COMP 257 SEC 003

Assignment #6: “Reinforcement Learning”

Group Omega:

- Andy Bandela 301282674
- Juan Arevalo 301323792
- Rasheed Adeyemi 301308878
- Mahfuzur Rahman 301336576
- Anthony Mariadas 301234143
- Mariela Ramos Vila 301324510
- Dinh Hoang Viet Phuong 301123263

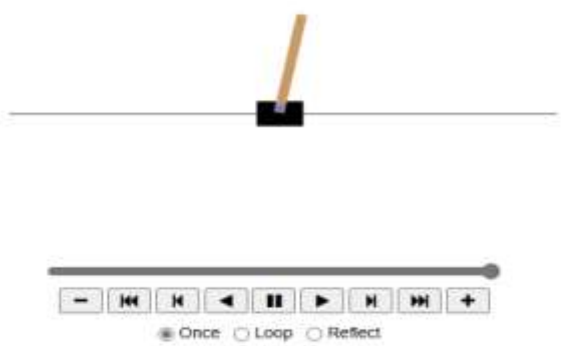
Use Reinforcement learning to train an agent that can successfully move the cartpole to the left using OpenAI Gym’s CartPole environment.

1. Using Neural Network Policy to move the CartPole to left direction

In [13]:

```
# Set a random seed for reproducibility (CPU)
tf.random.set_seed(42)

# Define a simple neural network policy
model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])

# Function for the neural policy to decide the action
def neural_policy(obs):
    left_proba = model.predict(obs[np.newaxis], verbose=0)
    return int(np.random.rand() > left_proba)

np.random.seed(42)
show_one_episode(neural_policy)
```

Out[13]:



2. Implement the Policy Gradient algorithm to the above scenario

- Policy gradient Function:

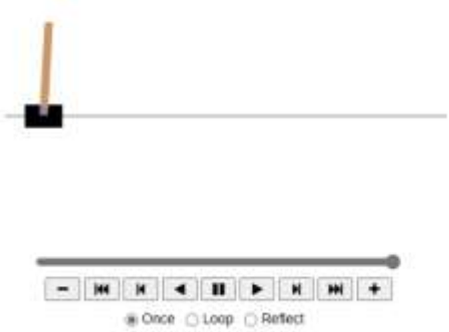
```
In [14]:
# Function to choose action based on policy model
def policy_gradient(obs):
    left_proba = model_pg.predict(obs[np.newaxis], verbose=0)

    return int(np.random.rand() > left_proba)
np.random.seed(42)
```

- Play one stop Function:
  Compared with the code that we work on class; two changes were done. First, the target label was changed, it was updated to encourage move left, and second, a condition for reward was implemented, if the action is 0, which means go to left, the reward is incremented in 2.

```
In [15]:
# Function to play one step using the model and return the gradients
def play_one_step(env, obs, model_pg, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model_pg(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba).numpy()

        # Update the target label to encourage moving left
        target_label_left = tf.constant([[1.]])
        y_target = target_label_left - tf.cast(action, tf.float32)

        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model_pg.trainable_variables)
    obs, reward, done, truncated, info = env.step(int(action))

    # Adjust the reward based on the desired behavior (moving left)
    if action == 0:  # Action 0 corresponds to moving left
        reward += 2  # Positive reward for moving left

    return obs, reward, done, truncated, grads
```

- After 180 iterations training the model, the agent moves to left .

```
In [20]:
# Show an episode using the trained neural policy
np.random.seed(42)
show_one_episode(policy_gradient)

Out[20]:
```



3. Implement Markov Decision Process defining the random transition probabilities, rewards and possible actions to move the CartPole towards left

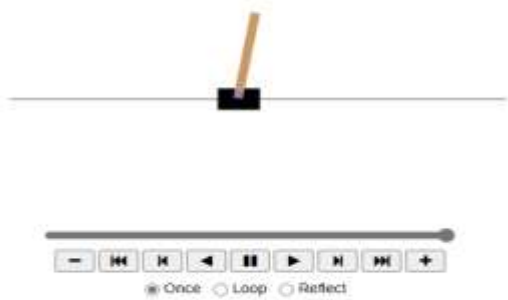
- A Discretize state Function was implemented for the CartPole environment adeptly handles continuous state observations through the discretize_state function.
- The CartPoleMDP class encapsulates key MDP components, defining discrete actions, deterministic state transitions in get_next_state, and reward computation in get_reward.
- The mdp_policy consistently directs the CartPole to move left.

In [21]:
```
# Discretize the state space
def discretize_state(obs):
    # Simplified discretization: consider only the position and velocity of t
    cart_pos, cart_vel, pole_angle, pole_vel = obs
    return (round(cart_pos, 1), round(cart_vel, 1))
```

In [22]:
```
# Define the MDP model
class CartPoleMDP:
    def __init__(self):
        self.actions = [0, 1]  # 0: left, 1: right
        self.state_transition = {}  # Transition probabilities
        self.rewards = {}  # Rewards

    def get_next_state(self, current_state, action):
        # Simplified transition model
        # In reality, this would be based on the physics of the CartPole
        next_state = (current_state[0] + (0.1 if action == 1 else -0.1), curr
        return next_state

    def get_reward(self, current_state, action):
        # Higher reward for moving left or staying still
        return 1 if action == 0 else 0

    def step(self, current_state, action):
        next_state = self.get_next_state(current_state, action)
        reward = self.get_reward(current_state, action)
        return next_state, reward

# Implement the MDP policy
def mdp_policy(state, mdp_model):
    # Simple policy: always move left
    return 0  # Action 0 corresponds to moving left
```

In [24]:
```
# Create the CartPole environment and MDP model
env = gym.make("CartPole-v1", render_mode="rgb_array")
mdp_model = CartPoleMDP()

# Visualize the CartPole using the MDP policy
np.random.seed(42)
show_mdp(mdp_policy, mdp_model, env)
```

Out[24]:



4. **Implement Deep Q Learning (Epsilon Greedy Policy) to move the CartPole towards left.**

- Epsilon Greedy Policy Function:

In [25]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility on the CPU

input_shape = [4]  # == env.observation_space.shape
n_outputs = 2  # == env.action_space.n

model_dq = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])


optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error


def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)  # random action
    else:
        Q_values = model_dq.predict(state[np.newaxis], verbose=0)[0]
        return np.argmax(Q_values)  # optimal action according to the DQN
```

5. **Train the agent on the Cartpole environment for a sufficient number of episodes to achieve a satisfactory level of performance.**

- Play one stop Function:
  Compared to the code we worked on in class, we implemented a condition for the reward, if the action is 0, which means go left, the reward is increased by 2, otherwise it is decreased by 1.

In [26]:
```
def play_one_step_dq(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)


    next_state, reward, done, truncated, info = env.step(action)

     # Adjust the reward based on the desired behavior (moving left)
    if action == 0:  # Action 0, corresponds to moving left
        reward += 2  # Positive reward for moving left
    else:
        reward -= 1

    replay_buffer.append((state, action, reward, next_state, done, truncated)

    return next_state, reward, done, truncated, info
```

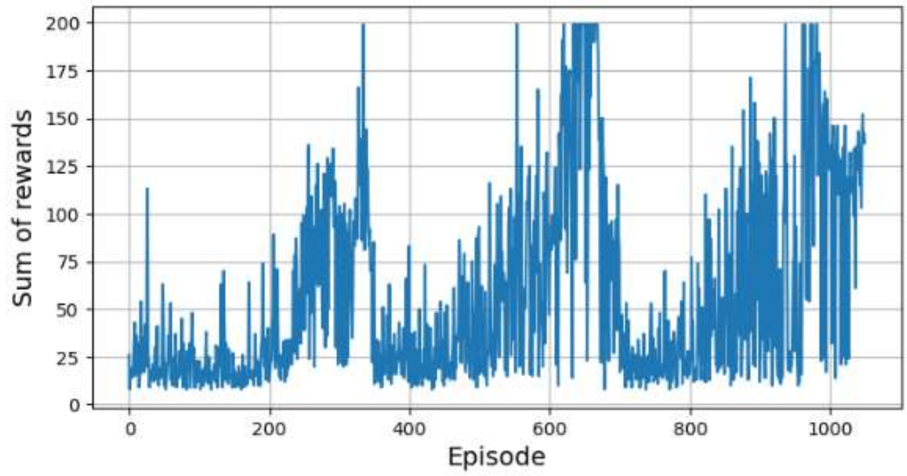
- The discount factor is 0.99 because we prioritize long-term rewards over short-term ones. This means that future rewards are considered with a high weight, promoting the agent to prioritize accumulating more cumulative rewards over time.

```
In [28]:
batch_size = 32
discount_factor = 0.99

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones, truncateds = experiences
    next_Q_values = model_dq.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds)  # episode is not done or truncated
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model_dq(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model_dq.trainable_variables)
    optimizer.apply_gradients(zip(grads, model_dq.trainable_variables))
```

- The loop runs for 350 episodes, and the choice of 500 in the epsilon-greedy strategy influences the rate at which the agent shifts from exploration to exploitation during training. It allows the agent to explore more in the early stages of training when it has less information about the environment, and gradually shifts towards exploitation as it gains more experience.

```
In [35]:
# Training Loop
for episode in range(350):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step_dq(env, obs, epsi
        if done or truncated:

            break

    # extra code – displays debug info, stores data for the next figure, and
    #              keeps track of the best model weights so far
    print(f"\rEpisode: {episode + 1}, Steps: {step + 1}, eps: {epsilon:.3f}",
          end="")
    rewards.append(step)
    if step >= best_score:
        best_weights = model_dq.get_weights()
        best_score = step

    if episode > 50:
        training_step(batch_size)

model_dq.set_weights(best_weights)  # extra code – restores the best model we

Episode: 350, Steps: 138, eps: 0.302
```

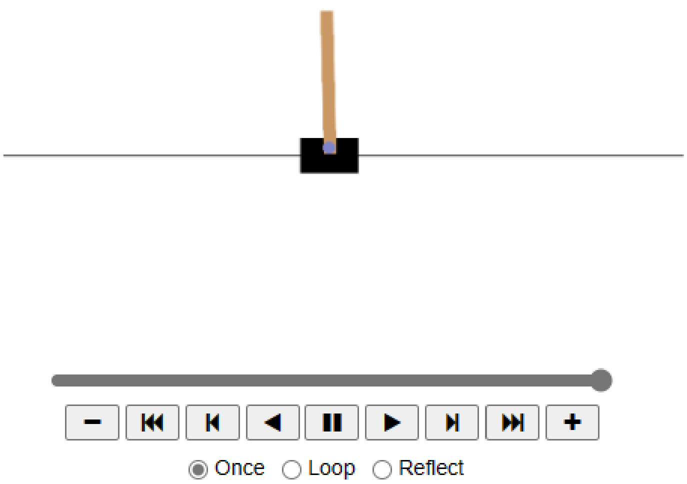
In [36]:
```
# extra code – this cell generates and saves Figure 18-10
plt.figure(figsize=(8, 4))
plt.plot(rewards)
plt.xlabel("Episode", fontsize=14)
plt.ylabel("Sum of rewards", fontsize=14)
plt.grid(True)
#save_fig("dqn_rewards_plot")
plt.show()
```



- The agent start to move left.

In [34]:
```
# shows an animation of the trained DQN playing one episode
show_one_episode(epsilon_greedy_policy)
```

Out[34]:



In [ ]:

6. **Discuss the challenges faced during training and potential strategies for further improving the agent's performance.**

- Reward Shaping:
  Designing an effective reward function that encourages the desired behavior without unintended consequences is difficult. In this case, encouraging the pole to lean left without it falling can be tricky.

- Exploration vs. Exploitation:
  Balancing exploration (trying new actions) and exploitation (using known information) is critical. Too much exploration can lead to slow learning, while too much exploitation can cause the agent to miss better strategies.

- Stability and Convergence:
  Ensuring stable learning and convergence to an optimal policy is difficult, especially in environments with high variance or complex dynamics.

- Sample Efficiency:
  RL agents often require a large number of episodes to learn effectively, which can be computationally expensive.

- Network Architecture and Hyperparameters:
  Choosing the right network architecture and hyperparameters is often a trial-and-error process.

- Generalization and Overfitting:
  The agent might overfit to specific scenarios or fail to generalize well across the state space.

- Evaluation Metrics:
  Determining the best metrics to evaluate the agent's performance can be non-trivial, especially for modified tasks.
  Improving the performance of an RL agent is often an iterative process that requires careful tuning of the reward function, exploration strategies, network architecture, and hyperparameters. Regular evaluation and a deep understanding of the environment's dynamics are key to guiding these adjustments.