Lecture 5 Reinforcement Learning

Deep Reinforcement Learning: Notes

These notes are based on the provided presentation and transcript on Deep Reinforcement Learning, aiming to synthesize the information and explain key concepts.

Introduction to Reinforcement Learning (RL)

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment. The core idea is to build an interaction between a model (the agent) and its environment, allowing the model to learn dynamically. Unlike other machine learning approaches, RL often aims to learn without explicit supervision or human labels, or with as few as possible.

Goal of RL: The primary goal in RL is for the agent to learn a strategy, known as a policy, to maximize a cumulative reward signal over many time steps.

Comparison with other Learning Paradigms:

- **Supervised Learning:**
  - **Data:** Uses labeled data, typically pairs of input (x) and corresponding output/label (y).
  - **Goal:** To learn a function that maps input x to output y.
  - **Example:** Given images of apples labeled "apple," the model learns to identify new images of apples.
- **Unsupervised Learning:**
  - **Data:** Uses unlabeled data (only input x).
  - **Goal:** To learn the underlying structure or patterns in the data.
  - **Example:** Given many images of different fruits without labels, the model learns to group similar-looking fruits together, identifying that a new apple is like other apples it has seen.
- **Reinforcement Learning:**
  - **Data:** Involves state-action pairs and the rewards received from the environment.
  - **Goal:** To learn a policy that maximizes cumulative future rewards over time by choosing optimal actions in different states.
  - **Example:** An agent (e.g., a person) learns that eating an apple (action) when hungry (state) provides nutrition and helps survival.

RL moves beyond learning from pre-collected static datasets to learning through active interaction in a dynamic environment.

Key Concepts in RL

The RL framework involves several core components:

- **Agent:** The learner or decision-maker that interacts with the environment by taking actions. This could be a robot, a character in a game (like Mario), or even a self-driving car.

- **Environment:** The external world in which the agent exists, operates, and makes decisions.
- **State (S_t):** A representation of the environment at a particular time t. It's the observation the agent receives from the environment. For example, in a game, the state could be the current screen image.
- **Action (A_t):** A move the agent can make in the environment at time t.
    - **Action Space ($\mathcal{A}$):** The set of all possible actions an agent can take. This can be discrete (e.g., up, down, left, right) or continuous (e.g., the steering angle of a car).
- **Reward (R_t):** A scalar feedback signal that the environment provides to the agent after it takes an action in a particular state. It indicates how good or bad the action was in terms of achieving the agent's goal.
    - **Return (Total Discounted Reward, G_t):** The agent's objective is to maximize the total accumulated reward over an entire "lifetime" or episode. Often, future rewards are discounted because immediate rewards can be more valuable. The discounted return is typically calculated as:

    $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

    where γ (gamma) is the discount factor ($0 \le \gamma \le 1$). A γ closer to 0 makes the agent prioritize immediate rewards, while a γ closer to 1 makes it value future rewards more.
- **Policy (π):** The agent's strategy or behavior function. It maps a given state to an action (or a probability distribution over actions). The policy defines how the agent acts.
    - **Deterministic Policy:** $\pi(s) = a$ (directly outputs an action for a state).
    - **Stochastic Policy:** $\pi(a|s) = P[A_t = a|S_t = s]$ (outputs a probability distribution over actions for a state).

Approaches to Reinforcement Learning

There are two main approaches to learning in RL:

1. **Value-Based Learning (e.g., Q-Learning):** Learn a value function that estimates how good it is to be in a particular state or to take a particular action in a state. Then, use this value function to select actions.
2. **Policy-Based Learning (e.g., Policy Gradients):** Directly learn the policy function that maps states to actions (or action probabilities) without necessarily learning a value function first.

1. Q-Learning (Value-Based)

Q-learning focuses on learning a specific value function called the **Q-function** (or action-value function).

- **Q-function ($Q^\pi(s, a)$):** Represents the expected total future discounted reward (return) an agent can achieve by taking action 'a' from state 's', and thereafter following policy π.

  $$Q^\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$$

  The goal is often to find the optimal Q-function, $Q^*(s, a)$, which gives the maximum expected return if the agent takes action 'a' in state 's' and then follows the optimal policy thereafter.
- **Deriving an Optimal Policy from $Q^*(s, a)$:** If we have the optimal Q-function, $Q^*(s, a)$, we can easily derive the optimal policy, $\pi^*(s)$, by choosing the action that maximizes the Q-value for the current state:

$$\pi^*(s) = \text{argmax}_a Q^*(s,a)$$

- **Deep Q-Networks (DQNs):** When the state space is large or continuous (e.g., images from a game), deep neural networks can be used to approximate the Q-function. This is known as a Deep Q-Network (DQN).
    - The DQN takes the current state (e.g., a game frame) as input and outputs the Q-values for all possible discrete actions.
    - **Training DQNs (Q-Loss):** DQNs are trained by minimizing a loss function. The Bellman equation provides the foundation for this. The target Q-value for a state-action pair (s,a) is the immediate reward r plus the discounted maximum Q-value of the next state s':

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

      where θ^- are the parameters of a target network (often a periodically updated copy of the online network) for stability.
      The loss function (often Mean Squared Error) is the difference between this target Q-value and the Q-value predicted by the network:

      $$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

      This difference is known as the Temporal Difference (TD) error. The network's weights are updated using backpropagation to minimize this loss.
- **DQN Process Example (Atari Breakout):**
    1. The agent (paddle) observes the current state (game screen).
    2. The DQN processes this state and outputs Q-values for each possible action (e.g., move left, move right, stay).
    3. The agent selects the action with the highest Q-value (or uses an exploration strategy like epsilon-greedy).
    4. The agent performs the action, and the environment provides a new state and a reward (e.g., points for breaking blocks).
    5. This experience (state, action, reward, next state) is stored in a replay buffer.
    6. The DQN is trained by sampling mini-batches of experiences from the replay buffer and updating its weights to minimize the Q-loss.
       Google DeepMind successfully used DQNs to play a variety of Atari games, often achieving super-human performance.
- **Disadvantages of Q-Learning:**
    - **Discrete Action Spaces:** Standard DQNs are designed for discrete action spaces, as they output a Q-value for each action. Handling continuous action spaces is challenging.
    - **Deterministic Policy:** The policy derived by taking the argmax of Q-values is deterministic. This can be problematic in stochastic environments or when exploration is needed.

2. Policy Learning (Policy Gradients)

Policy gradient methods aim to directly learn the parameters of the agent's policy $\pi_\theta(a|s)$ without necessarily learning a value function first.

- **Concept:** The neural network directly outputs the action to take, or a probability distribution over actions. For a given state, instead of Q-values, the network might output probabilities for each discrete action (e.g., P(left)=0.9, P(stay)=0.1, P(right)=0.0).
- **Action Selection:** Actions are selected by sampling from this output probability distribution. This naturally incorporates exploration.
- **Advantages:**
    - **Continuous Action Spaces:** Policy gradient methods can more naturally handle continuous action spaces. Instead of outputting probabilities for discrete actions, the network can output the parameters of a continuous probability distribution (e.g., the mean and variance of a Gaussian distribution for a steering angle).
    - **Stochastic Policies:** These methods learn stochastic policies directly, which can be beneficial in environments where the optimal policy is stochastic or for exploration purposes.
- **Policy Gradient Theorem:** The core idea is to adjust the policy parameters θ in the direction that increases the expected return. The gradient of the expected return $J(\theta)$ with respect to the policy parameters θ is given by:

  $$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$$

  where $R(\tau)$ is the total return of an episode (trajectory τ). A common variant uses the return from that time step onwards, G_t.
- **Loss Function (Simplified REINFORCE algorithm):** The objective is often to maximize expected rewards. In practice, this often translates to minimizing a loss function like:

  $$L(\theta) = -E_t[\log \pi_\theta(a_t|s_t) G_t]$$

  where G_t is the discounted return from time step t. The term $\log \pi_\theta(a_t|s_t)$ is the log-likelihood of selecting the action a_t given state s_t. If an action taken led to a high positive return (G_t), its probability is increased (loss becomes more negative, leading to a larger update in the positive gradient direction). If it led to a low or negative return, its probability is decreased.
- **Example: Training an Autonomous Vehicle:**
    1. **Initialize:** Start the agent (car) in the environment.
    2. **Run Policy & Collect Data:** Let the agent run its current policy (initially random). At each step, the policy network computes action probabilities (e.g., for steering angle), an action is sampled, and the car executes it. Record states, actions, and rewards (e.g., penalty for crashing, reward for staying in lane). This continues until the episode ends (e.g., car crashes).
    3. **Update Policy:**
        - For each state-action pair in the trajectory, calculate the return G_t.
        - Adjust the policy network's weights to:
            - Increase the probability of actions that led to high returns (e.g., actions taken long before a crash).
            - Decrease the probability of actions that led to low returns (e.g., actions taken just before a crash).
    4. **Repeat:** Reinitialize the agent and repeat the process with the updated policy.
       Through this iterative process, the agent learns to perform the desired task, like staying in its lane, without explicit instructions on driving rules, relying only on the sparse reward signal.

- **Challenges of RL in the Real World:** A major challenge is that for many real-world tasks (like autonomous driving), running the policy until "termination" (e.g., a crash) is unacceptable.
- **Photorealistic Simulators:** To address this, highly realistic simulators are used for training RL agents. This allows the agent to experience many "crashes" and learn from them safely before being deployed in the real world. Models trained in such simulators can then be transferred to real-world systems.

Applications and Advances

Deep RL has shown remarkable success in various domains:

- **Robotics and Autonomy:** As seen in the autonomous vehicle example.
- **Game Play and Strategy:**
    - **Atari Games:** DQNs mastered many Atari games.
    - **AlphaGo:** Developed by DeepMind, AlphaGo defeated world champion Go players. Go is a game with an enormous action space and requires complex strategic thinking.
        - **AlphaGo's Approach:**
            1. **Supervised Learning Policy Network:** Initially trained a policy network on human expert games to predict human moves.
            2. **Reinforcement Learning Policy Network:** Refined the policy network by having it play against itself (self-play) and learn from the outcomes (wins/losses) using RL.
            3. **Value Network:** Trained a value network to evaluate board positions, predicting the winner from a given state. This helps provide a denser reward signal than just the game outcome, which is very sparse.
            These components are combined with Monte Carlo Tree Search (MCTS) for move selection.
    - **AlphaGo Zero:** An advanced version that learned to play Go from scratch, without any human data, purely through self-play and reinforcement learning, surpassing AlphaGo's performance.

Summary

The lecture covered the fundamentals of Reinforcement Learning, distinguishing it from supervised and unsupervised learning. Key terminologies like agent, environment, state, action, and reward were introduced. Two primary RL methodologies were explored:

1. **Q-Learning:** Learning a value function (Q-function) to estimate the quality of actions in states, exemplified by Deep Q-Networks (DQNs) and their success in Atari games.
2. **Policy Learning:** Directly learning a policy that maps states to actions (or action probabilities), with policy gradients being a key technique, suitable for continuous action spaces.

Applications in autonomous driving and complex game playing (like Go with AlphaGo and AlphaGo Zero) highlighted the power and potential of Deep Reinforcement Learning. The lecture emphasized...