Reinforcement Learning for Algorithmic Trading

Complete Theory, Algorithms, and Trading Applications

Chapter 1: Introduction to Reinforcement Learning

1.1 What is Reinforcement Learning?

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment to maximize cumulative reward.

Key Difference from Supervised Learning:

- **Supervised:** Learn from labeled examples (input → output)
- **RL:** Learn from trial-and-error, rewards/penalties, delayed consequences

Trading Analogy:

- **Agent:** Trading algorithm
- **Environment:** Financial market
- **Actions:** Buy, sell, hold
- **State:** Market conditions, positions, P&L
- **Reward:** Profit/loss from actions

1.2 Components of RL Problems

1. Agent: Decision-maker (trading algorithm)

2. Environment: External system (market) that responds to actions

3. State s_t: Complete description of the environment at time t

- Example: Current price, historical returns, technical indicators, position

4. Action a_t: Decision made by agent

- Discrete: {Buy, Sell, Hold}
- Continuous: Position size [-1, 1]

5. Reward r_t: Scalar feedback signal

- Immediate: P&L from trade
- Delayed: Risk-adjusted return over horizon

6. Policy π: Mapping from states to actions

- $\pi(a|s)$: Probability of action a in state s
- Deterministic: $a = \pi(s)$

7. Value Function $V^\pi(s)$: Expected cumulative reward from state s following policy π

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]$$

8. Action-Value Function (Q-function) $Q^\pi(s, a)$: Expected return from taking action a in state s, then following π

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right]$$

9. Discount Factor $\gamma \in [0, 1]$: Importance of future rewards

- $\gamma = 0$: Only immediate reward
- $\gamma = 1$: All rewards equally important (undiscounted)

1.3 The RL Objective

Goal: Find optimal policy π^* that maximizes expected cumulative discounted reward:

$$\pi^* = \arg\max_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

Chapter 2: Markov Decision Processes (MDPs)

2.1 MDP Formulation

A **Markov Decision Process** is a tuple (S, A, P, R, γ):

- S: State space
- A: Action space
- $P(s'|s, a)$: Transition probability (dynamics)
- $R(s, a, s')$: Reward function
- γ: Discount factor

Markov Property:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t)$$

Future depends only on present state, not history.

2.2 Bellman Equations

Bellman Expectation Equation for V^π:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^\pi(s') \right]$$

Bellman Expectation Equation for Q^π:

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a') \right]$$

Bellman Optimality Equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s') \right]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

Optimal Policy from Q^*:

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

Chapter 3: Q-Learning Algorithm

3.1 Tabular Q-Learning

Q-Learning is a model-free, off-policy TD control algorithm.

Update Rule:

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

where:

- α: Learning rate (step size)
- $r_t + \gamma \max_{a'} Q(s_{t+1},a')$: TD target
- $r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)$: TD error

Algorithm:

```
Initialize Q(s,a) arbitrarily for all s, a
For each episode:
    Initialize state s
    For each step of episode:
        Choose action a from s using ε-greedy policy
        Take action a, observe reward r, next state s'
        Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') - Q(s,a)]
```

```
        s ← s'
    Until s is terminal
```

3.2 Exploration vs Exploitation

ε-Greedy Policy:

$$a = \begin{cases} \arg\max_a Q(s,a) & amp; \text{with probability } 1-\epsilon \\ \text{random action} & amp; \text{with probability } \epsilon \end{cases}$$

Decay Schedule:

- Start: $\epsilon = 1.0$ (pure exploration)
- End: $\epsilon = 0.01$ (mostly exploitation)
- Decay: $\epsilon_t = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-\lambda t}$

3.3 Python Implementation: Q-Learning for Trading

```
import numpy as np
import pandas as pd
import gym

class TradingEnv(gym.Env):
    """Simple trading environment"""

    def __init__(self, price_data, initial_cash=10000):
        self.prices = price_data.values
        self.n_steps = len(self.prices)
        self.current_step = 0

        self.initial_cash = initial_cash
        self.cash = initial_cash
        self.position = 0  # Number of shares

        # Action space: 0=Hold, 1=Buy, 2=Sell
        self.action_space = gym.spaces.Discrete(3)

        # State: [price_pct_change, position, cash_ratio]
        self.observation_space = gym.spaces.Box(
            low=-np.inf, high=np.inf, shape=(3,), dtype=np.float32
        )

    def reset(self):
        self.current_step = 0
        self.cash = self.initial_cash
        self.position = 0
        return self._get_state()

    def _get_state(self):
        price = self.prices[self.current_step]
        prev_price = self.prices[max(0, self.current_step - 1)]
        pct_change = (price - prev_price) / prev_price if prev_price != 0 else 0
```

```
        cash_ratio = self.cash / self.initial_cash
        position_norm = self.position / 100  # Normalize

        return np.array([pct_change, position_norm, cash_ratio], dtype=np.float32)

    def step(self, action):
        current_price = self.prices[self.current_step]

        # Execute action
        reward = 0
        if action == 1:  # Buy
            shares_to_buy = min(10, self.cash // current_price)
            if shares_to_buy &gt; 0:
                self.position += shares_to_buy
                self.cash -= shares_to_buy * current_price

        elif action == 2:  # Sell
            if self.position &gt; 0:
                self.cash += self.position * current_price
                reward = self.position * current_price  # Reward for selling
                self.position = 0

        # Move to next step
        self.current_step += 1
        done = (self.current_step &gt;= self.n_steps - 1)

        # Calculate portfolio value
        portfolio_value = self.cash + self.position * current_price

        # Reward: change in portfolio value
        reward = portfolio_value - self.initial_cash

        return self._get_state(), reward, done, {}

    def render(self):
        price = self.prices[self.current_step]
        portfolio = self.cash + self.position * price
        print(f"Step: {self.current_step}, Price: {price:.2f}, "
              f"Position: {self.position}, Cash: {self.cash:.2f}, "
              f"Portfolio: {portfolio:.2f}")


class QLearningAgent:
    """Q-Learning agent with discrete state discretization"""

    def __init__(self, n_states, n_actions, learning_rate=0.1, gamma=0.95,
                 epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995):
        self.n_states = n_states
        self.n_actions = n_actions
        self.lr = learning_rate
        self.gamma = gamma

        self.epsilon = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
```

```
        # Q-table: [state, action]
        self.q_table = np.zeros((n_states, n_actions))

    def discretize_state(self, state):
        """Convert continuous state to discrete bin"""
        # Simple binning strategy
        bins = 10
        discretized = []
        for s in state:
            if s &lt; -0.1:
                discretized.append(0)
            elif s &gt; 0.1:
                discretized.append(2)
            else:
                discretized.append(1)

        # Convert to single index
        idx = discretized[0] * 9 + discretized[1] * 3 + discretized[2]
        return min(idx, self.n_states - 1)

    def choose_action(self, state):
        """ε-greedy action selection"""
        state_idx = self.discretize_state(state)

        if np.random.random() &lt; self.epsilon:
            return np.random.randint(self.n_actions)  # Explore
        else:
            return np.argmax(self.q_table[state_idx])  # Exploit

    def learn(self, state, action, reward, next_state, done):
        """Q-learning update"""
        state_idx = self.discretize_state(state)
        next_state_idx = self.discretize_state(next_state)

        # Current Q-value
        current_q = self.q_table[state_idx, action]

        # TD target
        if done:
            td_target = reward
        else:
            td_target = reward + self.gamma * np.max(self.q_table[next_state_idx])

        # Q-learning update
        self.q_table[state_idx, action] += self.lr * (td_target - current_q)

        # Decay epsilon
        if self.epsilon &gt; self.epsilon_end:
            self.epsilon *= self.epsilon_decay


# Training loop
def train_q_learning(env, agent, n_episodes=1000):
    """Train Q-learning agent"""
```

```
    episode_rewards = []

    for episode in range(n_episodes):
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)

            agent.learn(state, action, reward, next_state, done)

            state = next_state
            total_reward += reward

        episode_rewards.append(total_reward)

        if (episode + 1) % 100 == 0:
            avg_reward = np.mean(episode_rewards[-100:])
            print(f"Episode {episode+1}, Avg Reward: {avg_reward:.2f}, "
                  f"Epsilon: {agent.epsilon:.3f}")

    return episode_rewards


# Usage
prices = pd.Series([100, 102, 101, 103, 105, 104, 106, 108, 107, 110])
env = TradingEnv(prices)

agent = QLearningAgent(
    n_states=27,  # 3^3 discretized states
    n_actions=3,
    learning_rate=0.1,
    gamma=0.95
)

rewards = train_q_learning(env, agent, n_episodes=500)
```

Chapter 4: Deep Q-Networks (DQN)

4.1 From Tabular to Function Approximation

Problem with Q-tables:

- State space explosion for continuous/high-dimensional states
- Cannot generalize to unseen states

Solution: Neural Network Q-function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

where θ are network weights.

4.2 DQN Algorithm Components

1. Experience Replay:
Store transitions (s, a, r, s') in replay buffer, sample random mini-batches for training.

Benefits:

- Breaks correlation between consecutive samples
- Improves data efficiency (reuse experiences)

2. Target Network:
Use separate network θ^- for TD target, updated periodically.

Loss Function:

$$L(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

3. Gradient Descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

4.3 DQN Implementation with PyTorch

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
import random

class DQN(nn.Module):
    """Deep Q-Network"""

    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(DQN, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
        )

    def forward(self, state):
        return self.network(state)


class ReplayBuffer:
    """Experience replay buffer"""

    def __init__(self, capacity=10000):
```

```
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (np.array(states), np.array(actions), np.array(rewards),
                np.array(next_states), np.array(dones))

    def __len__(self):
        return len(self.buffer)


class DQNAgent:
    """DQN agent for trading"""

    def __init__(self, state_dim, action_dim, lr=0.001, gamma=0.99,
                 epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995):

        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma

        # Q-networks
        self.q_network = DQN(state_dim, action_dim)
        self.target_network = DQN(state_dim, action_dim)
        self.target_network.load_state_dict(self.q_network.state_dict())

        self.optimizer = optim.Adam(self.q_network.parameters(), lr=lr)
        self.loss_fn = nn.MSELoss()

        # Replay buffer
        self.replay_buffer = ReplayBuffer(capacity=10000)

        # Exploration
        self.epsilon = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay

    def choose_action(self, state):
        """ε-greedy action selection"""
        if np.random.random() &lt; self.epsilon:
            return np.random.randint(self.action_dim)

        with torch.no_grad():
            state_tensor = torch.FloatTensor(state).unsqueeze(0)
            q_values = self.q_network(state_tensor)
            return q_values.argmax().item()

    def train(self, batch_size=64):
        """Train on mini-batch from replay buffer"""

        if len(self.replay_buffer) &lt; batch_size:
            return
```

```
        # Sample batch
        states, actions, rewards, next_states, dones = \
            self.replay_buffer.sample(batch_size)

        # Convert to tensors
        states = torch.FloatTensor(states)
        actions = torch.LongTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones)

        # Current Q-values
        current_q = self.q_network(states).gather(1, actions.unsqueeze(1)).squeeze(1)

        # Target Q-values
        with torch.no_grad():
            next_q = self.target_network(next_states).max(1)[0]
            target_q = rewards + (1 - dones) * self.gamma * next_q

        # Loss and backprop
        loss = self.loss_fn(current_q, target_q)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Decay epsilon
        if self.epsilon &gt; self.epsilon_end:
            self.epsilon *= self.epsilon_decay

        return loss.item()

    def update_target_network(self):
        """Copy weights from Q-network to target network"""
        self.target_network.load_state_dict(self.q_network.state_dict())


# Training loop
def train_dqn(env, agent, n_episodes=1000, batch_size=64,
              target_update_freq=10):
    """Train DQN agent"""

    episode_rewards = []

    for episode in range(n_episodes):
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)

            # Store transition
            agent.replay_buffer.push(state, action, reward, next_state, done)
```

```
            # Train
            loss = agent.train(batch_size)

            state = next_state
            total_reward += reward

        # Update target network periodically
        if (episode + 1) % target_update_freq == 0:
            agent.update_target_network()

        episode_rewards.append(total_reward)

        if (episode + 1) % 100 == 0:
            avg_reward = np.mean(episode_rewards[-100:])
            print(f"Episode {episode+1}, Avg Reward: {avg_reward:.2f}, "
                  f"Epsilon: {agent.epsilon:.3f}")

    return episode_rewards
```

Chapter 5: Policy Gradient Methods

5.1 REINFORCE Algorithm

Idea: Directly optimize policy $\pi_\theta(a|s)$ by gradient ascent on expected return.

Objective:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

Policy Gradient Theorem:

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

where $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ is the return from time t.

Algorithm:

```
Initialize policy parameters θ
For each episode:
    Generate trajectory τ = (s_0, a_0, r_0, ..., s_T)
    For t = 0 to T:
        R_t ← sum of discounted rewards from time t
        θ ← θ + α ∇_θ log π_θ(a_t|s_t) R_t
```

5.2 Actor-Critic Methods

Combine:

- **Actor:** Policy $\pi_\theta(a|s)$
- **Critic:** Value function $V_w(s)$ or $Q_w(s,a)$

Advantage Function:

$$A(s,a) = Q(s,a) - V(s)$$

Measures how much better action a is compared to average.

Actor update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a|s) A(s,a)$$

Critic update (TD learning):

$$w \leftarrow w + \beta(r + \gamma V_w(s') - V_w(s)) \nabla_w V_w(s)$$

Chapter 6: Advanced RL for Trading

6.1 State Representation

Financial Features:

```
def create_state(price_history, position, cash, lookback=20):
    """Create state vector from market data"""

    # Price features
    returns = np.diff(np.log(price_history[-lookback:]))

    # Technical indicators
    ma_5 = np.mean(price_history[-5:])
    ma_20 = np.mean(price_history[-20:])
    ma_ratio = ma_5 / ma_20

    volatility = np.std(returns)

    # Position features
    position_ratio = position / 100  # Normalize
    cash_ratio = cash / 10000

    state = np.array([
        returns[-1],  # Last return
        ma_ratio,
        volatility,
        position_ratio,
        cash_ratio
    ])
```

```
    return state
```

6.2 Reward Shaping

Reward Design Options:

1. Simple P&L:

$$r_t = \text{Portfolio Value}_t - \text{Portfolio Value}_{t-1}$$

2. Sharpe Ratio:

$$r_t = \frac{\text{Return}_t - R_f}{\sigma_t}$$

3. Risk-Adjusted:

$$r_t = \text{Return}_t - \lambda \cdot \text{Volatility}_t$$

4. Transaction Cost Penalty:

$$r_t = \text{P\&L}_t - c \cdot |\text{Trade Size}_t|$$

6.3 Continuous Action Space (DDPG)

For continuous position sizing $a \in [-1, 1]$:

Deep Deterministic Policy Gradient (DDPG):

- Actor: $\mu_\theta(s)$ outputs continuous action
- Critic: $Q_w(s, a)$ evaluates action
- Deterministic policy gradient

Chapter 7: Practical Trading Strategy with RL

7.1 Complete Trading System

```
import gym
import numpy as np
import pandas as pd

class RealWorldTradingEnv(gym.Env):
    """Realistic trading environment with transaction costs"""

    def __init__(self, price_data, initial_cash=10000,
                 commission=0.001, slippage=0.0005):
        self.prices = price_data
        self.returns = price_data.pct_change().fillna(0)
```

```
        self.initial_cash = initial_cash
        self.commission = commission
        self.slippage = slippage

        self.current_step = 0
        self.cash = initial_cash
        self.shares = 0
        self.portfolio_values = []

        # Action: continuous [-1, 1] (sell all to buy all)
        self.action_space = gym.spaces.Box(
            low=-1, high=1, shape=(1,), dtype=np.float32
        )

        # State: [returns, volatility, RSI, position, cash]
        self.observation_space = gym.spaces.Box(
            low=-np.inf, high=np.inf, shape=(5,), dtype=np.float32
        )

    def reset(self):
        self.current_step = 20  # Start after warmup
        self.cash = self.initial_cash
        self.shares = 0
        self.portfolio_values = []
        return self._get_state()

    def _get_state(self):
        # Returns (last 5 days)
        recent_returns = self.returns.iloc[self.current_step-5:self.current_step].mean()

        # Volatility (20-day)
        volatility = self.returns.iloc[self.current_step-20:self.current_step].std()

        # RSI
        rsi = self._calculate_rsi()

        # Position and cash
        current_price = self.prices.iloc[self.current_step]
        position_value = self.shares * current_price
        total_value = self.cash + position_value

        position_ratio = position_value / total_value if total_value &gt; 0 else 0
        cash_ratio = self.cash / total_value if total_value &gt; 0 else 0

        return np.array([recent_returns, volatility, rsi, position_ratio, cash_ratio])

    def _calculate_rsi(self, period=14):
        """Calculate RSI indicator"""
        deltas = self.returns.iloc[self.current_step-period:self.current_step]
        gain = deltas.where(deltas &gt; 0, 0).mean()
        loss = -deltas.where(deltas &lt; 0, 0).mean()

        if loss == 0:
            return 100
        rs = gain / loss
        rsi = 100 - (100 / (1 + rs))
```

```
        return rsi / 100  # Normalize to [0,1]

    def step(self, action):
        action = action[0]  # Unwrap
        current_price = self.prices.iloc[self.current_step]

        # Calculate target position
        total_value = self.cash + self.shares * current_price
        target_shares = int((total_value / current_price) * (action + 1) / 2)

        # Execute trade
        shares_to_trade = target_shares - self.shares
        trade_value = abs(shares_to_trade) * current_price

        # Transaction costs
        cost = trade_value * (self.commission + self.slippage)

        # Update position
        if shares_to_trade &gt; 0:  # Buy
            max_affordable = int(self.cash / (current_price * (1 + self.commission + sel
            shares_to_trade = min(shares_to_trade, max_affordable)
            self.cash -= shares_to_trade * current_price * (1 + self.commission + self.sl
            self.shares += shares_to_trade

        elif shares_to_trade &lt; 0:  # Sell
            shares_to_trade = max(shares_to_trade, -self.shares)
            self.cash += abs(shares_to_trade) * current_price * (1 - self.commission - se
            self.shares += shares_to_trade

        # Move to next step
        self.current_step += 1
        done = (self.current_step &gt;= len(self.prices) - 1)

        # Calculate reward (Sharpe-like)
        new_value = self.cash + self.shares * self.prices.iloc[self.current_step]
        self.portfolio_values.append(new_value)

        if len(self.portfolio_values) &gt; 1:
            portfolio_return = (new_value - total_value) / total_value
            reward = portfolio_return - cost / total_value  # Penalize costs
        else:
            reward = 0

        return self._get_state(), reward, done, {}
```

Chapter 8: Evaluation and Backtesting

8.1 Performance Metrics

```
def evaluate_strategy(portfolio_values, risk_free_rate=0.0):
    """Calculate performance metrics"""

    returns = pd.Series(portfolio_values).pct_change().dropna()

    # Total return
    total_return = (portfolio_values[-1] / portfolio_values[0]) - 1

    # Sharpe ratio (annualized)
    excess_returns = returns - risk_free_rate / 252
    sharpe = np.sqrt(252) * excess_returns.mean() / returns.std()

    # Maximum drawdown
    cumulative = (1 + returns).cumprod()
    running_max = cumulative.expanding().max()
    drawdown = (cumulative - running_max) / running_max
    max_drawdown = drawdown.min()

    # Win rate
    win_rate = (returns &gt; 0).sum() / len(returns)

    return {
        'Total Return': total_return,
        'Sharpe Ratio': sharpe,
        'Max Drawdown': max_drawdown,
        'Win Rate': win_rate,
        'Volatility': returns.std() * np.sqrt(252)
    }
```

Chapter 9: Project Ideas

Project 1: Q-Learning trend-following strategy on S&P 500
Project 2: DQN for portfolio allocation (multi-asset)
Project 3: Actor-Critic for options delta hedging
Project 4: PPO for market-making simulation

Further Reading

- Sutton & Barto: *Reinforcement Learning: An Introduction*
- Mnih et al.: *Playing Atari with Deep Reinforcement Learning* (DQN paper)
- Silver et al.: *Deterministic Policy Gradient Algorithms*
- OpenAI Spinning Up: RL tutorials and implementations

This textbook provides complete theory, mathematical foundations, and production-ready code for applying reinforcement learning to algorithmic trading.