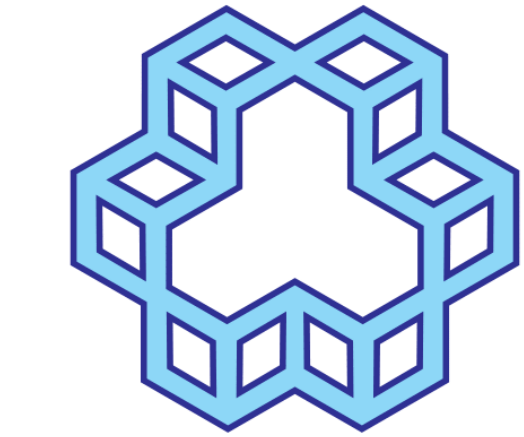


دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی برق- گروه مهندسی کنترل

درس یادگیری ماشین

گزارش مینی پروژه شماره چهار

نام و نام خانوادگی	علیرضا جهانی
شمارهٔ دانشجویی	۴۰۲۲۳۰۴۴
استاد درس	دکتر علیاری
تاریخ	بهمن‌ماه ۱۴۰۲

colab github

فهرست مطالب

۱ سوال دوم

۱.۱ توضیحات بازی و روش‌های DQN و DDQN

۱.۱.۱ توضیحات بازی

هدف این پروژه کسب بیش از ۲۰۰ امتیاز در هر اپیزود به طور متوسط در ۱۰۰ اپیزود از بازی Lander Lunar است. برای مقابله با این چالش، یک شبکه عصبی دوبل عمیق Double Deep Q-Network یا به اختصار DDQN (Hasselt et al. 2016) [۱] و DQN معرفی و پیاده‌سازی می‌شود که با توضیحات دقیق همراه خواهد بود.

فضای حالت:

محیط OpenAI Gym (Brockman et al. 2016) از Lunar Lander یک محیط تعاملی برای یک عامل برای فرود آوردن یک موشک روی یک سیاره است. یک حالت در اینجا می‌تواند با یک فضای پیوسته ۸ بعدی نمایش داده شود:

$$(x, y, v_x, v_y, \theta, v_\theta, leg_{left}, leg_{right})$$

که در آن x و y مختصات موقعیت فرودگر هستند؛ v_x و v_y اجزای سرعت در دو محور هستند؛ θ و v_θ زاویه و سرعت زاویه‌ای به طور جداگانه هستند؛ leg_{left} و leg_{right} دو مقدار باینری هستند که نشان می‌دهند آیا پای چپ یا راست فرودگر با زمین تماس دارد یا خیر. در نتیجه ۸ state داریم.

اکشن ها: برای هر گام زمانی، چهار اکشن گسسته موجود است، یعنی هیچ کاری نکردن، شلیک موتور جهت‌گیری چپ، شلیک موتور اصلی، و شلیک موتور جهت‌گیری راست. در نتیجه ۴ اکشن داریم.

امتیاز دهی:

بازی تمام می‌شود یا عبور می‌کند اگر فرودگر سقوط کند یا متوقف شود. پاداش برای یک پایان بد −۱۰۰ است، در حالی که برای یک پایان خوش +۱۰۰ است. تماس پای فرودگر با زمین می‌تواند +۱۰ پاداش بدهد، اما هر بار شلیک موتور اصلی یک جریمه -0.3 دارد. بنابراین، کل پاداش برای یک اپیزود از ۱۰۰ تا بیش از ۲۰۰ بسته به مکان نهایی فرودگر روی سکوی فرود متغیر است. فاصله بین سکوی فرود و فرودگر باعث جریمه می‌شود که برابر با پاداش کسب شده از نزدیک شدن به سکو است. بازی را میتوان از کتابخانه gym فراخواند.

```
env = gym.make('LunarLander-v2')
n_states, n_actions = env.observation_space.shape[0], env.action_space.n
print('state space: {}'.format(n_states))
print('action space: {}'.format(n_actions))
```

```
state space: 8
action space: 4
```

۱.۱.۲ DQN:

شبکه عصبی عمیق یکDQN روش یادگیری تقویتی است که از شبکه‌های عصبی عمیق برای تقریب تابع Q استفاده می‌کند. تابع Q ارزش یک حالت–اکشن را نشان می‌دهد که بیانگر ارزش مورد انتظار پاداش کلی است که از انجام اکشن a در حالت s و پیروی از سیاست بهینه بعد از آن حاصل می‌شود.

[۱] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence.

در روش DQN، شبکه عصبی عمیق به عنوان یک تقریب‌گر تابع Q استفاده می‌شود تا مقدار Q را برای جفت‌های حالت-اکشن محاسبه کند. معادله به‌روزرسانی تابع Q در DQN به صورت زیر است:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

که در آن:

- s_t: حالت در زمان t
- a_t: اکشن انتخاب شده در زمان t
- r_t: پاداش دریافت شده در زمان t
- s_{t+1}: حالت جدید بعد از انجام اکشن a_t
- α: نرخ یادگیری
- γ: discount factor

روش DQN از بازپخش تجربیات (Experience Replay) و شبکه هدف (Target Network) برای بهبود پایداری و کارایی یادگیری استفاده می‌کند. در بازپخش تجربیات، تجربیات عامل در یک حافظه ذخیره می‌شود و به صورت تصادفی از این حافظه نمونه‌برداری می‌شود تا همبستگی بین تجربیات کاهش یابد. شبکه هدف نیز کپی‌ای از شبکه اصلی است که به صورت اپیزودای به‌روزرسانی می‌شود تا از نوسانات زیاد در فرآیند یادگیری جلوگیری کند.

```
class DQN_Graph(nn.Module):
    def __init__(self, n_states, n_actions, hidden_size=32):
        super(DQN_Graph, self).__init__()
        self.dense_layer_1 = nn.Linear(n_states, hidden_size)
        self.dense_layer_2 = nn.Linear(hidden_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, n_actions)

    def forward(self, state):
        x = F.relu(self.dense_layer_1(state))
        x = F.relu(self.dense_layer_2(x))
        return self.output_layer(x)
```

کلاس `DQN_Graph` مدل شبکه عصبی برای یادگیری Q عمیق را تعریف می‌کند. این مدل شبکه‌ای از فضای حالت R^{n_states} به فضای اکشن $R^{n\_actions}$ است.

- تبدیل مدل از فضای حالت state به فضای action
  - `dense_layer_1`: لایه اول با ابعاد `n_states` به `hidden_size`.
  - `dense_layer_2`: لایه دوم با ابعاد `hidden_size` به `hidden_size`.
  - `output_layer`: لایه خروجی با ابعاد `hidden_size` به `n_actions`.
- `forward`: تابع (forward propagation) که انتقال ورودی از لایه‌ها را انجام می‌دهد.

کلاس ReplayMemory

```
class ReplayMemory():
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = [None] * capacity
        self.position = 0
        self.Transition = namedtuple('Transition',
                                     ('state', 'action', 'reward', 'next_state', 'done'))

    def size(self):
        return len(self.memory) - self.memory.count(None)

    def push(self, *args):
        self.memory[self.position] = self.Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def pull(self):
        return [exp for exp in self.memory if exp is not None]

    def sample(self, batch_size):
        exps = random.sample(self.pull(), batch_size)
        states = torch.tensor(np.vstack([e.state for e in exps if e is not None])).float()
        actions = torch.tensor(np.vstack([e.action for e in exps if e is not None])).long()
        rewards = torch.tensor(np.vstack([e.reward for e in exps if e is not None])).float
        next_states = torch.tensor(np.vstack([e.next_state for e in exps if e is not None]
        dones = torch.tensor(np.vstack([e.done for e in exps if e is not None]).astype(np.

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.memory)
```

کلاس ReplayMemory حافظه‌ی تجربی را پیاده‌سازی می‌کند که برای ذخیره و نمونه‌برداری تجربیات عامل استفاده می‌شود.

- حافظه و موقعیت کنونی را ذخیره میکنیم.

  - capacity: ظرفیت حافظه.

- memory: لیستی برای ذخیره تجربیات.
- position: موقعیت فعلی برای افزودن تجربه جدید.
- Transition: یک namedtuple برای ذخیره حالات، اکشن ها، پاداش‌ها، حالت‌های بعدی و وضعیت پایان.

- size: اندازه حافظه را برمی‌گرداند.
- push: یک تجربه جدید به حافظه اضافه می‌کند.
- pull: تجربیات غیر تهی را از حافظه بازمی‌گرداند.
- sample: نمونه‌ای تصادفی از تجربیات را برمی‌گرداند.
- __len__: طول حافظه را برمی‌گرداند.

کلاس DQN_Agent

```
class DQN_Agent():
    def __init__(self, n_states, n_actions, batch_size, hidden_size, memory_size,
                 update_step, learning_rate, gamma, tau):
        self.n_states = n_states
        self.n_actions = n_actions
        self.batch_size = batch_size
        self.hidden_size = hidden_size
        self.update_step = update_step
        self.lr = learning_rate
        self.gamma = gamma
        self.tau = tau
        self.setup_gpu()
        self.setup_model()
        self.setup_opt()
        self.memory = ReplayMemory(memory_size)
        self.memory.device = self.device  # Ensure memory uses the correct device
        self.prepare_train()

    def setup_gpu(self):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    def setup_model(self):
        self.policy_model = DQN_Graph(self.n_states, self.n_actions, self.hidden_size).to(
```

```
        self.target_model = DQN_Graph(self.n_states, self.n_actions, self.hidden_size).to(

    def setup_opt(self):
        self.opt = torch.optim.Adam(self.policy_model.parameters(), lr=self.lr)

    def prepare_train(self):
        self.steps = 0

    def act(self, state, epsilon):
        state = torch.tensor(state).reshape(1, -1).float().to(self.device)
        self.policy_model.eval()
        with torch.no_grad():
            action_vs = self.policy_model(state)
        self.policy_model.train()
        if np.random.random() > epsilon:
            return np.argmax(action_vs.cpu().detach().numpy())
        else:
            return np.random.randint(self.n_actions)

    def step(self, s, a, r, s_, done):
        self.memory.push(s, a, r, s_, done)
        self.steps = (self.steps + 1) % self.update_step
        if self.steps == 0 and self.memory.size() >= self.batch_size:
            exps = self.memory.sample(self.batch_size)
            self.learn(exps)

    def learn(self, exps):
        states, actions, rewards, next_states, dones = exps
        target_next_action_vs = self.target_model(next_states).detach().max(1)[0].unsqueeze
        target_q_vs = rewards + (self.gamma * target_next_action_vs * (1 - dones))
        policy_q_vs = self.policy_model(states).gather(1, actions)
        loss = F.mse_loss(policy_q_vs, target_q_vs)
        self.opt.zero_grad()
        loss.backward()
        for p in self.policy_model.parameters():
            p.grad.data.clamp_(-1, 1)
```

```
        self.opt.step()
        for tp, lp in zip(self.target_model.parameters(), self.policy_model.parameters()):
            tp.data.copy_(self.tau*lp.data + (1.0-self.tau)*tp.data)
```

کلاس DQN_Agent برای ایجاد یک عامل یادگیری تقویتی با استفاده از شبکه عصبی عمیق Q (DQN) طراحی شده است. این عامل به محیط بازی متصل می‌شود و با استفاده از تجربیات خود، سیاست بهینه‌ای را یاد می‌گیرد. فرآیند کار عامل را می‌توان به چندین مرحله تقسیم کرد که هر کدام جداگانه توضیح داده میشود.

۱. مقداردهی اولیه

در ابتدای کار، عامل باید پارامترهای اولیه شامل تعداد حالات (n_{states})، تعداد اکشن ها ($n_{actions}$)، اندازه دسته ($batch\_size$)، اندازه لایه‌های مخفی ($hidden\_size$)، اندازه حافظه ($memory\_size$)، گام به‌روزرسانی ($update\_step$)، نرخ یادگیری (α)، ضریب گاما (γ) و ضریب به‌روزرسانی شبکه هدف (τ) را مقداردهی کند. همچنین، حافظه تجربی و مدل‌های شبکه عصبی ایجاد و تنظیم می‌شوند.

۲. انتخاب اکشن

عامل با استفاده از سیاست ϵ-حریص اکشن می‌کند. به این صورت که با احتمال ϵ یک اکشن تصادفی انتخاب می‌شود و با احتمال $1-\epsilon$ بهترین اکشن ممکن بر اساس شبکه سیاست انتخاب می‌شود.

$$Action = \begin{cases} RandomAction & withprobability\epsilon \\ \arg\max_a Q(s,a;\theta) & withprobability1-\epsilon \end{cases}$$

جلوتر توضیح داده میشود اپسیلون را چگونه تغییر میدهیم که در ابتدا سرچ کند و در آخرای تمرین به سمت بهترین حالت حرکت کند یعنی با اپسیلون کنترل میکنیم و احتمال هارا کم و زیاد میکنیم.

۳. ذخیره‌سازی تجربیات

تجربیات جدید (شامل حالت s، اکشن a، پاداش r، حالت بعدی s' و وضعیت پایان $done$) در حافظه تجربی ذخیره می‌شود.

۴. به‌روزرسانی شبکه‌ها

در هر گام، اگر تعداد تجربیات به اندازه کافی باشد، شبکه‌های سیاست و هدف به‌روزرسانی می‌شوند. این به‌روزرسانی با استفاده از معادله‌های زیر انجام می‌شود:

$$Q = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

که در آن θ^- پارامترهای شبکه هدف است.

مقدار Q پیش‌بینی شده توسط شبکه سیاست:

$$Q = Q(s, a; \theta)$$

خطای مربع میانگین بین مقادیر Q هدف و سیاست محاسبه می‌شود:

$$loss = \frac{1}{N}\sum_{i=1}^{N}(Q - Q)^2$$

۵. به‌روزرسانی پارامترهای شبکه‌ها

پارامترهای شبکه سیاست با استفاده از الگوریتم بهینه‌سازی Adam به‌روزرسانی می‌شوند و پارامترهای شبکه هدف نیز به صورت اپیزودای به‌روزرسانی می‌شوند:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

این مراحل تکرار می‌شوند تا عامل به سیاست بهینه برای بازی دست یابد و امتیاز مورد نظر را کسب کند.

تابع dqn_learn_op

```
def dqn_learn_op(n_episodes, rewards_window_size, epsilon_array):
    best_avg_rewards = -30.
    total_rewards = []
    rewards_deque = deque(maxlen=rewards_window_size)
    t = trange(n_episodes)
    for episode in t:
        cur_state = env.reset()
        done = False
        rewards = 0
        epsilon = epsilon_array[episode]
        while not done:
            action = agent.act(cur_state, epsilon)
            next_state, reward, done, _ = env.step(action)
            agent.step(cur_state, action, reward, next_state, done)
            cur_state = next_state
            rewards += reward
        total_rewards.append(rewards)
        rewards_deque.append(rewards)
        avg_rewards = np.mean(rewards_deque)
        t.set_description(
            'Episode␣{}␣Epsilon␣{:.2f}␣Reward␣{:.2f}␣Avg_Reward␣{:.2f}␣Best_Avg_Reward␣{:.2
                episode + 1, epsilon, rewards, avg_rewards, best_avg_rewards))
        t.refresh()
        if avg_rewards >= best_avg_rewards:
            best_avg_rewards = avg_rewards
            torch.save(agent.policy_model.state_dict(), DQN_CHECKPOINT_PATH)
        if best_avg_rewards > 200:
            break
    return total_rewards, rewards_deque
```

تابع `dqn_learn_op` فرآیند آموزش عامل DQN را پیاده‌سازی می‌کند.

- `n_episodes`: تعداد اپیزودهای آموزش.
- `rewards_window_size`: اندازه پنجره برای محاسبه میانگین پاداش.
- `epsilon_array`: آرایه‌ای از مقادیر ϵ برای سیاست ϵ-حریص.
- `best_avg_rewards`: بهترین میانگین پاداش تاکنون.
- `total_rewards`: لیستی از کل پاداش‌های هر اپیزود.
- `rewards_deque`: صفی برای ذخیره پاداش‌های اخیر به منظور محاسبه میانگین پاداش.
- `t`: نوار پیشرفت برای نمایش وضعیت آموزش.

در هر اپیزود:

- وضعیت اولیه محیط تنظیم می‌شود.
- تا زمانی که اپیزود تمام نشده است، عامل اکشن می‌کند و وضعیت جدید و پاداش را دریافت می‌کند.
- تجربه جدید به حافظه اضافه می‌شود و مدل به روز رسانی می‌شود.
- پاداش اپیزود جمع‌آوری می‌شود و میانگین پاداش محاسبه می‌شود.
- اگر میانگین پاداش جدید بهتر از بهترین میانگین پاداش باشد، مدل ذخیره می‌شود.
- اگر بهترین میانگین پاداش بیشتر از ۲۰۰ شود، آموزش متوقف می‌شود.

تابع در نهایت کل پاداش‌ها و صف پاداش‌های اخیر را بازمی‌گرداند. مقدار اولیه بهترین پاداش ها را از عمد کم گذاشتم چون در مراحل جلو تر از ما خواسته شده است که با اپیزود های کم هم خروجی بگیریم و باید مدل ایجاد شود وگرنه با این شرط میتوان تا وقتی که مدل یاد نگرفته است مدل ذخیره نکند و حداقلی یاد بگیرد.

۳.۱.۱ **DDQN**:

Double Deep Q-Network (DDQN) نسخه‌ای پیشرفته‌تر از شبکه عصبی عمیق Q (DQN) است که برای بهبود پایداری و دقت یادگیری تقویتی معرفی شده است. مشکل اصلی DQN بیش‌برآورد ارزش‌های Q است که به دلیل استفاده از همان شبکه برای انتخاب و ارزیابی اکشن‌ها به وجود می‌آید. DDQN با جدا کردن این دو وظیفه به دو شبکه مجزا، این مشکل را کاهش می‌دهد.
در DDQN، دو شبکه عصبی به کار گرفته می‌شوند:

- شبکه سیاست (Policy Network) یا شبکه اصلی (Main Network)
- شبکه هدف (Target Network)

شبکه سیاست برای انتخاب اکشن‌ها استفاده می‌شود و شبکه هدف برای ارزیابی ارزش‌های Q به کار می‌رود.
معادلات به‌روزرسانی در DDQN به صورت زیر است:

انتخاب اکشن

در ابتدا، اکشن با استفاده از شبکه سیاست انتخاب می‌شود:

$$a^* = \arg\max_a Q(s, a; \theta)$$

محاسبه مقدار Q هدف

سپس، مقدار Q هدف با استفاده از شبکه هدف محاسبه می‌شود:

$$Q = r + \gamma Q(s', a^*; \theta^-)$$

که در آن:

- r: پاداش دریافتی
- γ: ضریب گاما
- s': حالت بعدی
- θ: پارامترهای شبکه سیاست
- θ^-: پارامترهای شبکه هدف

به‌روزرسانی پارامترهای شبکه سیاست

خطای مربع میانگین بین مقادیر Q هدف و مقادیر پیش‌بینی شده توسط شبکه سیاست محاسبه می‌شود:

$$loss = \frac{1}{N} \sum_{i=1}^{N} (Q - Q(s, a; \theta))^2$$

پارامترهای شبکه سیاست با استفاده از الگوریتم بهینه‌سازی (مانند Adam) به‌روزرسانی می‌شوند.

مقایسه DDQN و DQN

تفاوت اصلی بین DDQN و DQN در نحوه محاسبه مقدار Q هدف است. در DQN، مقدار Q هدف به صورت زیر محاسبه می‌شود:

$$Q = r + \gamma \max_{a'} Q(s', a'; \theta)$$

این باعث می‌شود که شبکه DQN به دلیل استفاده از همان شبکه برای انتخاب و ارزیابی اکشن‌ها، مقادیر Q را بیش‌برآورد کند. در مقابل، DDQN از دو شبکه مجزا برای این کار استفاده می‌کند که به کاهش بیش‌برآورد کمک می‌کند.

به طور کلی:

- DQN از یک شبکه برای انتخاب و ارزیابی اکشن‌ها استفاده می‌کند که منجر به بیش‌برآورد می‌شود.

- DDQN از دو شبکه مجزا استفاده می‌کند که انتخاب و ارزیابی اکشن‌ها را جدا می‌کند و به کاهش بیش‌برآورد کمک می‌کند.

این بهبود باعث می‌شود که DDQN پایداری بیشتری در فرآیند یادگیری داشته باشد و عملکرد بهتری در مسائل مختلف یادگیری تقویتی از خود نشان دهد.

کد زیر پیاده‌سازی شبکه عصبی دوبل عمیق Q (DDQN) را نشان می‌دهد که توسط Hasselt و همکاران در سال ۲۰۱۶ معرفی شده است. این کد شامل سه بخش اصلی است: مدل شبکه عصبی، حافظه بازپخش، و عامل .DDQN در ادامه هر بخش از کد به تفصیل توضیح داده شده است.

کلاس `DDQN_Graph`

این کلاس مدل شبکه عصبی برای DDQN را پیاده‌سازی می‌کند.

```
class DDQN_Graph(nn.Module):

    def __init__(self, n_states, n_actions, hidden_size=32):
        super(DDQN_Graph, self).__init__()
        self.n_actions = n_actions
        self.half_hidden_size = int(hidden_size/2)
        # hidden representation
        self.dense_layer_1 = nn.Linear(n_states, hidden_size)
        self.dense_layer_2 = nn.Linear(hidden_size, hidden_size)
        # V(s)
        self.v_layer_1 = nn.Linear(hidden_size, self.half_hidden_size)
        self.v_layer_2 = nn.Linear(self.half_hidden_size, 1)
        # A(s, a)
        self.a_layer_1 = nn.Linear(hidden_size, self.half_hidden_size)
        self.a_layer_2 = nn.Linear(self.half_hidden_size, n_actions)

    def forward(self, state):
        # state: batch_size, state_size
        # x: batch_size, hidden_size
        x = F.relu(self.dense_layer_1(state))
        # x: batch_size, hidden_size
        x = F.relu(self.dense_layer_2(x))
        # v: batch_size, half_hidden_size
        v = F.relu(self.v_layer_1(x))
        # v: batch_size, 1
        v = self.v_layer_2(v)
        # a: batch_size, half_hidden_size
```

```
        a = F.relu(self.a_layer_1(x))
        # a: batch_size, action_size
        a = self.a_layer_2(a)

        # Q(s,a) = V(s) + (A(s,a) - 1/|A| * sum A(s,a'))
        # batch_size, action_size
        return v + a - a.mean(dim=-1, keepdim=True).expand(-1, self.n_actions)
```

تابع `forward` در کلاس `DDQN_Graph` ورودی حالت (`state`) را از طریق لایه‌های مختلف شبکه عبور می‌دهد و مقدار Q را محاسبه می‌کند. این فرآیند به شرح زیر است:

- ورودی `state` با ابعاد `batch_size` و `state_size` وارد اولین لایه متراکم (`dense_layer_1`) می‌شود و تابع فعال‌سازی ReLU بر روی آن اعمال می‌شود:

$$x = F.relu(\texttt{dense\_layer\_1}(state))$$

- خروجی لایه اول (`x`) با ابعاد `batch_size` و `hidden_size` وارد دومین لایه متراکم (`dense_layer_2`) می‌شود و دوباره تابع فعال‌سازی ReLU بر روی آن اعمال می‌شود:

$$x = F.relu(\texttt{dense\_layer\_2}(x))$$

- خروجی لایه دوم (`x`) به دو مسیر جداگانه تقسیم می‌شود: یکی برای محاسبه ارزش حالت ($V(s)$) و دیگری برای محاسبه مزیت اکشن ($A(s,a)$).

- برای محاسبه $V(s)$، خروجی `x` وارد اولین لایه متراکم مربوط به $V(s)$ (`v_layer_1`) می‌شود و تابع ReLU بر روی آن اعمال می‌شود:

$$v = F.relu(\texttt{v\_layer\_1}(x))$$

- سپس خروجی `v` وارد دومین لایه متراکم مربوط به $V(s)$ (`v_layer_2`) می‌شود تا مقدار نهایی $V(s)$ محاسبه شود:

$$v = \texttt{v\_layer\_2}(v)$$

- برای محاسبه $A(s,a)$، خروجی `x` وارد اولین لایه متراکم مربوط به $A(s,a)$ (`a_layer_1`) می‌شود و تابع ReLU بر روی آن اعمال می‌شود:

$$a = F.relu(\texttt{a\_layer\_1}(x))$$

- سپس خروجی `a` وارد دومین لایه متراکم مربوط به $A(s,a)$ (`a_layer_2`) می‌شود تا مقدار نهایی $A(s,a)$ محاسبه شود:

$$a = \texttt{a\_layer\_2}(a)$$

- در نهایت، مقدار Q با استفاده از معادله زیر محاسبه می‌شود:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|}\sum A(s,a')\right)$$

که در اینجا `mean` مقدار میانگین $A(s,a')$ را برای تمام اعمال محاسبه می‌کند و از $A(s,a)$ کسر می‌شود تا مزیت نسبی هر اکشن را به دست آورد.

- تابع expand برای مطابقت دادن ابعاد خروجی استفاده می‌شود تا مقدار نهایی $Q(s,a)$ با ابعاد batch_size و action_size به دست آید.

کلاس ReplayMemory

این کلاس برای ذخیره و بازپخش تجربیات عامل استفاده می‌شود.

```
class ReplayMemory():
    """
    Replay memory records previous observations for the agent to learn later
    by sampling from the memory randomly
    """
    def __init__(self, capacity):
        super(ReplayMemory, self).__init__()
        self.capacity = capacity
        # to avoid empty memory list to insert transitions
        self.memory = [None] * capacity
        self.position = 0
        self.Transition = namedtuple('Transition',
                                     ('state', 'action', 'reward', 'next_state', 'done'))

    def size(self):
        return len(self.memory) - self.memory.count(None)

    def push(self, *args):
        # save a transition at a certain position of the memory
        self.memory[self.position] = self.Transition(*args)
        # update position
        self.position = (self.position + 1) % self.capacity

    def pull(self):
        return [exp for exp in self.memory if exp is not None]

    def sample(self, batch_size):
        exps = random.sample(self.pull(), batch_size)
        states = torch.tensor(np.vstack([e.state for e in exps if e is not None])).float()
        actions = torch.tensor(np.vstack([e.action for e in exps if e is not None])).long()
```

```
        rewards = torch.tensor(np.vstack([e.reward for e in exps if e is not None])).float
        next_states = torch.tensor(np.vstack([e.next_state for e in exps if e is not None]
        dones = torch.tensor(np.vstack([e.done for e in exps if e is not None]).astype(np.

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.memory)
```

همانند DQN، این کلاس شامل موارد زیر است:

- `__init__`: این تابع سازنده، حافظه بازپخش را مقداردهی اولیه می‌کند.
- `size`: اندازه حافظه را برمی‌گرداند.
- `push`: یک انتقال جدید به حافظه اضافه می‌کند.
- `pull`: تمام تجربیات غیر تهی را بازمی‌گرداند.
- `sample`: یک نمونه تصادفی از حافظه بازپخش بازمی‌گرداند.
- `__len__`: طول حافظه را بازمی‌گرداند.

کلاس `DDQN_Agent`

این کلاس عامل DDQN را پیاده‌سازی می‌کند که برای تعامل با محیط و یادگیری از تجربیات خود استفاده می‌شود.

```
class DDQN_Agent():
    """docstring for ddqn_agent"""
    def __init__(self, n_states, n_actions, batch_size, hidden_size, memory_size,
                 update_step, learning_rate, gamma, tau):
        super(DDQN_Agent, self).__init__()
        # state space dimension
        self.n_states = n_states
        # action space dimension
        self.n_actions = n_actions
        # configuration
        self.batch_size = batch_size
        self.hidden_size = hidden_size
        self.update_step = update_step
        self.lr = learning_rate
```

```
        self.gamma = gamma
        self.tau = tau
        # check cpu or gpu
        self.setup_gpu()
        # initialize model graph
        self.setup_model()
        # initialize optimizer
        self.setup_opt()
        # enable Replay Memory
        self.memory = ReplayMemory(memory_size)
        # others
        self.prepare_train()

    def setup_gpu(self):
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    def setup_model(self):
        self.policy_model = DDQN_Graph(
            self.n_states,
            self.n_actions,
            self.hidden_size).to(self.device)
        self.target_model = DDQN_Graph(
            self.n_states,
            self.n_actions,
            self.hidden_size).to(self.device)

    def setup_opt(self):
        self.opt = torch.optim.Adam(self.policy_model.parameters(), lr=self.lr)

    def prepare_train(self):
        self.steps = 0

    def act(self, state, epsilon):
      state = torch.tensor(state).reshape(1, -1).to(self.device)
      self.policy_model.eval()
      with torch.no_grad():
```

```
            action_vs = self.policy_model(state)
        self.policy_model.train()
        if np.random.random() > epsilon:
            return np.argmax(action_vs.cpu().detach().numpy())
        else:
            return np.random.randint(self.n_actions)


    def step(self, s, a, r, s_, done):
        # add one observation to memory
        self.memory.push(s, a, r, s_, done)
        # update model for every certain steps
        self.steps = (self.steps + 1) % self.update_step
        if self.steps == 0 and self.memory.size() >= self.batch_size:
            exps = self.memory.sample(self.batch_size)
            self.learn(exps)
        else:
            pass

    def learn(self, exps, soft_copy=True):
        states, actions, rewards, next_states, dones = exps
        states = states.to(self.device)
        actions = actions.to(self.device)
        rewards = rewards.to(self.device)
        next_states = next_states.to(self.device)
        dones = dones.to(self.device)

        # target side
        _, next_idx = self.policy_model(next_states).detach().max(1)
        target_next_action_vs = self.target_model(next_states).detach().gather(1, next_idx.u
        target_q_vs = rewards + (self.gamma * target_next_action_vs * (1 - dones))

        # policy side
        policy_q_vs = self.policy_model(states).gather(1, actions)
        loss = F.mse_loss(policy_q_vs, target_q_vs)
```

```
        self.opt.zero_grad()
        loss.backward()
        for p in self.policy_model.parameters():
            p.grad.data.clamp_(-1, 1)
        self.opt.step()

        if soft_copy:
            for tp, lp in zip(self.target_model.parameters(), self.policy_model.parameters()
                tp.data.copy_(self.tau * lp.data + (1.0 - self.tau) * tp.data)
        else:
            self.target_model.load_state_dict(self.policy_model.state_dict())
```

کلاس `DDQN_Agent` یک عامل یادگیری تقویتی دوبل عمیق Q (DDQN) را پیاده‌سازی می‌کند. این کلاس شامل مراحل مختلفی برای مقداردهی اولیه، انتخاب اکشن، ذخیره‌سازی تجربیات و به‌روزرسانی مدل‌ها است. در ادامه به توضیح این مراحل می‌پردازیم:

مقداردهی اولیه

در ابتدای کار، پارامترهای اولیه شامل ابعاد فضای حالت (n_{states})، ابعاد فضای اکشن ($n_{actions}$)، اندازه دسته ($batch\_size$)، اندازه لایه‌های مخفی ($hidden\_size$)، اندازه حافظه ($memory\_size$)، گام به‌روزرسانی ($update\_step$)، نرخ یادگیری (α)، ضریب گاما (γ) و ضریب به‌روزرسانی شبکه هدف (τ) تنظیم می‌شوند. همچنین، حافظه بازپخش و مدل‌های شبکه عصبی ایجاد و تنظیم می‌شوند.

تنظیم GPU یا CPU

تابعی برای تنظیم مدل GPU یا CPU استفاده می‌شود تا از منابع محاسباتی بهینه استفاده شود. اگر GPU در دسترس باشد، مدل به `cuda` تنظیم می‌شود و در غیر این صورت از `cpu` استفاده می‌شود.

ایجاد مدل‌های شبکه عصبی

مدل‌های شبکه عصبی سیاست (policy) (model و هدف (target) (model با استفاده از کلاس `DDQN_Graph` ایجاد می‌شوند و به مدل تنظیم شده منتقل می‌شوند.

تنظیم بهینه‌ساز

بهینه‌ساز `Adam` برای به‌روزرسانی پارامترهای شبکه سیاست تنظیم می‌شود. نرخ یادگیری (α) نیز در این مرحله تنظیم می‌شود.

آماده‌سازی برای آموزش

مراحل اولیه برای شروع آموزش آماده می‌شوند. شمارنده گام‌ها (`steps`) نیز در این مرحله مقداردهی اولیه می‌شود.

انتخاب اکشن

عملیات انتخاب اکشن با استفاده از سیاست ϵ-حریص انجام می‌شود. اگر مقدار تصادفی کوچکتر از ϵ باشد، یک اکشن تصادفی انتخاب می‌شود. در غیر این صورت، بهترین اکشن ممکن بر اساس مقادیر Q محاسبه شده توسط شبکه سیاست انتخاب می‌شود.

ذخیره‌سازی تجربیات

تجربیات جدید شامل حالت، اکشن، پاداش، حالت بعدی و وضعیت پایان در حافظه بازپخش ذخیره می‌شوند. این تجربیات در به‌روزرسانی‌های بعدی مدل‌ها استفاده خواهند شد.

به‌روزرسانی مدل‌ها

در هر گام، اگر تعداد تجربیات در حافظه بازپخش به اندازه کافی باشد و گام به‌روزرسانی فرا رسیده باشد، نمونه‌ای تصادفی از تجربیات انتخاب و مدل‌ها به‌روزرسانی می‌شوند.

محاسبه مقدار Q هدف

در سمت هدف، اکشن بهینه بعدی با استفاده از شبکه سیاست انتخاب می‌شود و مقدار Q هدف محاسبه می‌شود:

$$Q = r + \gamma Q(s', a^*; \theta^-)$$

محاسبه مقدار Q سیاست

در سمت سیاست، مقدار Q برای حالت‌ها و اعمال فعلی محاسبه می‌شود و با مقدار Q هدف مقایسه می‌شود. خطای مربع میانگین (`loss MSE`) بین این مقادیر محاسبه و برای به‌روزرسانی پارامترهای شبکه سیاست استفاده می‌شود.

به‌روزرسانی نرم شبکه هدف

در نهایت، پارامترهای شبکه هدف به صورت نرم (`update soft`) با استفاده از ترکیبی از پارامترهای شبکه سیاست و شبکه هدف به‌روزرسانی می‌شوند:

$$\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$$

این مراحل به طور مداوم تکرار می‌شوند تا عامل به سیاست بهینه برای محیط یادگیری تقویتی دست یابد.

به‌روزرسانی مدل‌ها

در هر گام، اگر تعداد تجربیات در حافظه بازپخش به اندازه کافی باشد و گام به‌روزرسانی فرا رسیده باشد، نمونه‌ای تصادفی از تجربیات انتخاب و مدل‌ها به‌روزرسانی می‌شوند.

محاسبه مقدار Q هدف

در سمت هدف، اکشن بهینه بعدی با استفاده از شبکه سیاست انتخاب می‌شود و مقدار Q هدف محاسبه می‌شود:

$$Q = r + \gamma Q(s', a^*; \theta^-)$$

محاسبه مقدار Q سیاست

در سمت سیاست، مقدار Q برای حالت‌ها و اعمال فعلی محاسبه می‌شود و با مقدار Q هدف مقایسه می‌شود. خطای مربع میانگین (`loss MSE`) بین این مقادیر محاسبه و برای به‌روزرسانی پارامترهای شبکه سیاست استفاده می‌شود.

به‌روزرسانی نرم شبکه هدف

در نهایت، پارامترهای شبکه هدف به صورت نرم (`update soft`) با استفاده از ترکیبی از پارامترهای شبکه سیاست و شبکه هدف به‌روزرسانی می‌شوند:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

این مراحل به طور مداوم تکرار می‌شوند تا عامل به سیاست بهینه برای محیط یادگیری تقویتی دست یابد. به طور کلی همانند قبل اپسیلون را اول زیاد میگیریم تا در اپیزود های اولیه مدل سرچ کند و شانسی گشتن بیشتر باشد و در نهایت شانسی آنرا کمتر و کمتر میکنیم جلو تر تابع انتخب اپسیلون آورده شده است.

تابع `ddqn_learn_op`

همانند روش DQN است.
در کل هر اپیزود:

- وضعیت اولیه محیط تنظیم می‌شود.
- تا زمانی که اپیزود تمام نشده است، عامل اکشن می‌کند و وضعیت جدید و پاداش را دریافت می‌کند.
- تجربه جدید به حافظه اضافه می‌شود و مدل به روز رسانی می‌شود.
- پاداش اپیزود جمع‌آوری می‌شود و میانگین پاداش محاسبه می‌شود.
- اگر میانگین پاداش جدید بهتر از بهترین میانگین پاداش باشد، مدل ذخیره می‌شود.
- اگر بهترین میانگین پاداش بیشتر از ۲۰۰ شود، آموزش متوقف می‌شود.

```python
n_episodes = 400
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.004
rewards_window_size = 100

epsilon_array = np.zeros((n_episodes))
for i in range(n_episodes):
    epsilon = min_epsilon + (max_epsilon-min_epsilon)*np.exp(-decay_rate*i)
    epsilon_array[i] = epsilon

plt.plot(epsilon_array)
plt.show()
```

۴.۱.۱ انتخاب اپسیلون

در یادگیری تقویتی با سیاست ϵ-حریص، عامل با احتمال ϵ یک عمل تصادفی انتخاب می‌کند و با احتمال $1 - \epsilon$ بهترین اکشن ممکن بر اساس مقادیر Q را انتخاب می‌کند. هدف از این سیاست، ایجاد توازن بین کاوش (exploration) و بهره‌برداری (exploitation) است.

معادله ϵ در این کد به صورت زیر تعریف شده است:

$$\epsilon = min\_epsilon + (max\_epsilon - min\_epsilon) \cdot \exp(-decay\_rate \cdot i)$$

که در آن:

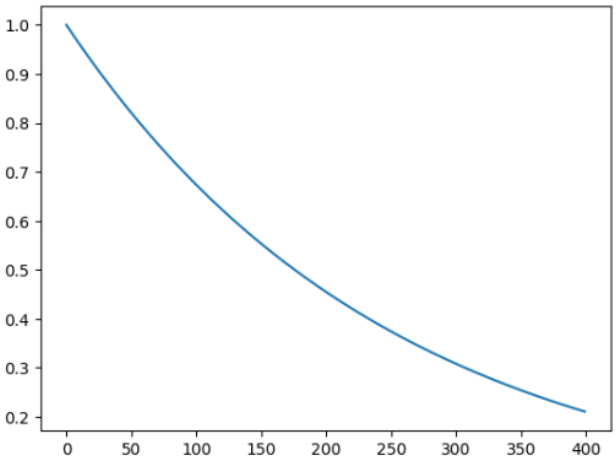
- $min\_epsilon$: حداقل مقدار ϵ که عامل به آن نزدیک می‌شود (در اینجا 0.01).
- $max\_epsilon$: حداکثر مقدار ϵ در شروع آموزش (در اینجا 1.0).
- $decay\_rate$: نرخ کاهش ϵ با افزایش تعداد اپیزودها (در اینجا 0.004).
- i: شماره اپیزود جاری.

این معادله باعث می‌شود که مقدار ϵ در طول زمان کاهش یابد و عامل از حالت کاوش بیشتر به سمت بهره‌برداری بیشتر حرکت کند. به عبارتی، در ابتدای آموزش، عامل بیشتر به کاوش می‌پردازد و با گذشت زمان و افزایش دانش خود، بیشتر از سیاست بهینه استفاده می‌کند.

در کد زیر، آرایه‌ای از مقادیر ϵ برای تعداد اپیزودهای مشخص ($n\_episodes = 400$) محاسبه و ذخیره می‌شود. سپس مقادیر ϵ به ازای هر اپیزود در یک نمودار رسم می‌شوند.

نمودار زیر تغییرات ϵ را در طول ۴۰۰ اپیزود نشان می‌دهد:

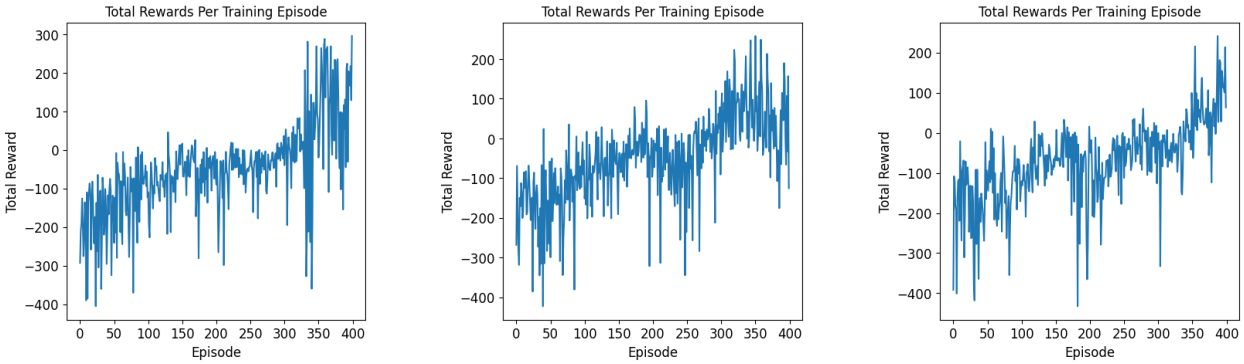
این نمودار نشان می‌دهد که مقدار ϵ به صورت نمایی کاهش می‌یابد و به حداقل مقدار خود نزدیک می‌شود، که نشان‌دهنده انتقال تدریجی عامل از کاوش به بهره‌برداری است.



شکل ۱: مقدار اپسیلون بر حسب اپیزود

۲.۱ قسمت ب

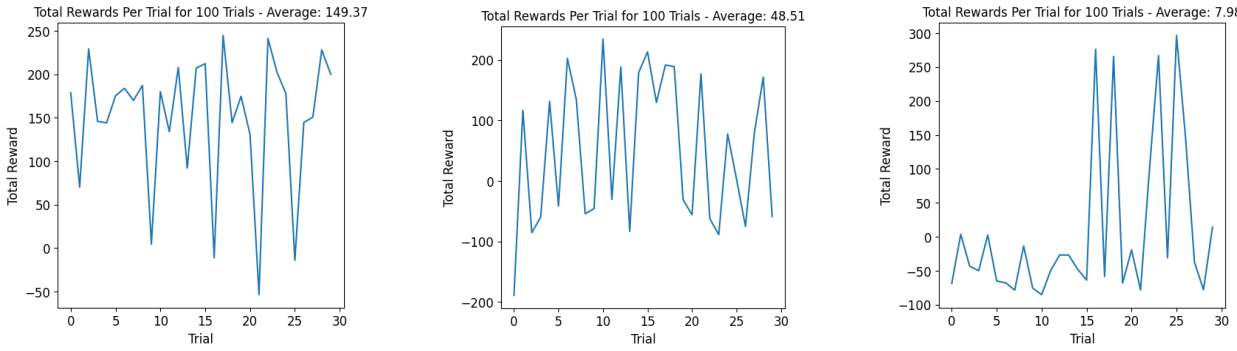
۱.۲.۱ تاثیر **batch size**



(آ) با batch size ۳۲ (ب) با batch size ۶۴ (ج) با batch size ۱۲۸

شکل ۲: نتایج پاداش تجمیعی تمرین به ازای ۴۰۰ اپیزود

همانگونه که از شکل مشهود است، افزایش batch size موجب کمتر شدن نویز شده است، همچنین افزایش آن موجب افزایش سرعت همگرایی شده است.



(آ) با batch size ۳۲ (ب) با batch size ۶۴ (ج) با batch size ۱۲۸

شکل ۳: نتایج پاداش تجمیعی با ۳۰ اپیزود برای تست مدل

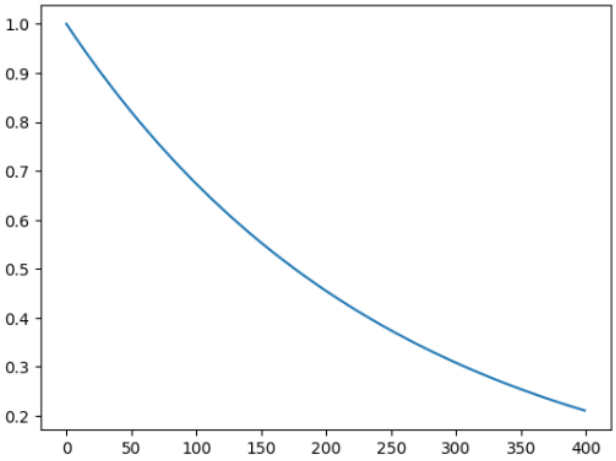
با توجه به تست گرفتن با batch size های مختلف روی ۳۰ اپیزود و گرفتن میانگین آنها مشهود است که افزایش batch size موجب بهبود عملکرد مدل درشرایط یکسان شده است. همچنین بصورت شهودی از نمودار ها مشهود است که batch sizeبرابر با ۱۲۸ به بهترین حالت واقعی خود نزدیک تر است در نتیجه از لحاظ معیار regret نیز بهتر عملکرده است و کمتر در بهینه های محلی گیر افتاده است.

به طور کلی، افزایش batch size می‌تواند منجر به افزایش سرعت همگرایی شود، اما این موضوع به شرایط خاص و مدل مورد نظر بستگی دارد. نوسانات کوچکی که در گرادیان‌های محاسبه‌شده با batch size کوچک وجود دارند، می‌توانند مدل را از بهینه های محلی نجات دهند. با افزایش batch size، این نوسانات کاهش یافته و ممکن است مدل در بهینه های محلی گیر کند.

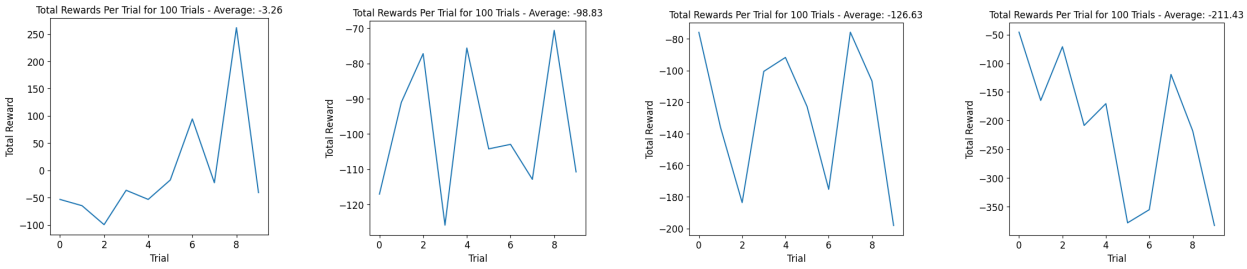
برای دانلود مدل ها و فیلم های این بخش، اینجا کلیک کنید.

۲.۲.۱ عملکرد مدل در اپیزودهای ۵۰، ۱۰۰، ۱۵۰، ۲۰۰، ۲۵۰، ۳۰۰، ۳۵۰

با توجه به نمودار اپسیلون بر حسب اپیزود توقع داریم مدل در ابتدا گردش کند و در انتها به سمت هدف حرکت کند. درواقع هر چه اپیزود های نهایی نزدیک میشویم، مدل شانس کمتری برای گردش پیدا میکند.



شکل ۴: مقدار اپسیلون بر حسب اپیزود



(آ) اپیزود ۵۰ (ب) اپیزود ۱۰۰ (ج) اپیزود ۱۵۰ (د) اپیزود ۲۰۰

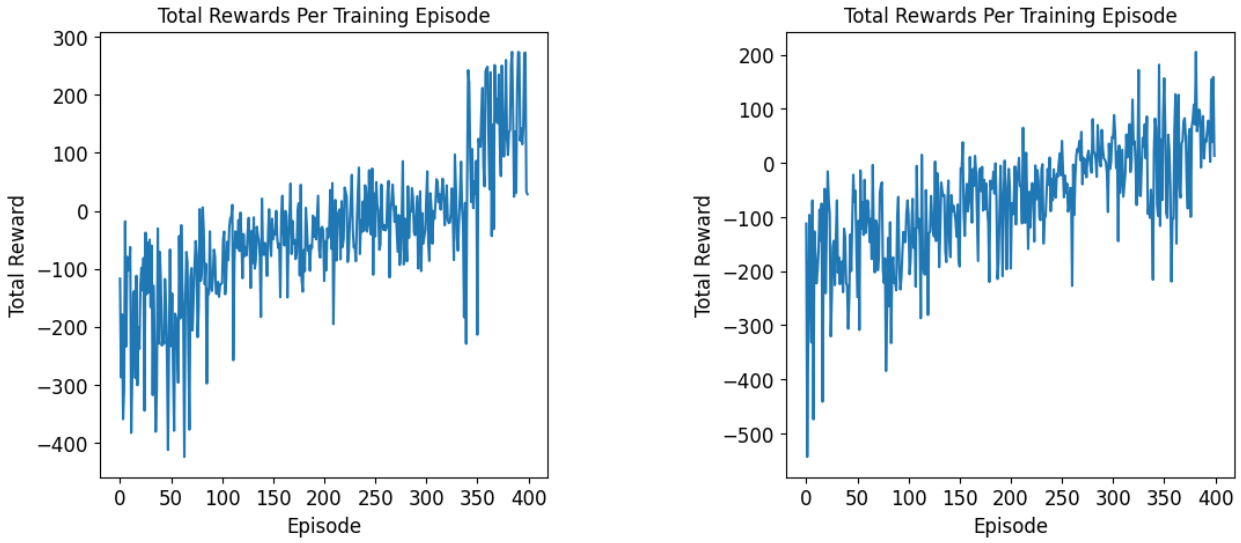
شکل ۵: نتایج پاداش تجمیعی با ۳۰ اپیزود برای تست مدل

همانگونه که مشهود است هر چه به اپیزود های پایانی نزدیک میشویم پاداش تجمیعی بیشتری نیز روی تست بدست می‌اوریم دلیل این امر گردش و سرچ کردن مدل در اپیزود های اول است.

برای دانلود مدل و فیلم ها اینجا کلیک کنید.

۳.۱ مقایسه DDQN با DQN

۱.۳.۱ مقایسه کلی آموزش



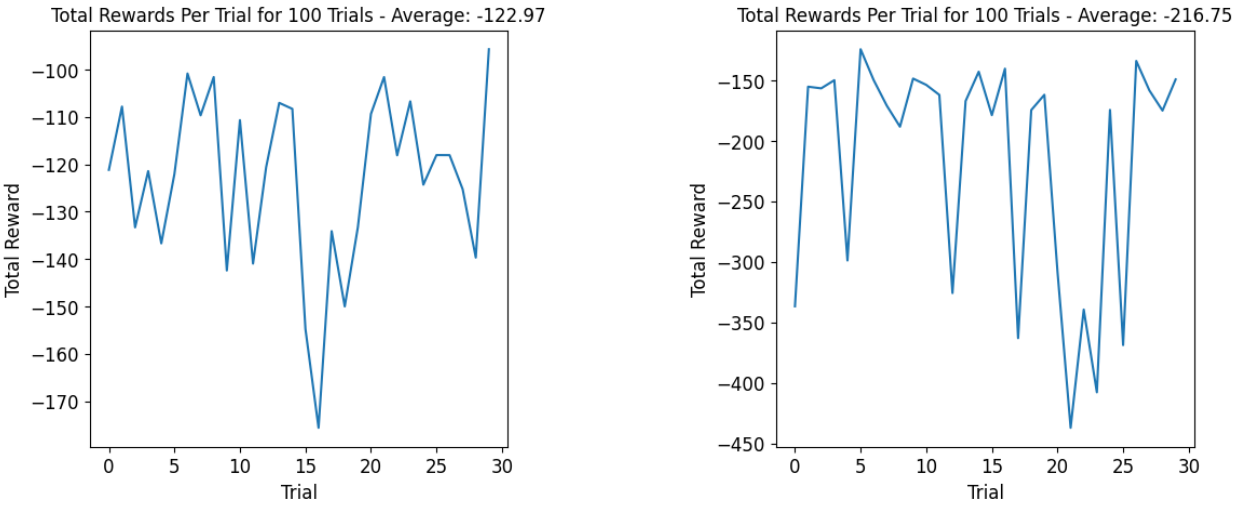
(آ) روش DQN

(ب) روش DDQN

شکل ۶: مقایسه پاداش تجمیعی در روند آموزش در ۴۰۰ اپیزود

همانگونه که مشهود است مدل DDQN در اپیزود های اولیه امتیاز های کمتری گرفته است ولی هرچی به اپیزود نهایی نزدیک میشویم مدل بهتر و به پاداش های خیلی زیاد میرسد. که نشان میدهد مدل DDQN توانسته است بهتر از DQN در کل عمل کند.

۲.۳.۱ مقایسه با اپیزود ۱۰۰



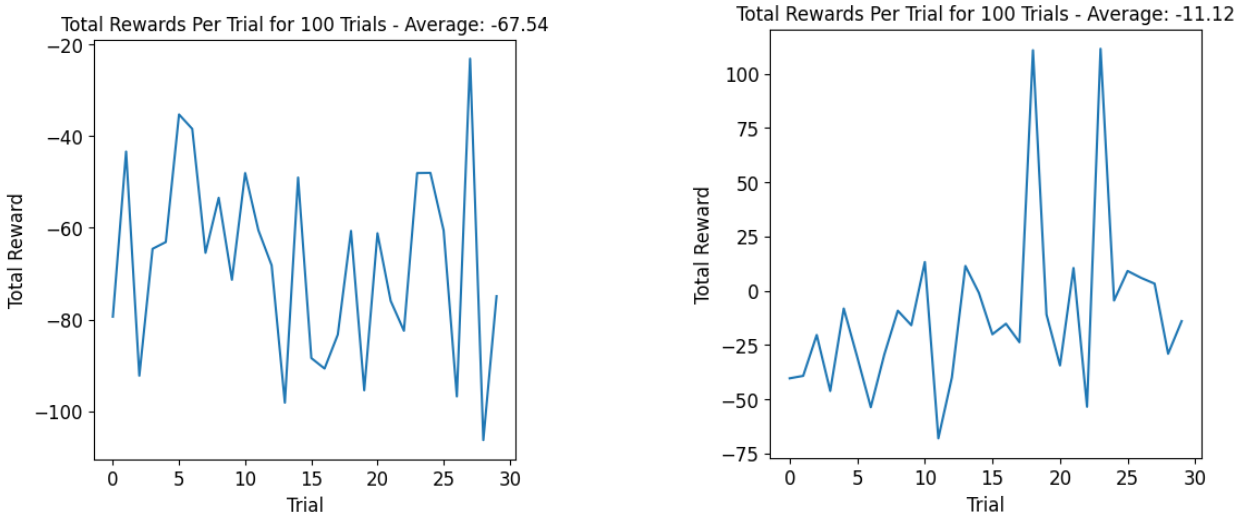
(آ) روش DQN

(ب) روش DDQN

شکل ۷: مقایسه عملکرد دو مدل در اپیزود ۱۰۰ روی ۳۰ اپیزود تستی

با تست گرفت مدل با اپیزود ۱۰۰ و تست گرفتن با ۳۰ اپیزود مشهود است میانگین پاداش تجمعی در روش DDQN بیشتر از DQN است که نشان میدهد مدل DDQN بهتر عملکرده است.

۳.۳.۱ مقایسه با اپیزود ۲۵۰



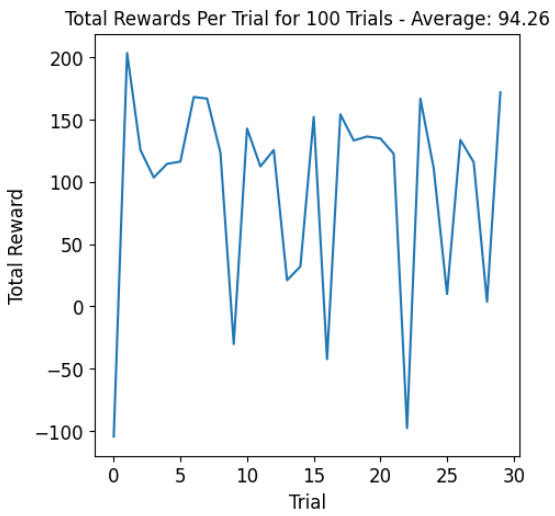
(آ) روش DQN

(ب) روش DDQN

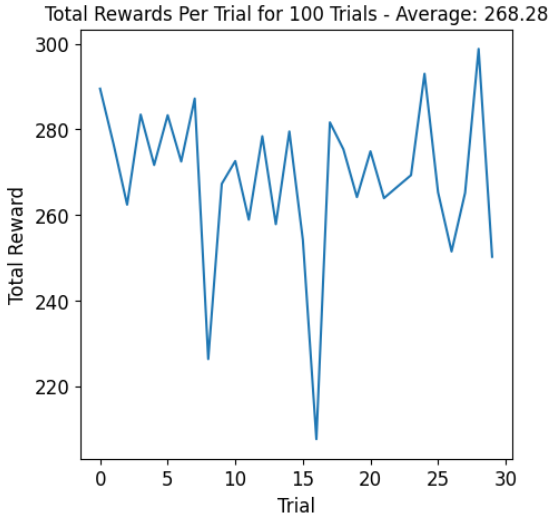
شکل ۸: مقایسه عملکرد دو مدل در اپیزود ۲۵۰ روی ۳۰ اپیزود تستی

با تست گرفت مدل با اپیزود ۲۵۰ و تست گرفتن با ۳۰ اپیزود مشهود است میانگین پاداش تجمعی در روش DDQN بیشتر از DQN است که نشان میدهد مدل DDQN بهتر عملکرده است.

۴.۳.۱ مقایسه با بهترین حالت در ۴۰۰ اپیزود



(آ) روش DQN



(ب) روش DDQN

شکل ۹: مقایسه عملکرد دو مدل در اپیزود ۴۰۰ روی ۳۰ اپیزود تستی

در کل نیز مدل DDQN خیلی بهتر از DQN عمل کرده است، مشهود است که میانگین پاداش آن ۲۶۸ است که اختلاف قابل توجهی با روش DQN دارد. دلیل این امر در بخش اول کامل توضیح داده شده است.

برای دانلود فیلم ها و مدل ها اینجا کلیک کنید.