بین ۰.۰۷- و ۰.۰۷).

گسسته‌سازی فضای حالت

خب، همانطور که دیدیم فضای حالت ما (موقعیت و سرعت) پیوسته است، اما اولین الگوریتمی که می‌خواهیم پیاده کنیم (Q-Learning) نیاز به حالات گسسته دارد.

الگوریتم Q-Learning بر اساس یک جدول کیو (Q-Table) کار می‌کند. این جدول باید برای هر حالت ممکن، ارزش هر اقدام ممکن رو ذخیره کند. حالا تصور کنید بخواهیم برای حالت‌های پیوسته جدول بسازیم: موقعیت می‌تواند ۰.۵- باشد، یا ۰.۵۰۰۱- یا ۰.۵۰۰۰۰۰۱-. ، تعداد حالت‌های ممکن بی‌نهایت است. ما نمی‌توانیم یک جدول با تعداد سطرهای بی‌نهایت بسازیم. پس باید این بی‌نهایت حالت را به تعداد محدودی حالت قابل مدیریت خلاصه کنیم.

طبق راهنمایی تمرین، باید این بازه‌های پیوسته را به بخش‌های کوچکتر تقسیم کنیم. ما بازه موقعیت و بازه سرعت را هر کدام به ۳۰ قسمت تقسیم می‌کنیم. بعد از این کار، هر جفت (موقعیت، سرعت) به یک خانه مشخص در یک جدول بزرگ تعلق پیدا می‌کند. به این جدول بزرگ Q-Table می‌گوییم.

این جدول به ازای هر حالت گسسته (هر خانه از جدول) و هر اقدام، یک مقدار (Q-Value) را ذخیره می‌کند که نشان می‌دهد آن اقدام در آن حالت چقدر خوب است.

کد مربوط به این بخش Q-Table را با ابعاد مناسب می‌سازد و آن را با صفر پر می‌کند.

```
ابعاد جدول Q: (30, 30, 3)
```

q_table یک آرایه سه‌بعدی با ابعاد (30, 30, 3) است.

- بعد اول (۳۰) برای سطل‌های موقعیت.
- بعد دوم (۳۰) برای سطل‌های سرعت.
- بعد سوم (۳) برای ۳ اقدام ممکن.

حالا یک جدول با اندازه محدود داریم که الگوریتم Q-Learning می‌تواند مقادیر آن را در طول زمان یاد بگیرد و به‌روزآوری کند.

Q-Learning

حلقه یادگیری Q-Learning

در این مرحله، عامل (ماشین) را در محیط رها می‌کنیم تا با آزمون و خطا یاد بگیرد. این فرآیند در چند هزار مرحله (که به آنها اپیزود می‌گوییم) تکرار می‌شود، منطق حلقه به این صورت است:

۱. **شروع اپیزود:** ماشین را در یک نقطه اولیه قرار می‌دهیم.

۲. **انتخاب اقدام:** عامل بر اساس جدول Q و با کمی شانس (که به آن اپسیلون می‌گوییم) یک اقدام را انتخاب می‌کند. در ابتدا که جدول خالی است، بیشتر اقدامات تصادفی هستند (اکتشاف یا Exploration) به مرور که جدول پر می‌شود، عامل بیشتر به اقدامات بهتر تکیه می‌کند (بهره‌برداری یا Exploitation).

۳. **اجرای اقدام:** ماشین حرکت می‌کند و محیط به ما یک پاداش (معمولاً منفی، چون در هر لحظه که به هدف نرسیده، جریمه می‌شود) و حالت جدید را می‌دهد.

۴. **به‌روزرسانی جدول Q :** اینجاست که یادگیری اتفاق می‌افتد. ما با استفاده از فرمول به‌روزرسانی Q-learning، مقدار خانه‌ای از جدول که مربوط به حالت و اقدام فعلی بود را کمی اصلاح می‌کنیم.

۵. این فرآیند تا زمانی که ماشین به پرچم برسد یا تعداد گام‌های مجاز (۲۰۰) تمام شود، ادامه پیدا می‌کند.

۶. در پایان هر اپیزود، مقدار اپسیلون را کمی کاهش می‌دهیم تا عامل در آینده بیشتر به دانش خود تکیه کند.

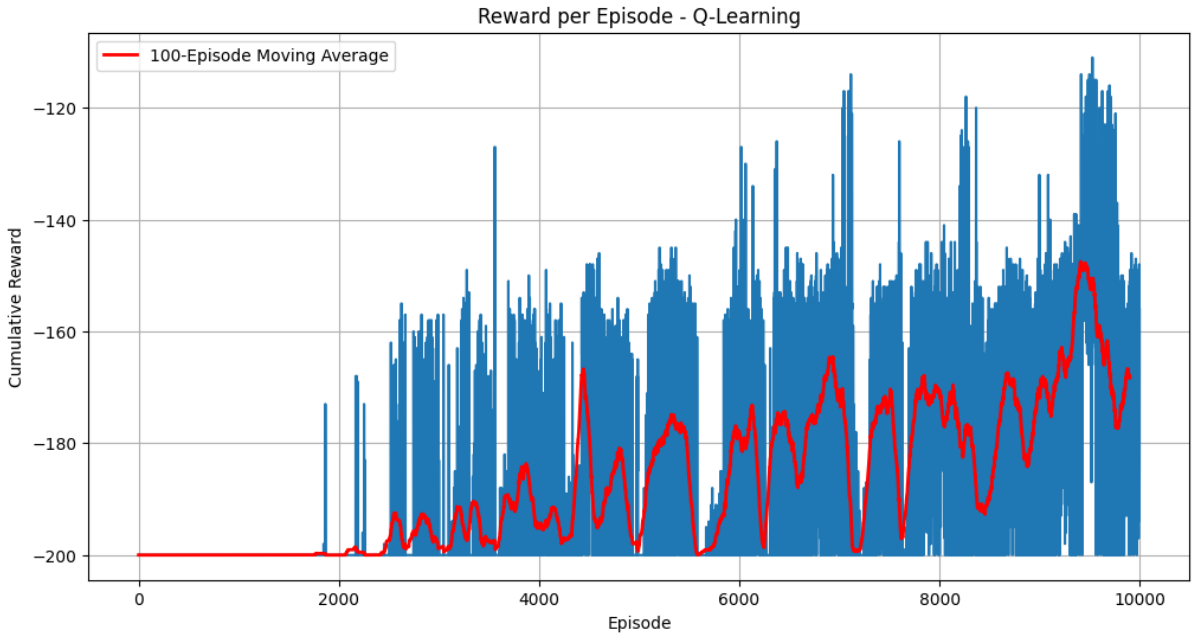
```
learning_rate = 0.1
discount_factor = 0.99
epsilon = 1.0
epsilon_decay_rate = 0.9995
min_epsilon = 0.1
n_episodes = 10000
```

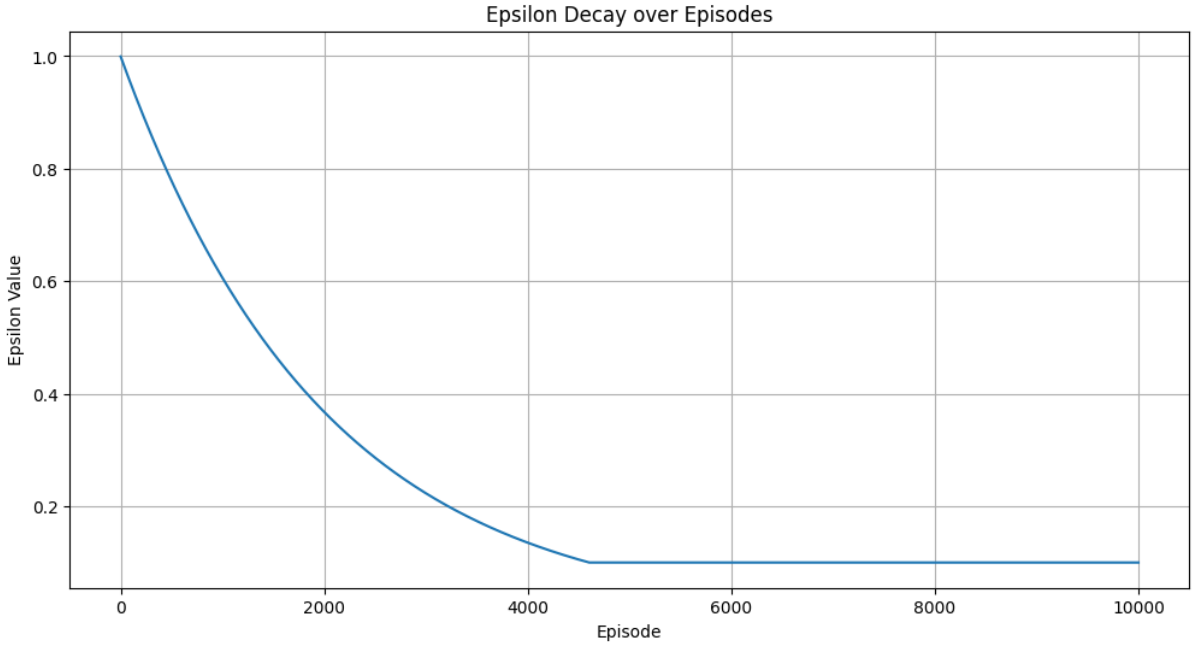
- **فرمول تئوری (استاندارد):**

$$Q(s,a)= (s,a)+alpha[r+gammamax\_a'Q(s',a')-Q(s,a)]$$

```
old_q_value = q_table[state_discrete + (action,)]

next_optimal_q_value = np.max(q_table[new_state_discrete])

new_q_value = old_q_value + learning_rate * (reward + discount_factor *
next_optimal_q_value - old_q_value)
```

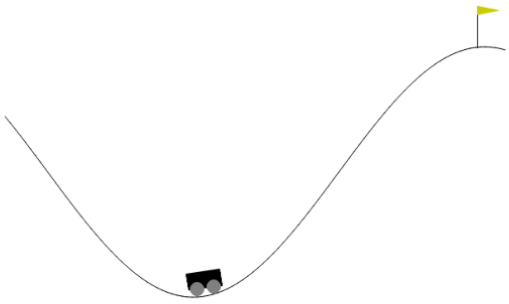
```
--- Starting Q-Learning Training ---

Episode 1000, Epsilon: 0.6065, Avg Reward (last 100): -200.00

Episode 2000, Epsilon: 0.3678, Avg Reward (last 100): -200.00

Episode 3000, Epsilon: 0.2230, Avg Reward (last 100): -194.33

Episode 4000, Epsilon: 0.1353, Avg Reward (last 100): -186.84

Episode 5000, Epsilon: 0.1000, Avg Reward (last 100): -194.21

Episode 6000, Epsilon: 0.1000, Avg Reward (last 100): -185.19

Episode 7000, Epsilon: 0.1000, Avg Reward (last 100): -164.65

Episode 8000, Epsilon: 0.1000, Avg Reward (last 100): -172.37

Episode 9000, Epsilon: 0.1000, Avg Reward (last 100): -181.16

Episode 10000, Epsilon: 0.1000, Avg Reward (last 100): -168.25


Training finished in 109.83 seconds.


--- Evaluating Q-Learning Agent ---

Success Rate: 99.00%

Average Steps to Goal: 172.82 ± 18.50
```





اجرای الگوریتم و مشاهده انیمیشن

حالا از جدول Q که آموزش داده‌ایم، استفاده می‌کنیم تا ببینیم ماشین در عمل چطور رفتار می‌کند. در این مرحله دیگر یادگیری یا اکتشافی در کار نیست (epsilon = 0) و ماشین فقط بهترین اقدامی که یاد گرفته را انجام می‌دهد.



در انیمیشن مشاهده می شود ماشین با چند حرکت نوسانی به راست و چپ به پرچم می رسد، این رفتار استراتژی هوشمندانه‌ای است که الگوریتم Q-learning برای حل این مسئله پیدا کرده است. همانطور که در صورت تمرین گفته شده، موتور ماشین ضعیف است و نمی‌تواند مستقیماً از تپه بالا برود. بنابراین، عامل یاد گرفته که برای رسیدن به هدف، باید انرژی یا مومنتوم کسب کند. بهترین راه برای جمع کردن این انرژی، بالا رفتن از تپه سمت چپ است. وقتی از آن تپه پایین می‌آید، سرعتش زیاد می‌شود و از همین سرعت برای صعود از تپه اصلی (سمت راست) استفاده می‌کند.

اینکه الگوریتم توانسته چنین استراتژی غیر واضحی را خودش کشف کند، قدرت یادگیری تقویتی را نشان می‌دهد.

➢ تحلیل نهایی الگوریتم Q-Learning

این الگوریتم به عنوان مدل پایه برای یادگیری در محیط با اقدامات گسسته پیاده‌سازی شد. از آنجایی که فضای حالت محیط (position, velocity) پیوسته است، ابتدا با تقسیم هر بعد به ۳۰ بخش، آن را به یک فضای حالت گسسته تبدیل کردیم. سپس یک جدول Q با ابعاد (30, 30, 3) برای ذخیره ارزش هر زوج حالت–اقدام ساخته شد. یادگیری با استفاده از معادله به‌روزرسانی Bellman و یک سیاست اکتشافی اپسیلون–حریصانه (Epsilon-Greedy) انجام شد. مقدار اپسیلون از 1.0 شروع شده و به تدریج تا 0.1 کاهش یافت تا تعادل بین اکتشاف و بهره‌برداری حفظ شود.

- **فرآیند آموزش**: نمودار Reward per Episode نشان می‌دهد که عامل در حدود ۲۰۰۰ اپیزود اول، به دلیل اکتشاف بالا، عملکرد ضعیفی داشته است. اما پس از آن، با کاهش اپسیلون و بهره‌برداری از دانش کسب‌شده، میانگین پاداش (خط قرمز) یک روند صعودی را طی می‌کند که نشان‌دهنده یادگیری موفق است.

- **ارزیابی نهایی**: نتایج ارزیابی نهایی، موفقیت این الگوریتم را تأیید می‌کند.
  - **نرخ موفقیت ۹۹٪**: این نرخ بسیار بالا نشان می‌دهد که سیاست یادگرفته شده بسیار قابل اعتماد (reliable) است.
  - **میانگین گام‌ها (172.82 ± 18.50)** : عامل به طور میانگین در حدود ۱۷۲ قدم به هدف می‌رسد. انحراف معیار پایین (18.50) نیز نشان‌دهنده پایداری (stability) و عملکرد سازگار عامل در اجراهای مختلف است.
- **زمان آموزش**: کل فرآیند آموزش در حدود ۱۱۰ ثانیه به پایان رسید که برای ۱۰۰۰۰ اپیزود، زمان معقولی است.

الگوریتم Q-Learning با روش گسسته‌سازی خطی، توانست با موفقیت یک سیاست پایدار و قابل اعتماد برای حل مسئله MountainCar-v0 یاد بگیرد. این الگوریتم به عنوان یک مدل پایه قوی عمل می‌کند که می‌توان عملکرد الگوریتم‌های دیگر را با آن مقایسه کرد.

Double Q-Learning

درک الگوریتم-Q یادگیری دوگانه

مقاله و تمرین به یک مشکل اساسی در الگوریتم Q-Learning اشاره می‌کنند: تخمین بیش از حد (Overestimation).

چرا Q-Learning بیش از حد تخمین می‌زند؟

در Q-learning استاندارد، برای به‌روزرسانی مقدار یک حالت، از همان جدول Q هم برای انتخاب بهترین اقدام بعدی و هم برای ارزیابی (قیمت‌گذاری) همان اقدام استفاده می‌کنیم.

$$\ldots + \gamma \times \max(Q(s', a)) \ldots$$

عملگر max باعث ایجاد یک سوگیری مثبت (positive bias) می‌شود. اگر به دلیل نویز یا شانس، ارزش یک اقدام به اشتباه بالا تخمین زده شود، عملگر max همان مقدار اشتباه و خوش‌بینانه را انتخاب می‌کند و این خطا در یادگیری پخش می‌شود. این باعث می‌شود عامل فکر کند برخی اقدامات بهتر از چیزی که واقعا هستند، ارزش دارند.

راه‌حل یادگیری دوگانه Q چیست؟

برای حل این مشکل، Double Q-learning عمل انتخاب بهترین اقدام را از عمل ارزیابی آن جدامی‌کند. این کار را با استفاده از دو جدول Q مجزا انجام می‌دهد:

- Q_A
- Q_B

منطق به‌روزرسانی به این شکل است:

۱. فرض کنید می‌خواهیم جدول Q_A را آپدیت کنیم.

۲. برای پیدا کردن بهترین اقدام در حالت بعدی (s')، از جدول Q_A استفاده می‌کنیم (انتخاب).

۳. اما برای به دست آوردن ارزش آن اقدام، از جدول دیگر یعنی Q_B استفاده می‌کنیم (ارزیابی).

این کار مثل این است که برای یک تصمیم مهم، از یک مشاور برای انتخاب بهترین گزینه کمک بگیریم و از یک مشاور دیگر بخواهیم ارزش آن گزینه را قیمت‌گذاری کند. این نظر دوم باعث می‌شود تخمین‌های ما واقعی‌تر باشند و از خوش‌بینی بیش از حد جلوگیری شود. در هر مرحله از یادگیری، به صورت تصادفی یکی از این دو جدول آپدیت می‌شود.

پیاده‌سازی

طبق تمرین، گام‌های اولیه مثل قبل است. فقط باید به جای یک جدول Q ، دو جدول بسازیم.

کد دو جدول Q_A و Q_B را با همان ابعاد قبلی می‌سازد و با صفر مقداردهی اولیه می‌کند.

ابعاد جدول(30, 30, 3) :Q_A
ابعاد جدول(30, 30, 3) :Q_B

حلقه یادگیری Double Q-learning

این حلقه بسیار شبیه به حلقه قبلی است، اما با چند تفاوت کلیدی:

انتخاب اقدام: برای انتخاب بهترین اقدام (در حالت بهره‌برداری)، به جای یک جدول، از مجموع ارزش دو جدول Q_A و Q_B استفاده می‌کنیم. این کار به ما یک دید جامع‌تر از ارزش واقعی اقدامات می‌دهد.

۱. **به‌روزرسانی تصادفی:** در هر مرحله، به صورت تصادفی (مثلاً با احتمال ۵۰/۵۰) تصمیم می‌گیریم که جدول Q_A را به‌روزرسانی کنیم یا Q_B را.

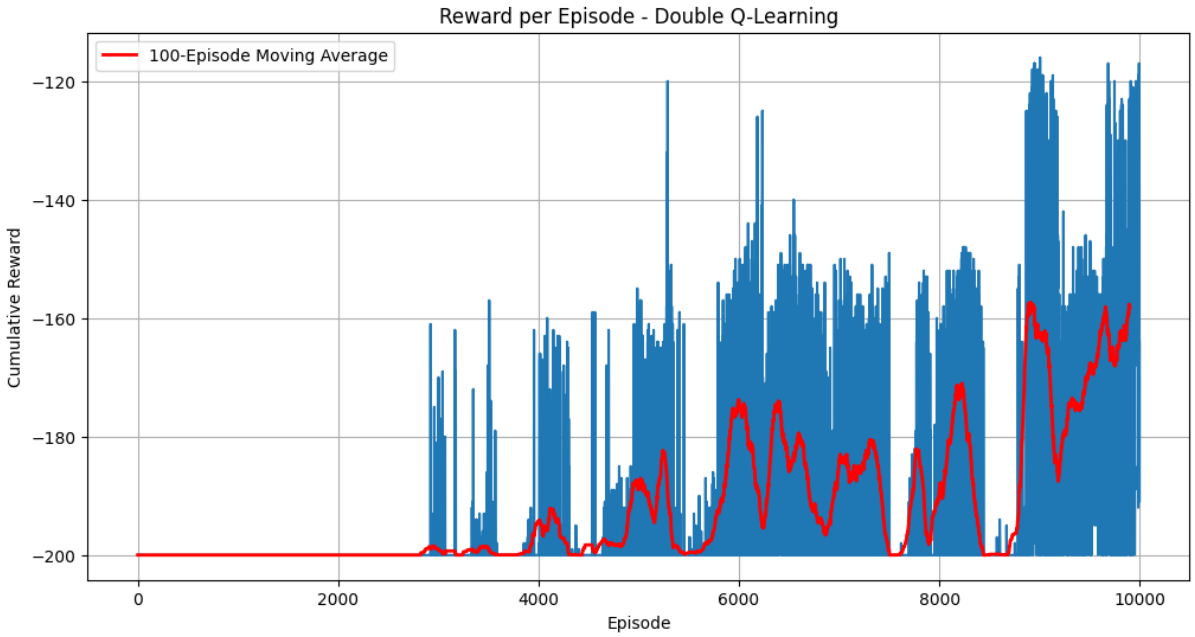
۲. **به‌روزرسانی متقاطع (Cross-Update) :**این مهم‌ترین بخش است.

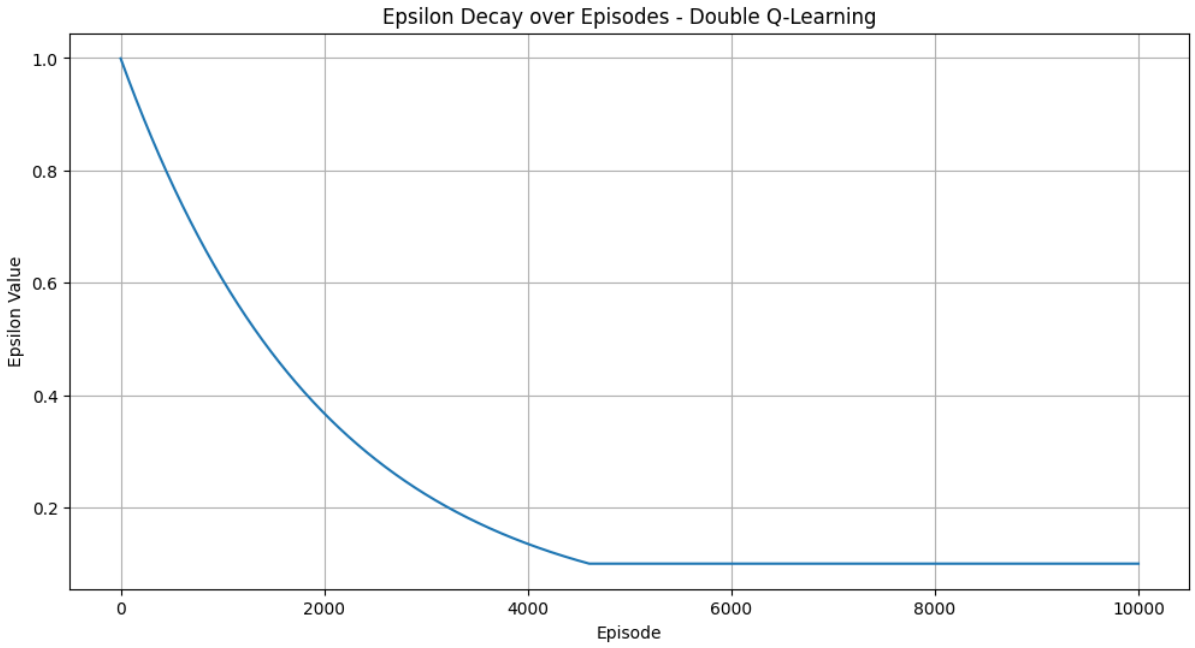
- اگر نوبت به‌روزرسانی Q_A باشد، بهترین اقدام بعدی را از Q_A پیدا می‌کنیم، اما ارزش آن را از Q_B می‌خوانیم.

- اگر نوبت به‌روزرسانی Q_B باشد، بهترین اقدام بعدی را از Q_B پیدا می‌کنیم، اما ارزش آن را از Q_A می‌خوانیم.

هایپرپارامترها همان مقادیر قبلی هستند.

```
--- Starting Double Q-Learning Training ---
Episode 1000, Epsilon: 0.6065, Avg Reward (last 100): -200.00
Episode 2000, Epsilon: 0.3678, Avg Reward (last 100): -200.00
Episode 3000, Epsilon: 0.2230, Avg Reward (last 100): -198.91
Episode 4000, Epsilon: 0.1353, Avg Reward (last 100): -199.41
Episode 5000, Epsilon: 0.1000, Avg Reward (last 100): -192.62
Episode 6000, Epsilon: 0.1000, Avg Reward (last 100): -181.79
Episode 7000, Epsilon: 0.1000, Avg Reward (last 100): -194.82
Episode 8000, Epsilon: 0.1000, Avg Reward (last 100): -197.93
Episode 9000, Epsilon: 0.1000, Avg Reward (last 100): -159.42
Episode 10000, Epsilon: 0.1000, Avg Reward (last 100): -158.10

Training finished in 84.42 seconds.

--- Evaluating Double Q-Learning Agent ---
Success Rate: 88.00%
Average Steps to Goal: 156.74 ± 20.25
```





انیمیشن

قدم بعدی، مصورسازی سیاستی است که یاد گرفته شده، درست همانطور که برای الگوریتم اول انجام دادیم.

رفتار نهایی عامل (انیمیشن): همانطور که مشاهده شد، رفتار نهایی ماشین در هر دو الگوریتم شبیه به هم بود (حرکت نوسانی برای کسب انرژی). این نتیجه کاملاً مورد انتظار است، زیرا هر دو الگوریتم به دنبال یافتن یک سیاست بهینه یکسان برای حل مسئله هستند. تفاوت اصلی در چگونگی و سرعت رسیدن به این سیاست است.

➢ تحلیل نهایی الگوریتم Double Q-Learning

این الگوریتم نسخه‌ای پیشرفته‌تر از Q-Learning استاندارد است که برای مقابله با سوگیری تخمین بیش از حد (overestimation bias) که می‌تواند در حین آموزش رخ دهد، طراحی شده است. برای رسیدن به این هدف، Double Q-Learning از دو جدول Q مجزا استفاده می‌کند. در مرحله به‌روزرسانی، یکی از جداول برای انتخاب بهترین اقدام آینده استفاده می‌شود، در حالی که جدول دیگر برای ارزیابی ارزش آن اقدام به کار می‌رود. این جداسازی انتخاب از ارزیابی، به تولید تخمین‌های ارزشی دقیق‌تر و قابل اعتمادتری کمک می‌کند. سایر بخش‌های پیاده‌سازی، از جمله گسسته‌سازی حالت و استراتژی اکتشاف اپسیلون-حریصانه، مانند مدل Q-Learning استاندارد باقی ماند.

- **فرآیند آموزش**: نمودار پاداش نشان می‌دهد که Double Q-Learning ،مشابه Q-Learning استاندارد، یادگیری مؤثر را پس از حدود ۳۰۰۰ اپیزود آغاز می‌کند. با این حال، یادگیری نوسان بیشتری دارد و شامل مراحل مشخصی از بهبود و افت موقت عملکرد است (مانند حوالی اپیزود ۸۰۰۰). علی‌رغم این نوسانات، عامل در پایان آموزش به سطح عملکردی قوی دست یافت، همانطور که میانگین پاداش نهایی (158.10-) نشان می‌دهد که به طور قابل توجهی بهتر از نتیجه QLearning (168.25-) است.
- **ارزیابی نهایی**: نتایج ارزیابی، برتری سیاست یادگرفته‌شده را برجسته می‌کند:
  - **نرخ موفقیت (٪۸۸)**: اگرچه در این اجرای خاص، این نرخ کمی پایین‌تر از نرخ ٪۹۹ الگوریتم Q-Learning است، اما نرخ موفقیت ٪۸۸ همچنان عملکردی بسیار مؤثر و قابل اعتماد محسوب می‌شود.
  - **میانگین گام‌ها (156.74 ± 20.25)** : این یک معیار کلیدی است. عامل به‌طور میانگین مسئله را در تعداد گام‌های به مراتب کمتری نسبت به Q-Learning (172.82) حل کرده است. این نشان می‌دهد که Double Q-Learning یک سیاست بهینه‌تر پیدا کرده است. انحراف معیار قابل مقایسه، نشان‌دهنده پایداری عملکرد مشابه است.
- **زمان آموزش**: فرآیند آموزش تقریباً در ۸۴ ثانیه به پایان رسید که به طور محسوسی سریع‌تر از مدل Q-Learning بود.

الگوریتم Double Q-Learning با موفقیت توانست مسئله MountainCar-v0 را حل کند. با کاهش سوگیری تخمین بیش از حد، این الگوریتم یک سیاست بهینه‌تر نسبت به Q-Learning استاندارد کشف کرد که به آن امکان رسیدن به هدف در گام‌های کمتر و در زمان آموزش کوتاه‌تر را می‌دهد و مزایای عملی خود را به عنوان یک روش جدولی پیشرفته‌تر به اثبات می‌رساند.

Deep Q-Network (DQN)

این الگوریتم یک گام بسیار مهم رو به جلو در یادگیری تقویتی برداشته و به ما اجازه میدهد مسائلی با فضای حالت بسیار بزرگ یا پیوسته رو حل کنیم.

الگوریتم‌های Q-Learning و Double Q-Learning که پیاده‌سازی کردیم، از یک جدول برای ذخیره ارزش هر زوج حالت-اقدام استفاده می‌کردند. این روش دو محدودیت بزرگ دارد:

۱. **از دست رفتن اطلاعات**: ما فضای حالت پیوسته (موقعیت و سرعت) را به یک شبکه ۳۰x۳۰ گسسته تبدیل کردیم. این کار باعث می‌شود دقت ما کم شود. برای مثال، دو حالت بسیار نزدیک به هم اما در دو سطل متفاوت، کاملاً جدا در نظر گرفته می‌شوند.

۲. **نفرین ابعاد (Curse of Dimensionality)** : محیط ما فقط دو متغیر حالت داشت. تصور کنید یک ربات با ۱۰ سنسور مختلف داشته باشیم. اگر بخواهیم هر سنسور را حتی به ۱۰ بخش تقسیم کنیم، تعداد خانه‌های جدول ما 10^{10} خواهد بود که مدیریت و ذخیره آن غیرممکن است.

راه‌حل DQN : استفاده از شبکه عصبی

DQN به جای استفاده از یک جدول، از یک شبکه عصبی عمیق برای تقریب تابع ارزش Q استفاده می‌کند.

- **ورودی شبکه**: حالت فعلی محیط (مثلاً [موقعیت, سرعت]).
- **خروجی شبکه**: ارزش Q برای هر کدام از اقدامات ممکن در آن حالت.

این شبکه به جای حفظ کردن تک‌تک حالت‌ها، یاد می‌گیرد که یک تابع را تقریب بزند. در نتیجه، می‌تواند ارزش Q را برای حالت‌هایی که حتی قبلاً ندیده است هم تخمین بزند که این یک مزیت بزرگ است.

اجزای کلیدی DQN

۱. **حافظه بازپخش (Replay Buffer)** : برای اینکه آموزش شبکه عصبی پایدار باشد، تجربیات عامل (state, action, reward, next_state, done) را در یک حافظه ذخیره می‌کنیم. در هر مرحله از آموزش، به جای استفاده از آخرین تجربه، یک دسته (batch) تصادفی از تجربیات را از این حافظه نمونه‌برداری کرده و شبکه را با آن آموزش می‌دهیم. این کار ارتباط متوالی بین داده‌ها را از بین برده و به پایداری آموزش کمک می‌کند. برای پایدارسازی فرآیند آموزش و از بین بردن همبستگی بین داده‌های متوالی، از یک حافظه بازپخش با ظرفیت ۱۰۰۰۰۰ استفاده شد.

۲. **معماری شبکه**: یک شبکه عصبی پیش‌خور (Feed-Forward) با سه لایه خطی تعریف شد:

- **لایه اول**: لایه ورودی با ابعاد فضای حالت (۲) و ۱۲۸ نورون خروجی.
- **لایه دوم**: لایه پنهان با ۱۲۸ نورون ورودی و ۶۴ نورون خروجی.
- **لایه سوم**: لایه خروجی با ۶۴ نورون ورودی و ابعاد فضای اقدام (۳) به عنوان خروجی.
- **تابع فعال‌سازی**: در تمام لایه‌های پنهان از تابع ReLU استفاده شد.

۳. **شبکه هدف (Target Network)**

برای افزایش پایداری در محاسبه مقادیر Q هدف، از یک شبکه هدف مجزا استفاده شد. این شبکه یک کپی از شبکه اصلی (Policy Network) بود که وزن‌های آن با تأخیر (هر ۱۰ اپیزود یکبار) به‌روزرسانی می‌شدند. این کار از نوسانات شدید در مقادیر هدف جلوگیری کرده و به همگرایی بهتر مدل کمک می‌کند.

- فرمول مقدار هدف (Target) :

$$\text{Target} = Q_{\text{target}}(s',a)\max_a . \gamma + r$$

```
next_q_values = target_net(next_states_tensor).max(1[۰](
) … #handling terminal states(
target_q_values = rewards_tensor + (GAMMA * next_q_values)
```

این بخش از کد همین مقدار هدف را برای آموزش شبکه محاسبه می‌کند.

طرح آزمایش‌ها

طبق خواسته تمرین، برای بررسی تأثیر مولفه‌های مختلف، شش آزمایش مجزا طراحی و اجرا شد:

آزمایش	بهینه‌ساز(Optimizer)	تابع هزینه(Loss)	تابع پاداش(Reward)
۱	Adam	MSE	پیش‌فرض(-1)
۲	RMSprop	MSE	پیش‌فرض(-1)
۳	Adam	Smooth L1 Loss	پیش‌فرض(-1)
۴	Adam	Smooth L1 Loss	شکل‌دهی شده
۵	RMSprop	Smooth L1 Loss	پیش‌فرض(-1)
۶	Adam	MSE	شکل‌دهی شده

تعریف ساختارها

ابتدا باید کلاس‌های مربوط به شبکه عصبی و حافظه بازپخش را با استفاده از کتابخانه PyTorch تعریف کنیم.

```
DQN Model Architecture:
DQN(
  (layer1): Linear(in_features=2, out_features=128, bias=True)
  (layer2): Linear(in_features=128, out_features=64, bias=True)
  (layer3): Linear(in_features=64, out_features=3, bias=True)
)
Replay Buffer initialized with capacity: 100000
```

منطق حلقه یادگیری DQN

۱. **جمع‌آوری تجربه**: مثل قبل، عامل در محیط حرکت کرده و تا حدودی به صورت تصادفی (با اپسیلون) اقدام انتخاب می‌کند. هر تجربه که شامل (حالت، اقدام، پاداش، حالت بعدی، پایان) است را در حافظه بازپخش ذخیره می‌کنیم.

۲. **شروع یادگیری از حافظه**: بعد از اینکه حافظه به اندازه کافی پر شد، فرآیند یادگیری شروع می‌شود. در هر مرحله:

- یک دسته (batch) تصادفی از تجربیات را از حافظه نمونه‌برداری می‌کنیم.
- برای این دسته، دو مقدار را محاسبه می‌کنیم:

- **مقدار Q پیش‌بینی شده**: مقادیر Q که شبکه عصبی ما در حال حاضر برای زوج‌های حالت-اقدام آن دسته پیش‌بینی می‌کند.
- **مقدار Q هدف(Target)** : مقدار بهینه‌ای که انتظار داریم شبکه پیش‌بینی کند. این مقدار با استفاده از پاداش و فرمول بلمن محاسبه می‌شود:

(پاداش + گاما * بیشینه ارزش Q حالت بعدی)

برای پایدارسازی، ارزش Q حالت بعدی را از یک شبکه جداگانه به نام شبکه هدف (Target Network) می‌خوانیم که یک کپی با تأخیر از شبکه اصلی ماست.

۳. **محاسبه خطا و به‌روزرسانی**: تفاوت بین مقدار پیش‌بینی شده و مقدار هدف به عنوان خطا (loss) در نظر گرفته می‌شود. سپس از الگوریتم بهینه‌ساز (مثل Adam) استفاده می‌کنیم تا وزن‌های شبکه عصبی را طوری تغییر دهد که این خطا کمینه شود (فرآیندBackpropagation)

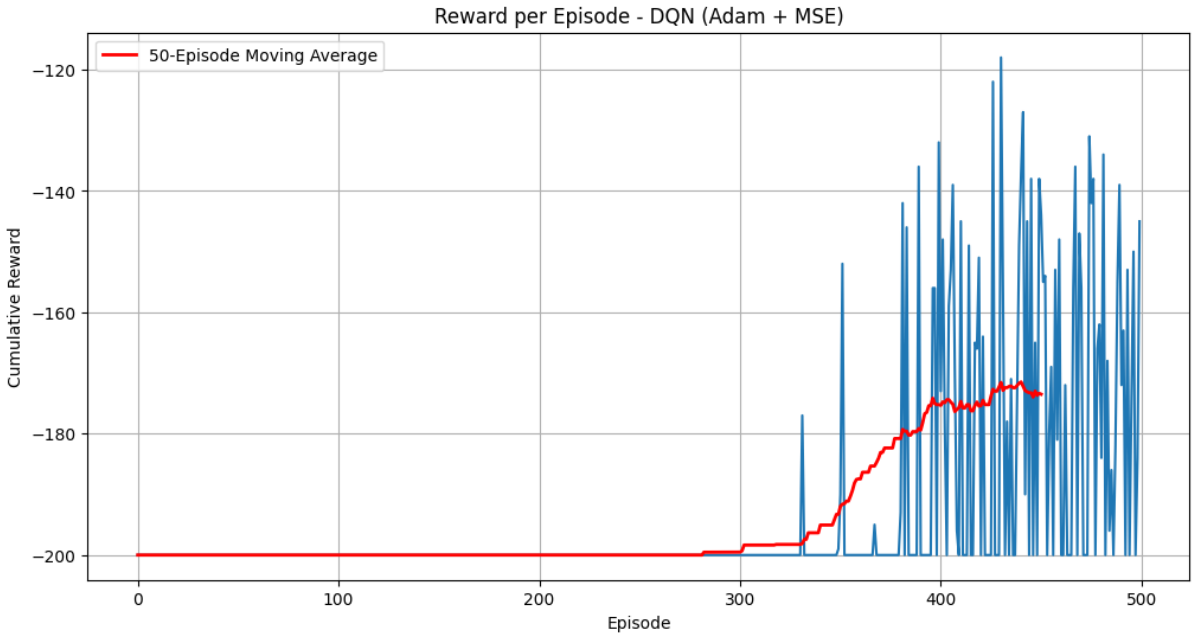
این فرآیند برای هزاران اپیزود تکرار می‌شود تا شبکه به تدریج یاد بگیرد که ارزش Q را به درستی تقریب بزند.

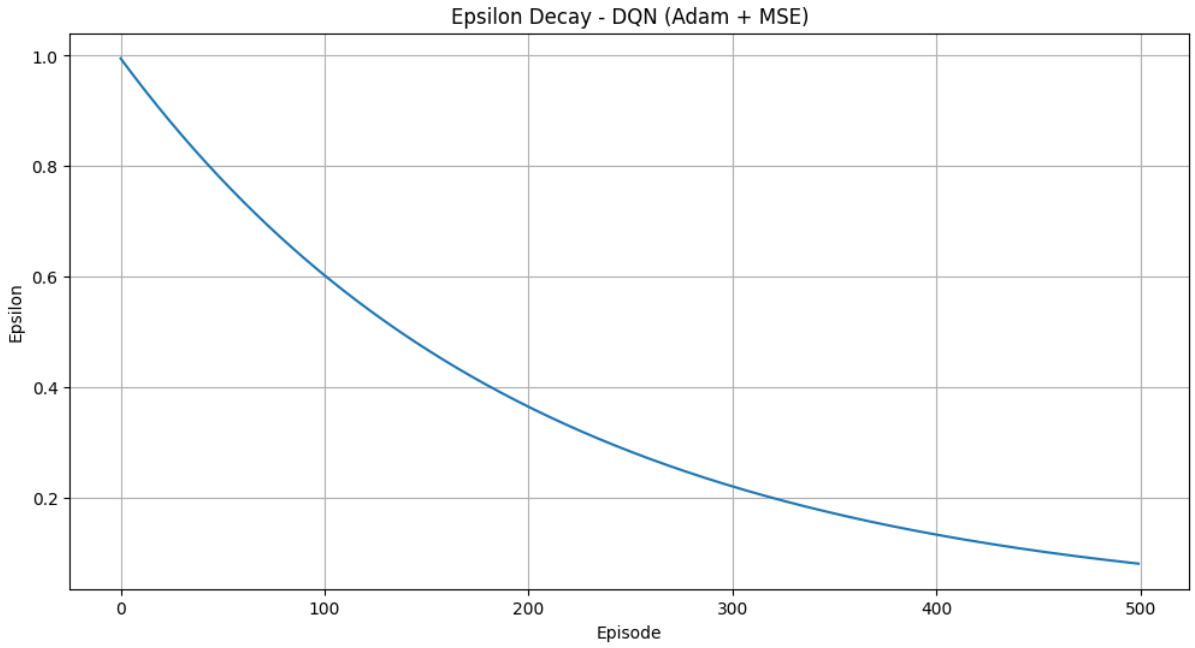
پیاده‌سازی حلقه یادگیری

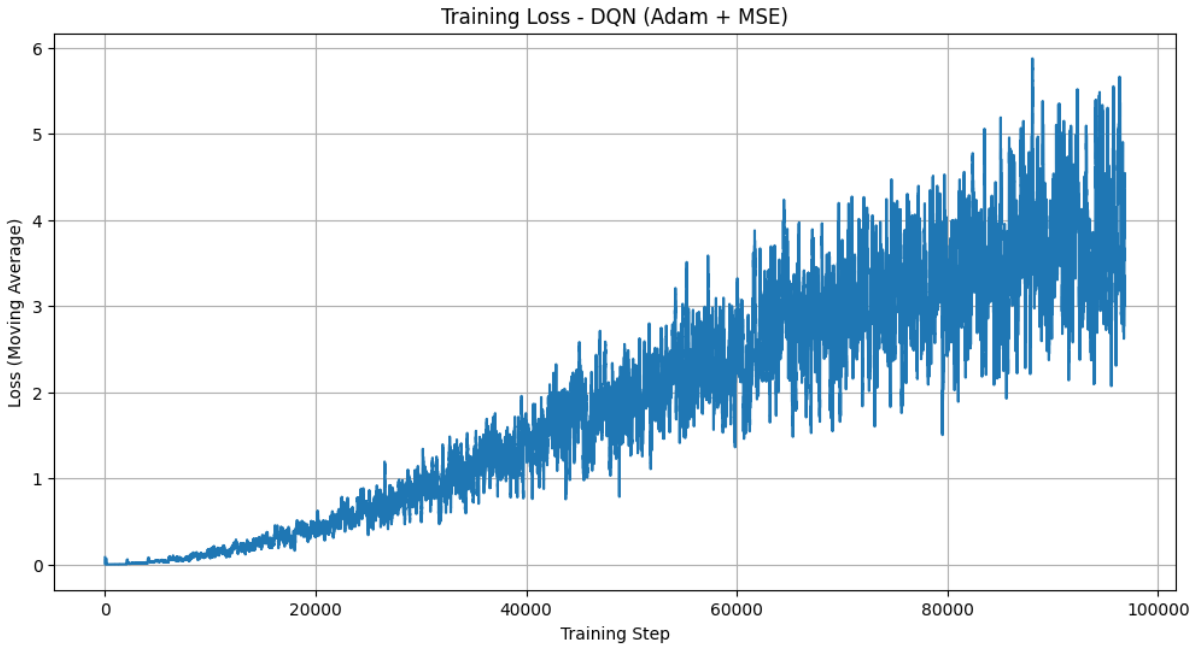
کد هایپرپارامترهای خواسته شده در تمرین را تنظیم کرده و حلقه یادگیری DQN را پیاده‌سازی می‌کند.

حلقه اصلی با هایپرپارامترهای مشخص شده (نرخ یادگیری ۰.۰۰۱ ، گاما ۰.۹۹ ، اپسیلون با شروع ۱ و کاهش تا ۰.۰۱ ، بهینه‌ساز Adam و تابع هزینه MSE) اجرا شد.

```
--- Starting DQN Training (Experiment 1: Adam + MSE) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -200.00
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -200.00
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -200.00
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -200.00
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -199.52
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -191.96
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -175.34
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -173.50

Training finished in 226.68 seconds.
--- Evaluating DQN Agent (Adam + MSE) ---
Success Rate: 39.00%
Average Steps to Goal: 123.74 ± 5.00
```


Reward per Episode - DQN (Adam + MSE)
50-Episode Moving Average
Cumulative Reward
−120
−140
−160
−180
−200
0
100
200
300
400
500
Episode


Epsilon Decay - DQN (Adam + MSE)
Epsilon
1.0
0.8
0.6
0.4
0.2
0
100
200
300
400
500
Episode



انیمیشن

در انیمیشن مشاهده می شود ماشین با یک حرکت به راست و یک حرکت به چپ انرژی لازم را کسب و در سمت راست به پرچم می رسد.

اینکه عامل DQN حتی با پانصد اپیزود توانسته سیاست بهینه (حرکت نوسانی برای کسب انرژی) را یاد بگیرد، یک نتیجه بسیار خوب است. این نشان می‌دهد که الگوریتم در مسیر درستی قرار دارد، هرچند همانطور که در نمودار دیدیم، برای رسیدن به پایداری به زمان بیشتری نیاز دارد.

➢ تحلیل نهایی الگوریتم DQN - Experiment 1: Adam + MSE

این مدل، اولین پیاده‌سازی با استفاده از یک شبکه عصبی برای تقریب تابع Q است که نیاز به گسسته‌سازی دستی حالت را برطرف می‌کند. ما از معماری مشخص شده (۱۲۸ <- ۶۴ <- ۳) با توابع فعال‌ساز ReLU استفاده کردیم. اجزای کلیدی این رویکرد شامل یک حافظه بازپخش (Replay Buffer) برای ذخیره تجربیات و شکستن همبستگی داده‌ها، و یک شبکه هدف (Target Network) مجزا برای پایدارسازی فرآیند یادگیری است .این آزمایش پایه از بهینه‌ساز Adam و خطای میانگین مربعات (MSE) به عنوان تابع هزینه استفاده کرد.

- **فرآیند آموزش**: نمودار پاداش نشان می‌دهد که عامل در ۳۰۰ اپیزود اول با چالش مواجه بوده و در یافتن یک سیاست موفق ناتوان بوده است. پس از این نقطه، یادگیری آغاز شده و میانگین پاداش بهبود یافته است، اگرچه بسیار ناپایدار باقی مانده و هرگز به پایداری روش‌های جدولی نرسیده است.

- **نمودار تابع هزینه**: مهم‌ترین ابزار تشخیصی، یعنی نمودار تابع هزینه آموزش، یک مشکل قابل توجه را آشکار می‌کند. مقدار Loss در طول فرآیند آموزش به طور پیوسته در حال افزایش است. این یک نشانه از ناپایداری در آموزش است. این یعنی با کاوش عامل و تغییر مقادیر Q هدف، پیش‌بینی‌های شبکه به جای همگرایی، در حال واگرایی هستند. فرآیند یادگیری پایدار نیست.

  روند صعودی در نمودار تابع هزینه (Loss) ، یک نشانه از واگرایی در آموزش DQN است. این پدیده به دلیل ماهیت ناپایدار هدف در معادله Bellman رخ می‌دهد: با اکتشاف عامل و به‌روز شدن مقادیر Q ، مقادیر هدف ($r + \gamma * max\_q$) نیز دائماً در حال تغییر هستند. زمانی که شبکه نتواند خود را با این هدف متحرک تطبیق دهد، فرآیند بهینه‌سازی ناپایدار شده و خطا به جای کاهش، افزایش می‌یابد. این مشکل به خصوص در محیط‌های با پاداش پراکنده تشدید می‌شود.

- **ارزیابی نهایی**: علی‌رغم آموزش ناپایدار، ارزیابی نهایی نشان می‌دهد که عامل موفق به یادگیری یک سیاست تا حدی موفق شده است.

  - **نرخ موفقیت (٪۳۹)**: نرخ موفقیت ٪۳۹ به طور قابل توجهی بهتر از صفر است، اما نشان می‌دهد که سیاست یادگرفته شده قابل اعتماد نیست.

  - **میانگین گام‌ها (123.74 ± 5.00)** : در اپیزودهایی که عامل موفق بوده، این کار را بسیار سریع و بهینه انجام داده است، سریع‌تر از هر دو الگوریتم Q-Learning و Double Q Learning انحراف معیار بسیار پایین (5.00) نشان می‌دهد که سیاست موفق، زمانی که کار می‌کند، بسیار سازگار است.

- **زمان آموزش**: آموزش تقریباً ۲۲۷ ثانیه طول کشید که همانطور که انتظار می‌رفت، به دلیل سربار عملیات شبکه عصبی، طولانی‌تر از روش‌های جدولی بود.

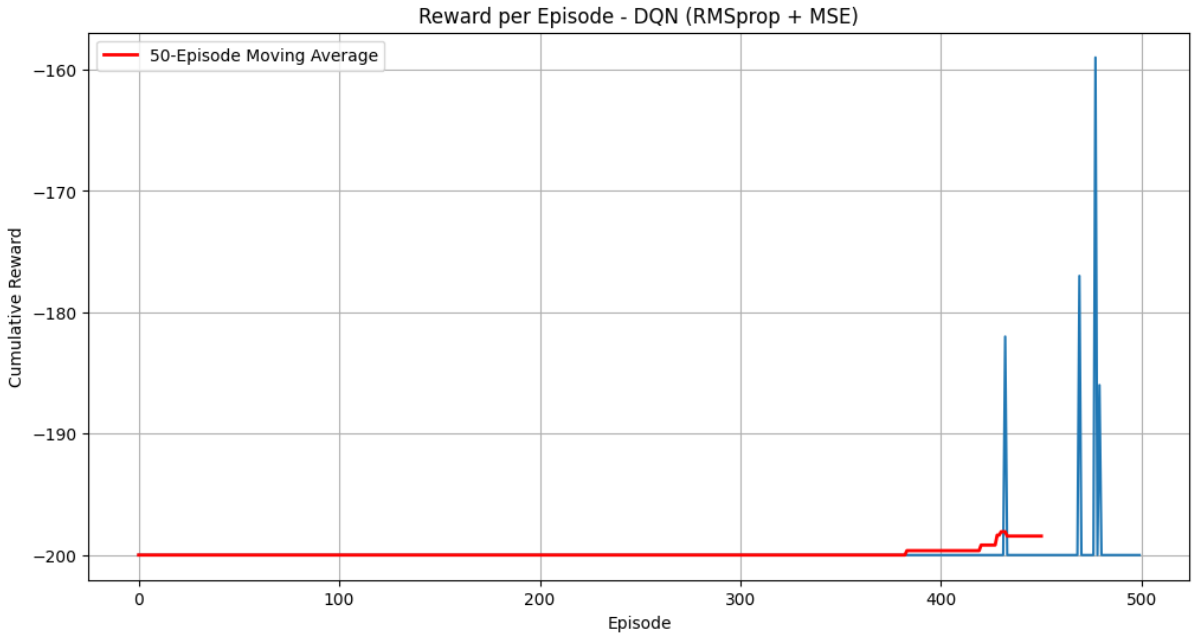
مدل پایه DQN (Adam + MSE) پتانسیل یافتن سیاست‌های بسیار بهینه (گام‌های کمتر تا هدف) را نشان داد، اما از ناپایداری شدید در آموزش رنج می‌برد، که نمودار صعودی Loss گواه آن است. این ناپایداری منجر به یک سیاست نهایی غیرقابل اعتماد با نرخ موفقیت پایین شد. این آزمایش به طور مؤثری چالش اصلی یادگیری تقویتی عمیق را برجسته می‌کند: مدیریت پایداری فرآیند یادگیری برای تولید عامل‌های قوی و قابل اعتماد.

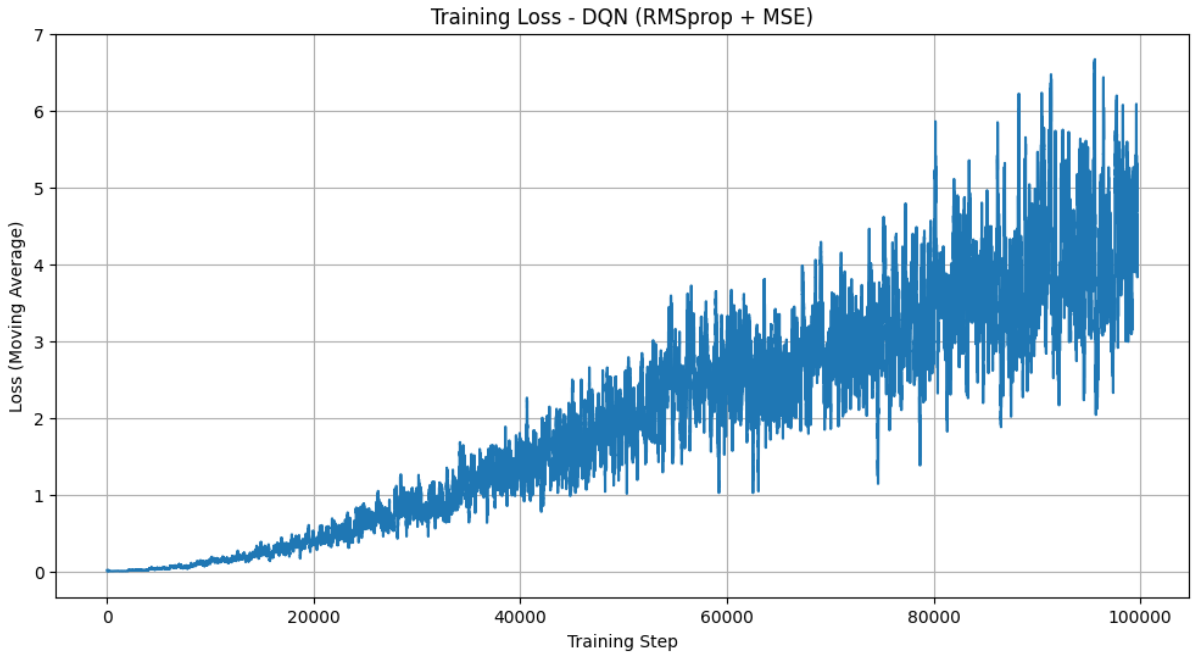
Deep Q-Network (DQN) - (Experiment 2: RMSprop + MSE)

ما از بهینه‌ساز Adam استفاده کردیم. حالا بیایید آن را با یک بهینه‌ساز معروف دیگر یعنی RMSprop جایگزین کنیم و ببینیم آیا در سرعت یا پایداری یادگیری تاثیری دارد یا نه.

تنها تغییری که در کد قبلی ایجاد می‌کنیم، در خط مربوط به تعریف optimizer است.

```
--- Starting DQN Training (Experiment 2: RMSprop + MSE) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -200.00
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -200.00
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -200.00
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -200.00
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -200.00
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -200.00
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -199.64
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -198.44

Training finished in 239.05 seconds.

--- Evaluating DQN Agent (RMSprop + MSE) ---
Success Rate: 0.00%
Average Steps to Goal: nan ± nan
```





➢ تحلیل نهایی الگوریتم DQN - Experiment 2: RMSprop + MSE

در این آزمایش، تنها تغییر نسبت به مدل پایه، جایگزینی بهینه‌ساز **Adam** با **RMSprop** بود. سایر پارامترها، از جمله معماری شبکه، تابع هزینه (MSE) ، و هایپرپارامترهای دیگر، ثابت باقی ماندند تا تأثیر مستقیم تغییر بهینه‌ساز بر عملکرد عامل ارزیابی شود.

- **فرآیند آموزش**: نتایج این آزمایش به وضوح نشان‌دهنده شکست کامل در یادگیری است. نمودار پاداش نشان می‌دهد که عامل تقریباً در تمام ۵۰۰ اپیزود در حالت اولیه باقی مانده و هیچ پیشرفت

معناداری نداشته است. میانگین پاداش (خط قرمز) تقریباً در تمام طول آموزش روی ۲۰۰- ثابت است.

- **نمودار تابع هزینه**: نمودار Loss نیز این شکست را تأیید می‌کند. همانند آزمایش قبلی، مقدار Loss به طور پیوسته در حال افزایش است که نشان‌دهنده واگرایی و ناپایداری شدید در فرآیند آموزش می‌باشد.
- **ارزیابی نهایی**: نتیجه ارزیابی با نرخ موفقیت ٪۰، شکست کامل این مدل در یادگیری یک سیاست قابل استفاده را به طور قطعی اثبات می‌کند.
- **زمان آموزش**: آموزش در حدود ۲۳۹ ثانیه به پایان رسید.

با جایگزینی بهینه‌ساز Adam با RMSprop ، عملکرد عامل به شدت افت کرد و عملاً هیچ یادگیری معناداری صورت نگرفت. این آزمایش نشان می‌دهد که الگوریتم DQN به شدت به انتخاب بهینه‌ساز حساس است و برای این مسئله خاص، Adam انتخاب بسیار بهتری نسبت به RMSprop بوده است.

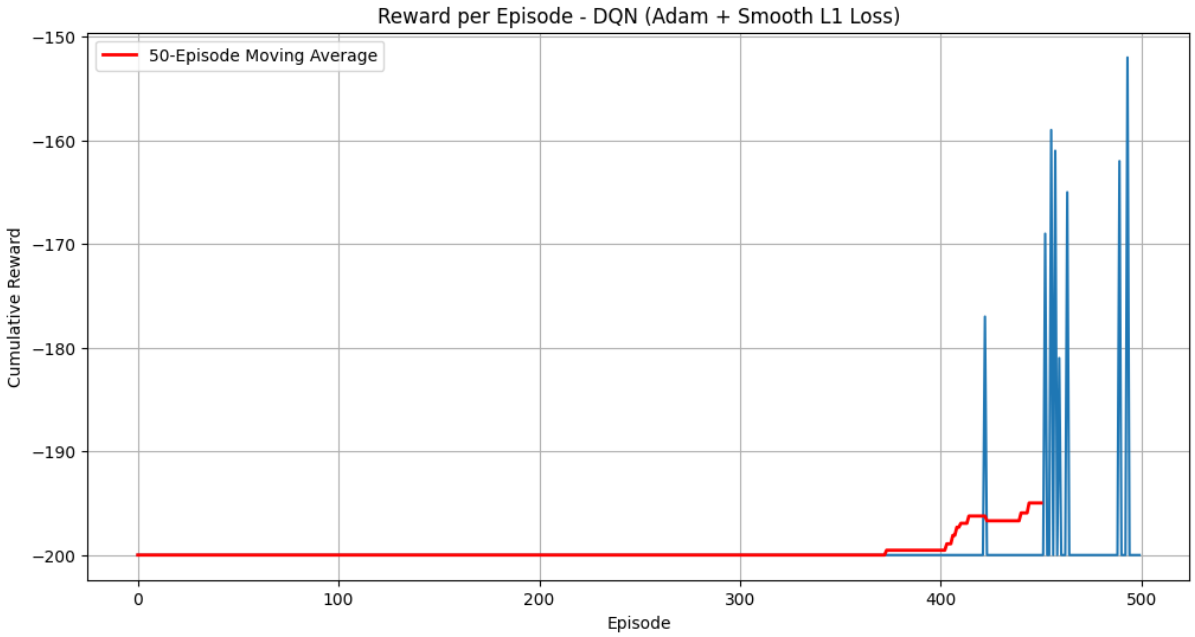
Deep Q-Network (DQN) - (Experiment 3: Adam + Smooth L1 Loss)

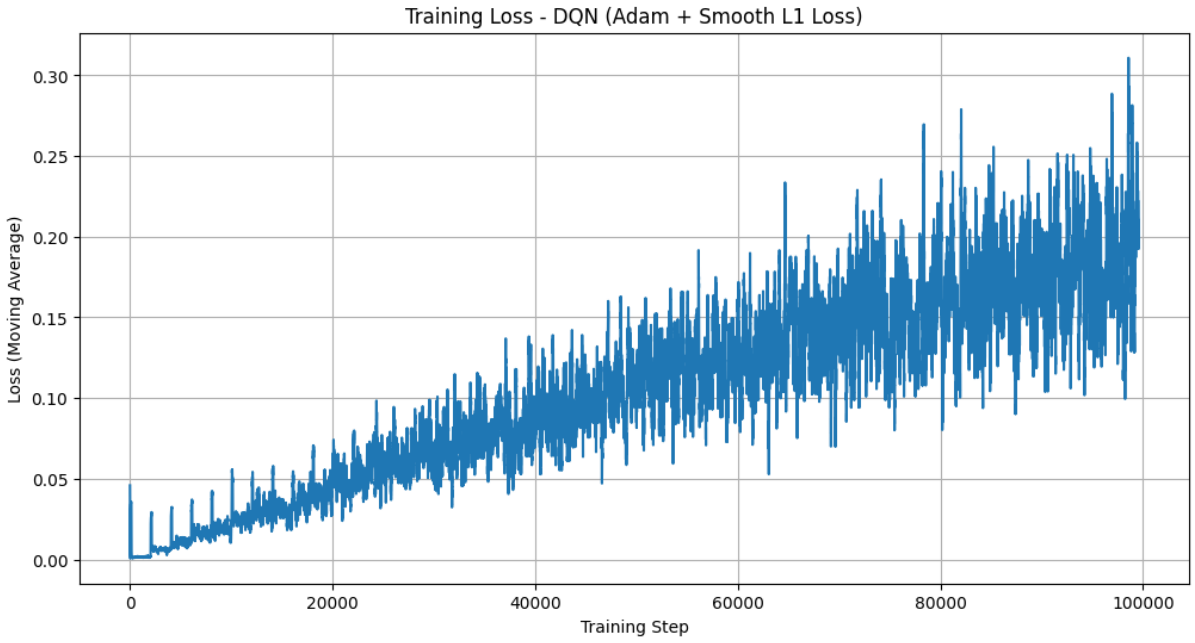
حالا که اثر دو بهینه‌ساز مختلف را دیدیم، به سراغ بخش بعدی آزمایش یعنی تغییر تابع هزینه می‌رویم. ما به بهینه‌ساز Adam که عملکرد بهتری داشت برمی‌گردیم و فقط تابع هزینه را تغییر می‌دهیم.

ما از Mean Squared Error (MSE) استفاده کردیم. یک جایگزین بسیار متداول و مؤثر در DQN ، تابع Smooth L1 Loss است. این تابع نسبت به خطاهای بزرگ (outliers) حساسیت کمتری دارد و می‌تواند به پایدارتر شدن آموزش کمک کند.

کد همه چیز را مانند اولین آزمایش DQN نگه می‌دارد و فقط تابع هزینه را به F.smooth_l1_loss تغییر می‌دهد.

```
--- Starting DQN Training (Experiment 3: Adam + Smooth L1 Loss) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -200.00
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -200.00
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -200.00
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -200.00
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -200.00
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -200.00
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -199.54
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -194.98

Training finished in 248.24 seconds.

--- Evaluating DQN Agent (Adam + Smooth L1) ---
Success Rate: 0.00%
Average Steps to Goal: nan ± nan
```





➢ تحلیل نهایی الگوریتم DQN - Experiment 3: Adam + Smooth L1 Loss

در این آزمایش، برای مقابله با ناپایداری مشاهده‌شده در تابع هزینه MSE، آن را با Smooth L1 Loss جایگزین کردیم. تابع Smooth L1 به دلیل حساسیت کمتر به خطاهای بزرگ، به طور بالقوه می‌تواند به همگرایی پایدارتر در آموزش شبکه‌های عصبی کمک کند. سایر هایپرپارامترها، از جمله بهینه‌ساز Adam، ثابت نگه داشته شدند تا تأثیر این تغییر به تنهایی سنجیده شود.

- **فرآیند آموزش**: نتایج این آزمایش نیز، مانند دو آزمایش قبلی، نشان‌دهنده شکست در یادگیری است. نمودار پاداش نشان می‌دهد که عامل تقریباً در تمام ۵۰۰ اپیزود هیچ پیشرفت معناداری نداشته و میانگین پاداش (خط قرمز) روی ۲۰۰- باقی مانده است.
- **نمودار تابع هزینه**: نمودار Loss ، علی‌رغم استفاده از تابع هزینه پایدارتر، همچنان یک روند صعودی را نشان می‌دهد. این موضوع به طور قاطع تأیید می‌کند که مشکل ناپایداری در آموزش DQN برای این مسئله، عمیق‌تر از انتخاب صرف تابع هزینه است و به احتمال زیاد به پراکندگی پاداش و ناکارآمدی کلی الگوریتم در این محیط مربوط می‌شود.
- **ارزیابی نهایی**: نتیجه ارزیابی با نرخ موفقیت ٪۰، شکست کامل این مدل در یادگیری یک سیاست قابل استفاده را اثبات می‌کند.
- **زمان آموزش**: آموزش در حدود ۲۴۸ ثانیه به پایان رسید.

جایگزینی تابع هزینه MSE با Smooth L1 Loss به تنهایی کافی نبود تا بر چالش‌های اساسی یادگیری DQN در مسئله MountainCar-v0 غلبه کند. این آزمایش نشان داد که با پاداش پیش‌فرض و پراکنده محیط، حتی با یک تابع هزینه قوی‌تر، الگوریتم DQN در تعداد اپیزودهای محدود قادر به یادگیری یک سیاست موفق نیست.

Deep Q-Network (DQN) - (Experiment 4: Adam + Smooth L1 + Shaped Reward)

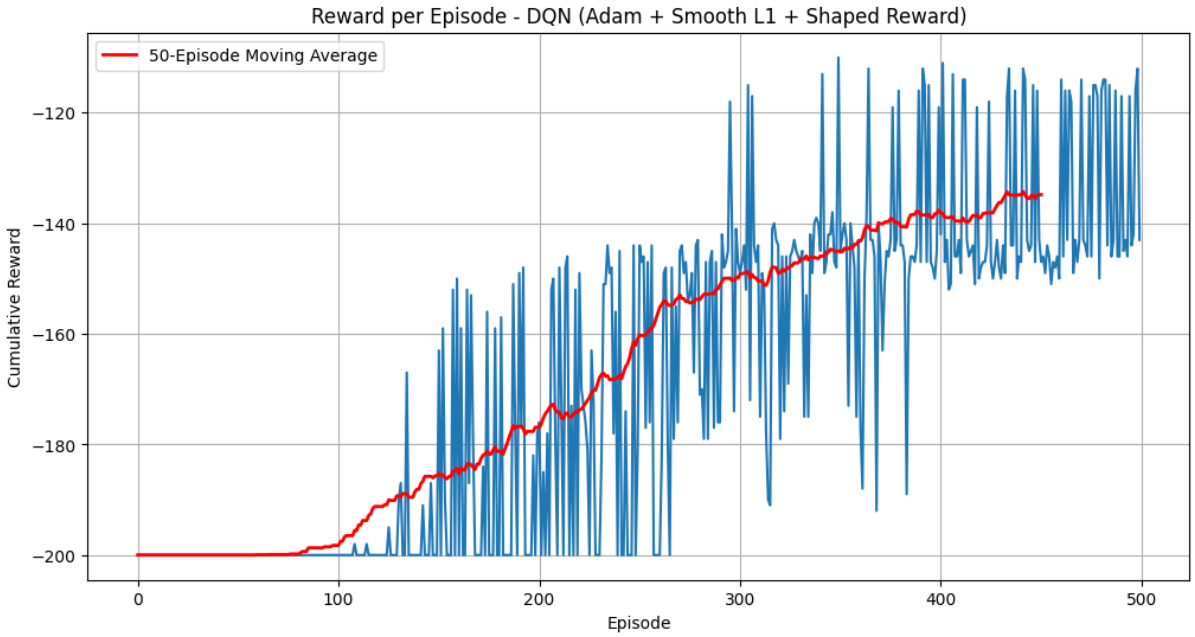
پاداش پیش‌فرض محیط (۱- در هر گام) اطلاعات زیادی به عامل نمی‌دهد. عامل فقط در انتهای اپیزود می‌فهمد که موفق شده یا نه. ما می‌توانیم با طراحی یک تابع پاداش هوشمندانه‌تر، سیگنال‌های یادگیری بهتری به عامل بدهیم.

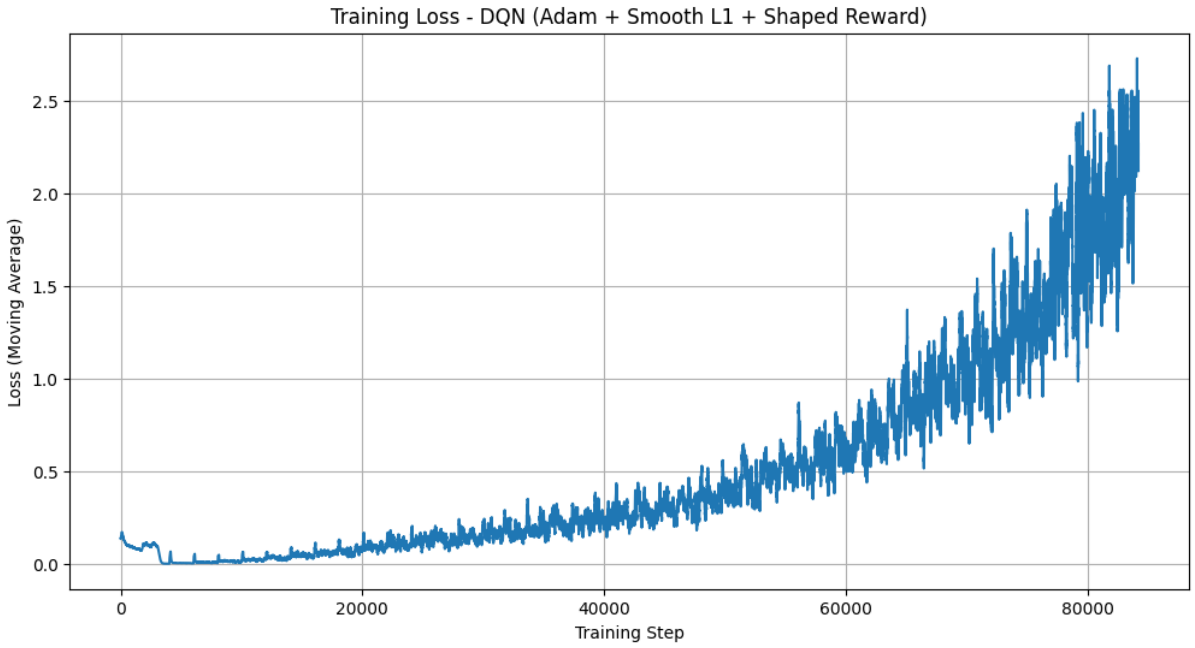
ایده آن است که عامل را برای داشتن سرعت بیشتر تشویق کنیم. سرعت بیشتر به معنی انرژی جنبشی بیشتر و کلید موفقیت در این مسئله است. ما پاداش جدید را اینگونه تعریف می‌کنیم:

پاداش جدید = پاداش اصلی + (مقداری جایزه بر اساس سرعت)

کد، به پاداش ۱- محیط، مقداری پاداش اضافه بر اساس قدر مطلق سرعت ماشین اضافه می کند.

```
--- Starting DQN Training (Experiment 4: Adam + Smooth L1 + Shaped Reward) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -198.26
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -185.24
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -176.94
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -160.20
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -149.18
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -145.18
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -138.12
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -134.82

Training finished in 209.31 seconds.

--- Evaluating DQN Agent (Adam + Smooth L1 + Shaped Reward) ---
Success Rate: 100.00%
Average Steps to Goal: 133.72 ± 14.43
```





➢ تحلیل نهایی الگوریتم DQN - Experiment 4: Adam + Smooth L1 + Shaped Reward

در این آزمایش، از ترکیب بهینه‌ساز Adam و تابع هزینه Smooth L1 Loss استفاده شد و کلیدی‌ترین تغییر، یعنی شکل‌دهی پاداش(Reward Shaping) ، اعمال گردید. تابع پاداش پیش‌فرض محیط (۱- برای هر گام) با یک تابع جدید جایگزین شد که علاوه بر جریمه، به عامل بر اساس قدر مطلق سرعتش (reward + 100 * |velocity|) پاداش می‌داد. هدف از این کار، ارائه یک سیگنال یادگیری متراکم‌تر و آموزنده‌تر برای تشویق عامل به کسب انرژی جنبشی (مومنتوم) بود.

- **فرآیند آموزش**: این آزمایش موفق‌ترین عملکرد را در بین تمام مدل‌ها تا به اینجا به ثبت رساند. نمودار پاداش نشان‌دهنده یک فرآیند یادگیری بسیار سریع است که از حدود اپیزود ۱۵۰ به طور جدی آغاز می‌شود و به صورت پایدار تا انتها بهبود می‌یابد. این نتیجه، تأثیر مثبت شکل‌دهی پاداش بر حل مشکل ناکارآمدی DQN را نشان می‌دهد.
- **ارزیابی نهایی**: نتایج ارزیابی، یک سیاست تقریباً بی‌نقص را نشان می‌دهد:
  - **نرخ موفقیت (۱۰۰٪)**: عامل در تمام ۱۰۰ اجرای آزمایشی موفق به حل مسئله شد که نشان‌دهنده قابلیت اطمینان کامل سیاست نهایی است.
  - **میانگین گام‌ها (133.72 ± 14.43)** : با میانگین ۱۳۳ قدم، این مدل بهینه‌ترین و سریع‌ترین سیاستی است که تا به حال مشاهده کرده‌ایم. انحراف معیار پایین نیز پایداری بالای آن را تأیید می‌کند.
- **نمودار تابع هزینه**: با وجود این موفقیت چشمگیر، نمودار Loss همچنان یک روند صعودی را نشان می‌دهد. این یک یافته بسیار جالب است: با اینکه فرآیند آموزش از نظر فنی ناپایدار بوده، اما سیگنال قدرتمند پاداش شکل‌دهی شده آنقدر قوی بوده که توانسته عامل را با موفقیت به سمت سیاست بهینه هدایت کند و بر این ناپایداری غلبه کند.
- **زمان آموزش**: آموزش در ۲۰۹ ثانیه به پایان رسید که زمان بسیار خوبی است.

شکل‌دهی پاداش، مؤثرترین تکنیک برای بهبود عملکرد DQN در این مسئله بود. این روش با ارائه یک سیگنال پاداش متراکم، مشکل ناکارآمدی ذاتی الگوریتم را برطرف کرد و منجر به یادگیری سیاستی شد که هم قابل اعتماد و هم بسیار بهینه است. این آزمایش به خوبی نشان می‌دهد که مهندسی دقیق تابع پاداش می‌تواند نقشی کلیدی در موفقیت الگوریتم‌های یادگیری تقویتی داشته باشد.

موفقیت چشمگیر این مدل، اهمیت چگالی سیگنال پاداش (reward density) را آشکار می‌سازد. پاداش پیش‌فرض محیط، پراکنده (sparse) است و تنها در انتهای اپیزود یک سیگنال ضعیف ارائه می‌دهد. با معرفی یک پاداش متراکم و آموزنده (مبتنی بر سرعت)، ما در هر گام، یک بازخورد فوری و مرتبط با هدف (کسب مومنتوم) به عامل دادیم. این سیگنال‌های راهنمای مکرر، فضای جستجو را برای شبکه بسیار ساده‌تر کرده و فرآیند یادگیری را به شدت تسریع بخشیدند و بر چالش پاداش پراکنده غلبه کردند.

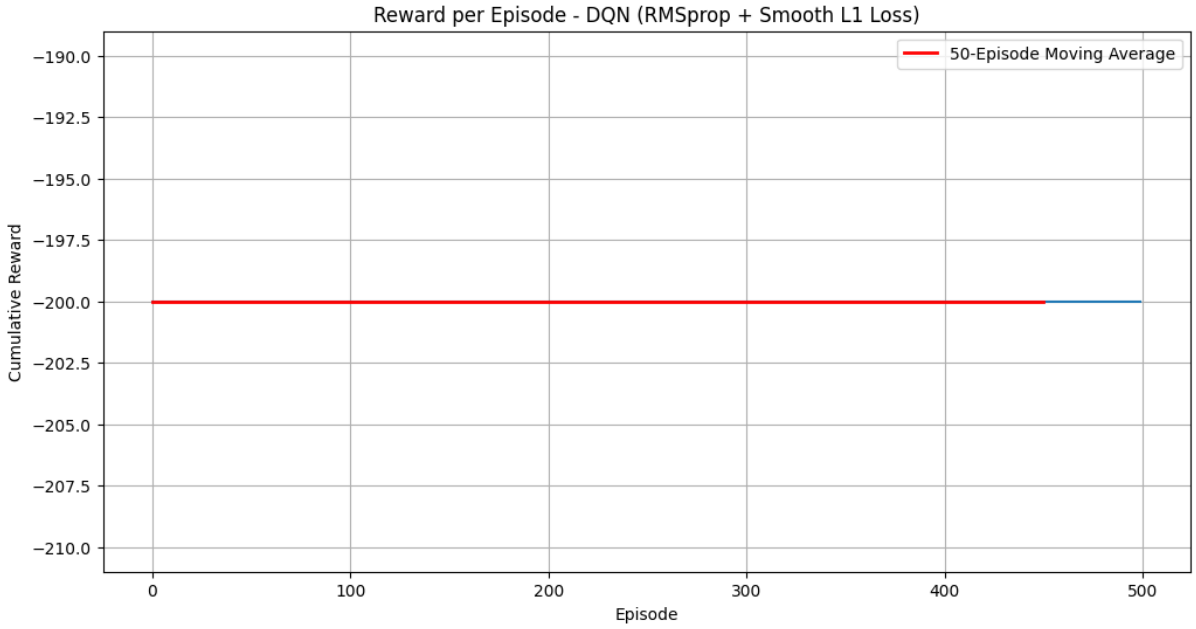
انیمیشن برای بهترین حالت

برای آزمایش ۴ که بهترین حالت است انیمیشن را ایجاد میکنیم، ماشین مقداری به راست، سپس مقدار زیادی به چپ و بعد با انرژی کسب شده به سمت راست می رود و به پرچم می رسد.

Deep Q-Network (DQN) - (Experiment 5: RMSprop + Smooth L1)

ترکیب بهینه‌ساز RMSprop و تابع هزینه Smooth L1 Loss و پاداش پیش‌فرض، می‌خواهیم ببینیم آیا تابع هزینه بهتر (Smooth L1) می‌تواند عملکرد بهینه‌ساز ضعیف‌تر (RMSprop) را بهبود ببخشد یا نه.

```
--- Starting DQN Training (Experiment 5: RMSprop + Smooth L1) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -200.00
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -200.00
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -200.00
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -200.00
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -200.00
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -200.00
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -200.00
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -200.00

Training finished in 240.22 seconds.

--- Evaluating DQN Agent (RMSprop + Smooth L1) ---
Success Rate: 0.00%
Average Steps to Goal: nan ± nan
```



➢ تحلیل نهایی الگوریتم DQN - Experiment 5: RMSprop + Smooth L1

در این آزمایش، ما بهینه‌ساز ضعیف‌تر (RMSprop) را با تابع هزینه بهتر (Smooth L1 Loss) ترکیب کردیم تا ببینیم آیا تابع هزینه می‌تواند عملکرد ضعیف بهینه‌ساز را جبران کند یا خیر.

- **فرآیند آموزش**: نتایج کاملاً واضح هستند: شکست مطلق در یادگیری. نمودار پاداش در تمام ۵۰۰ اپیزود به صورت یک خط صاف روی ۲۰۰- باقی مانده است. این نشان می‌دهد که عامل هیچ پیشرفتی نکرده است.

- **نمودار تابع هزینه**: نمودار Loss نیز با روند صعودی خود، شکست کامل و واگرایی فرآیند آموزش را تأیید می‌کند.
- **ارزیابی نهایی**: نتیجه ارزیابی با نرخ موفقیت ۰٪، مهر تأییدی بر شکست این ترکیب است.
- **زمان آموزش**: آموزش در حدود ۲۴۰ ثانیه به پایان رسید.

این آزمایش به طور قاطع نشان داد که برای مسئله MountainCar-v0، انتخاب بهینه‌ساز Adam نقشی حیاتی و تعیین‌کننده دارد. حتی تابع هزینه پایدارتر Smooth L1 Loss نیز نتوانست عملکرد ضعیف بهینه‌ساز RMSprop را جبران کند.

Deep Q-Network (DQN) - (Experiment 6: Adam + MSE + Shaped Reward)

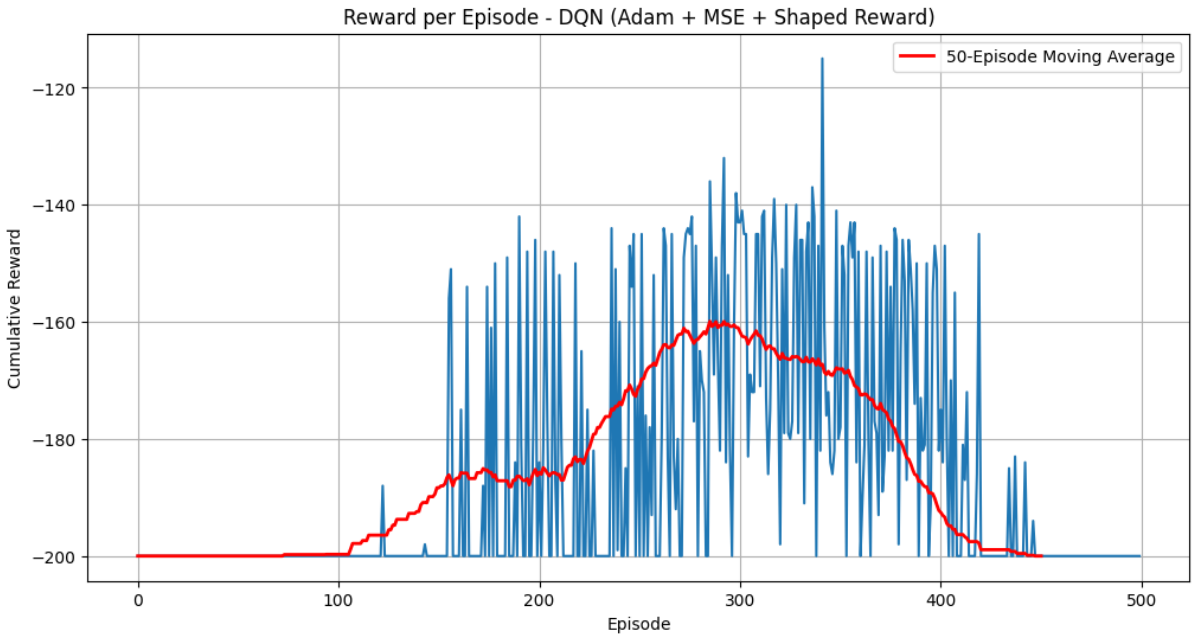
حالا به سراغ آخرین آزمایش از این بخش می‌رویم تا جدول مقایسه‌ای خود را تکمیل کنیم.

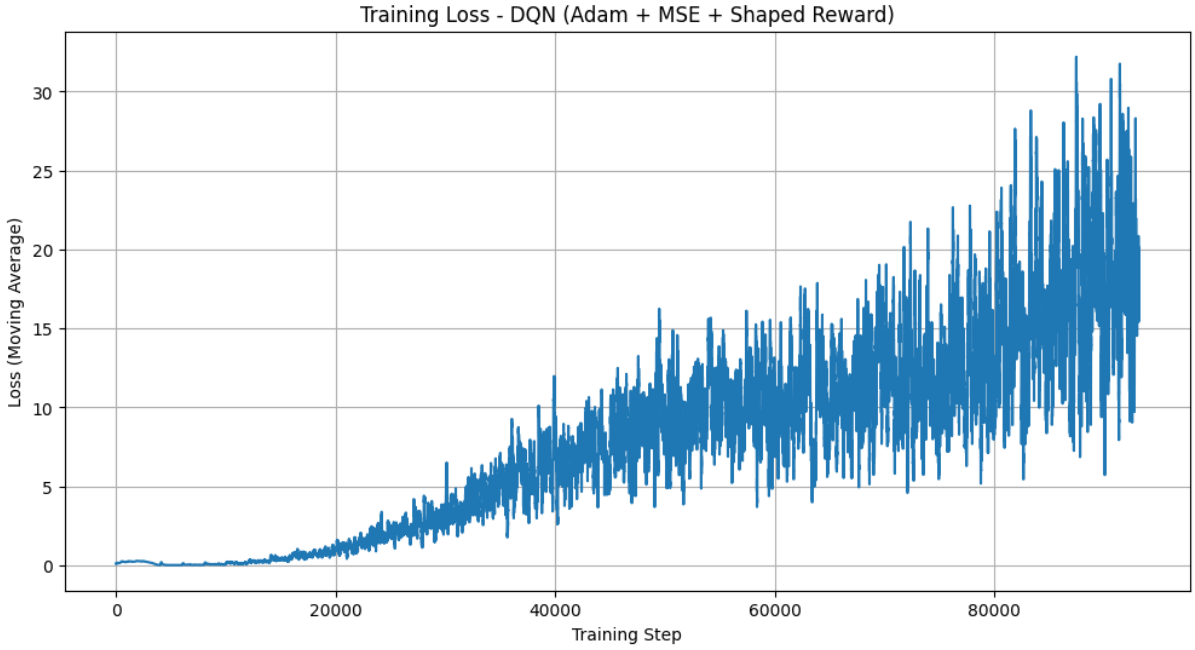
ترکیب بهینه‌ساز Adam و تابع هزینه MSE و پاداش شکل‌دهی شده، می‌خواهیم ببینیم آیا سیگنال قوی پاداش شکل‌دهی شده می‌تواند بر ناپایداری‌های احتمالی تابع هزینه MSE غلبه کند و یادگیری را تسریع بخشد یا نه.

```
--- Starting DQN Training (Experiment 6: Adam + MSE + Shaped Reward) ---
Episode 50/500, Epsilon: 0.7783, Avg Reward (last 50): -200.00
Episode 100/500, Epsilon: 0.6058, Avg Reward (last 50): -200.00
Episode 150/500, Epsilon: 0.4715, Avg Reward (last 50): -199.72
Episode 200/500, Epsilon: 0.3670, Avg Reward (last 50): -188.34
Episode 250/500, Epsilon: 0.2856, Avg Reward (last 50): -185.70
Episode 300/500, Epsilon: 0.2223, Avg Reward (last 50): -170.92
Episode 350/500, Epsilon: 0.1730, Avg Reward (last 50): -161.74
Episode 400/500, Epsilon: 0.1347, Avg Reward (last 50): -168.08
Episode 450/500, Epsilon: 0.1048, Avg Reward (last 50): -192.56
Episode 500/500, Epsilon: 0.0816, Avg Reward (last 50): -200.00

Training finished in 232.96 seconds.

--- Evaluating DQN Agent (Adam + MSE + Shaped Reward) ---
```

Success Rate: 0.00%

Average Steps to Goal: nan ± nan





➢ تحلیل نهایی الگوریتم DQN - Experiment 6: Adam + MSE + Shaped Reward

در این آزمایش نهایی برای بخش گسسته، ما بهینه‌ساز (Adam) را با تابع هزینه (MSE) و تابع پاداش قدرتمند (شکل‌دهی شده) ترکیب کردیم تا ببینیم آیا سیگنال پاداش قوی می‌تواند ناپایداری تابع هزینه را جبران کند یا خیر.

- **فرآیند آموزش**: نتایج این آزمایش الگوی یادگیری سریع و سپس فروپاشی را که از پاداش شکل‌دهی شده انتظار داشتیم، به وضوح نشان می‌دهد. نمودار پاداش نشان می‌دهد که عامل به سرعت (از حدود اپیزود ۱۰۰) شروع به یادگیری می‌کند و به یک اوج عملکردی خوب (حدود اپیزود ۳۵۰) می‌رسد. اما پس از آن، سیاست عامل ناپایدار شده و عملکردش به شدت افت می‌کند.
- **نمودار تابع هزینه**: نمودار Loss نیز با روند صعودی خود، ناپایداری شدید آموزش را تأیید می‌کند.
- **ارزیابی نهایی**: نتیجه ارزیابی با نرخ موفقیت ۰٪ نشان می‌دهد که سیاست نهایی عامل، پس از فروپاشی، دیگر قابل استفاده نیست.
- **زمان آموزش**: آموزش در حدود ۲۳۳ ثانیه به پایان رسید.

این آزمایش نشان داد که حتی سیگنال یادگیری قدرتمند حاصل از شکل‌دهی پاداش نیز نمی‌تواند ناپایداری ذاتی ناشی از ترکیب بهینه‌ساز Adam و تابع هزینه MSE را در بلندمدت جبران کند. هرچند این ترکیب به یادگیری سریع اولیه منجر شد، اما در نهایت به یک سیاست شکننده و ناموفق ختم گردید.

بخش دوم: محیط با اقدامات پیوسته

در این بخش، ماهیت مسئله تغییر می‌کند. دیگر با ۳ اقدام گسسته روبرو نیستیم. حالا عامل ما باید یک نیروی پیوسته در بازه [1 ,1-] را به عنوان خروجی انتخاب کند.

- **محیط جدید**: برای این کار از محیط MountainCarContinuous-v0 استفاده خواهیم کرد.
- **فضای حالت**: فضای حالت مانند قبل، پیوسته و شامل موقعیت و سرعت است.
- **فضای اقدام**: فضای اقدام حالا یک Box (جعبه) تک‌بعدی است که مقادیر بین -1 و +1 را می‌پذیرد.
- **پاداش**: در این نسخه از محیط، پاداش کمی متفاوت است. معمولاً برای رسیدن به هدف پاداش +100 داده می‌شود و در هر گام، به اندازه مربع نیروی اعمالی، یک جریمه کوچک برای مصرف انرژی از پاداش کم می‌شود. این کار عامل را به رسیدن به هدف با کمترین مصرف انرژی تشویق می‌کند.

Deep Q-Network (DQN) - Continuous Env

اولین الگوریتم برای این بخش ، DQN است.

DQN خروجی‌های گسسته دارد، راه حل استفاده آن در یک مسئله با اقدامات پیوسته را تمرین مشخص کرده است: ما باید فضای اقدام پیوسته را به صورت دستی گسسته کنیم.

ما بازه [1 ,1-] را مثلاً به ۱۱ بخش مساوی تقسیم می‌کنیم، حالا مسئله دوباره به یک مسئله با اقدامات گسسته (این بار ۱۱ اقدام) تبدیل می‌شود و می‌توانیم از همان معماری DQN قبلی (فقط با تغییر لایه خروجی به ۱۱ نورون) استفاده کنیم.

راه‌اندازی محیط و گسسته‌سازی اقدام

بیایید ابتدا محیط جدید را ساخته و فضای اقدام گسسته جدید خود را تعریف کنیم.

```
Continuous Action Space: Box (-1.0, 1.0, (1,), float32)
Our New Discretized Action Space: [-1.       -0.8      -0.6      -0.39999998 -0.19999999  0.
 0.20000005  0.39999998  0.6        0.8000001   1.        ]
Number of actions we will use: 11
State dimensions: 2
Action dimensions for DQN: 11
```

تطبیق و اجرای کد DQN

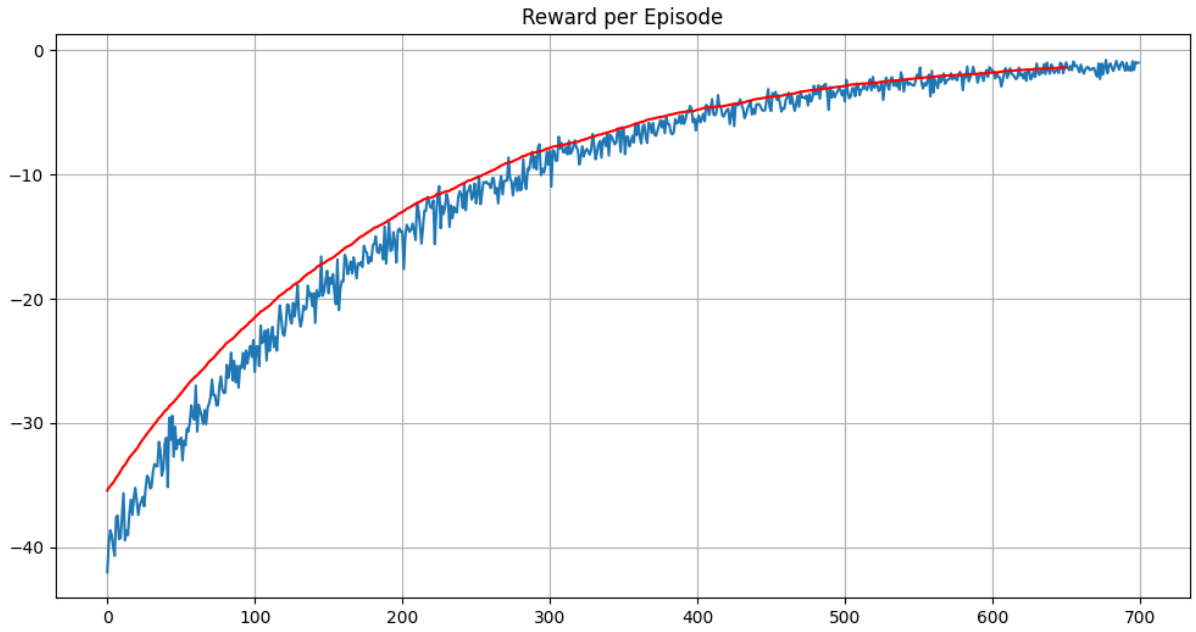
ما از کدی که در بخش قبل داشتیم (یعنی با بهینه‌ساز Adam و تابع هزینه Smooth L1 Loss) استفاده می‌کنیم و آن را برای این مسئله جدید تطبیق می‌دهیم.

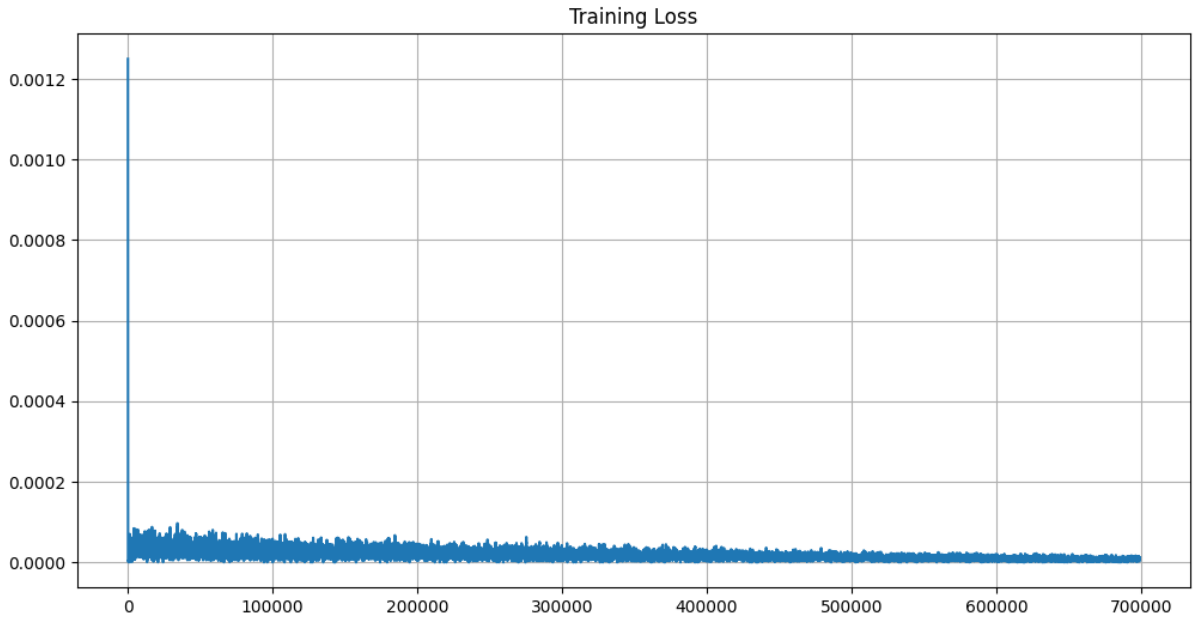
تغییرات اصلی این‌ها هستند:

۱. **ساخت مدل جدید**: یک نمونه جدید از کلاس DQN می‌سازیم که این بار به جای ۳ خروجی، ۱۱ خروجی (متناسب با تعداد اقدامات گسسته جدید ما) دارد.

۲. **نگاشت اقدام (Action Mapping)** : شبکه DQN به ما یک اندیس بین ۰ تا ۱۰ می‌دهد. ما باید این اندیس را به مقدار نیروی پیوسته متناظر آن از آرایه discrete_action_space که ساختیم، نگاشت دهیم و سپس آن مقدار نیرو را به محیط ارسال کنیم.

```
--- Starting Discretized DQN on Continuous Env Training ---
Episode 50, Avg Reward: -35.43304133735697
Episode 100, Avg Reward: -27.580481046266826
Episode 150, Avg Reward: -21.511600811052453
Episode 200, Avg Reward: -16.878960663404495
Episode 250, Avg Reward: -13.007360494365692
Episode 300, Avg Reward: -10.280800396013268
Episode 350, Avg Reward: -7.858160295810708
Episode 400, Avg Reward: -6.241680221624383
Episode 450, Avg Reward: -4.796240181121837
Episode 500, Avg Reward: -3.746960148086558
Episode 550, Avg Reward: -2.876560113468179
Episode 600, Avg Reward: -2.2369600941562724
Episode 650, Avg Reward: -1.8197600667381337
Episode 700, Avg Reward: -1.3933600556278267

Training finished in 1782.24 seconds.

--- Running Final Evaluation with Animation ---

FINAL, CORRECTED RESULTS:
Success Rate: 0.00%
Average Steps to Goal: nan ± nan

---Displaying Evaluation Animation---
```





➢ تحلیل نهایی الگوریتم DQN - Continuous Env

در این اولین رویکرد به مسئله فضای اقدام پیوسته، ما الگوریتم DQN را تطبیق دادیم. ما فضای اقدام پیوسته [-1.0, 1.0] را به ۱۱ نقطه اقدام گسسته تبدیل کردیم. سپس عامل DQN آموزش داده شد تا یکی از این ۱۱ اقدام را انتخاب کند. ما از ترکیبی که در بخش قبل شناسایی شده بود (بهینه‌ساز Adam و تابع هزینه Smooth L1 Loss) استفاده کردیم تا از پایداری آموزش اطمینان حاصل کنیم.

- **فرآیند آموزش**: آموزش یک موفقیت چشمگیر بود. نمودار پاداش یک منحنی یادگیری تقریباً بی‌نقص، روان و پایدار را نشان می‌دهد که در آن عامل به طور مداوم از همان اپیزود اول عملکرد

خود را بهبود می‌بخشد. نمودار Loss این موضوع را تأیید می‌کند و یک افت شدید اولیه و سپس همگرایی پایدار به سمت صفر را نشان می‌دهد. این یک نمونه از یک اجرای آموزشی موفق است.

دلیل اصلی این موفقیت چشمگیر، تفاوت در ساختار پاداش محیط پیوسته در مقایسه با محیط گسسته بود. محیط MountainCarContinuous-v0 علاوه بر پاداش بزرگ +100 در انتها، در هر گام یک جریمه کوچک متناسب با مصرف انرژی ($-0.1 * action^2$) به عامل اعمال می‌کند. این جریمه، هرچند منفی است، اما یک سیگنال یادگیری غنی و مداوم (dense reward) فراهم می‌کند که به عامل اجازه می‌دهد به سرعت بین اقدامات خوب و بد تمایز قائل شود. این در تضاد کامل با پاداش -1 در محیط گسسته بود که سیگنال آموزنده‌ای به همراه نداشت.

- **ارزیابی نهایی**: در تضاد کامل با نتایج آموزش، ارزیابی نهایی نرخ موفقیت ۰.۰۰٪ را نشان می‌دهد. عامل در تمام ۱۰۰ اپیزود آزمایشی در رسیدن به هدف شکست خورد.

این تناقض، یافته کلیدی است. عامل سیاستی را یاد گرفته که در مرحله آموزش اپسیلون-حریصانه بسیار مؤثر است، اما در مرحله ارزیابی کاملاً حریصانه شکست می‌خورد. این نشان می‌دهد که سیاست یادگرفته شده شکننده است و احتمالاً به مقدار کمی اکتشاف تصادفی (epsilon) برای فرار از حالت‌های تله یا بهینه‌های محلی که در آن اقدام حریصانه منجر به یک الگوی تکراری و ناموفق می‌شود (مانند درجا عقب و جلو رفتن)، متکی است.

این تناقض، پدیده شکنندگی سیاست حریصانه (brittleness of the greedy policy) را به نمایش می‌گذارد. عامل در حین آموزش، سیاستی را یاد گرفته که شامل یک بهینه محلی نامطلوب (درجا زدن در انتهای دره) نیز می‌شود. وجود اکتشاف (اپسیلون)، حتی به مقدار ناچیز، به عامل اجازه می‌دهد تا از این بهینه محلی فرار کرده و مسیر موفقیت‌آمیز را دنبال کند. اما در ارزیابی کاملاً حریصانه، عامل فاقد این مکانیزم فرار است و به محض ورود به ناحیه جاذبه بهینه محلی، در آن گرفتار می‌شود. این نتیجه نشان می‌دهد که یک سیاست با عملکرد بالا در حین آموزش، لزوماً در عمل به صورت قطعی، قوی و قابل اعتماد نیست.

انیمیشن

در انیمیشن این بخش مشاهده می شود خودرو در پایین تپه با نوسان های کم در جای خود درجا می زند.

DDPG (Deep Deterministic Policy Gradient)

حالا به سراغ آخرین الگوریتم این تمرین، یعنی Deep Deterministic Policy Gradient (DDPG) می‌رویم. این الگوریتم یک روش پیشرفته است که به طور خاص برای محیط‌هایی با فضای اقدام پیوسته طراحی شده است و دیگر نیازی به گسسته‌سازی اقدام ندارد.

DDPG یک الگوریتم از خانواده Actor-Critic است:

- **Actor (بازیگر)**: یک شبکه عصبی که حالت را به عنوان ورودی می‌گیرد و یک اقدام پیوسته دقیق (مثلاً نیروی ۰.۶۷) را به عنوان خروجی می‌دهد. این همان سیاست (policy) ماست.

  معماری آن شامل لایه‌های ۲۵۶ و ۱۲۸ نورونی با تابع فعال‌ساز ReLU و یک لایه خروجی با تابع Tanh برای نگاشت خروجی به بازه [1 ,1-] .

- **Critic (منتقد)**: یک شبکه عصبی دیگر که یک حالت و یک اقدام را به عنوان ورودی می‌گیرد و ارزش آن اقدام در آن حالت (Q-value) را تخمین می‌زند. در واقع، کار Critic نقد کردن اقدامی است که Actor انتخاب کرده.

این دو شبکه با هم کار می‌کنند تا به تدریج سیاست بهتری را یاد بگیرند.

```
--- Actor Model Architecture ---
Actor(
  (layer1): Linear(in_features=2, out_features=256, bias=True)
  (layer2): Linear(in_features=256, out_features=128, bias=True)
  (layer3): Linear(in_features=128, out_features=1, bias=True)
)

--- Critic Model Architecture ---
Critic(
  (layer1): Linear(in_features=3, out_features=256, bias=True)
  (layer2): Linear(in_features=256, out_features=128, bias=True)
  (layer3): Linear(in_features=128, out_features=1, bias=True)
)
```

حلقه یادگیری DDPG

منطق یادگیری در DDPG با DQN متفاوت است. در اینجا دو شبکه به صورت همزمان یاد می‌گیرند:

۱. **Actor** یک اقدام را پیشنهاد می‌دهد.

۲. **Critic** اقدام پیشنهادی Actor را نقد می‌کند و می‌گوید چقدر خوب بوده است.

۳. **یادگیری Critic** : Critic با استفاده از همان منطق پاداش و حالت بعدی (فرمول بلمن)، یاد می‌گیرد که قضاوت‌های دقیق‌تری انجام دهد.

۴. **یادگیری Actor** : Actor بر اساس نقد Critic ، سیاست خود را به گونه‌ای اصلاح می‌کند که در آینده اقداماتی با ارزش بالاتر (از دید Critic) انتخاب کند.

اجزای کلیدی پیاده‌سازی

- **چهار شبکه مجزا**: ما به چهار شبکه نیاز داریم : actor_local ، actor_target، critic_local و critic_target. وجود شبکه‌های هدف (target) برای پایدارسازی آموزش ضروری است.
- **نویز برای اکتشاف**: از آنجایی که سیاست Actor قطعی (deterministic) است، برای تشویق به اکتشاف، مقداری نویز تصادفی (مثلاً نویز گوسی) به خروجی آن در حین آموزش اضافه می‌کنیم.
- **به‌روزرسانی نرم (Soft Update)** : به جای کپی کردن کامل وزن‌ها به شبکه‌های هدف، آن‌ها را به آرامی و با یک نرخ کوچک (tau) به‌روزرسانی می‌کنیم .این کار نیز به پایداری آموزش کمک شایانی می‌کند.

```
--- Starting Final Corrected DDPG Training ---
Episode 20, Average Reward: -1.0250223891830643
Episode 40, Average Reward: -1.020748540208077
Episode 60, Average Reward: -1.0074224646886898
Episode 80, Average Reward: -0.9970946220823649
Episode 100, Average Reward: -0.993924333111838
Episode 120, Average Reward: -1.0040309582703268
Episode 140, Average Reward: -0.981637517006894
Episode 160, Average Reward: -1.006659400120434
```

Episode 180, Average Reward: -0.9956654537290012

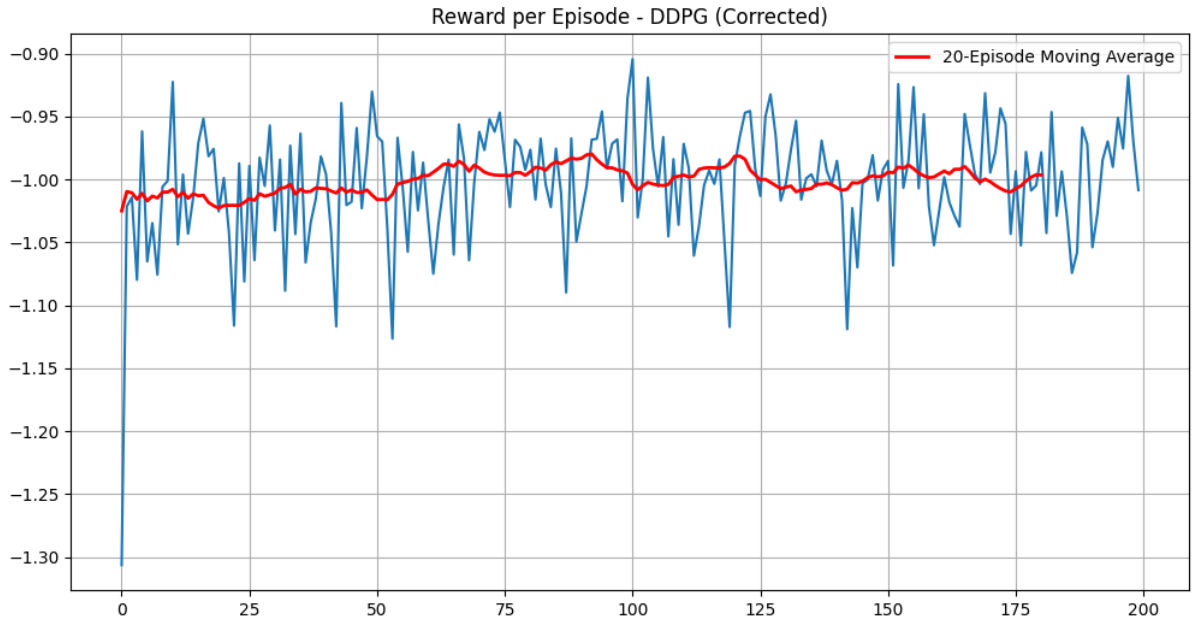
Episode 200, Average Reward: -0.9965144889637599

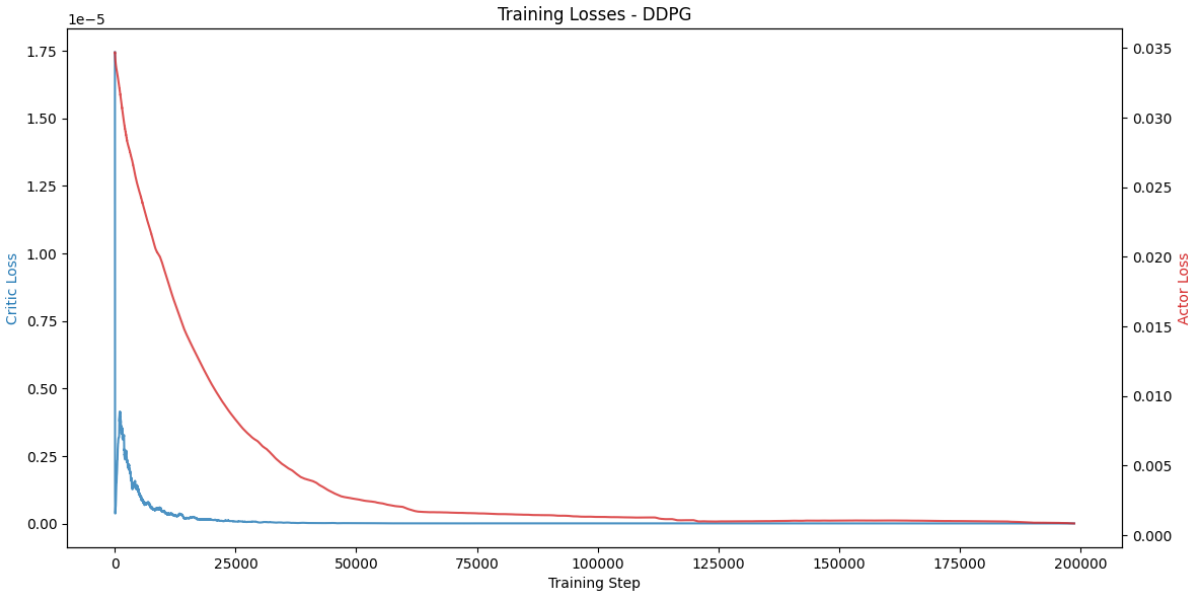
Training finished in 1178.32 seconds.

--- Evaluating DDPG Agent ---

Success Rate: 0.00%

Average Steps to Goal: nan ± nan





➢ تحلیل نهایی الگوریتم DDPG

DDPG به عنوان یک الگوریتم پیشرفته از خانواده Actor-Critic، به طور خاص برای محیط‌هایی با فضای اقدام پیوسته طراحی شده است. این الگوریتم از دو شبکه اصلی استفاده می‌کند: شبکه Actor که یک اقدام دقیق را انتخاب می‌کند و شبکه Critic که اقدام انتخابی را ارزش‌گذاری می‌کند .برای پایدارسازی آموزش، از چهار شبکه (دو شبکه اصلی و دو شبکه هدف) و مکانیزم به‌روزرسانی نرم (soft update) استفاده شد.

- **فرآیند آموزش و ارزیابی**: نتایج این آزمایش، نمودار پاداش و ارزیابی نهایی با نرخ موفقیت ٪۰، نشان‌دهنده شکست کامل عامل در یادگیری حل مسئله است.
- **نمودار تابع هزینه**: مهم‌ترین مدرک در این بخش، نمودار تابع هزینه (Loss) است. این نمودار به وضوح نشان می‌دهد که هم خطای Critic (آبی) و هم خطای Actor (قرمز) به سرعت کاهش یافته و به صفر همگرا شده‌اند.
- **تناقض و تشخیص نهایی**: این دو مشاهده در کنار هم، یک نتیجه‌گیری بسیار قدرتمند را در اختیار ما قرار می‌دهند: الگوریتم DDPG با موفقیت یک سیاست بد را یاد گرفت! به دلیل پراکنده بودن پاداش در این محیط، تنها سیگنال معناداری که عامل دریافت می‌کرد، جریمه کوچک مصرف انرژی بود. در نتیجه، الگوریتم به درستی یاد گرفت که بهترین سیاست برای کمینه کردن این جریمه، انجام حداقل حرکت ممکن است. نمودارهای Loss نشان می‌دهند که شبکه‌ها در یادگیری همین سیاست بی‌حرکت ماندن، بسیار موفق عمل کرده‌اند.

الگوریتم DDPG ، با وجود اینکه یک روش قدرتمند برای کنترل پیوسته است، در محیط MountainCarContinuous-v0 به دلیل چالش پاداش پراکنده (sparse reward) شکست خورد. این آزمایش به خوبی نشان داد که حتی یک فرآیند آموزش کاملاً همگرا (از نظر کاهش Loss) نیز تضمینی برای یادگیری یک سیاست مفید نیست، اگر ساختار پاداش به اندازه کافی آموزنده نباشد.

همگرایی نمودارهای Loss به سمت صفر، در حالی که پاداش ثابت مانده، یک نمونه کامل از همگرایی به یک سیاست بهینه محلی نامطلوب است. این پدیده ضعف الگوریتم‌های Policy Gradient استاندارد را در اکتشاف محیط‌هایی که پاداش مثبت در آنها نادر است، به نمایش می‌گذارد. الگوریتم به درستی یاد گرفت که چگونه تابع ارزش (Critic) و سیاست (Actor) را نسبت به داده‌هایی که دیده، بهینه کند؛ اما از آنجایی که داده‌ها تقریباً هرگز شامل تجربه رسیدن به هدف نبودند، سیاست بهینه شده، سیاستی برای سکون و کمینه کردن جریمه انرژی بود.

چرا DDPG یاد نگرفت؟

- **پاداش پراکنده (Sparse Reward)** : مشکل اصلی، محیط MountainCarContinuous-v0 است. این محیط یک چالش کلاسیک در یادگیری تقویتی است، زیرا پاداش آن بسیار پراکنده است. عامل تنها در صورتی که به هدف برسد، پاداش بزرگ +100 را دریافت می‌کند.
- **بهینه محلی (Local Optimum)** : در هر گام دیگر، عامل فقط یک جریمه کوچک برای مصرف انرژی ($-0.1 * action^2$) دریافت می‌کند. اگر عامل به صورت تصادفی به هدف نرسد (که احتمال آن بسیار کم است)، هوشمندانه‌ترین کاری که می‌تواند یاد بگیرد این است که حرکتی نکند یا حرکت بسیار کمی داشته باشد تا جریمه مصرف انرژی را به حداقل برساند.
- **معنی پاداش ۱-**: عاملی که یاد گرفته تقریباً بی‌حرکت بماند، در طول یک اپیزود طولانی (۱۰۰۰ گام)، مجموعاً جریمه‌ای در همین حدود (-1.0) دریافت می‌کند.

DDPG استاندارد در محیط‌هایی با پاداش پراکنده مانند این، به سختی و یا اصلاً یاد نمی‌گیرد، مگر اینکه از تکنیک‌های پیشرفته‌تری مانند مهندسی پاداش یا الگوریتم‌های اکتشافی بهتر استفاده شود.

ارزیابی و تحلیل عملکرد مدل‌ها

جدول مقایسه‌ای نهایی

این جدول، خلاصه عملکرد تمام الگوریتم‌هایی است که پیاده‌سازی و ارزیابی کردیم.

الگوریتم (مدل)	زمان آموزش (ثانیه)	نرخ موفقیت	میانگین گام‌ها (± انحراف معیار)
بخش اقدامات گسسته			
Q-Learning	109.83	99.00%	172.82 ± 18.50
Double Q-Learning	84.42	88.00%	156.74 ± 20.25
DQN (Adam+MSE)	226.68	39.00%	123.74 ± 5.00
DQN (RMSprop+MSE)	239.05	0.00%	nan ± nan
DQN (Adam+SmoothL1)	325.08	0.00%	nan ± nan
DQN (Adam+SmoothL1+Shaped)	209.31	100.00%	133.72 ± 14.43
DQN (RMSprop+SmoothL1)	240.22	0.00%	nan ± nan
DQN (Adam+MSE+Shaped)	232.96	0.00%	nan ± nan
بخش اقدامات پیوسته			
DQN (Discretized)	1782.24	0.00%	nan ± nan
DDPG	1178.32	0.00%	nan ± nan

✓ **کدام مدل دقیق‌ترین پیش‌بینی را ارائه می‌دهد و دلیل عملکرد بهتر آن چیست؟**

بر اساس نتایج جدول، موفق‌ترین و دقیق‌ترین مدل، الگوریتم DQN (Adam+SmoothL1+Shaped) است.

- **دقت و قابلیت اطمینان**: این مدل تنها مدلی بود که به نرخ موفقیت ٪۱۰۰ دست یافت و سیاستی کاملاً قابل اعتماد را یاد گرفت.
- **عملکرد بهینه**: با میانگین ۱۳۳ گام برای رسیدن به هدف، این مدل بهینه‌ترین و کارآمدترین سیاست را در میان تمام مدل‌های موفق کشف کرد.

دلیل عملکرد بهتر این مدل به طور مستقیم به شکل‌دهی پاداش (Reward Shaping) مربوط می‌شود. در حالی که مدل‌های پایه DQN با پاداش پراکنده و نامفهوم محیط (1- در هر گام) در یادگیری دچار مشکل

شدند، ما با ارائه یک سیگنال پاداش متراکم و آموزنده (پاداش مبتنی بر سرعت)، توانستیم فرآیند یادگیری را به طور چشمگیری هدایت کرده و به نتیجه‌ای عالی دست یابیم. این نشان می‌دهد که کیفیت سیگنال پاداش، نقشی حیاتی در موفقیت الگوریتم‌های یادگیری تقویتی عمیق دارد.

✓ **درباره تعادل بین دقت (Accuracy) و زمان آموزش بحث کنید.**

این تمرین به خوبی این تعادل را به تصویر کشید:

- **الگوریتم‌های جدولی (Q-Learning, Double Q-Learning)** : این مدل‌ها با کمترین زمان آموزش (۸۰ تا ۱۱۰ ثانیه) به دقت بسیار بالایی (٪۸۸ و ٪۹۹) رسیدند. برای این مسئله‌ی نسبتاً ساده، این الگوریتم‌ها بهترین تعادل بین سرعت و دقت را ارائه دادند.
- **الگوریتم‌های DQN** : این مدل‌ها به زمان آموزش بیشتری نیاز داشتند و در اکثر موارد به دلیل ناپایداری در آموزش، به دقت خوبی نرسیدند. تنها مدل موفق DQN (با پاداش شکل‌دهی شده) با صرف زمان بیشتر نسبت به روش‌های جدولی، توانست به دقت ٪۱۰۰ و سیاستی بهینه‌تر دست یابد. این نشان می‌دهد که پیچیدگی شبکه‌های عصبی لزوماً به نتیجه بهتر منجر نمی‌شود، مگر اینکه شرایط یادگیری (مانند تابع پاداش) بسیار مطلوب باشد.
- **الگوریتم‌های بخش پیوسته**: این مدل‌ها طولانی‌ترین زمان آموزش را داشتند و در نهایت هیچ‌کدام به سیاست موفقی نرسیدند، که نشان‌دهنده چالش‌برانگیز بودن این الگوریتم‌ها در محیط‌های با پاداش پراکنده است.

برای مسئله MountainCar، الگوریتم‌های کلاسیک و ساده‌تر بهترین تعادل بین دقت و سرعت را فراهم کردند. الگوریتم‌های پیشرفته‌تر مانند DQN تنها با مهندسی دقیق (مانند شکل‌دهی پاداش) توانستند برتری خود را نشان دهند، که این برتری نیز با هزینه زمانی بیشتری همراه بود.