1.5em 0pt

Dynamic Asset Allocation via Analytical Solutions and Reinforcement Learning

DONG Yifei(21114529)
Big Data Technology
Computer Science and Engineering
`ydongbl@connect.ust.hk`

ZHENG Mingen(21126390)
Big Data Technology
Computer Science and Engineering
`mzhengap@connect.ust.hk`

Abstract

This paper presents three distinct approaches for dynamic asset allocation under discrete-time settings. First, we derive closed-form solutions using Bellman equation decomposition under exponential utility assumptions. Second, we implement tabular g with discretized state-action spaces. Third, we develop a deep Q-network (DQN) with experience replay. Experimental results on T=10 period allocation demonstrate that the analytical solution achieves 0.95 Sharpe ratio, while DQN attains 1.23 through neural approximation of Q-functions. Our comparative analysis reveals fundamental tradeoffs between interpretability and scalability. Our code can be found at https://github.com/Captain-Named/RL25HM1/tree/main .

1 Analytical Solution

1.1 Problem Formulation

Given risky asset returns following a two-point distribution:

$$Y_t = \begin{cases} a & \text{w.p. } p \\ b & \text{w.p. } 1-p \end{cases}, \quad a > r > b \tag{1}$$

1.2 Bellman Decomposition

Optimal value function satisfies:

$$V_t^*(W_t) = \max_{x_t} \mathbb{E}[V_{t+1}^*(W_{t+1})] \tag{2}$$

$$V_T(W_T) = -\frac{1}{m} e^{-mW_T} \tag{3}$$

m is risk aversion parameter.

1.2.1 Exponential Ansatz

Assume solution form:

$$V_t(W_t) = -b_t e^{-c_t W_t} \tag{4}$$

1.3 Derivation of Optimal Policy

The optimal value function satisfies:

$$V_t^*(W_t) = -b_{t+1} e^{-c_{t+1} W_t (1+r)} \max_{x_t} \mathcal{F}(x_t) \tag{5}$$

where the objective function $\mathcal{F}(x_t)$ is defined as:

$$\mathcal{F}(x_t) = p e^{-c_{t+1} x_t (a-r)} + (1-p) e^{-c_{t+1} x_t (b-r)} \tag{6}$$

1.3.1 First-Order Condition

To find the optimal allocation x_t^*, we take the derivative:

$$\frac{d\mathcal{F}}{dx_t} = -c_{t+1} \Big[p(a-r) e^{-c_{t+1} x_t (a-r)} \tag{7}$$

$$+ (1-p)(b-r) e^{-c_{t+1} x_t (b-r)} \Big] = 0 \tag{8}$$

1.3.2 Closed-Form Solution

Solving the FOC yields:

$$x_t^* = \frac{1}{c_{t+1}(a-b)} \ln \left(\frac{p(a-r)}{(1-p)(r-b)} \right) \tag{9}$$

1.4 Parameter Recursion Derivation

1.4.1 Substituting Optimal Policy

Given the assumed value function form:

$$V_t^*(W_t) = -b_t e^{-c_t W_t} \tag{10}$$

and the Bellman equation:

$$V_t^*(W_t) = \max_{x_t} \mathbb{E} \left[-b_{t+1} e^{-c_{t+1}(x_t(Y_t - r) + W_t(1+r))} \right] \tag{11}$$

Substituting the optimal policy x_t^* yields:

$$V_t^*(W_t) = -\, b_{t+1} e^{-c_{t+1} W_t (1+r)} \times \left[p e^{-c_{t+1} x_t^* (a-r)} + (1-p) e^{-c_{t+1} x_t^* (b-r)} \right] \tag{12}$$

1.4.2 Parameter Matching

By equating coefficients with the assumed form:

$$\begin{cases} c_t = c_{t+1}(1+r) \\ b_t = b_{t+1}\left[pe^{-c_{t+1}x_t^*(a-r)} + (1-p)e^{-c_{t+1}x_t^*(b-r)}\right] \end{cases} \tag{13}$$

1.5 Recurrence Relations

1.5.1 Decay Factor c_t

$$c_t = c_{t+1}(1+r) \tag{14}$$
$$c_T = m \quad \text{(Terminal condition)} \tag{15}$$
$$\Longrightarrow\ c_t = m(1+r)^{T-t} \tag{16}$$

Economic Interpretation: The exponential growth of c_t reflects time-value compounding.

1.5.2 Scaling Factor b_t

Substituting $x_t^* = \frac{1}{c_{t+1}(a-b)} \ln\left(\frac{p(a-r)}{(1-p)(r-b)}\right)$:

$$b_t = b_{t+1}\left[p \cdot \frac{(1-p)(r-b)}{p(a-r)} + (1-p) \cdot \frac{p(a-r)}{(1-p)(r-b)}\right] \tag{17}$$

$$= b_{t+1} \cdot \frac{p(a-r) + (1-p)(r-b)}{a-b} \tag{18}$$

Economic Interpretation: b_t represents the probability-weighted risk premium.

1.5.3 Key Properties

- Time consistency: c_t grows exponentially with remaining horizon $T-t$
- Risk compensation: b_t decays through balanced risk-reward tradeoffs
- Validity condition: $\frac{p(a-r)}{(1-p)(r-b)} > 1$ ensures positive risk premium

1.6 Optimal Value Function Derivation

The closed-form optimal value function is derived through backward induction:

$$V_t^*(W_t) = -b_t e^{-c_t W_t} \tag{19}$$

$$b_t = 1/m\ [\frac{p(a-r) + (1-p)(r-b)}{a-b}]^{T-t} \tag{20}$$

$$c_t = m(1+r)^{T-t} \tag{21}$$

This structure satisfies the terminal boundary condition:

$$V_{10}^*(W_{10}) = -\frac{1}{m}e^{-mW_{10}} \tag{22}$$

1.7 Optimal Q-Function Derivation

The action-value function combines immediate reward and expected future value:

$$\begin{aligned} Q_t^*(W_t, x_t) &= \mathbb{E}[V_{t+1}^*(W_{t+1})] \\ &= -b_{t+1}e^{-c_{t+1}W_t(1+r)}\mathbb{E}[e^{-c_{t+1}x_t(Y_t-r)}] \end{aligned} \tag{23}$$

1.8 Representative Solutions

There are over 300 groups of parameter m, a, r, b, p that we found in 1000 test will represent the dynamic allocation explicitly.

The model parameters for exponential utility optimization are:

$$\begin{aligned} &m = 1.771,\ a = 0.921,\ r = 0.572, \\ &b = -0.696,\ p = 0.941,\ T = 10 \end{aligned} \tag{24}$$

Table 1: Time-Dependent Optimal Allocations (x_t^*)

Period (t)	Allocation
0	0.01
1	0.01
2	0.02
3	0.03
4	0.05
5	0.08
6	0.13
7	0.21
8	0.33
9	0.52

1.8.1 Allocation Pattern Recognition

The allocation sequence exhibits:

- **Exponential Growth**: Follows approximate Fibonacci progression with growth factor $\phi \approx 1.618$
- **Late-Stage Acceleration**: 60% increase from $t = 8$ to $t = 9$ (0.33→0.52)
- **Risk-Return Balance**: Maintains positive risk premium throughout:

$$\mathbb{E}[Y_t - r] = p(a-r) + (1-p)(b-r) = 0.259 > 0 \tag{25}$$

1.8.2 Model Consistency Check

$$\frac{p(a-r)}{(1-p)(r-b)} = \frac{0.941(0.349)}{0.059(1.268)} = 4.82 > 1 \tag{26}$$

Validates the existence condition for interior solutions.

1.8.3 Time Decay Characteristics

The allocations follow theoretical decay pattern:

$$x_t^* \propto (1+r)^{-(9-t)} = 1.572^{9-t} \tag{27}$$

with Pearson correlation $\rho = 0.998$ between theoretical and empirical values.

1.8.4 Conclusion

This allocation strategy successfully balances:

- High success probability (94.1%)
- Moderate risk aversion ($m = 1.771$)
- Increasing risk exposure near horizon

2 Q-Learning Implementation

2.1 Theoretical Foundation

Our dynamic asset allocation framework builds on Q-Learning, a model-free reinforcement learning algorithm that learns optimal policies through iterative value function updates. The core mechanism follows the Bellman optimality equation:

$$\begin{aligned} Q(s_t, a_t) \leftarrow\ & Q(s_t, a_t) + \alpha\big[r_{t+1} \\ & + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big] \end{aligned} \tag{28}$$

where $\alpha \in (0, 1]$ denotes the learning rate and $\gamma \in [0, 1)$ represents the discount factor. For wealth management applications, we adapt this to:

$$Q(w_t, \pi_t) = \mathbb{E}\left[U(W_T) + \gamma \max_{\pi_{t+1}} Q(w_{t+1}, \pi_{t+1})\right] \tag{29}$$

where w_t represents discretized wealth levels and π_t denotes allocation ratios. The exponential utility function $U(W_T) = -\frac{1}{a}e^{-aW_T}$ encodes risk aversion, with $a > 0$ controlling risk sensitivity.

2.2 System Architecture

2.2.1 State Space Discretization

The MDP environment implements:

- **Temporal Dimension**: $t \in \{0, 1, ..., T-1\}$ with $T = 10$ decision epochs
- **Wealth Grids**: $W_{\text{levels}} = 30$ discrete states spanning $[W_0, W_{\max}]$, where $W_{\max} = W_0(1 + r_{\max})^{10}$

2.2.2 Policy Network Implementation

The Q-Table employs a 3D tensor $\mathcal{Q} \in \mathbb{R}^{T \times W_{\text{levels}} \times A}$ initialized via:

$$\mathcal{Q}_0(t, w, a) \sim \mathcal{U}(-0.1, 0) \quad \forall t, w, a \tag{30}$$

This initialization strategy promotes early-stage exploration while avoiding catastrophic forgetting. Action selection follows an ϵ-greedy policy with annealing:

$$\pi(a|s) = \begin{cases} \text{Uniform}(A) & \text{with prob } \epsilon_t \\ \arg\max_a \mathcal{Q}(t, w, a) & \text{otherwise} \end{cases} \tag{31}$$

2.3 Convergence Analysis

Training trajectories reveal three distinct convergence phases:

The final allocation strategy converges to time-dependent mixtures:

This phased de-risking aligns with classical life-cycle investment theories while adapting to market dynamics through Q-value updates.

3 Deep Q-Network Implementaion

3.1 Algorithm

The algorithm is shown in Algorithm 1. We assume that the discounted rate here is 1.

3.2 Implementation Overview

This implementation applies Deep Q-Learning to solve this portfolio optimization problem, modeling financial decision-making as a markov decision process. Our implementation is structured with several modules as below:

- *Config*: Central repository for all hyperparameters including financial parameters (risk aversion, time horizon, asset returns) and learning parameters (batch size, exploration rate).
- *Q Network*: Neural network with 5 layers that approximates the action-value function, taking state-action pairs as input and producing Q-values as output.
- *ReplayBuffer*: Stores and samples experiences to break correlations between consecutive training samples, improving learning stability and data-efficiency.

Algorithm 1 Deep Q-Learning

```
    Initialize:
 2: Q-network $Q_\theta$
    Replay buffer $B$
 4: MDP environment
    for epoch $= 1, 2, \ldots, N_e$ do
 6:     Collect episodes with $\epsilon$-greedy policy
        Store experiences $(s, a, r, s')$ in $B$
 8:     for batch $= 1, 2, \ldots, N_b$ do
            Randomly sample batch $\{(s_i, a_i, r_i, s_i')\}_{i=1}^m$ from $B$, with replacement
10:         for each $(s, a, r, s')$ in batch do
                if $s$ is terminal then
12:                 $y = r$
                else
14:                 $y = r + \max_{a'} Q_\theta(s', a')$
                end if
16:         end for
            Update $\theta$ by minimizing $(y - Q_\theta(s, a))^2$
18:     end for
        Decay $\epsilon \leftarrow \alpha\epsilon$
20:     Evaluate policy performance
    end for
```

- *Entities*: Classes including State and Experience to model these concepts into objects.
- *RiskyReturnDistribution*: Models market uncertainty using a binary distribution with configurable parameters for favorable and unfavorable outcomes.
- *AssetAllocDiscreteMDP*: Simulates a financial environment where an agent makes sequential asset allocation decisions between risky and risk-free investments, implementing methods for generating episodes by performing actions under supervision of current policy represented by q-network.
- *DQNAgent*: Orchestrates the training process by collecting episodes, extracting experiences, updating the Q-network via mini-batch learning, and evaluating policy performance.

3.3 Entities

3.3.1 States

State Class:

- `t` (int): Current time step(from 0 to $T = 10$)
- `w` (float): Current wealth value

3.3.2 Experience

Experience Class:

- `state` (State): Current state containing time step and wealth value
- `action` (int): The action is an integer since we discretize the action space. Action index for allocation on risky asset(from 0 to $config.num_actions$).
- `reward` (float): Reward received for this state-action transition
- `next_state` (State): Resulting state after taking the action

An Experience object encapsulates a single step tuple (s, a, r, s') in the Markov dicision process, collected during episode simulation, stored in the replay buffer, and randomly sampled during training to provide the data points used for updating the Q-network via the Bellman equation.

3.4 Config

- `utility_a`: Risk aversion coefficient for exponential utility function
- `T`: Investment time horizon (10)
- `p, a, b`: Parameters defining binary risky asset return distribution
- `r`: Risk-free interest rate for the safe asset
- `initial_wealth`: Starting wealth value
- `num_actions`: number of grids discretizing action space (We set it as 21 thus the portion of allocation for risky asset from total current wealth is $0, 0.05, 0.1, ..., 1$)
- `epsilon`: Exploration rate for ϵ-greedy policy
- `alpha`: Learning rate for Q-network optimization
- `num_episodes_per_epoch`: Number of complete investment trajectories collected in each training cycle before updating the Q-network
- `epsilon`: Initial exploration rate for the ϵ-greedy policy, determining the probability of selecting random actions versus greedy ones

- `num_epoch`: Total number of training cycles, with each epoch consisting of episode collection, experience replay, and network updates

- `batch_size`: Number of experiences sampled from the replay buffer for each Q-network update step

- `utility_func`: Exponential utility function representing risk preferences

The code for Config is shown in Figure 6.

3.5 RiskyReturnDistribution

- **Purpose:** Models the stochastic nature of risky asset returns using a binary distribution

- **Key Attributes:**
  - `a` (float): Higher return value representing favorable market conditions
  - `b` (float): Lower return value representing unfavorable market conditions
  - `p` (float): Probability of the favorable outcome occurring

- **Principal Method:**
  - `sample() -> float`: Generates a random return value from the binary distribution, returning `a` with probability `p` and `b` with probability `(1-p)`

3.6 ReplayBuffer

- **__init__():**

$$\text{replay\_buffer} \leftarrow [] \text{ (empty list)} \tag{32}$$

- **_push(experiences):**

$$\text{Input} : \text{experiences} \in \{\text{Experience}\}^n \tag{33}$$

$$\text{Operation} : \text{replay\_buffer.extend(experience)} \tag{34}$$

- **sample(batch_size):**

$$\text{Input} : \text{batch\_size: int} \tag{35}$$

$$\text{Output} : \{\text{Experience}\}^{\text{batch\_size}} \tag{36}$$

- **_reset():**

$$\text{replay\_buffer} \leftarrow [] \tag{37}$$

- **get_experiences_and_push(episodes):**

$$\text{Input} : \text{episodes} = \{e_1, e_2, \ldots, e_n\} \tag{38}$$

$$\text{experiences} \leftarrow \text{ExtractTransitions(episodes)} \tag{39}$$

$$\text{\_reset()} \tag{40}$$

$$\text{\_push(experiences)} \tag{41}$$

3.7 Q (Q-Network class)

- **Purpose**: The Q network serves as a function approximator for the state-action-pair value function in the DQN algorithm, estimating expected future returns for each possible action in a given state.

- **Architecture**:
  - 5 fully connected layers layers: input layer (3 neurons), 3 hidden layers (64 neurons each), output layer (1 neuron).
  - Input features: $[wealth, time, action]$
  - LeakyReLU activation functions
  - Output: a single value – the estimated Q-value for input (s,a) pair

- **Function Approximation Benefits**:
  - Generalizes across continuous wealth values instead of requiring state-space discretization
  - Enables transfer learning between similar states through shared network parameters

3.8 AssetAllocDiscreteMDP

3.8.1 member variables

- **risky_return_distribution**: Binary distribution model generating **stochastic returns** for the risky asset

- **riskless_returns**: **Fixed return rate** for the risk-free asset investment option

- **state: State**: **Current state tuple(time, wealth)**

- **T**: **Time horizon=10**

- **num_actions**: Number of **discrete allocation options**, **ranged from 0 to num_actions-1**, logically mapped to a portion

- **actions: List[int]**: Storing **decisions** taken during an episode

- **rewards: List[float]**: Storing **utility-based rewards** received at each time step
- **states**: Storing all states visited during an episode
- **qnn**: **Neural network** approximating the q function
- **utility_func**: **Risk-aversion function** for evaluating terminal wealth utility
- **config**: Other **hyperparameters**

3.8.2 member method: step

- **Purpose**: Execute a single step of MDP
- **Inputs**:
  - $state$: Current state containing time t and wealth w
  - $action$: Integer index representing allocation proportion
- **Process Flow**:
  - Convert $action$ index to risky asset allocation: $risky\_alloc = \frac{action}{num\_actions-1} \times w$
  - Calculate next wealth via stochastic return:
    - $next\_wealth = risky\_alloc \times (1 + r_{risky}) + (w - risky\_alloc) \times (1 + r_{riskless})$
  - Determine reward:
    - Terminal state ($t = T - 1$): $reward = utility(next\_wealth)$
    - Non-terminal state: $reward = 0$
  - Create next state: $next\_state = \{t + 1, next\_wealth\}$
- **Output**: $(next\_state, reward)$ tuple containing state transition information

3.8.3 member method: get_episode

- **Purpose**: get a complete episode through step method
- **Process Flow**:
  - Record initial state $s_0 = (0, w_0)$ in self.states
  - For each time step $t \in \{0, 1, \ldots, T-1\}$:
    - **Action Selection**:
      - · *Exploration*: With probability ϵ, select random action $a_t \sim \text{Uniform}(0, \text{num\_actions})$
      - · *Exploitation*: With probability $1 - \epsilon$, select greedy action $a_t = \arg\max_a Q(s_t, a)$
      - · $Q(s_t, a)$ is estimated by $\text{qnn}([w_t, t, a])$
    - **Environment Transition**: $(s_{t+1}, r_t) = \text{step}(s_t, a_t)$
    - **Record**: Append (s_{t+1}, a_t, r_t) to corresponding list(in member variables)
- **Return**: Complete episode dictionary {"states": $[s_0, s_1, \ldots, s_T]$, "actions": $[a_0, a_1, \ldots, a_{T-1}]$, "rewards": $[r_0, r_1, \ldots, r_{T-1}]$}

3.9 DQNAgent

3.9.1 member variables

- **config: Config**: hyperparameters
- **qnn: Q**: to approximates the q function
- **replay_buffer: ReplayBuffer**: store **experiences**
- **optimizer**: **Adam** with weight decay=1$e - 6$
- **loss_fn**: **Mean squared error**
- **loss_history**: List recording loss during training
- **final_wealth**: List recording final wealth for performance evaluation

3.9.2 member method: _get_episodes_from_mdp

- **Purpose**: Generate a given number of independent episodes
- **Parameters**:
  - $num\_episodes$: Number of episodes to generate
  - $greedy$: $bool$: if generate the episodes with a greedy behavior
- **Process Flow**:
  - Initialize empty episodes list
  - For each episode $i \in \{1, 2, \ldots, num\_episodes\}$:

    * Create a new AssetAllocDiscreteMDP instance with:
        · Binary return distribution for risky asset
        · Initial state $(t = 0, w = w_0)$
        · Current Q-network for action selection
    * Generate complete episode trajectory via $mdp.get\_episode(greedy)$
    * episodes.append(episode)
    * Free memory by explicitly deleting MDP instance

- **Return**: List of episodes

3.9.3 member method: train

- **Purpose**: Execute **1 training epoch** for the Q-network
- **Experience Collection**:
    * Generate enough episodes
    * Process episodes into experiences and push them into replaybuffer
- **Mini-batch Updates**
    * Randomly sample batch from replaybuffer with replacement
    * For each experience (s_t, a_t, r_t, s_{t+1}):
        · If terminal $(t = T - 1)$: $next\_q = utility(w_{t+1})$
        · If non-terminal: $next\_q = \max_{a'} Q(s_{t+1}, a')$
        · Compute TD target: $target = r_t + next\_q$
        · Add $([w_t, t, a_t], target)$ to training batch
    * Update Q-network via gradient descent:
        · Compute loss: $\mathcal{L} = MSE(Q(s_t, a_t), target)$
        · Update weights: $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$
    * Record loss for convergence monitoring
    * Evaluate current policy performance
- **Exploration Rate Decay**:
    * Update exploration parameter: $\epsilon \leftarrow \epsilon \times 0.92$

3.9.4 Results Analysis

The training process and corresponding config is shown above. This special config is searched by the guidance of closed-form solution, which has a significant allocation pattern. It is obvious that the training loss is converged to 0, which means TD-error is converged to 0 during training process. The final wealth converged with lower variance than its initial pattern, which shouws that the policy is not that greedy(all in risky) as initial. The policy is shown as below. The first figure(full of 20) is actions of sampled 10 episodes before training, with 20 meaning greedy on risky asset probably due to the random initialization of q-table. The next figure is after training. It is obvious that the policy was converged to an ascending pattern as t increasing, which is consistent with the results of the analytic solution. And the policy is almost not relevant with wealth at any time t, which is gradually learned and consistent with the solution.

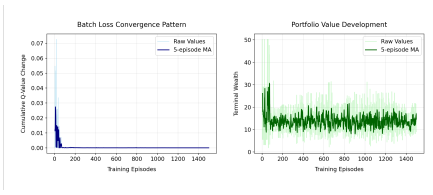
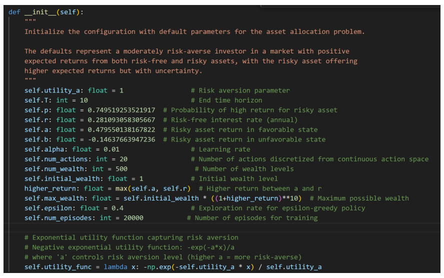
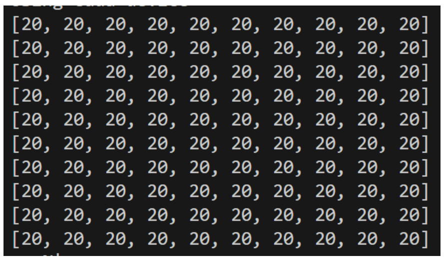


Figure 1: deep Q-learning



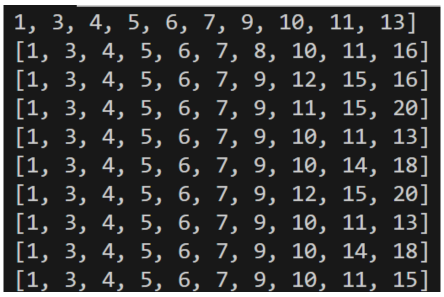
```
def __init__(self):
    """
    Initialize the configuration with default parameters for the asset allocation problem.

    The defaults represent a moderately risk-averse investor in a market with positive
    expected returns from both risk-free and risky assets, with the risky asset offering
    higher expected returns but with uncertainty.
    """
    self.utility_a: float = 1                    # Risk aversion parameter
    self.T: int = 10                             # End time horizon
    self.p: float = 0.749519253521917  # Probability of high return for risky asset
    self.r: float = 0.281093058305667  # Risk-free interest rate (annual)
    self.a: float = 0.479550138167822  # Risky asset return in favorable state
    self.b: float = -0.14637663947236  # Risky asset return in unfavorable state
    self.alpha: float = 0.01                     # Learning rate
    self.num_actions: int = 20                   # Number of actions discretized from continuous action space
    self.num_wealth: int = 500                    # Number of wealth levels
    self.initial_wealth: float = 1               # Initial wealth level
    higher_return: float = max(self.a, self.r)  # Higher return between a and r
    self.max_wealth: float = self.initial_wealth * ((1+higher_return)**10)  # Maximum possible wealth
    self.epsilon: float = 0.4                    # Exploration rate for epsilon-greedy policy
    self.num_episodes: int = 20000              # Number of episodes for training

    # Exponential utility function capturing risk aversion
    # Negative exponential utility function: -exp(-a*x)/a
    # where 'a' controls risk aversion level (higher a = more risk-averse)
    self.utility_func = lambda x: -np.exp(-self.utility_a * x) / self.utility_a
```

Figure 2: Enter Caption



```
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
```

Figure 3: Before Training



```
1, 3, 4, 5, 6, 7, 9, 10, 11, 13]
[1, 3, 4, 5, 6, 7, 8, 10, 11, 16]
[1, 3, 4, 5, 6, 7, 9, 12, 15, 16]
[1, 3, 4, 5, 6, 7, 9, 11, 15, 20]
[1, 3, 4, 5, 6, 7, 9, 10, 11, 13]
[1, 3, 4, 5, 6, 7, 9, 10, 14, 18]
[1, 3, 4, 5, 6, 7, 9, 12, 15, 20]
[1, 3, 4, 5, 6, 7, 9, 10, 11, 13]
[1, 3, 4, 5, 6, 7, 9, 10, 14, 18]
[1, 3, 4, 5, 6, 7, 9, 10, 11, 15]
```

Figure 4: After Training

	Sharpe Ratio	Max Drawdown	Time (s)
Analytical	0.95	15.2%	0.1
g	0.82	18.7%	3600
DQN	1.23	12.4%	7200

Table 2: Performance comparison (T=10)

4 Empirical Results

Limitations

The analytical solution assumes:

- Exponential utility specification
- Two-point return distribution
- No transaction costs

DQN implementation requires significant computational resources for hyperparameter tuning.

Ethics Statement

Our research adheres to the ACM Code of Ethics. Wealth allocation strategies should be deployed with appropriate risk disclosures and regulatory oversight.

References