ابتدا SARSA را هم توضیح میدهیم سپس نمودار ها را بررسی میکنیم:

SARSA:

الگوریتم SARSA

1: Initialize learning rate α
2: Initialize ε
3: Randomly initialize the network parameters θ
4: **for** $m = 1 \ldots MAX\_STEPS$ **do**
5: Gather N experiences $(s_i, a_i, r_i, s'_i, a'_i)$ using the current ε-greedy policy
6: **for** $i = 1 \ldots N$ **do**
7: # Calculate target Q-values for each example
8: $y_i = r_i + \delta_{s'_i} \gamma \, Q^{\pi_\theta}(s'_i, a'_i)$ where $\delta_{s'_i} = 0$ if s'_i is terminal, 1 otherwise
9: **end for**
10: # Calculate the loss, for example using MSE
11: $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$
12: # Update the network's parameters
13: $\theta = \theta - \alpha \nabla_\theta L(\theta)$
14: Decay ε
15: **end for**

همانند انچه در این الگوریتم دیده میشود، دیگر از replay_momory نباید استفاده کرد. صرفا تعدادی state, action, reward, next state, next action را جمع آوری میکنیم، و آن ها را با شبکه اصلی و تارگت خود بررسی میکنیم و پس از backpropagation، دیگر برخلاف DQN، از آنها استفاده مجدد نمیکنیم.
پس از پیاده سازی SARSA بدلیل به نتیجه نرسیدن، تحلیل های بسیاری انجام شد که پس از توضیح کد ها به سراغ آنها میرویم. همانند الگوریتم ذکر شده:

ابتدا یک حافظه میسازیم که صرفا این بخش از الگوریتم را پیاده سازی میکند:

Gather and store h experiences (s_i, a_i, r_i, s'_i) using the current policy

```python
class Memory:
    def __init__(self, capacity):
        # Initialize replay memory with a specified capacity
        self.capacity = capacity
        self.states = deque(maxlen=capacity)
        self.actions = deque(maxlen=capacity)
        self.next_states = deque(maxlen=capacity)
        self.next_actions = deque(maxlen=capacity)
        self.rewards = deque(maxlen=capacity)
        self.dones = deque(maxlen=capacity)

    def store(self, state, action, next_state, next_action, reward, done):
        # Store an experience (state, action, next_state, reward, done) in the replay memory
        self.states.append(state)
        self.actions.append(action)
        self.next_states.append(next_state)
        self.next_actions.append(next_action)
        self.rewards.append(reward)
        self.dones.append(done)
```

```python
    def get_all(self):
        # Get all the contents of memory
        states = torch.stack([torch.as_tensor(state, dtype=torch.float32, device=device) for state in self.states])
        actions = torch.as_tensor(self.actions, dtype=torch.long, device=device)
        next_states = torch.stack([torch.as_tensor(next_state, dtype=torch.float32, device=device) for next_state in self.next_states])
        next_actions = torch.stack([torch.as_tensor(next_action, dtype=torch.long, device=device) for next_action in self.next_actions])
        rewards = torch.as_tensor(self.rewards, dtype=torch.float32, device=device)
        dones = torch.as_tensor(self.dones, dtype=torch.bool, device=device)
        return states, actions, next_states, next_actions, rewards, dones

    def __len__(self):
        # Return the current size of the memory
        return len(self.dones)

    def clear(self):
        # Clear all the deque objects
        self.states.clear()
        self.actions.clear()
        self.next_states.clear()
        self.next_actions.clear()
        self.rewards.clear()
        self.dones.clear()
```

همانطور که مشاهده میکنیم همیشه پس از دریافت محتوای حافظه، همه آن را با دستور clear پاک میکنیم.

```python
class SARSA_Network(nn.Module):

    def __init__(self, num_actions, input_dim):
        super(SARSA_Network, self).__init__()

        # Define the neural network layers
        self.FC = nn.Sequential(
            nn.Linear(input_dim, 512),  # Input layer to first hidden layer with 512 nodes
            nn.ReLU(),                  # ReLU activation for non-linearity
            nn.Linear(512, 256),        # Second hidden layer with 256 nodes
            nn.ReLU(),                  # ReLU activation for non-linearity
            nn.Linear(256, 64),         # Third hidden layer with 64 nodes
            nn.ReLU(),                  # ReLU activation for non-linearity
            nn.Linear(64, num_actions) # Output layer with 'num_actions' nodes
        )
        self._initialize_weights()

    def _initialize_weights(self):

        for m in self.modules():
```

این بخش ساخت مدل، همانند DQN است.

در بخش SARSA_agent، به سراغ توضیح بخش های تغییرداده شده نسبت به DQN میرویم:

```python
def learn(self, done):

    if self.memory.__len__() > self.memory.capacity:
        return

    states, actions, next_states, next_actions, rewards, dones = self.memory.get_all()
    actions = actions.unsqueeze(1)
    rewards = rewards.unsqueeze(1)
    dones = dones.unsqueeze(1)
    next_actions = next_actions.unsqueeze(1)

    predicted_q = self.main_network(states)
    predicted_q = predicted_q.gather(dim=1,index=actions)

    with torch.no_grad():
        next_q_values = self.target_network(next_states)
        next_target_q_value = next_q_values.gather(dim=1, index=next_actions)

    next_target_q_value[dones] = 0  # Set the Q-value for terminal states to zero
    y_js = rewards + (self.discount * next_target_q_value)  # Compute the target Q-values
    loss = self.critertion(predicted_q, y_js)  # Compute the loss
```

تفاوت این بخش با DQN، استفاده از q در s’ با استفاده از a’ یا همان next_state است. در حالی که در بخش قبلی DQN از ماکسیمم استفاده میکردیم:

```
# Compute the maximum Q-value for the next states using the target network
with torch.no_grad():
    next_target_q_value = self.target_network(next_states).max(dim=1, keepdim=True)[
        0]  # not argmax (cause we want the maxmimum q-value, not the action that maximize it)
```

در بخش train_test:

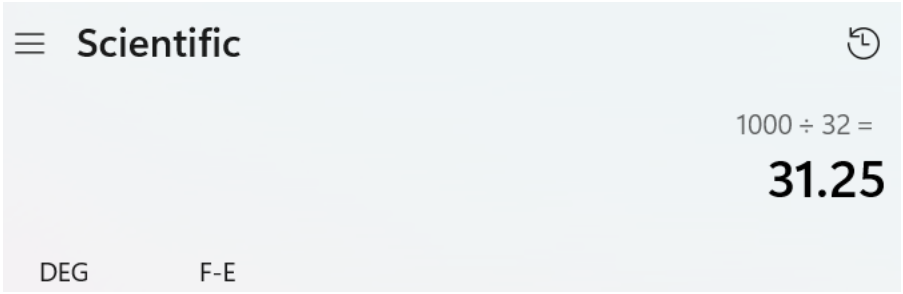
```
episode_reward = 0
action = self.agent.select_action(state)
while not done and not truncation:
    next_state, reward, done, truncation, _ = self.env.step(action)

    next_state = self.state_preprocess(next_state, num_states=self.num_states)
    next_action = self.agent.select_action(next_state)

    angle = next_state[2]  # Assuming next_state[2] is the pole angle
    angle_reward = 1.0 - abs(angle) / np.pi  # Normalize and invert the angle to get reward
    reward += angle_reward  # Modify reward with additional reward shaping
    x_pos = abs(next_state[0])
    if x_pos > 4:
        reward -= x_pos
    elif x_pos > 2:
        reward -= x_pos * 0.5
    if (reward > 0):
            reward /= 2
    self.agent.memory.store(state, action, next_state, next_action, reward, done)

    if self.agent.memory.__len__() >= self.memory_capacity and sum(self.reward_history) > 0:
        self.agent.learn((done or truncation))
```

برای انتخاب action بعدی از سیاست فعلی استفاده میکنیم، و هربار این وقایع را در مموری ذخیره میکنیم. اگر حافظه پر شده باشد، شروع به یادگیری این رخداد ها میکنیم،

```
if self.agent.memory.__len__() >= self.memory_capacity and sum(self.reward_history) > 0:
    self.agent.learn((done or truncation))

    # Update target-network weights
    if total_steps % self.update_frequency == 0:
        self.agent.hard_update()

    self.agent.memory.clear()

state = next_state
action = next_action
episode_reward += reward
step_size += 1
```

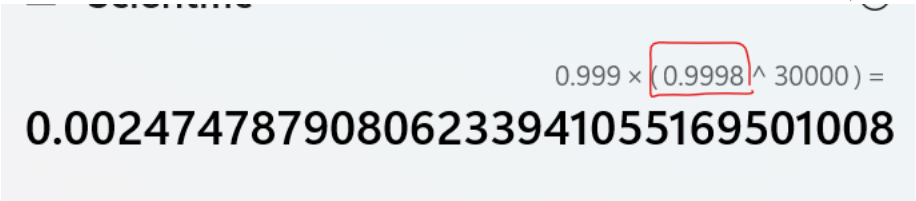
و پس از یادگیری، دوباره حافظه را خالی میکنیم.

پس از بررسی همه این موارد، شروع به یادگیری با استفاده از SARSA کردیم، اما نتایج پس از تعداد اپیزود های برابر با DQN، بسیار ضعیف بود.
البته که DQN، چون از حافظه استفاده میکند، هربار پس از رخ دادن یک SARSA، یکبار دیگر دچار اپدیت میشود. درحالی که چون در SARSA باید منتظر پر شدن حافظه باشیم، پر کردن ان طولانی میشود، پس حتما به تعداد اپیزود های بسیار بیشتری از DQN نیاز دارد. در واقع تعداد دفعاتی که DQN دچار اپدیت میشود برابر است با:
max_episode – batch_size (صرفا در مرتبه اول منتظر میشود که حافظه اش به اندازه یک batch پر شود، پس از آن دائما دچار اپدیت میشود)
= ۹۶۸
که این تعداد اپدیت ها، تعداد زیادی رویداد تکراری دارد (به اندازه حافظه ممور‌ی ما که برابر با ۴۰۰۰ است – بسیار بیشتر از تعداد تجربه ها – و هر بار به اندازه یک batch از آن را لرن میکنیم).
حال اگر با ۱۰۰۰ اپیزود در SARSA منتظر به جواب رسیدن بودیم، درواقع باید انتظار داشتیم که



پس از 31 بار اپدیت، به نتیجه برسد، که کاملا ناعالانه بود.
رویکرد دیگر، استفاده مجدد از داده ها و درواقع replay_memory مانند DQN است که پس از بررسی ها، متوجه میشدیم تا حد خوبی به جواب میرسد، اما این کار خودش را از ذات الگوریتم SARSA کاملا جدا میکند.
پس تنها موجود ما، افزایش تعداد max_episode به همین روش، افزایش نرخ کاهش، به طریقی بتواند در میان یادگیری، از سیاست های exploration استفاده کند، بود.

پس max_episode را برابر 30 000 قرار داده، و برای epsilon_decay داریم:



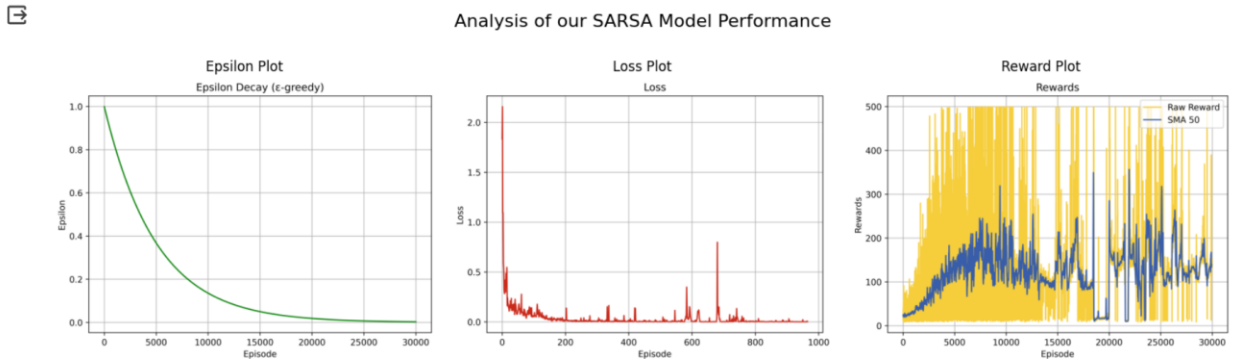
پس 0.9998 میتواند مقدار مناسبی باشد.

حال با همه این بررسی ها:

```python
def set_hyperparameters(train_mode):
    render = not train_mode
    RL_hyperparams = {
        "train_mode": train_mode,
        "RL_load_path": f'./final_weights_{"1000"}.pth',
        "save_path": f'./final_weights',
        "early_stop_threshold": 496,
        "early_stop_window": 30,
        "save_interval": 15000,
        "report": 250,
        "clip_grad_norm": 3,
        "learning_rate": 6e-4,
        "discount_factor": 0.8,
        "update_frequency": 1,
        "max_episodes": 30000 if train_mode else 1,
        "max_steps": 200,
        "render": render,
        "epsilon_max": 0.999 if train_mode else -1,
        "epsilon_min": 0.001,
        "epsilon_decay": 0.9998,
        "memory_capacity": 32 if train_mode else 0,
        "map_size": 16,
        "num_states": 8 ** 2,
```

هایپر پارامتر هارا انتخاب میکنیم که یادگیری طولانی داشته باشیم!:

```
load_and_display_figures(main_title, epsilon_path, loss_path, reward_path)
```



پس از یادگیری، متوجه میشویم که گرچه مدل بتواند به نتیجه برسد، اما میانگین عملکرد آن، باز هم ضعیف تر از مدل DQN ما میباشد. DQN از حافظه خود استفاده میکند و چون هرکدام را صرفا سمپل کرده و دوباره برمیگرداند، احتمال انتخاب دوباره رویدار ها وجود دارد و این گونه قبلا آن ها را دیده است، با رخ دادن چندباره این اتفاق، سعی در کاهش لاس خود با داده های از پیش دیده شده دارد.
اما SARSA داده ها را همیشه دور میاندازد و تجربه هایش همیشه برایش تازه است اما همین تازگی میتواند مشکلاتی را برایش ایجاد کند. از طرفی چون learning_rate ها معمولا زیاد نیست، از همان تجربه ها هم، نمیتواند به طور کامل یادگیری مناسبی داشته باشد. شاید اگر این مسئله نیز کنترل میشد (که همان داده هایی که یکبار میبند را خوب اموزش ببیند – یا چندبار روی آن یادگیری انجام شود و سپس دور انداخته شود) آنگاه SARSA در برابر حالت تست هایی که قبلا تجربه نشده اند، بهتر از DQN عمل میکرد.

توجه داریم که اگر در SARSA نیز از حافظه استفاده میشد که داده های قبلی را ببیند، بدون شک به جواب مناسبی میرسید اما از این هدف الگوریتم دور است.

ویدیو این بخش نیز در تایتل SARSA و سپس سابتایتل visualizing وجود دارد و لینک کولب قابل مشاهده است.

```
print("Testing the SARSA agent: ")
RL_hyperparameters = set_hyperparameters(train_mode=False)
handler(train_mode=False, RL_hyperparameters=RL_hyperparameters)
```

```
Testing the SARSA agent:
Episode: 1, Steps: 145, Reward: 145.00,
Moviepy - Building video temp-{start}.mp4.
Moviepy - Writing video temp-{start}.mp4

t:  86%|████████  | 126/147 [00:00<00:00, 191.66it/s, now=None]WARNING:py.warnings:/usr/local/lib/python3.10/dist-packages/moviepy/video/io/f
  warnings.warn("Warning: in file %s, "%(self.filename)+

Moviepy - Done !
Moviepy - video ready temp-{start}.mp4
```

Moviepy - video ready temp-{start}.mp4