Reinforcement Learning

Introduction

Reinforcement Learning

- Supervised (inductive) learning is the simplest and most studied type of learning

- How can an agent ***learn behaviors*** when it doesn't have a teacher to tell it how to perform?
  - The agent has a task to perform
  - It takes some actions in the world
  - At some later point, it gets feedback telling it how well it did on performing the task
  - The agent performs the same task over and over again

- This problem is called **reinforcement learning:**
  - The agent gets *positive reinforcement* for tasks done well
  - The agent gets *negative reinforcement* for tasks done poorly

Reinforcement Learning

- Learning from interaction with an environment to achieve some long-term goal that is related to the state of the environment
- The goal is defined by reward signal, which must be maximised
- Agent must be able to partially/fully sense the environment state and take actions to influence the environment state
- The state is typically described with a feature-vector

Reinforcement Learning: Exploration versus Exploitation

- We want a reinforcement learning agent to earn lots of reward
- The agent must prefer past actions that have been found to be effective at producing reward
- The agent must exploit what it already knows to obtain reward
- The agent must select untested actions to discover reward-producing actions
- The agent must explore actions to make better action selections in the future
- Trade-off between exploration and exploitation

Reinforcement Learning

- Learning to interact with an environment
  - Robots, games, process control
  - With limited human training
  - Where the 'right thing' isn't obvious

- Supervised Learning:
  - Goal: $f(x) = y$
  - Data: $[< x_1, y_1 >, \ldots, < x_n, y_n >]$

- Reinforcement Learning:
  - Goal:
    Maximize $\sum_{i=1}^{\infty} Reward(State_i, Action_i)$
  - Data:
    $Reward_i, State_{i+1} = Interact(State_i, Action_i)$

Reinforcement Learning -Introduction

Supervised Learning:

Example Class

Reinforcement Learning:

Situation Reward … Situation Reward

Playing chess: Reward comes at end of game
Ping-pong: Reward on each point scored

Reinforcement Learning - Introduction

- The goal is to get the agent to act in the world so as to maximize its rewards

- The agent has to figure out what it did that made it get the reward/punishment
  - This is known as the **credit assignment** problem

- Reinforcement learning approaches can be used to train computers to do many tasks
  - backgammon and chess playing
  - job shop scheduling
  - controlling robot limbs

Reinforcement Learning Systems

- Reinforcement learning systems have 4 main elements:
  - Policy
  - Reward signal
  - Value function
  - Optional model of the environment

Reinforcement Learning - Policy

- A policy is a mapping from the perceived states of the environment to actions to be taken when in those states
- A reinforcement learning agent uses a policy to select actions given the current environment state

Reinforcement Learning - Reward Signal

- The reward signal defines the goal
- On each time step, the environment sends a single number called the reward to the reinforcement learning agent
- The agent's objective is to maximise the total reward that it receives over the long run
- The reward signal is used to alter the policy

Value Function (1)

- The reward signal indicates what is good in the short run while the value function indicates what is good in the long run
- The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting in that state
- Compute the value using the states that are likely to follow the current state and the rewards available in those states
- Future rewards may be time-discounted with a factor in the interval [0, 1]

Value Function (2)

- Use the values to make and evaluate decisions
- Action choices are made based on value judgements
- Prefer actions that bring about states of highest value instead of highest reward
- Rewards are given directly by the environment
- Values must continually be re-estimated from the sequence of observations that an agent makes over its lifetime

Reinforcement Learning - Formalization

- Given:
  - a state space S
  - a set of actions $a_1, \ldots, a_k$
  - reward value at the end of each trial (may be positive or negative)
- Output:
  - a mapping from states to actions

example: Alvinn (driving agent)
state: configuration of the car
learn a steering action for each state

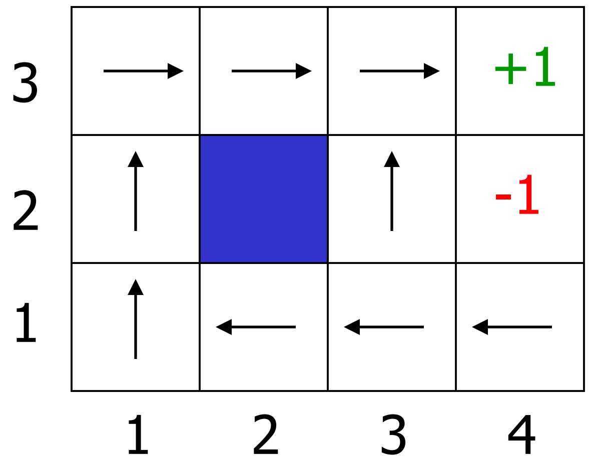
Reactive Agent Algorithm

Repeat:

- s ← sensed state
- If s is terminal then exit
- a ← choose action (given s)
- Perform a

Accessible or observable state

Policy (Reactive/Closed-Loop Strategy)



- A policy Π is a complete mapping from states to actions

Reactive Agent Algorithm

Repeat:

- s ← sensed state
- If s is terminal then exit
- a ← Π(s)
- Perform a

Reinforcement Learning - Approaches

- Learn policy directly– function mapping from states to actions
- Learn utility values for states (i.e., the value function)

Value Function

- The agent knows what state it is in
- The agent has a number of actions it can perform in each state.
- Initially, it doesn't know the value of any of the states
- If the outcome of performing an action at a state is deterministic, then the agent can update the utility value U() of states:
  - U(oldstate) = reward + U(newstate)
- The agent learns the utility values of states as it works its way through the state space

Exploration

- The agent may occasionally choose to explore suboptimal moves in the hopes of finding better outcomes
  - Only by visiting all the states frequently enough can we guarantee learning the true values of all the states

- A discount factor is often introduced to prevent utility values from diverging and to promote the use of shorter (more efficient) sequences of actions to attain rewards

- The update equation using a discount factor γ is:

  - $U(oldstate) = reward + \gamma * U(newstate)$

- Normally, γ is set between 0 and 1

Model-free versus Model-based

- A model of the environment allows inferences to be made about how the environment will behave
- Example: Given a state and an action to be taken while in that state, the model could predict the next state and the next reward
- Models are used for planning, which means deciding on a course of action by considering possible future situations before they are experienced
- Model-based methods use models and planning. Think of this as modelling the dynamics $p(s' \mid s, a)$
- Model-free methods learn exclusively from trial-and-error (i.e. no modelling of the environment)

Q-Learning

- Q-learning augments value iteration by maintaining an *estimated utility value* *Q(s,a)* for *every action* at every state

- The utility of a state U(s), or Q(s), is simply the maximum Q value over all the possible actions at that state

- Learns utilities of actions (not states) $\Rightarrow$ ***model-free learning***

Q-Learning

- foreach state s
  foreach action a
    Q(s,a)=0
  s=currentstate
  do forever
    a = select an action
    do action a
    r = reward from doing a
    t = resulting state from doing a
    $Q(s,a) = (1 - \alpha)\ Q(s,a) + \alpha\ (r + \gamma\ Q(t))$
    s = t

- The *learning coefficient*, α, determines how quickly our estimates are updated

- Normally, α is set to a small positive constant less than 1

Selecting an Action

- Simply choose action with highest (current) expected utility?

- ***Problem***: each action has two effects
  - yields a reward (or penalty) on current sequence
  - information is received and used in learning for future sequences
- ***Trade-off***: immediate good for long-term well-being

try a shortcut – you might get lost; you might learn a new, quicker route!

Exploration policy

- Wacky approach *(exploration)*: act randomly in hopes of eventually exploring entire environment

- Greedy approach *(exploitation)*: act to maximize utility using current estimate

- Reasonable balance: act more wacky (exploratory) when agent has little idea of environment; more greedy when the model is close to correct

Credit Assignment Problem

- Given a sequence of states and actions, and the final sum of time-discounted future rewards, how do we infer which actions were effective at producing lots of reward and which actions were not effective?

- How do we assign credit for the observed rewards given a sequence of actions over time?

- Every reinforcement learning algorithm must address this problem

Reward Design

- We need rewards to guide the agent to achieve its goal

- Option 1: *Hand-designed reward functions*
  - This is a black art

- Option 2: *Learn rewards from demonstrations*
  - Instead of having a human expert tune a system to achieve the desired behaviour, the expert can demonstrate desired behaviour and the robot can tune itself to match the demonstration

What is Deep Reinforcement Learning?

- Deep reinforcement learning is standard reinforcement learning where a deep neural network is used to approximate either a policy or a value function

- Deep neural networks require lots of real/simulated interaction with the environment to learn

- Lots of trials/interactions is possible in simulated environments

- We can easily parallelise the trials/interaction in simulated environments

- We cannot do this with robotics (no simulations) because action execution takes time, accidents/failures are expensive and there are safety concerns

Finite Markov Decision Processes

Slide of Karan Kathpalia

Markov Decision Process (MDP)

- Set of states S
- Set of actions A
- State transition probabilities p(s' | s, a). This is the probability distribution over the state space given we take action a in state s
- Discount factor γ in [0, 1]
- Reward function R: S x A -> set of real numbers
- For simplicity, assume discrete rewards
- Finite MDP if both S and A are finite

Time Discounting

- The undiscounted formulation $\gamma = 0$ across episodes is appropriate for episodic tasks in which agent-environment interaction breaks into episodes (multiple trials to perform a task).
- Example: Playing Breakout (each run of the game is an episode)
- The discounted formulation $0 < \gamma <= 1$ is appropriate for continuing tasks, in which the interaction continues without limit
- Example: Vacuum cleaner robot
- This presentation focuses on episodic tasks

Agent-Environment Interaction (1)

- The agent and environment interact at each of a sequence of discrete time steps t = {0, 1, 2, 3, …, T} where T can be infinite
- At each time step t, the agent receives some representation of the environment state S_t in S and uses this to select an action A_t in the set $A(S_t)$ of available actions given that state
- One step later, the agent receives a numerical reward R_{t+1} and finds itself in a new state S_{t+1}

Agent-Environment Interaction (2)

- At each time step, the agent implements a stochastic policy or mapping from states to probabilities of selecting each possible action
- The policy is denoted π_t where $\pi_t(a \mid s)$ is the probability of taking action a when in state s
- A policy is a stochastic rule by which the agent selects actions as a function of states
- Reinforcement learning methods specify how the agent changes its policy using its experience

Action Selection

- At time t, the agent tries to select an action to maximise the sum G_t of discounted rewards received in the future

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

MDP Dynamics

- Given current state s and action a in that state, the probability of the next state s’ and the next reward r is given by:

$$p(s', r | s, a) = \Pr(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

State Transition Probabilities

- Suppose the reward function is discrete and maps from S x A to W
- The state transition probability or probability of transitioning to state s' given current state s and action a in that state is given by:

$$p(s'|s,a) = \sum_{r \in W} p(s',r|s,a)$$

Expected Rewards

- The expected reward for a given state-action pair is given by:

$$r(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in W} r \sum_{s' \in S} p(s', r | s, a)$$

State-Value Function (1)

- Value functions are defined with respect to particular policies because future rewards depend on the agent's actions
- Value functions give the expected return of a particular policy
- The value $v_\pi(s)$ of a state s under a policy π is the expected return when starting in state s and following the policy π from that state onwards

State-Value Function (2)

- The state-value function $v_\pi(s)$ for the policy π is given below. Note that the value of the terminal state (if any) is always zero.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

Action-Value Function

- The value $q_\pi(s, a)$ of taking action a in state s under a policy π is defined as the expected return starting from s, taking the action a and thereafter following policy π
- q_π is the action-value function for policy π

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

$$q_\pi(s, a) = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

Bellman Equation (1)

- The equation expresses the relationship between the value of a state s and the values of its successor states
- The value of the next state must equal the discounted value of the expected next state, plus the reward expected along the way

Bellman Equation (2)

- The value of state s is the expected value of the sum of time-discounted rewards (starting at current state) given current state s
- This is expected value of r plus the sum of time-discounted rewards (starting at successor state) over all successor states s' and all next rewards r and over all possible actions a in current state s

$$\forall s \in \mathcal{S} : v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

Optimality

- A policy is defined to be better than or equal to another policy if its expected return is greater than or equal to that of the other policy for all states
- There is always at least one optimal policy π_* that is better than or equal to all other policies
- All optimal policies share the same optimal state-value function v^*, which gives the maximum expected return for any state s over all possible policies
- All optimal policies share the same optimal action-value function q^*, which gives the maximum expected return for any state-action pair (s, a) over all possible policies

Temporal-Difference Learning

What is TD learning?

- Temporal-Difference learning = TD learning
- The prediction problem is that of estimating the value function for a policy π
- The control problem is the problem of finding an optimal policy π_*
- Given some experience following a policy π, update estimate v of v_π for non-terminal states occurring in that experience
- Given current step t, TD methods wait until the next time step to update $V(S_t)$
- Learn from partial returns

Value-based Reinforcement Learning

- We want to estimate the optimal value $V^*(s)$ or action-value function $Q^*(s, a)$ using a function approximator $V(s; \theta)$ or $Q(s, a; \theta)$ with parameters θ
- This function approximator can be any parametric supervised machine learning model
- Recall that the optimal value is the maximum value achievable under any policy

Update Rule for TD(0)

- At time t + 1, TD methods immediately form a target $R_{t+1} + \gamma V(S_{t+1})$ and make a useful update with step size α using the observed reward R_{t+1} and the estimate $V(S_{t+1})$
- The update is the step size times the difference between the target output and the actual output

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

Update Rule Intuition

- The target output is a more accurate estimate of $V(S_t)$ given the reward R_{t+1} is known
- The actual output is our current estimate of $V(S_t)$
- We simply take one step with our current value function estimate to get a more accurate estimate of $V(S_t)$ and then perform an update to move $V(S_t)$ closer towards the more accurate estimate (i.e. temporal difference)

Tabular TD(0) Algorithm

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

SARSA – On-policy TD Control

- SARSA = State-Action-Reward-State-Action
- Learn an action-value function instead of a state-value function
- q_π is the action-value function for policy π
- Q-values are the values $q_\pi(s, a)$ for s in S, a in A
- SARSA experiences are used to update Q-values
- Use TD methods for the prediction problem

SARSA Update Rule

- We want to estimate $q_\pi(s, a)$ for the current policy π, and for all states s and action a
- The update rule is similar to that for TD(0) but we transition from state-action pair to state-action pair, and learn the values of state-action pairs
- The update is performed after every transition from a non-terminal state S_t
- If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is zero
- The update rule uses $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

SARSA Algorithm

Sarsa: An on-policy TD control algorithm

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

Q-learning – Off-policy TD Control

- Similar to SARSA but off-policy updates
- The learned action-value function Q directly approximates the optimal action-value function q_* independent of the policy being followed
- In update rule, choose action a that maximises Q given S_{t+1} and use the resulting Q-value (i.e. estimated value given by optimal action-value function) plus the observed reward as the target
- This method is off-policy because we do not have a fixed policy that maps from states to actions. This is why A_{t+1} is not used in the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(Q(S_{t+1}, a)) - Q(S_t, A_t)]$$

One-step Q-learning Algorithm

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until S is terminal

Epsilon-greedy Policy

- At each time step, the agent selects an action
- The agent follows the greedy strategy with probability 1 – epsilon
- The agent selects a random action with probability epsilon
- With Q-learning, the greedy strategy is the action a that maximises Q given S_{t+1}

- The policy (i.e. the behavior learned by the agent) can be represented in a Q table (where Q stands for Q-learning, the algorithm derives from the Q function, which calculates the expected benefit of an action in the state, to create the best possible policy).
- The rows report all possible observations while the columns include all possible actions.
- The cells are therefore filled during training with values that represent the expected reward.
- The Q table only works with small observation spaces.
- When the possibilities increase, the agent must use neural networks.

Deep Q-Networks (DQN)

- Introduced deep reinforcement learning
- It is common to use a function approximator $Q(s, a; \theta)$ to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Example: Breakout has 3 actions – move left, move right, no movement
- Uses epsilon-greedy policy to select actions

Q-Networks

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

Experience Replay

- The state is a sequence of actions and observations $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$
- Store the agent's experiences at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D = e_1, \ldots, e_n$ pooled over many episodes into a replay memory
- In practice, only store the last N experience tuples in the replay memory and sample uniformly from D when performing updates

State representation

- It is difficult to give the neural network a sequence of arbitrary length as input
- Use fixed length representation of sequence/history produced by a function $\varphi(s_t)$
- Example: The last 4 image frames in the sequence of Breakout gameplay

Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D
- Similar to Q-learning update rule but:
  - Use mini-batch stochastic gradient updates
  - The gradient of the loss function for a given iteration with respect to the parameter θ_i is the difference between the target value and the actual value is multiplied by the gradient of the Q function approximator $Q(s, a; \theta)$ with respect to that specific parameter
- Use the gradient of the loss function to update the Q function approximator